An Approach to Managing Feature Dependencies for Product Releasing in Software Product Lines*

Yuqin Lee, Chuanyao Yang, Chongxiang Zhu, and Wenyun Zhao

Computer Science and Technology Department, Fudan University,Shanghai 200433, China
li_yuqin@yahoo.com.cn, Yangcy9216@163.com,
{cxzhu, wyzhao}@fudan.edu.cn

Abstract. Product line software engineering is a systematic approach to realize large scale software reuse. Software product lines deal with reusable assets across a domain by exploring requirements commonality and variability. Requirements dependencies have very strong influence on all development phases of member products in a product line. There are many feature oriented approaches on requirement dependencies. However, most of them are limited to the problem domain. Among those few focusing on the solution domain, they are limited to modeling requirement dependencies. This paper presents a feature oriented approach to managing domain requirements dependencies. Not only is a requirement dependencies model presented, but a directed graph-based approach is also developed to analyze domain requirement dependencies for effective release of member products in a product line. This approach returns a simple directed graph, and uses an effective algorithm to get a set of requirements to be released in a member product. A case study for spot and futures transaction domain is described to illustrate the approach.

1 Introduction

One of the approaches to successfully realizing large scale software reuse is product line engineering. Its goal is to support the systematic development of a set of similar software systems by understanding and controlling their common and particular characteristics [1]. Analyzing requirements commonality and managing variability is an important activity for domain engineering (DE) and software product line engineering (SPLE). Managing requirements for a software product line(SPL) is a lot more complex and difficult than that for an individual application. Domain requirements commonality and variability are developed into reusable assets of a SPL. Obtaining a proper set of reusable requirements is the key to achieve successful DE activities, for these requirements are not only the inputs of subsequent steps in DE, but also help form requirement models for application engineering (AE) [2]. Many researchers

* Supported by the National Natural Science Foundation of China under Grant No. 60473061; the National High Technology Development 863 Program of China under Grant No.2005AA113120.

M. Morisio (Ed.): ICSR 2006, LNCS 4039, pp. 127 – 141, 2006.

have recognized that individual requirements are seldom independent of each other, and various kinds of dependencies exist among them [3, 4, 5, 6, 7, 8, 9].

Dependencies are essential elements among the requirements of a real software system, because of the cohesion of a system. The cohesion is a basic quality that is necessary for a system to be a system, and to achieve certain customer-desired goals [10]. A SPL deals with a set of member products, so dependencies exist among requirements of SPL. Otherwise the requirements would be unrelated and it would be unnecessary to use SPL to develop them. Dependencies among the requirements of a domain have not only positive but also negative effects. Positive dependencies are useful to achieve a requirements set for a member product from SPL, and guide the release planning for a software system. Negative dependencies will lead to requirement conflicts and inconsistencies. These conflicts and inconsistencies have to be managed during analysis phase and along with system evolution.

A feature is a set of tight-related requirements from the stakeholders' viewpoint. Features are not independent in a system. Feature dependencies reflect requirement dependencies. There are many feature oriented approaches on managing requirement dependencies. However, most of them are limited to the problem domain. Approaches focusing on the problem domain emphasize static relations among features. Dynamic relations and behavior characteristics of requirements are difficult to represent. Among those few focusing on the solution domain, they are limited to modeling requirement dependencies. Few of them deal with managing feature dependencies and getting requirement sets for a member product from a SPL.

Feature dependencies have to be represented in domain models. A feature dependencies model influences how effective to configure member products in a SPL. Tree structure can not represent feature dependencies, because not all feature relations are hierarchical. Representing the nonhierarchical dependencies relationships into a tree structure has left the current feature modeling methods with the possibility of either omitting dependencies or losing control over the feature model [11]. Feature dependencies relationships are a graph intuitively. The graph can be processed using a matrix or an adjoining table. It's the groundwork in achieving the concise graph representing feature dependencies.

The proposed method not only defines a classification of feature dependencies and presents a feature dependencies model, but also uses a directed graph to analyze domain requirement dependencies for effective release of member products in SPL. This approach returns a simple directed graph that includes only direct feature dependencies, and uses an effective algorithm to get the set of requirements which are to be released in a member product.

The paper is organized as follows: Section 2 discusses related work on feature dependencies. Section 3 defines a classification of feature dependencies in a SPL. Section 4 presents a feature dependencies model based on directed-graph, and provides an algorithm that generates the maximum connective dependencies graphs. It can be easily used to get the features set for a member product from a SPL, and detect conflicting dependencies. Section 5 illustrates and analyses our approach by an example of spot and futures transaction domain. Section 6 draws a conclusion and some suggestions for future work.

