Mining Binary Constraints in the Construction of Feature Models

Li Yi, Wei Zhang, Haiyan Zhao, Zhi Jin, Hong Mei
Institute of Software, School of EECS, Peking University
Key Laboratory of High Confidence Software Technology, Ministry of Education of China
Beijing, China
{yili07, zhangw, zhhy, zhijin}@sei.pku.edu.cn, meih@pku.edu.cn

***Abstract*—Feature models provide an effective way to organize and reuse requirements in a specific domain. A feature model consists of a feature tree and cross-tree constraints. Identifying features and then building a feature tree takes a lot of effort, and many semi-automated approaches have been proposed to help the situation. However, finding cross-tree constraints is often more challenging which still lacks the help of automation. In this paper, we propose an approach to mining cross-tree binary constraints in the construction of feature models. Binary constraints are the most basic kind of cross-tree constraints that involve exactly two features and can be further classified into two sub-types, i.e. *requires* and *excludes.* Given these two sub-types, a pair of any two features in a feature model falls into one of the following classes: no constraints between them, a *requires* between them, or an *excludes* between them. Therefore we perform a 3-class classification on feature pairs to mine binary constraints from features. We incorporate a support vector machine as the classifier and utilize a genetic algorithm to optimize it. We conduct a series of experiments on two feature models constructed by third parties, to evaluate the effectiveness of our approach under different conditions that might occur in practical use. Results show that we can mine binary constraints at a high recall (near 100% in most cases), which is important because finding a missing constraint is very costly in real, often large, feature models.**

Keywords-feature model; binary constraints; support vector machine

I. INTRODUCTION

Feature models provide an effective way to organize and reuse requirements in a specific domain. Requirements are encapsulated in a set of *features*, and features are organized into a feature tree according to *refinements* between them. Furthermore, cross-tree *constraints* are constructed to capture additional dependencies between the features. The requirements are then reused through selecting a subset of features without violating the constraints.

In the construction of feature models, the first step is to identify features and organize them into a feature tree. This step is time-consuming because it needs a comprehensive review of requirements of applications in a domain [9].

However, finding cross-tree constraints among identified features is even more challenging for two reasons. First, the size of problem space of finding constraints is the square of identifying features. In other words, a certain feature may be constrained by any other features, and all the possibilities need to be checked to avoid any miss. Second, features are often concrete, which means that they can often be directly observed from an existing application or its documents [9]. By contrast, constraints are often abstract, which means that they often have to be learned from a systematic review of several similar applications.

Many semi-automated approaches have been proposed to reduce human workload during the construction of feature models, e.g. [4][12]. However, these approaches mainly focus on identifying features or constructing feature trees. In this paper, we focus on finding binary constraints. Binary constraints are the most basic kind of constraints that capture dependencies between exactly two features. We focus on them for three reasons. First, they have been adopted in most feature-oriented methods therefore our approach may have a wide applicability. Second, they are often the mostly used form of constraints in real feature models, although there are some more complicated kinds of constraints. Third, they are simple so that they could be a suitable starting point for the research of mining constraints.

Binary constraints can be further categorized into two sub-types: *requires* and *excludes.* Consider a *pair* of features in a feature model, it exactly falls into one of the following classes: (1) no constraint between the paired features, (2) a *requires* between them, or (3) an *excludes* between them. From such a perspective we treat the problem of mining binary constraints as a three-class classification problem on feature pairs. The input of our approach is a partially constructed feature model, in which features and their descriptions have been provided, the feature tree may be constructed, and a few binary constraints are already known. We first extract feature pairs from the input, and then train a classifier with known binary constraints. The classifier is implemented as a support vector machine and optimized by a genetic algorithm. Finally, the optimized classifier checks feature pairs to find candidates of binary constraints.

We conduct a series of experiments on two feature models constructed by third parties, to evaluate the effectiveness of our approach under different conditions that might occur in practical use. Results show that we can mine binary constraints at a high recall (near 100% in most cases), which is important because missing a constraint would be very costly in real, often large, feature models.

