Automating Feature-Oriented Domain Analysis

Fei Cao, Barrett R. Bryant, Carol C. Burt
Department of Computer and Information Sciences
University of Alabama at Birmingham
{caof, bryant, cburt}@cis.uab.edu

Zhisheng Huang, Rajeev R. Raje, Andrew M. Olson
Department of Computer and Information Science
Indiana University Purdue University at Indianapolis
{zhuang, rraje, aolson}@cs.iupui.edu

Mikhail Auguston
Computer Science Department
Naval Postgraduate School
auguston@cs.nps.navy.mil

Abstract

Feature modeling is commonly used to capture the commonalities and variabilities of systems in a domain during Domain Analysis. The output of feature modeling will be some reusable assets (components, patterns, domain-specific language, etc.) to be fed into the application engineering phase for ultimate software products. But current practice lacks an automatic approach for seamless generation of reusable assets from feature models. This paper presents an algorithm for generating sets of instance descriptions (feature instances) from feature models of a domain and applies this algorithm in creating a Generic Feature Modeling Environment for automating Feature-Oriented Domain Analysis.

Keywords: Feature Modeling, Domain Analysis, Generative Programming

1. Introduction

Generative Programming (GP) [Czar00] has emerged as a software development paradigm for automatic generation of software products based on modeling of software system families. The distinct property of GP is it is not only about a development *for reuse* in terms of building a Generative Domain Model (GDM) for software system families, but also about a development *with reuse* in terms of using GDM to generate concrete systems. To build a GDM, domain analysis has to be applied to scope a system family and to identify the commonalities, variabilities and dependencies among family members. A crucial outcome of the domain analysis phase is a feature model, which is usually represented as a feature diagram. However, the application of feature diagrams is quite limited, due to the fact that current practice is not fully automated, while the size of the set of feature instances may be expanded exponentially (which we will see later in this paper), thus it is difficult to apply constraint checking and other types of computing. In order to align with the goal of GP for the highest level of automation, to cope with family system processing (which is usually of a large scale), feature modeling should be carried out in an automatic fashion to seamlessly generate reusable assets to be used in application engineering for constructing a family of applications. This paper presents an algorithm for generating the set of all feature instances from a feature diagram and applies this algorithm in creating a Generic Feature Modeling Environment (GFME) for automating Feature-Oriented Domain Analysis (FODA). This paper is organized as follows: Section 2 briefly describes major related research efforts. Section 3 gives the algorithm for computing feature models. Section 4 presents the GFME created with the Generic Modeling Environment (GME) 2000 [GME01]. Section 5 draws the conclusion of this paper.

2. Related Work

Feature models were initially introduced by the FODA method [Kang90]. In the FODA method, a feature is defined as an end-user-visible characteristic of a system. This model uses a feature diagram to represent a hierarchical decomposition of features, which include mandatory, alternative or optional features. Feature constraints, stakeholders and rationales are also incorporated in this feature model. Czarnecki and Eisenecker [Czar00] give a more detailed account of feature diagrams including diagram normalization.

The FODA method uses Prolog in a prototype tool for doing checking over some sets of feature values. However, features have to be stored in the Prolog fact base first, rather than being analyzed directly over the feature diagram, thus the tool is not seamlessly integrated with the visual diagram setting. Czarnecki and Eisenecker [Czar00] also explore the possible implementation of feature diagrams by mapping into UML, which in turn may be used to generate some implementation codes using such CASE tools as Rational Rose[1]. The mapping process, however, is again a manual process. Also, what Rational Rose can generate are just some skeleton codes, which are far from being complete implementations.

Feature models can be represented not only in graphical form using feature diagrams, but also in textual form. Van Deursen and Klint [Deur02] propose a Feature Description Language (FDL) for textual representation of feature diagrams. Manipulation of features is achieved by Feature Diagram Algebra (FDA), which consists of four sets of rules: normalization rules, variability rules, expansion rules and satisfaction rules. The FDL can be fed into the a tool named "ASF+SDF Meta-Environment" [Bran01] for direct execution as a basis for prototype tool support, which again is not seamlessly integrated with graphical representations of feature diagrams; the capacity of constraint checking is quite limited; the FDA is separated from, rather than integrated as part of the feature diagram; the generation of reusable assets from FDL is not flexible.

Obviously for the related work mentioned in this section, there is a gap between using feature diagrams for feature modeling and a seamless, efficient generation of reusable assets. This paper presents an approach toward bridging this gap.

