ISSN 1884-0760

GRACE TECHNICAL REPORTS

Applying Bidirectional Transformation to Feature Model Refinement – Implementation Issues –

Qiang SUN Bo WANG Zhenjiang HU

GRACE-TR 2011–04 August 2011

CENTER FOR GLOBAL RESEARCH IN
ADVANCED SOFTWARE SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF INFORMATICS
2-1-2 HITOTSUBASHI, CHIYODA-KU, TOKYO, JAPAN

WWW page: http://grace-center.jp/

Applying Bidirectional Transformation to Feature Model Refinement – Implementation Issues –

Qiang SUN[1] Bo WANG[2] Zhenjiang HU[3]

[1]Department of Computer Science and Engineering
Shanghai Jiao Tong University, China
sun-qiang@sjtu.edu.cn
[2] Key Laboratory of High Confidence Software Technologies
Peking University, China
wangbo07@sei.pku.edu.cn
[3] GRACE Center
National Institute of Informatics, Japan
hu@nii.ac.jp

Abstract

In the research of software reuse, feature models have been widely adopted to capture, organize and reuse the requirements of a set of similar applications in a software domain. However, the construction, especially the refinement, of feature models is a labor-intensive process, and there lacks an effective way to aid domain engineers in refining feature models.

In this paper, we implement a tool named FMView to support interactive refinement of feature models based on the view updating technique. The procedure of our tool is to first extract features and relationships of interest from a possibly large and complicated feature model, then organize them into a comprehensible view, and finally refine the feature model through modifications on the view. We successfully apply FMView to refine the web store domain which shows the feasibility of the feature model refinement.

1 Introduction

In domain engineering, feature models [1][2][3] are widely used to capture, organize and reuse the requirements of applications in the same domain. An important step of constructing a feature model is to refine a big, abstract feature into small, concrete features. Different approaches have been proposed to guide the refinement of features. For example, in FODA [1], a set of guiding principles are proposed to help refine feature models. In FORM [2], features are refined in four layers ac-

cording to the feature hierarchy, i.e. capabilities, operating environments, domain technologies and implementation techniques.

Feature models grow large during refinement. Reports [4][5][6] show that real-would feature models often grow beyond a thousands of features, and the largest one reported [7] has more than 5000 features. On the other hand, feature model refinement becomes more and more difficult when the feature model grows. For one thing, it becomes more difficult to find all features related to the current refinement task. For another, it is difficult to locate a specific feature in the large model.

In this paper, we implement a tool FMView[1] supporting interactive refinement of feature models. In our tool, first, domain engineers choose features of interest in the marking phase; Second, all the marked features and relationships are automatically organized into an annotated feature model (updatable view); and finally, after domain engineers refine the view, we transform all view updates into updates on the feature model using the bidirectional transformation technique [8].

We successfully apply FMView to refine the web store domain, which shows that our approach to feature model refinement via modification on updatable view is promising and potentially useful in practice.

The rest of this paper is organized as follows. Section 2 gives some preliminary knowledge on feature model refinement. Section 3 introduces bidirectional transformation mechanism used in FMView tool. Section 4 gives the overview of FMView tool. Section 5 describes the implementation details of the FMView tool. Section 6 concludes the paper and highlights the future work.

2 Feature Model Refinement

Another technique report [9] presents the basic knowledge on the feature models and the approach details of feature model refinement using updatable view.

3 Bidirectional Transformation

The aim of GRoundTram is to solve this problem by providing a linguistic framework for bidirectional model transformation [10]. The framework includes (1) a new model transformation language with clear bidirectional semantics, being equipped with a powerful bidirectional inference mechanism and a virtual machine on which bidirectional transformation model can be efficiently realized; (2) an environment for supporting programming, debugging and maintaining bidirectional model transformation; and (3) a set of application examples and domain-specific libraries that can be used in practice. Figure 1 depicts an architecture (the basic idea) of the compositional framework. A model transformation is described in UnQL+, which is functional (rather than rule-based as in many existing tools) and compositional with high modularity for reuse and maintenance. The model

[1] See http://sei.pku.edu.cn/~ wangbo07/ for the detail.