The remainder of this paper is organized in the following way. Section II gives some preliminaries on feature models. Section III presents details of our approach. Section IV

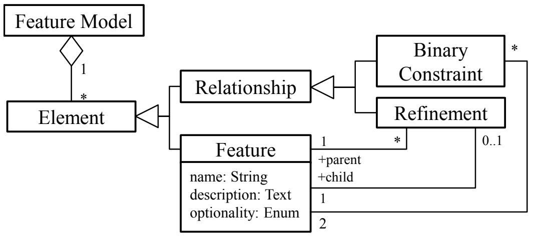


Figure 1. A meta-model of simple feature models.

describes our experiments. Section V discusses threats to validity, and Section VI presents some related work. Finally, Section VII concludes the paper and describes future work.

II. Preliminaries: Feature Models

In this section, we give a brief introduction on feature models, particularly the binary constraints that we want to mine from features.

Figure 1 shows a meta-model of simple feature models. Concepts in the meta-model appear in most feature modeling methods. A feature model consists of a set of *features* and *relationships*. The concept of feature can be understood in two aspects: intension and extension [15]. In intension, a feature denotes a cohesive set of individual requirements. In extension, a feature describes a software characteristic that has sufficient user/customer value.

There are two kinds of relationships between features, namely *refinements* and *binary constraint*s. Refinements organize features into a feature tree. A feature may have several child features, and it may have at most one parent feature. Besides, a feature may be *mandatory* or *optional* relating to its parent feature (i.e. the *optionality* of a feature). When a feature is select, its mandatory children must also be selected, while its optional children can be either selected or removed.

A binary constraint is either *requires* or *excludes.* Given two features *X* and *Y*, *X requires Y* means that if *X* is selected then *Y* must be selected as well, while *X excludes Y* means that *X* and *Y* cannot be selected at the same time.

III. A Classification-Based Approach to Mining Binary Constraints

In this section, we present our approach to mining binary constraints. We first give the overview of our approach, and then describe the details.

A. Overview of the Approach

The key technique in our approach is classification. We first learn a classifier from known data (i.e. training), and then use it to classify unknown data (i.e. test). Figure 2 gives an overview of our approach. The input is the training and test feature models. They can be the same, and in such cases, binary constraints within a fraction of the feature model are already known, and the rest of the feature model is tested. The first step is making feature pairs. The original feature pairs contain textual attributes, i.e. *names* and *descriptions*. The second step is quantifying feature pairs so that they are represented by numeric attributes and can be viewed as vectors. The classifier is then trained and optimized by the training vectors. The final step is to use the classifier to classify the test vectors.

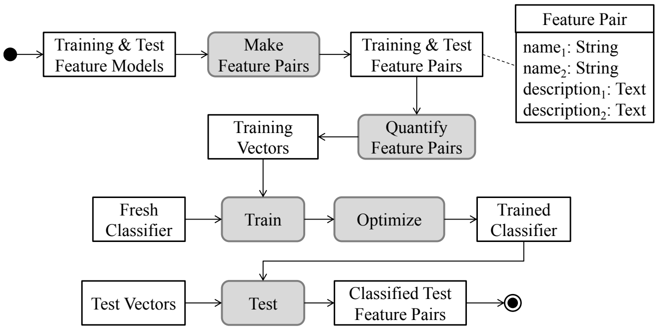


Figure 2. An overview of our approach.

B. Step 1: Make Feature Pairs

In our approach, feature pairs are *unordered*, that is, given two features *X* and *Y*, we do not distinguish between the pairs (*X, Y*) and (*Y, X*). If a feature pair (*X, Y*) is classified as *requires*, it means that *X requires Y* or *Y requires X,* or both. The reason for making feature pairs unordered is that *excludes* and *non-constrained* pairs are unordered by nature, so we treat *requires* as unordered pairs as well. Furthermore, given an unordered *requires* pair (*X, Y*), it is often trivial for people to further identify whether *X requires Y,* or *Y requires X*, or *X mutual-requires Y*. Therefore it does not harm the benefit brought by the automation.