2 Related Work

Some different approaches exist on how to deal with feature dependencies. Most of them focus on feature dependencies modeling, and some of them deal with analysis or management of feature dependencies. The following are some approaches that deal with software release planning.

Claes Wohlin and Aybüke Aurum [20] presented an empirical study of the decision criteria when selecting a set of requirements to implement in a forthcoming project, and hence to postpone the implementation of other requirements to a later point in time.

Omolade Saliu and Guenther Ruhe [21] described ten key technical and non technical aspects impacting release planning, and evaluated seven existing release planning methods. They proposed a new release planning framework that considers the effect of existing system characteristics on release planning decisions.

Par Carlshamre and Bjorn Regnel1 [22] compared two independently developed industrial market-driven requirements engineering processes, which both apply continuous requirements management using state-oriented life cycle models in the fostering of requirements from invention to release.

Hassan Gomaa, Michael E. Shin [12] proposed a multiple-view meta-model for SPLs to describe how each view relates semantically to other views. The meta-model depicts life cycle phases, views within each phase, and meta-classes within each view. The relationship between the meta-classes in the different views was described. Consistency checking rules were defined based on the relationships among the meta-classes in the meta-model. The approach dealt with feature dependencies but did not go deep into it.

K. Lee and K. C. Kang [13] extended the feature modeling into analyzing feature dependencies that are useful in the design of reusable and adaptable product line components, and presented design guidelines based on the extended model. Although the structural relationships and configuration dependencies are essential inputs for product line asset development, they are not sufficient for development of reusable and adaptable product line assets. They gave out six types of feature dependencies which have significant influences on the design of product line assets. They emphasized on solutions to hide variable features from the client features which use them.

H. Ye and H. Liu [15] presented a matrix-based approach to model feature dependencies in a scalable way. Three hierarchical relationships and three non- hierarchical relationships were identified, but the dependencies in each class were not representative.

W. Zhang, H. Mei et al [10] identified static, dynamic feature dependencies. They also identified feature dependencies on the specification level. They emphasized feature dependencies modeling and interaction among different dependencies types. Dependencies on specification level are not sharing the same classification condition with the two other types of dependencies.

We define a classification of feature dependencies in a static and dynamic way, and propose a feature dependencies model. The directed graph is used to analyze domain requirements dependencies for effective configuration of member products in a product line. This approach returns a simpler directed graph, and uses an effective algorithm to generate the maximum of connective dependencies graphs. It can be used to get a set of requirements for a member product more easily. The characteristics make

the approach more precise, easily understood and very effective for producing requirement set of a member product from a SPL.

3 Feature Dependencies

3.1 Feature Variability

Features in a SPL can be classified into two types: mandatory and variable features.

Mandatory features are those which must be present in all member products in a SPL.

Variable features are those that may not be present in all member products in a SPL.

3.2 Feature Dependencies

The dependencies among features can be classified into static and dynamic ones. Static feature dependencies show the intrinsic relations existing among features, such as whole-part relations and static constraints etc. Dynamic feature dependencies show the operational relations among features, such as sequential relation illustrates features that have to be active one after another. Each kind of dependencies will be discussed in detail. We use contiguous lines to represent static dependencies and broken lines for dynamic dependencies in the following figures.

3.2.1 Static Dependencies

The static dependencies reflect hierarchical feature relations and static constraints among features on the same level. They include decomposition, generalization, and static constraints. Static constraints include required and excluded. Decomposition and generalization reflects dependencies between parent and child features. Static constraints reflect dependencies between peer features, especially different variants in one variable point.

Decomposition: When a parent feature is decomposed into a number of children features, the relation between parent feature and child feature is called decomposition dependency. For instance, in spot and futures transaction product lines, a feature trade is decomposed into two features called order and match. The dependency between trade and order or match is decomposition. The figure 1 describes the decomposition dependency. We use rectangles to annotate feature, and diamonds to annotate decomposition dependency.

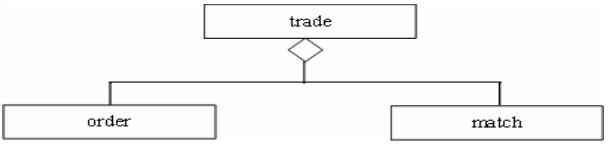

Fig. 1. Decomposition dependency example

Generalization: When a parent feature is generalized from a number of children features, the relation between parent feature and child feature is called generalization dependency. For instance, Match deal feature is generalized from forward, bidding, and auction etc.