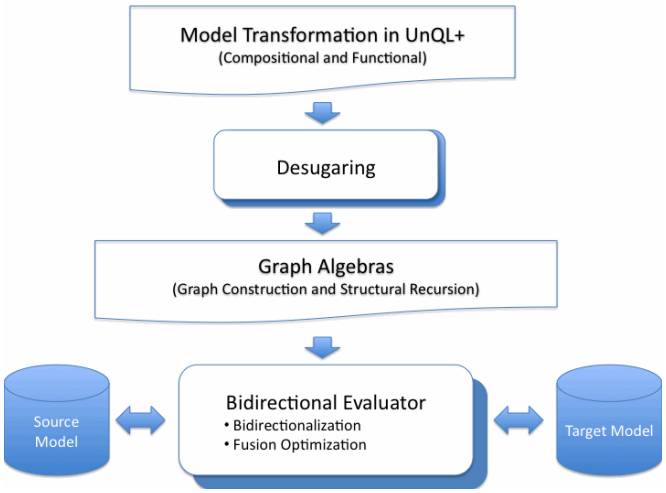


Figure 1: Architecture of GRoundTram Algebraic Framework

transformation is then desugared to a core algebra which consists of a set of constructors for building graphs and a powerful structural recursion for manipulating graphs. This graph algebra can have clear bidirectional semantics and be efficiently evaluated in a bidirectional manner.

Besides the easy implementation of the updatable view, another two advantages of using GRoundTram in our approach is that: 1) GRoundTram maintains the history of the modifications caused by backward transformation, so that we can easily implement an undo functionality to cancel a refinement; 2) GRoundTram records the traceability links between nodes in the source graph and nodes in the target graph, so that we can easily support tracing back from features on the view back to the features in the original model.

For example, Figure 2 shows a snapshot of GRoundTram system, the source graph model and target graph model are displayed in the left and right part, respectively. Suppose that in the target graph model, we first delete the feature *account* and perform the backward transformation. And then, we add the feature *cash* and perform the backward transformation again. History of these changes is reflected to the source graph model (left in Figure 2). In this modified source graph model, the feature deleted is represented by a set of dash nodes and edges and the feature added is colored with purple. In addition, when we select the feature *submitorder* on the target graph model, the selected feature can be traced back to the source graph model with red highlight.

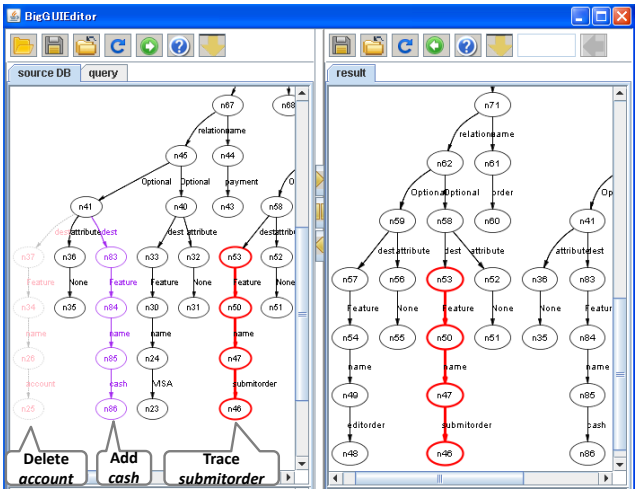


Figure 2: Snapshot of GRoundTram System

4 Overview of FMView

We have developed a tool (FMView) to support our idea of refining feature model with updatable view. FMView is a prototypical implementation and we are developing it within Eclipse as a Plug-In. Figure 3 illustrate the tool and the basic idea of our approach.

The feature model can be constructed in FMView. In this demo, a feature model with 18 features are built using our tool, as shown in the figure. The marked features are filled with green. Based on the marked features, the updatable view is built, as shown in the right part of the figure. Annotations, artificial features and artificial refinements are added to help domain engineer understand the view.

In FMView, modifications on the view can be reflected to the original feature model automatically. In this demo, feature O, P and the require constraint between features K and L are deleted in the view. The corresponding features and constraints in the original feature model are deleted (colored with orange) automatically. Two new features, S and T ,are added as the children of feature K in the view, these modifications are reflected to the original feature model (colored with purple) automatically. In the view, feature R is renamed with R'. In the original feature model, the corresponding feature is also renamed (colored with blue) automatically. To make the refined original feature model more clear, FMView can hide the deleted features in the refined original feature model. In this demo, the deleted feature S and T are hidden in FMView, as shown in the left part of the figure.