3. An Algorithm for Feature Diagram Computing

In contrast to computing features by transforming feature diagrams to some other representation forms (such as UML or FDL) first, we are going to apply the proposed algorithm directly over the feature diagram. We first briefly describe the representations used in [Czar00] illustrated in Figure 1. The *mandatory* feature is represented by being attached to an edge ending with a filled circle. So the feature F consists of both C1 and C2 in this case, and the feature instances here are {F, C1, C2}. The *optional* feature is represented by being attached to an edge ending with an unfilled circle. So the feature F may or may not contain C1. The *optional* feature instances here are {F, C2} and {F, C1, C2}. The *alternative* feature is represented by connecting edges with an arch. So the feature F consists of exactly one of its child features. The *alternative* feature instances here are {F, C1} and {F, C2}. Note that if C1 is optional while C2 is mandatory, then the *alternative* feature instances here are {F}, {F, C1} and {F, C2}, because the child feature instances derived from the C1 side contain an empty feature. The *Or* feature is represented by connecting edges with a filled arch. The *Or* feature instances here are {F, C1}, {F, C2} and {F, C1, C2}. If there is an optional child feature, then the *Or* representation is actually equivalent to the situation that all the child features are optional, i.e., the *Or* feature instances will be {F}, {F, C1}, {F, C2} and {F, C1, C2}.

These representations can also be intermingled in feature diagrams, such as in Figure 2. These mixture forms can be normalized so that it is easier to be processed. e.g., Figure 2 can be normalized into Figure 3.

This normalization can be performed iteratively over all such "mixture relation" nodes in the feature diagram. In this way, the father-feature in the feature diagram will only be either *XOR* (corresponding to *alternative*), or *OR*, or *AND* in relationship to child-features. Meanwhile, each child-feature may be either *optional* or *mandatory*. Obviously, the normalization process described here is fulfilled by adding hierarchy into the original feature tree

[1] www.rational.com

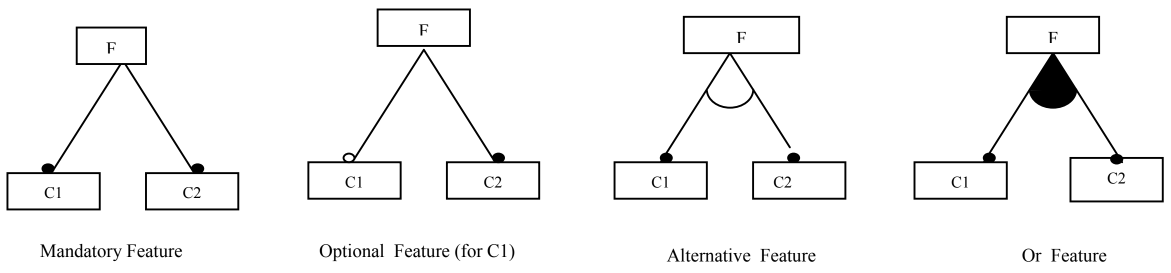


Figure 1: Feature Diagram Representation

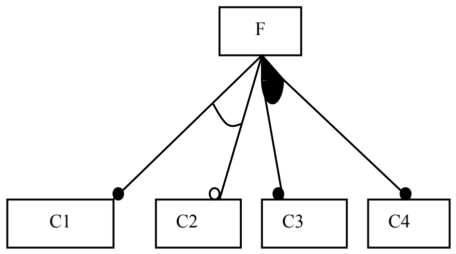


Figure 2: Mixture of Feature Representation

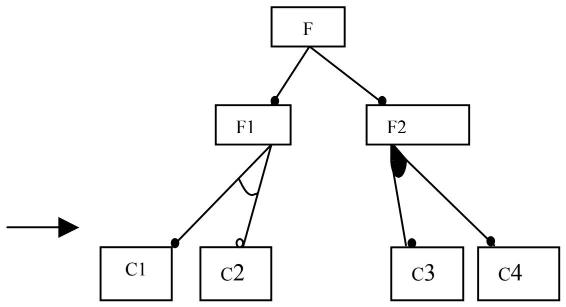


Figure 3: Normalized Feature Representation

without loss of any commonality and variability representations. After such normalization is performed, the feature diagram will be in the structure as in Figure 4. The proposed algorithm will be applied over such normalized feature diagrams thereafter.