In addition, if a feature tree is given, we only keep the cross-tree pairs, since the binary constraints to be mined are cross-tree. Given a feature model of *n* features, the maximal possible number of unordered, cross-tree pair is $(n^2-n)/2$.

C. Step 2: Quantify Feature Pairs

The classification technique in our approach assumes that feature pairs have numeric attributes. However, input feature pairs have only textual attributes, i.e. *name* and *description.* Therefore we quantify feature pairs by deriving four numeric attributes from the two textual attributes, as shown in Table I.

The rationale behind these attributes is that constraints are dependencies and interactions between features and the attributes reflect the possibility of such dependencies and interactions. Feature similarity and object overlap reveal that their function areas might be overlapped. Targeting shows that a feature directly affects another.

TABLE I. Numeric Attributes for a Feature Pair *(X, Y)*

Attribute	*Definition*
Feature Similarity	Similarity between *X* and *Y*'s descriptions.
Object Overlap	Similarity between objects and their adjective modifiers (see Figure 3) in *X* and *Y*'s descriptions.
Targeting (1)	Similarity between the name of *X* and objects and their adjective modifiers of *Y*.
Targeting (2)	Similarity between the name of *Y* and objects and their adjective modifiers of *X*.

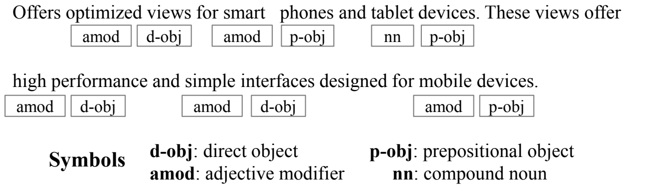

Offers optimized views for smart phones and tablet devices. These views offer
amod
d-obj
amod
p-obj
nn
p-obj
high performance and simple interfaces designed for mobile devices.
amod
d-obj
amod
d-obj
amod
p-obj
Symbols
d-obj: direct object
amod: adjective modifier
p-obj: prepositional object
nn: compound noun

Figure 3. An example of extracting objects and their adjective modifiers by Stanford Parser.

We utilize *Stanford Parser* [10] to find objects and their adjective modifiers in description of features. An object is an entity that is being affected by actions described in a feature, and the adjective modifiers distinguish between different characteristics and status of a certain kind of object. Figure 3 gives an example description from real products. Stanford Parser works well with incomplete sentences (e.g. missing subjects) that are common in descriptions of features in the real world.

The similarity between two textual documents (here a document is a feature name or a feature description) is defined as the cosine of their corresponding term vectors. First, the words in the documents are stemmed, and stop words and unrelated words are removed. Then the words are weighted by the widely used TF-IDF (term frequency and inversed document frequency) score defined as follows:

$$TF_{word\ w, document\ d} = \frac{\#\ of\ w\ occurs\ in\ d}{\#\ of\ words\ in\ d}, \text{ and}$$

$$IDF_{word\ w} = log\,\frac{\#\ of\ documents}{\#\ of\ documents\ containing\ w}$$

The term vector **D** for a document *d* is the vector of TF-IDF weights of its distinct words, that is:

$$\mathbf{D} = \left(TF_{w_1,d} \times IDF_{w_1}, \ldots, TF_{w_n,d} \times IDF_{w_n}\right), \\ \text{where } \mathrm{w}_1, \ldots, w_n \in d.$$

Given two documents d_1 and d_2, the similarity between them is defined as the cosine of the angle of corresponding term vectors $\mathbf{D_1}$ and $\mathbf{D_2}$:

$$Similarity_{d_1,d_2} = \frac{\mathbf{D_1} \cdot \mathbf{D_2}}{|\mathbf{D_1}| \times |\mathbf{D_2}|}$$

D. Step 3: Train the Classifier

The classification technique used in our approach is support vector machine (SVM), which has shown promising results in many practical applications [13]. In this subsection, we first introduce the concept of *maximal margin hyperplanes* that forms the basis of SVM. We then explain how an SVM can be trained to look for such hyperplanes to perform a *two-class* classification. Finally, we show how to extend the SVM so that it can be applied to a *three-class* classification problem as in classifying feature pairs.