5 Implementation of FMView

5.1 Framework Structure of FMView

FMView can be divided into two layers. The fist layer is graphical user interface that makes it easy for domain engineers to design the source model and refine the source model with the view. The second layer is GRoundTram API that makes the view undatable and keep the traceability between the source model and view.

We employ the graphical modeling framework to implement the GUI of FMView. The graphical modeling framework (GMF) is a framework within the Eclipse platform. It provides a generative component and runtime infrastructure for developing graphical editors based on the Eclipse Modeling Framework (EMF) and Graphical Editing Framework (GEF). The project aims to provide these components, in addition to exemplary tools for select domain models which illustrate its capabilities.

GMF has a set of models to create to generate a graphical editor. Figure 4 displays the process involved in creating these models. The first model is the graphical definition, which defines the visual aspects of generated editor. Next is the tooling definition, which comprises things related to editor palettes, menus, etc. Finally, the last model we need is the mapping definition that defines the mapping between the business logic (EMF model) and visual model (graphical and tooling definition).

The feature model is represent by EMF model in the GUI layer. We represent, in GRoundTram, a feature model by a source graph model, and an updatable view by a target graph model. A forward UnQL+ query is automatically created by analyzing the result of marking phase. Once a forward query is provided, the backward transformation comes for free by the GRoundTram system. In this way we only need to implement a forward query that extracts a view from the feature model, and do not have to write code to reflect the updates back into the source. In the first layer, we have to synchronized two representations of the feature model as illustrated in Figure 5.

5.2 Graph Representation of Feature Model

In order to convert the feature model to its edged-labeled graph representation, a feature model should be divided into some small parts. Each part corresponds to one element of the feature model. The kinds of the feature model element are listed as follows:

- Feature
- Feature Group
- Refinement Relationship: (1) Decomposition; (2) Characterization; (3) Specialization; (4) None.
- Simple Constraints: Require and Exclude

- Complex Constraints: Complex Require and Complex Exclude

A graph segment is a tuple (E, V, IN, OUT), where:

- E is an edge set;
- V is a node set;
- IN is the input node;
- OUT is the output node.

Each graph segment has only one input node and one output node. Two graph segment can be composed by merging the input node and output node.

One feature model element can be translated into a certain graph segment and vice versa. Figure 6 illustrates the corresponding relationship between the feature model elements and the graph segments. The node filled with blue color represents input node and the node filled with yellow color represents output node.

The composition of the feature model elements corresponds to the composition of the graph segments.

Feature Group. Because the feature group not including one feature is meaningless. One feature group has at least one feature. The feature group associates with the features through the edges labeled with *member*. Figure 7 illustrates that the feature group contains the feature $f_1, f_2, ..., f_n$.

Refinement Relationship. Figure 8 illustrates that two features are composed by refinement relationships.

Simple Constraints. Figure 9 illustrates that two features are composed by simple constraints.

Complex Constraints. Figure 10 illustrates that two feature groups are composed by complex constraints.

5.3 Operations on View

Table 1: Valid Operations on View

	Feature	Feature Group	RR	SC	CC	Others
Add	Valid	Valid	Valid	Valid	Valid	Invalid
Delete	Valid	Valid	Valid	Valid	Valid	Invalid
Rename	Valid	-	-	-	-	Invalid
Change Predicate	-	Valid	-	-	-	Invalid

The updatable view is defined by a view and a set of valid operations on it. Domain engineers can refine the feature model by using these valid operations to modify the view. These modifications on the view can be automatically transformed back to the original feature model. The valid operations are provided to facilitate the feature model refinement.

All the valid operations are described in Table 1. The first column lists the types of operations. The rest columns form the classification of the feature model elements in the updatable view. *RR*, *SC* and *CC* represent refinement relationship, simple constraints and complex constraints respectively. The infeasible operations are marked with "-".