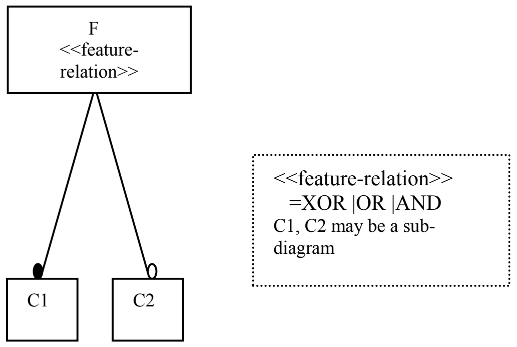


Figure 4: Variation of Feature Diagram

Suppose each feature node is represented as the following data structure (note that without loss of generality, the following data structure may not be strictly consistent with a specific C++ programming environment):

```
struct  FeatureNode{
 String featurename;
```

```
enum {XOR, OR, AND} feature-relation;
  /*denotes the father-child  relation */

 ChildConnectionList *edges;
  /*list of connections associated with
   its child-feature nodes */
  }

 struct ChildConnectionList {
   bool  isMandatory ;
     /*is a mandatory/optional  feature*/
   FeatureNode * aFeature;
        /*point to a feature node*/
                    }
```

From the data structure above we can see that we can get access to the child-nodes of a feature node by traversing its associated edges.

Currently, the result of the algorithm to compute the feature diagram is just the set of all feature instances of a feature diagram. The result will be represented as a list. Each element of the list corresponds to a feature instance. Each feature instance in turn is represented as a list, which consists of the list of pointers to the related feature node. The result is represented as follows:

```
 typedef List<FeatureInstance *> Result;
 typedef List<FeatureNode *>
               FeatureInstance;
```

Below is the pseudo code for the algorithm. The input parameter to the algorithm is the pointer to the root node of a feature diagram. The output will be all feature instances derived from the feature diagram. Note the variables are in italicized font while the types are in bold font.

```
Result * processFeatureDiagram (
   FeatureNode *node-root)
 {
 create a temp1:FeatureInstance  with
 only node-root in it;

 create a temp2: Result  with only one
 FeatureInstance temp1 in it;

 if(node-root has no child nodes)
 then  return temp2;

 else
 if (node-root->feature-relation==AND)

  {
  recursively call  processFeatureDiagram
  over each of node-root's child-nodes,
  each returning a child result;

  if corresponding child node is
  "Optional",
  add an empty FeatureInstance into the
  corresponding child result;

  calculate the production of all the
  returned child results as temp3:Result;

  return the production of temp2 and
  temp3;
  }

 else
 if(node-root->feature-relation==XOR)
  {
  recursively call processFeatureDiagram
  over each of node-root's child-nodes,
  each returning a child result;

  calculate the union of those returned
  child results as temp3:Result;

  if there is a child node that is
  "Optional",
  add an empty FeatureInstance into
  temp3;

  return  temp3;
  }

 else
 if(node-root->feature-relation==OR)
  {
   recursively call processFeatureDiagram
   over each of node-root's child-nodes,
   each returning a child result;

   for each of the child result returned
   in the above call,
   add an empty FeatureInstance into it;

   get the production of all the child
   results as temp3:Result;

   If all child features are mandatory,
   remove the empty FeatureInstance from
   temp3;

   return the production of temp2 and
   temp3;
   }
 }
```

Beware that a **Result** is actually a two-dimension data structure. If **Result** *A* has m **FeatureInstance**s while Result *B* has n **FeatureInstance**s, then the union of *A* and *B* has m+n **FeatureInstance**s while the production of *A* and *B* has m*n **FeatureInstance**s. To exemplify the above algorithm, we use ε to represent an empty **Result**, × for production, ∪ for union operation in Figures 5-7, which correspond to three types of cases for computing the set of feature instances. Also from Figure 7 we can easily see the size of feature set may grow exponentially (as to the extreme case where all *feature-relation*s are *OR* , the size will be 2^n, where n is the amount of leaf nodes).

Here we put the non-leaf node (like *F* here) into the feature instances in order to facilitate constraint checking. If one non-leaf feature *F* is supposed to be excluded in the final feature instance, then its child-features should not be included correspondingly, and we can eliminate those feature instances from the final result by identifying which feature instance contains feature *F*, rather than by tracking down all its child-features laboriously.