We use triangles to annotate generalization dependency. The generalization dependency is illustrated in figure 2.

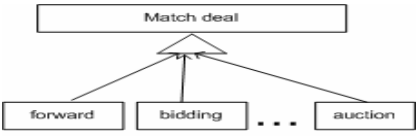

Fig. 2. Generalization dependency example

Static Constraints: If one feature is required or excluded by another feature to constitute a member product, the relation between the two features is called required or excluded. Required dependency is unidirectional. Excluded dependency is bidirectional. For instance, Cancel order feature requires Order feature, so Order feature is required by Cancel order feature. Local market feature can not coexist with Global market feature, so Local market feature is excluded by the Global market feature; meanwhile Global market is excluded by Local market. The static constraint dependencies are illustrated in figure 3.

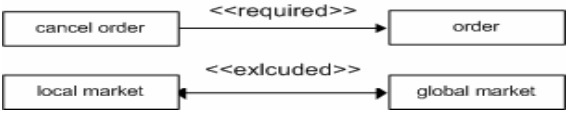

Fig. 3. Static constraints dependency example

3.2.2 Dynamic Dependencies

In addition to static dependencies, there are several relations reflecting dynamic dependencies between features. Dynamic dependencies are described as following.

Serial: If two features should be active immediately one after another, the relation of the two features is called serial dependency. Serial dependency may represent precondition and post condition relations. The feature being active first is called precondition feature, the other is called post condition feature. For instance, Match deal and Create order form are serial features. Match deal should be active immediately after Create order form, so Create order form feature is the precondition of Match deal feature. Serial dependency can represent control flow and data flow operations. The serial dependency is illustrated in figure 4.

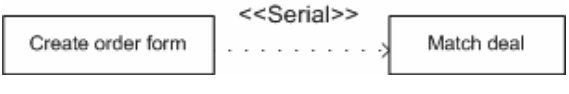

Fig. 4. Serial dependency example

`*Collateral:* If two or more features should be active at the same time, the relation between these features is called collateral dependency. Collateral dependency is bidirectional. For instance, Order and Match should be active when the trading system is operating. Order feature will deal with order request related operations. Match feature will deal with matching order requests. The system is a real time transaction product, so the two features should be active at the same time during trading period. The collateral dependency is illustrated in figure 5.

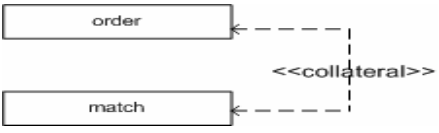

Fig. 5. Collateral dependency example

Synergetic: If two or more features should be synchronized sometime during their active period, the relation between them is called synergetic dependency. Synergetic dependency represents serial relation existing in concurrent operations. Synergetic dependency is bidirectional. For instance, in spot and futures transaction product line, feature Order and feature Match have to be active parallel, but Match has to wait to cooperate with Order by order queue. Synergetic dependency describes two or more features working concurrently to fulfill a task. The synergetic dependency is illustrated in figure 6.

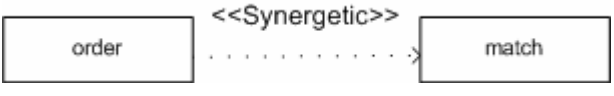

Fig. 6. Synergetic dependency example

Change: If one feature can be changed by another feature, the relation between these two features is called change dependency. The feature changing other features is called changer, and the feature being changed by others is called changee.

Change dependency can be classified into the following kinds: state change, behavior change, data change, and code change.

State change dependency represents the relation that one feature can change another feature's state when they are active. For instance, a feature is changed from unbound to bound when it is needed by other features. In spot and futures transaction product line, Start match market feature will need Match trading feature to be bound to process the corresponding transactions, so Start match market feature has a state change dependency with Match trading feature. Likewise, close match market feature will change Match trading feature's state from bound to unbound, so Close match market feature has a state change dependency with Match trading feature. The state change dependencies are illustrated in figure 7-1.