5.4 UnQL Generation

In GRoundTram, graphs are edged-labeled in the sense that all information is stored as labels on edges rather than on nodes (the labels on nodes have no particular meaning). It can be directly used to represent model. UnQL, like other query languages, has a convenient select-where structure for extracting information from a graph.

There are three phases in UnQL generation: marking phase, organization phase and generation phase.

Marking Phase. To obtain the view, domain engineers are requested to select some features they want to focus on. The domain engineer may want to refine these features, or they may want to use the feature to help refine other features. FMView helps the domain engineers find more features and relationships during marking phase. These features and relationships can help domain engineers to refine the feature model.

Organization Phase. To make the results of the extraction more comprehensible, we organize marking features and relationships into a view, which has additional annotations and maintains the relative level relations among the these features.

In our approach, we organize the results of the extraction by computing the lowest common ancestor (LCA) of them. For any two features, their LCA is their shared ancestor that is located farthest from the root feature. With LCAs, artificial features, annotations and artificial relationships are created to organize the results of the extraction.

The view is built by an organizing algorithm that adopts a bottom up tree construction strategy, as illustrated as follows. This algorithm takes the marking features as inputs and builds the view as output.

```
1   public static ArtiFeature buildView(HashSet<Feature> selectedSet)
2   {
3       HashSet<Feature> workSet = new HashSet<Feature>();
4       // initialize workSet
5       // selectedSet contains marking features
6       workSet.addAll(selectedSet);
7       // initialize LCATable
8       LCATable.iniLCATableForWorkSet(workSet);
9       // organization body
10      // main Loop
11      while(workSet.size()>1)
12      {
13          FeaturePair fp = findMaxDepthForLCA();
14          Feature lca = LCATable.get(fp);
15          Feature a = fp.getA();
16          Feature b = fp.getB();
17          if(lca == a)
18          {
19              if(!selected(lca))
20              {
```

```
21                                      addChild(lca, b);
22                              }
23                              remove(workSet,b);
24                      }
25                      else if(lca == b)
26                      {
27                              if(!selected(lca))
28                              {
29                                      addChild(lca, a);
30                              }
31                              remove(workSet,a);
32                      }
33                      else
34                      {
35                              if(!workSet.contains(lca))
36                                      add(workSet,lca);
37                              if(!selected(lca))
38                              {
39                                      addChild(lca, a);
40                                      addChild(lca, b);
41                              }
42                              remove(workSet,a);
43                              remove(workSet,b);
44                      }
45              }
46              Feature f = (Feature)workSet.toArray()[0];
47              ArtiFeature root = reference.get(f);
48              // Adjust Relative Level
49              if(root != null)
50                      adjust(root);
51              else
52              {
53                      root = StapFactory.eINSTANCE.createArtiFeature();
54                      root.setName("anonymous");
55                      root.getFeatures().add(f);
56              }
57              return root;
58      }
```

Generation Phase. This phase takes the view as input and generate UnQL query. The query consists two part: select clause and where clause. The select clause is generated according to the tree structure of the view. Each feature in where clause is accessed by it path in the graph representation of the feature model. For example, the following UnQL query is generated automatically by a given view.

```
select {
    ArtiFeature:{name:{Onlineshop:{}}, relations:{
        Refinement:{
            destination:{
                ArtiFeature:{
                    name:{anonymous:{}},
                    relations:{Refinement:$G98f9c2}
                }
            }
        },
        Refinement:$G1a30706}
    }
}
where
    $src in $db,
    $G98f9c2 in
        (select $G where
            {root.Feature.relations.Mandatory.Decomposition.destination.
                Feature.relations.Mandatory.Characterization.destination.
                Feature.relations.Optional.Specialization:$G} in $src,
            {destination:{Feature:{name:{Account:$g}}}} in $G),
    $G1a30706 in
        (select $G where
            {root.Feature.relations.Mandatory.Decomposition.destination.
                Feature.relations.Mandatory.Decomposition:$G} in $src,
            {destination:{Feature:{name:{Submitorder:$g}}}} in $G)
```

The UnQL sentence can be executed by GRoundTram to perform the forward transformation which will produce more additional files that are necessary in the backward transformation. The ei file that contains more editing information and

the xg file that represents the result graph being attached to the abstract syntax tree.