4. A Generic Feature Modeling Environment (GFME)

We use the Generic Modeling Environment (GME) [GME01] to build GFME. GME is a configurable toolkit for creating domain-specific modeling and program synthesis environments. The configuration is accomplished through metamodels specifying the modeling paradigm (modeling language) of the application domain. The modeling paradigm defines the family of models that can be created using the resultant modeling environment. The metamodels specifying the modeling paradigm are used to automatically generate the target domain-specific environment. GME provides the Builder Object Network (BON) framework for building interpreters to interpret domain models built in the domain-specific environment. The interpretation process can be used to generate reusable assets for the domain engineering phase. The BON API provides leverages for access to the domain models, which makes the above algorithm implementable. With all those facilities of GME, we believe it has the best tool support for feature modeling.

GFME provides the modeling environment for building feature diagrams with the structure as described in Figure 4. Figure 8 provides the screenshot of the GFME. Note at the lower-right corner is the interface to specify such attributes as the relationship with its child-nodes for a node under focus (here "TransactionSubsystem") in the environment. In the same way, we can specify the attributes for those connections

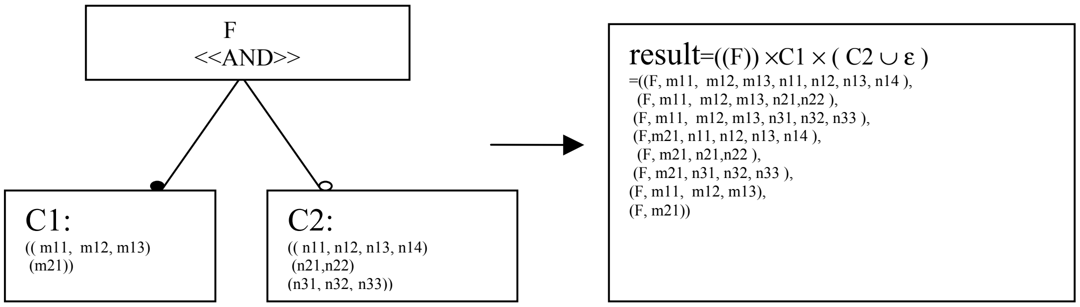

F
<<AND>>
C1:
((m11, m12, m13)
(m21))
C2:
((n11, n12, n13, n14)
(n21,n22)
(n31, n32, n33))
result=((F)) ×C1 × (C2 ∪ ε)
=((F, m11, m12, m13, n11, n12, n13, n14),
(F, m11, m12, m13, n21,n22),
(F, m11, m12, m13, n31, n32, n33),
(F,m21, n11, n12, n13, n14),
(F, m21, n21,n22),
(F, m21, n31, n32, n33),
(F, m11, m12, m13),
(F, m21))