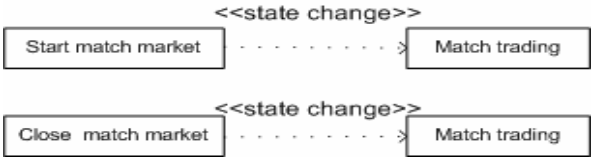

Fig. 7-1. State change dependency example

Behavior change dependency represents the relation that one feature may change another feature's behavior when they are active. For instance, when orders can't be matched, Match deal feature will be idle. When a client enters a new order, the new order will be added into the order queue, and Match deal feature will change from idle to run. So Create order form feature has a behavior change dependency with Match deal feature. The behavior change dependency is illustrated in figure 7-2.

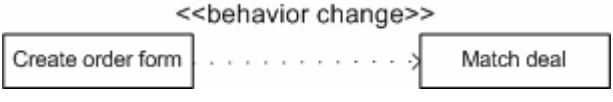

Fig. 7-2. Behavior change dependency example

Data change dependency represents the relation that one feature may change data when being used by another feature. For instance, one order may be matched partly when there is not enough suitable reversed order, e.g. partial match. So the matched and unmatched amount of the order is changed by Match deal. The matched and unmatched amount is not zero but less than the order amount. The Match deal feature has a data change dependency with order feature. The data change dependency is illustrated in figure 7-3.

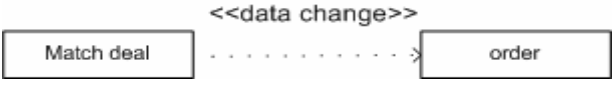

Fig. 7-3. Data change dependency example

Code change dependency represents a relation where one feature may change another feature's code. For instance, in the security model, Anti-trace feature will change Security protect feature's code. The code change dependency is illustrated in figure 7-4.

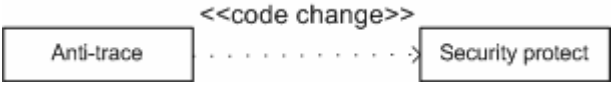

Fig. 7-4. Code change dependency example

4 Managing Feature Dependencies

Feature dependencies will influence how to get features set of a member product in a SPL, and how to release products in an incremental way. So how to represent feature dependencies in an understandable way is a challenge. We use directed graphs to represent feature dependencies, and assign an eigenvalue to each type of dependencies. In directed graph, eigenvalue of every directed path represents combined dependencies from one feature to the other.

4.1 Feature Dependencies Model

Among features in a SPL, usually majority dependencies exist among minority features. In order to decrease workload of feature dependencies analysis, we first set apart those features which have no dependencies with other features. These features are called isolated features. Then we analyze dependencies pair by pair.

Every type of dependency is assigned an eigenvalue. We use a byte to denote dependencies. Each bit of the byte represents a type of dependency. This method is used to represent combined dependencies easily and directly. If two features have more than one dependency, the eigenvalue of all dependencies between them is assigned a combined value with a corresponding bit representing a dependency. The eigenvalues of all dependencies are assigned in table 1.

Table 1. Eigenvalue of dependencies

Type of dependency	Assigned eigenvalue(binary)
decomposition	01000000 (0x 40)
generalization	00100000 (0x 20)
required	00010000 (0x 10)
excluded	10000000 (0x 80)
serial	00001000 (0x 08)
collateral	00000100 (0x 04)
synergetic	00000010 (0x 02)
change	00000001 (0x 01)

Table 2. Feature dependencies table

Feature(source)	Dependency	Feature(destination)
Feature1	decomposition (0x40)	Feature7
Feature2	Synergetic(0x02)	Feature1
Feature1	Synergetic(0x02)	Feature2
Feature3	required (0x10)	Feature4
Feature3	serial (0x08)	Feature4
Feature3	change (0x01)	Feature5
Feature5	Collateral(0x04)	Feature2
Feature2	Collateral(0x04)	Feature5
Feature6	excluded (0x80)	Feature1
Feature1	excluded (0x80)	Feature6

```
DependencyForestGenerator(T: in dependency table, F: out dependency forest)
F.TreeNo=0;
FeatureSet ; //feature set consists of all features;
While T not tableEnd
{
ReadOneRowFromTable(T, R);//T is the dependency table, R is the gotten row;
If R.sourceFeature exists in the dependency forest, and is the root of tree K
  Then If R.sourceFeature has a path to R.destinationFeature
                Then eigenvalue of the path = + R.dependency
Else create a directed path from R.sourceFeature to R.destinationFeature with eigenvalue R.dependency in tree K;
          Else {
Create tree F.TreeNo++;
F.TreeNo.root= R.sourceFeature;
FeatureSet=FeatureSet – F.TreeNo.root;
}
}
While FeatureSet not empty //generate trees with only roots to represent isolated features
{
      F.TreeNo ++;
      F.TreeNo.root= get a feature from FeatureSet;
FeatureSet=FeatureSet - F.TreeNo.root;
}
//end of algorithm DependencyForestGenerator.
```