6 Conclusion

In this paper, we present the implementation issues on applying bidirectional transformation to feature model refinement. With the view updating technique, we are able to refine large and complicated feature models. The updatable view allows domain engineers to refine feature models in an effective way; they can get the extracted and organized information, and refine the feature model by directly modifying the view. FMView shows the feasibility of this approach.

Acknowledgements

This work was partially supported by the NII Intern Program.

References

[1] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," Carnegie-Mellon University Software Engineering Institute, Tech. Rep., November 1990.

[2] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, "Form: A feature-oriented reuse method with domain-specific reference architectures," *Ann. Software Eng.*, vol. 5, pp. 143–168, 1998.

[3] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Formalizing cardinality-based feature models and their specialization," *Software Process: Improvement and Practice*, vol. 10, no. 1, pp. 7–29, 2005.

[4] D. S. Batory, D. Benavides, and A. R. Cortés, "Automated analysis of feature models: challenges ahead," *Commun. ACM*, vol. 49, no. 12, pp. 45–47, 2006.

[5] F. Loesch and E. Ploedereder, "Optimization of variability in software product lines," in *SPLC*, 2007, pp. 151–162.

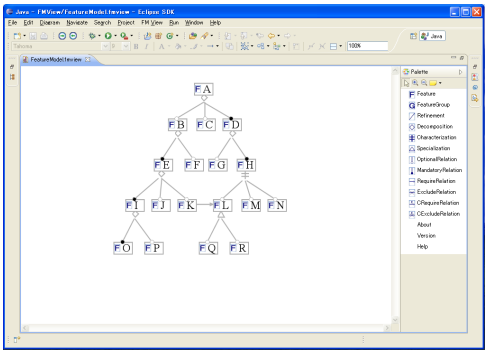
[6] M. Steger, C. Tischer, B. Boss, A. Müller, O. Pertler, W. Stolz, and S. Ferber, "Introducing pla at bosch gasoline systems: Experiences and practices," in *SPLC*, 2004, pp. 34–50.

[7] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "The variability model of the linux kernel," in *VaMoS*, 2010, pp. 45–51.

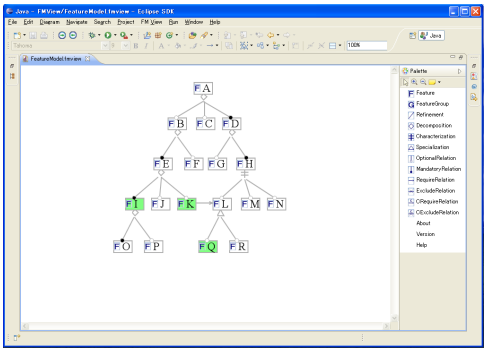
[8] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger, "Bidirectional transformations: A cross-discipline perspective," in *ICMT*, 2009, pp. 260–283.

[9] B. Wang, Z. Hu, Q. Sun, H. Zhao, Y. Xiong, and H. Mei, "Supporting feature model refinement with updatable view," Center for Global Research in Advanced Software Science and Engineering National Institute of Informatics, Tech. Rep., May 2010.

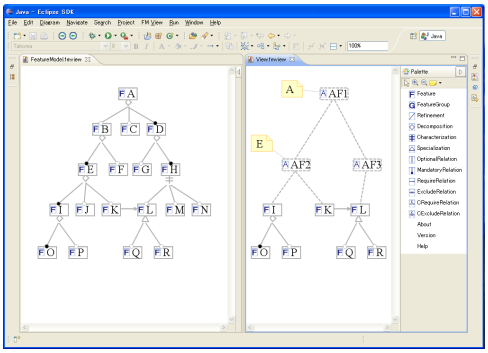
[10] S. Hidaka, Z. Hu, H. Kato, and K. Nakano, "A compositional approach to bidirectional model transformation," in *ICSE Companion*, 2009, pp. 235–238.