Figure 5: Computing AND result

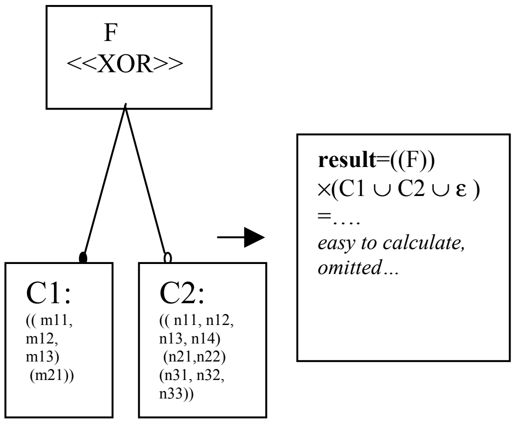


Figure 6: Computing XOR result

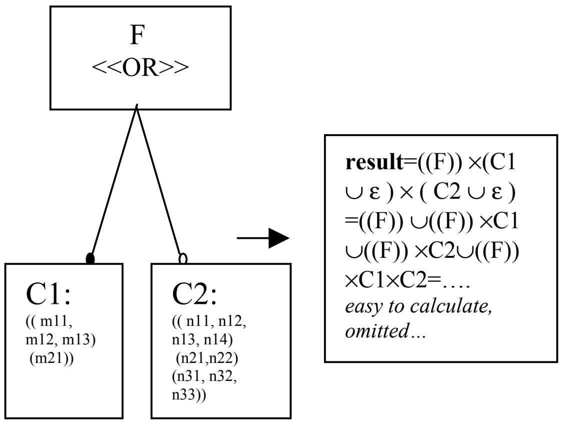
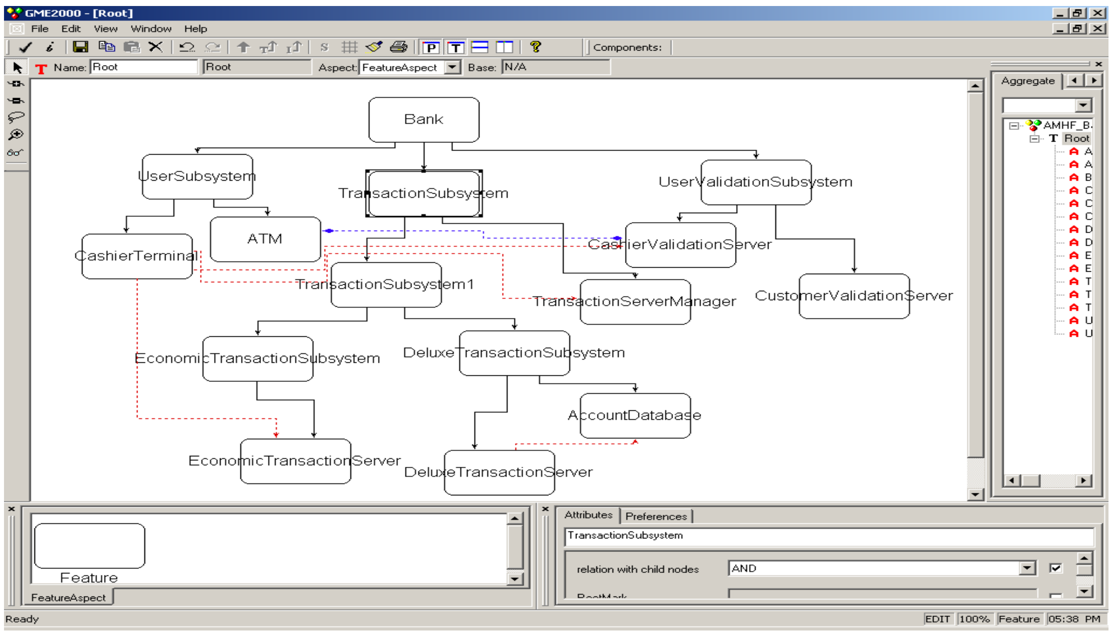


Figure 7: Computing OR result



Name: Root
Aspect: FeatureAspect
Base: N/A

TransactionSubsystem
relation with child nodes: AND

Figure 8: Generic Feature Modeling Environment

between feature nodes. The dashed lines denote the various kinds of dependencies or constraints to be enforced between feature nodes. Currently we just generate the set of feature instances from feature diagram satisfying all specified constraints. With full control of the interpretation process (i.e., writing interpreter code via BON API), we can generate application code from feature diagrams on demand.

5. Conclusion

Feature Modeling is the core part of FODA. Our ongoing UniFrame project [Raje02] requires feature modeling for building a generative domain model. The reusable assets generated from feature modeling after normalization, expansion and constraint checking will be output into XML files. The reusable assets serve two purposes: 1) for clients to initiate natural-language-like queries [Lee02] in the problem space [Czar00]; 2) to provide a guideline for component providers to produce component families in the solution space [Czar00]. The current practice of feature modeling remains at the manual or semi-automatic level, which hinders it from becoming widely applied. This paper applies normalization over the traditional feature diagram and presents an algorithm to generate complete feature instances from a feature diagram under constraints. The algorithm is adopted in GFME, which provides an efficient, automatic FODA environment.

Acknowledgements. This research is supported by the U. S. Office of Naval Research under the award number N00014-01-1-0746.

References

[Bran01] M.G.J. van den Brand, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, J. Visser. *The ASF+SDF Meta-Environment: a Component-Based Language Development Environment.* Compiler Construction (CC '01), vol. 2027, Lecture Notes in Computer Science, pp. 365-370, Springer-Verlag, 2001.

[Czar00] K. Czarnecki, U.W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2000.

[Deur02] A. van Deursen and P. Klint. *Domain-specific Language Design Requires Feature Descriptions*. Journal of Computing and Information Technology 10(1), pp. 1-17, 2002.

[GME01] *GME 2000 User's Manual, Version 2.0*. ISIS, Vanderbilt University, 2001.

[Kang90] K.C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. *Feature-oriented Domain Analysis (FODA) Feasibility Study*. Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[Lee02] B.-S. Lee, B. R. Bryant. *Contextual Processing and DAML for Understanding Software Requirements Specifications*. Proceedings of COLING 2002, the 19th International Conference on Computational Linguistics, pp. 516-522, 2002.

[Raje02] R. R. Raje, M. Auguston, B. R. Bryant, A. M. Olson, C. C. Burt. *A Quality of Service-Based Framework for Creating Distributed Heterogeneous Software Components*. Concurrency and Computation: Practice and Experience 14, pp. 1009-1034, 2002.