Fig. 8. Algorithm for generating feature dependency forest

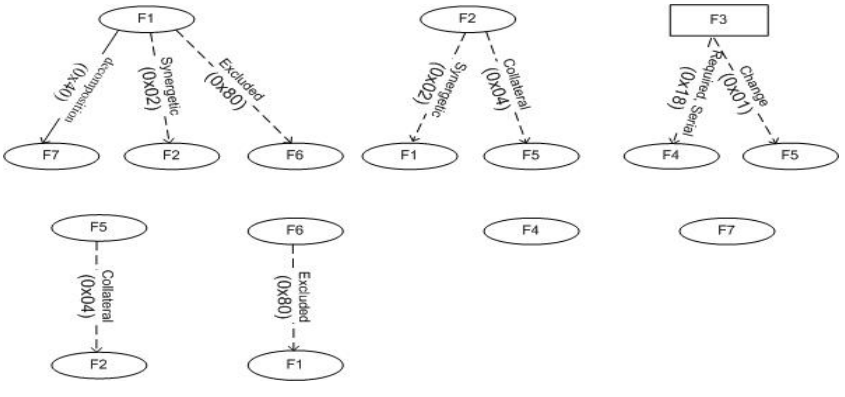

Fig. 9. Example of feature dependency forest generated

Rectangles are used to represent mandatory features, and ellipses are used to represent variable features. Directed path represents direct dependency from one feature to another feature. Eigenvalue of each path represents combined dependencies eigenvalues from one feature to another feature.

```
MaximalConnectiveDependencyGraphGenerator(F: in dependency forest, G: out graph)
 //generate maximum connective feature dependency graphs;
initial G is null;
initial F[i]=0, i=0..n, n is the number of features.
//F[i]=0 representing tree i has not be processed, the value will be 1 after processed;

Repeat
{
If a mandatory feature which is the root of one feature tree in the forest F exists, and F[i]=0
Then G.node =i ;  //i is the mandatory feature number in feature set.
Else If a variable feature j which is the root of one feature tree in the forest F exists, and F[j]=0

   Then G.node =j ; //j is the variable feature number in feature set.
Else G.node = null;

While G.node not null and F[G.node]=0
{
Take feature tree G.node;
If G.node tree has a child Then NextNode = G.node.child;
        While NextNode not null
        {
                If NextNode is in G
Then
        Add NextNode to G;
Add a path from G.node to NextNode;
eigenvalue of the path= eigenvalue from G.node to NextNode in tree G.node;

If F[NextNode]=0 then insert NextNode to queue WaitingProcessedTrees;
If tree G.node has another child unprocessed
Then
NextNode = next child of tree G.node
Else   NextNode = null;
        }  // tree G.node has been processed;
        F[G.node]=1; //tree G.node has been processed;
        If queue WaitingProcessedTrees is not empty Then G.node = the first node pop out of the queue;
        Else G.node = null;
}
// get one maximum connective dependency graph; the graph is directed, and two features may have
// bidirectional paths.

} until G.node is null;
// end of repeat; all maximum connective graphs returned.
// end of algorithm MaximalConnectiveDependencyGraphGenerator.
```

Fig. 10. Algorithm for generating maximum connective directed graphs

After analyzing dependencies between features in DE, we get a feature dependencies table. Each row in the dependency table represents a direct dependency between two features. The represented dependency is has a direction. Bidirectional dependency is represented by two rows; each feature of the dependency is the source feature in one row. In our classification, collateral, synergetic and excluded dependencies are bidirectional. A simple example is described in the following table 2. The example has seven features. Feature 3 is mandatory, and the other features are variable.

Based on feature dependency table, we design an algorithm DependencyForestGenerator to generate feature dependency forest. The feature dependency forest consists of n trees, where n is the number of features. Every feature is the root of one tree. The trees are called feature dependency trees. Only direct dependencies are presented in feature dependency trees, no implicit dependencies are presented in feature dependency trees. If there is more than one dependency between two features, all the dependencies are presented in one path by combined eigenvalue.

The algorithm DependencyForestGenerator is described in figure 8.