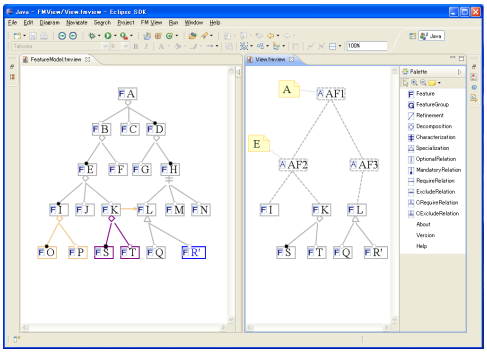
(a) Feature Model Construction



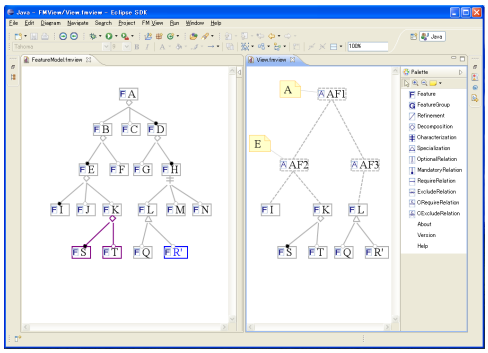
(b) Marking Phase



(c) View Generation



(d) Reflecting the View Modifications On the Original Feature Model



(e) Feature Model After Refinement

Figure 3: FMView Tool Demo

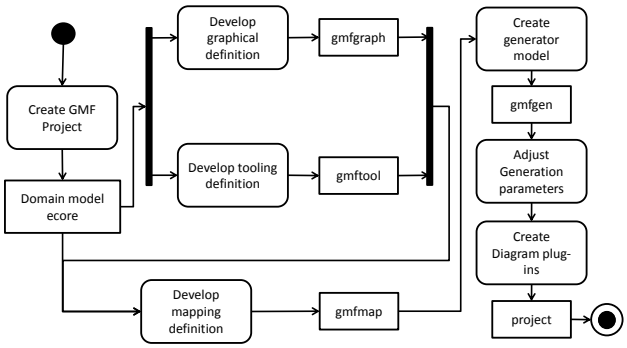


Figure 4: GMF Overview

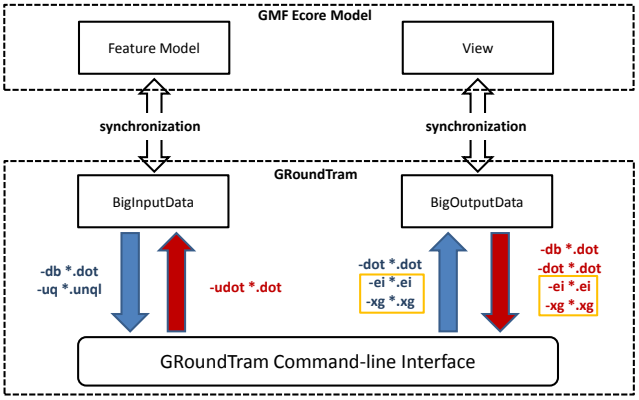


Figure 5: Data Models of FMView

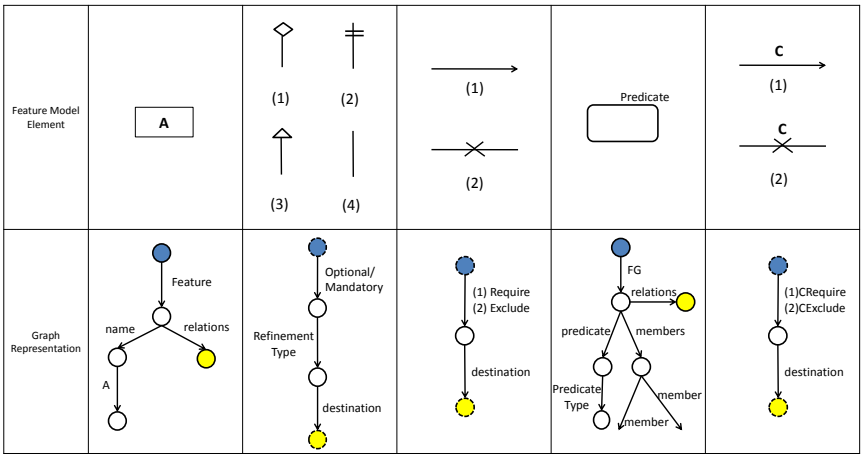


Figure 6: The Relationship between Feature Model Elements and Graph Segments.

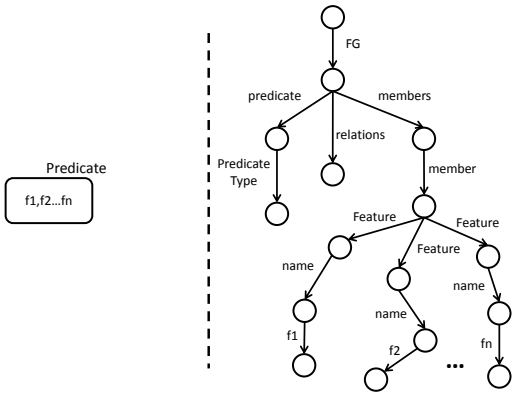
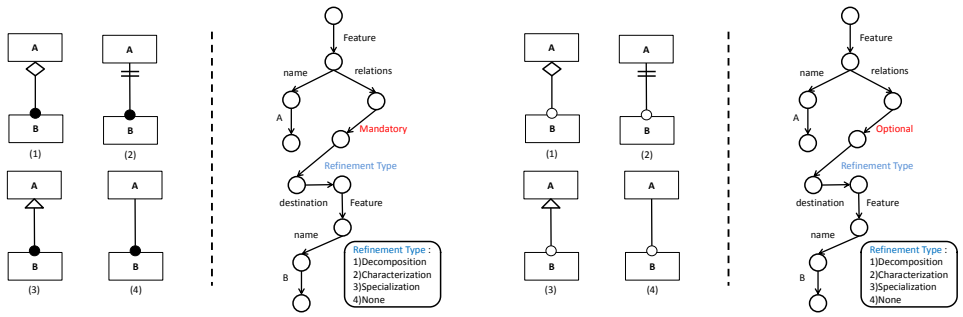
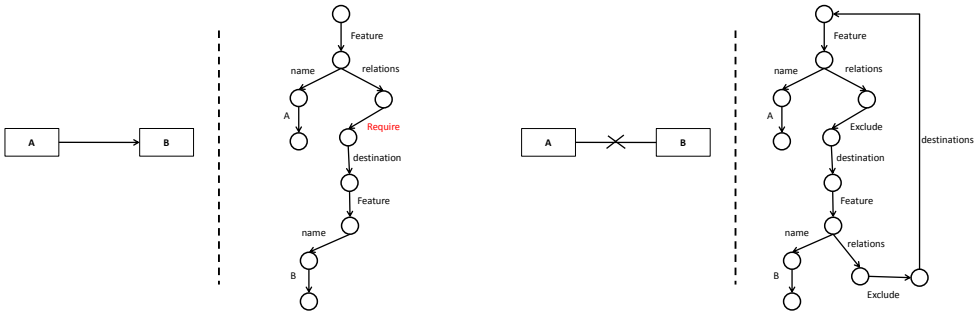


Figure 7: Feature Group Including Features.



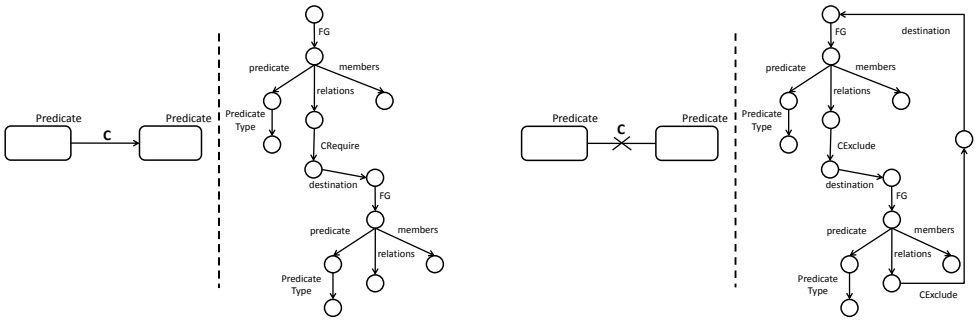
(a) Mandatory (b) Optional

Figure 8: Combining the Features with Refinement Relationships



(a) Require (b) Exclude

Figure 9: Combining the Features with Simple Constraints



(a) Complex Require (b) Complex Exclude

Figure 10: Combining the Feature Groups with Complex Constraints