Using algorithm DependencyForestGenerator, the feature dependency forest in figure 9 below is generated.

The feature dependency forest describes feature dependencies clearly. Each feature dependency tree represent dependencies related with one feature. This approach is simpler and more straightforward than matrix-based and table-based methods. Only direct dependencies are represented on trees. Implied dependencies can be gained by transitional relations reasoning.

4.2 Managing Feature Dependencies

The feature dependency forest discussed above is a representation of direct dependencies. The forest only describes direct dependencies, but implied dependencies are not represented. When we want to get a feature set for releasing a member product, implied dependencies have to be used. Based on feature dependency forest, an algorithm generating maximal connective graphs is developed in the following figure 10. Implied dependencies can easily be gained from connective graphs.

Using algorithm MaximalConnectiveDependencyGraphGenerator, maximum connective feature dependency graphs of forest in figure 9 are generated in figure 11.

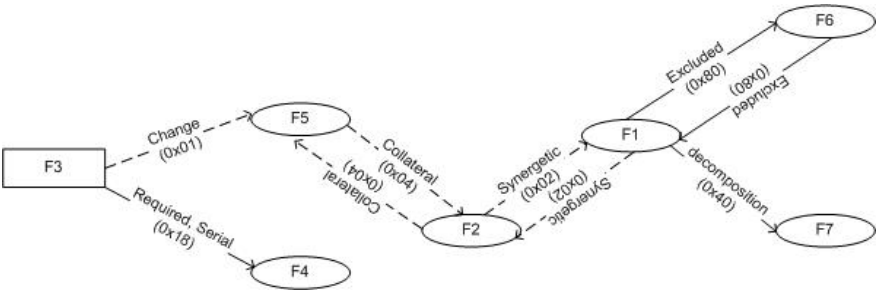

Fig. 11. Maximum connective dependency graphs generated from example above

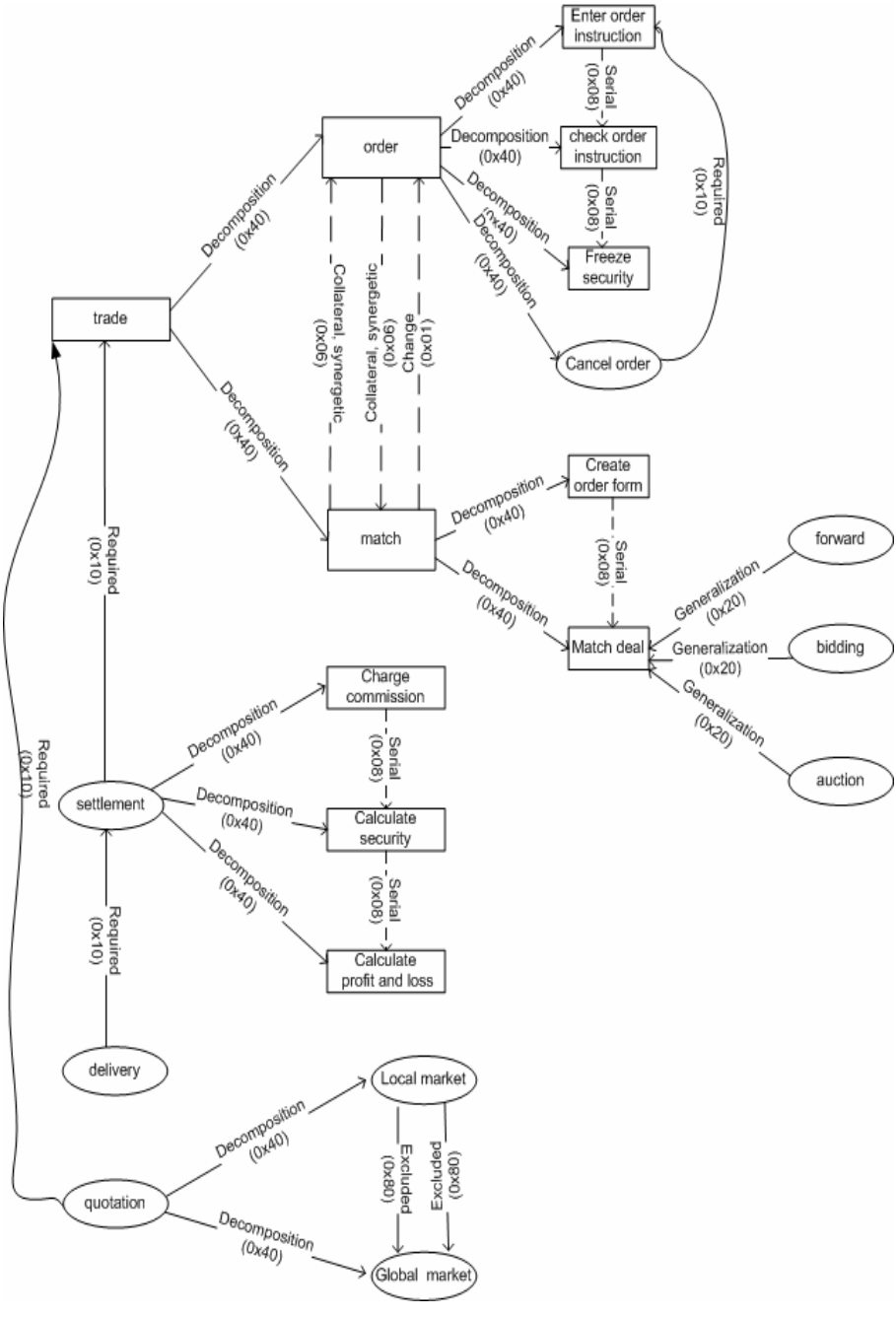

Fig. 12. Feature dependency graph generated for spot and futures transaction

The maximum connective dependency graphs are very effective for generating a features set for product release in SPL. Starting from a feature included in a product release, the features achieved by spreading maximum connective dependency graphs will constitute the member product. Only the graph consisting of the start feature need to spread, other maximal connective graphs can't be used. Until all required features are included in a set, generating process ends. From a few mandatory features, we can easily produce a features set of the product.

5 Case Study

We take the spot and futures transaction product line as an example. After analysis of domain requirements, we get a feature model and feature dependency model. The feature dependency model is the one we're concerned with here. In large SPL, there are a plenty hood of features, so the dependency forest will consist of many trees. A feature model can be included in a feature dependency model. We only describe the feature dependency graph generated in the following figure 12.

In figure 12, rectangles are used to represent mandatory features, and ellipses are used to represent variable features. Static dependencies are annotated by real lines, and dynamic dependencies are annotated by dashed lines. Every path is described by a dependency name and eigenvalue. From the dependencies graph, the relations of features are very concise and clear. For example, trade feature is mandatory in the product line, but settlement, delivery, and quotation are variable. If settlement is needed in a product release, its children features are needed. If settlement is included in a product release, trade feature is included implicitly. Similarly, delivery or quotation feature included in a product release implies that trade feature is included too. All kinds of feature dependencies are included in the figure, so children features of delivery feature are hidden to simplify the graph.

6 Conclusion and Future Work

Product line software engineering is a systematic approach to realize large scale software reuse. SPLs deal with reusable assets across a domain by exploring domain requirements commonality and variability. Domain requirements dependencies have very strong influence on all development phases of member products in a product line. Feature dependencies reflect domain requirement dependencies. How to manage feature dependencies will influence how to release products effectively.

A new feature oriented approach has been developed to model feature dependencies. Feature dependencies are classified as static and dynamic dependencies. Static dependencies reflect hierarchical feature relations and static constraints among features in the same level. They include decomposition, generalization, and static constraints. Dynamic feature dependency shows the operational relations among features. They include serial, collateral, synergetic, and change dependencies. Each kind of dependency is given a notation.

Each type of dependency is assigned an eigenvalue. The feature dependencies are analyzed in a dependency table, representing direct dependencies among features. The

dependency table is transferred to a feature dependency forest. The forest is concise and easily understood. Each feature has a dependency tree whose root is the feature.

Based on feature dependency forest, maximal connective feature dependency graphs are achieved by an effective algorithm. The graphs may include more than one graph when the features have non connected parts. The graphs are useful for releasing products incrementally.

This approach is more effective than other approaches and provides a reasonable dependency classifying method. A feature dependency forest is used to annotate whole domain feature dependencies. Each feature has a tree to represent dependencies related within it. Based on the forest, maximum connective graphs are generated to represent united dependencies in the SPL. It is used to decide which features need to be included in a release.

Based on the feature dependencies model, we will research how different feature dependencies influence architecture design in a SPL, and how to detect and handle conflicts within dependencies.

References

1. Mikyeong Moon, Keunhyuk Yeom, "An Approach to Developing Domain Requirements as a Core Asset Based on Commonality and Variability Analysis in a Product Line", IEEE transactions on software engineering, vol. 31, no. 7, July 2005, pp551-569.
2. Hong Mei, Wei Zhang, Fang Gu, "A feature oriented approach to modeling and reusing requirements of SPLs", Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC'03),2003.
3. P. Carlshamre, K. Sandahl, M. Lindvall, B. Regnell, and J. Natt och Dag, "An Industrial Survey of Requirements Interdependencies in Software Product Release Planning", In Proceedings of Fifth IEEE International Symposium on Requirements Engineering, IEEE Computer Society, 2001, pp. 84-91.
4. A.G. Dahlstedt, A. Persson, "Requirements Interdependencies–Moulding the State of Research into a Research Agenda", In Proceedings of Ninth International Workshop on Requirements Engineering: Foundation for Software Quality, Klagenfurt/Velden, Austria, June 2003, pp. 55-64.
5. S. Ferber, J. Haag, J. Savolainen, "Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line", The Second Software Product Line Conference 2002, LNCS 2379, August 2002, pp. 235–256.
6. J. Giesen, A. Volker, "Requirements Interdependencies and Stakeholders Preferences", In Proceedings of IEEE Joint International Conference on Requirements Engineering, Sep 2002, pp. 206-209.
7. J. Karlsson, S. Olsson, and K. Ryan, "Improved Practical Support for Large-scale Requirements Prioritizing", Requirements Engineering Journal, Vol. 2, No. 1, 1997, pp. 51-60.
8. K. Lee, K.C. Kang, "Feature Dependency Analysis for Product Line Component Design", The Third Software Product Line Conference 2004, LNCS 3107, Aug 2004, pp. 69–256.
9. B. Ramesh, M. Jarke, "Toward Reference Models for Requirements Traceability", IEEE Transactions on Software Engineering, Vol. 27, No. 1, January 2001, pp. 58-93.
10. Wei Zhang, Hong Mei, Haiyan Zhao, "A Feature-Oriented Approach to Modeling Requirements Dependencies", Proceedings of the 2005 13th IEEE International Conference on Requirements Engineering (RE'05),2005.

11. Hein, A., Schlick, M., and Vinga-Martins, R.: 'Applying feature model in industry setting' 'SPLs – experience and research directions' (Kluwer Academic Publishers, Boston, 2000), pp. 47–70
12. Hassan Gomaa, Michael E. Shin, "A multiple-View Meta-modeling Approach for Variability Management in SPLs", ICSR 2004, LNCS 3107, pp274-185, 2004
13. Kwanwoo Lee, Kyo C. Kang, "Feature Dependency Analysis for Product Line Component Design", ICSR 2004, LNCS 3107, pp69-85, 2004
14. M. Sinnema, S. Deelstra, J. Nijhuis, J. Bosch, "Managing Variability in Software Product Families"
15. H. Ye and H. Liu, "Approach to modeling feature variability and dependencies in SPLs", IEE Proc.-Softw., Vol. 152, No. 3, June 2005, pp101-109.
16. P. Carlshamre, K. Sandahl, M. Lindvall, B. Regnell, and J. Natt och Dag, "An Industrial Survey of Requirements Interdependencies in Software Product Release Planning", In Proceedings of Fifth IEEE International Symposium on Requirements Engineering, IEEE Computer Society, 2001, pp. 84-91.
17. J. Giesen, A. Volker, "Requirements Interdependencies and Stakeholders Preferences", In Proceedings of IEEE Joint International Conference on Requirements Engineering, Sep 2002, pp. 206-209.
18. von Knethen, B. Paech, F. Kiedaisch, and F. Houdek, "Systematic Requirements Recycling through Abstraction and Traceability", In Proceedings of IEEE Joint International Conference on Requirements Engineering, Sep 2002, pp. 273-281.
19. Martin S. Feather, Steven L. Cornford, Mark Gibbel, "Scalable mechanisms for requirements interaction management", 2000, IEEE.
20. Claes Wohlin, Aybüke Aurum," What is important when deciding to include a software requirement in a project or release", 2005 IEEE,p246-255
21. Omolade Saliu, Guenther Ruhe, "Supporting Software Release Planning Decisions for Evolving Systems", Proceedings of the 2005 29th Annual IEEE/NASA Software Engineering Workshop (SEW'05)
22. Par Carlshamre, Bjorn Regnell, "Requirements lifecycle management and release planning in market-driven requirements engineering processes", 2000 IEEE, p961-965