1) Maximal Margin Hyperplanes

For simplicity, suppose that our feature pairs have only two numeric attributes, A_1 and A_2. Furthermore, suppose that we have a set of training feature pairs that exactly belong to two classes. With the above assumptions we can draw the training feature pairs in a 2-dimensional space in which we draw the two classes as circles and squares, respectively (see Figure 4). We also assume that the training data are linearly separable, i.e. we can find a separating *hyperplane* (or a *line* in 2D space) such that all the circles reside on one side of the hyperplane and all the squares reside on the other side. In the example of Figure 4, there are infinitely many separating hyperplanes, and we show two of them (H_1 and H_2). Each hyperplane H_i is associated with a pair of boundary hyperplanes, denoted as h_{i1} and h_{i2}, respectively. h_{i1} (resp. h_{i2}) is obtained by moving a parallel hyperplane away from H_i until it touches the closest circle(s) (resp. square(s)). The distance between h_{i1} and h_{i2} is known as the *margin* of H_i.

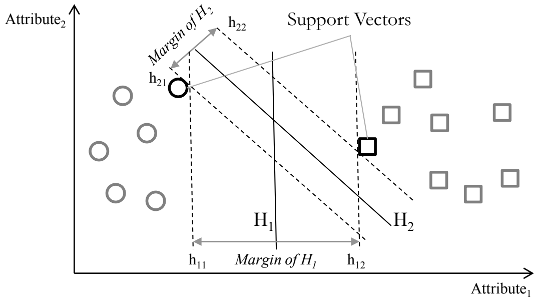


Figure 4. Maximal margin hyperplane.

The basic idea of SVM is to find separating hyperplanes with *maximal margin*, because wider margins have been proved to make fewer mistakes in classification [13]. In fact, wider margins are even preferable despite the fact that they make training errors sometimes. Figure 5 shows a training set that is similar to Figure 4 except it has two new items, *P* and *Q*. Although H_1 misclassifies *P* and *Q*, while H_2 does not, H_1 should still be preferred over H_2. The margin of such hyperplanes (like H_1 in Figure 5) is called a *soft margin*.

2) The Support Vector Machine

Support vectors are the vectors residing on the boundary hyperplanes of a maximal margin hyperplane (see Figure 4). A support vector machine tries to find support vectors in a given training set. Formally, in the classification of feature pairs, we assume that there are *N* training examples that belong to two classes. Each example has four attributes and thus is denoted by a 4-dimensional vector $\mathbf{x_i} = (x_{i1}, x_{i2}, x_{i3}, x_{i4})$, and its class label is denoted by $y_i \in \{-1, 1\}$, for *i* = 1 to *N*. We first consider the simplest case in which the training

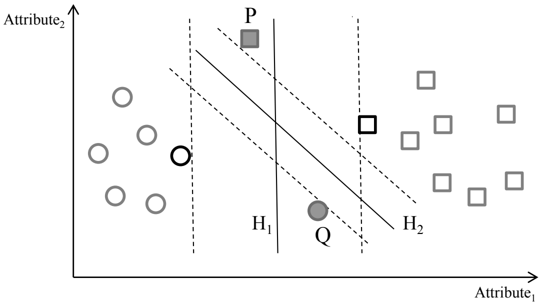


Figure 5. Maximal margin hyperplane with soft margin.

examples are linearly separable and no training errors. For any given pair of boundary hyperplanes, there are infinitely many separating hyperplanes lie between them (all of the hyperplanes are parallel with each other). The trick of SVM is that it always chooses the middle one, that is, we formulate the separating hyperplane as (**v** is a vector and *t* is a constant):

$$\mathbf{v} \cdot \mathbf{x} + t = 0 \tag{1}$$

Since it lies in the middle of boundary hyperplanes, we can express the boundary hyperplanes as (*d* is a constant):

$$\mathbf{v} \cdot \mathbf{x} + t = -d \tag{2}$$
$$\mathbf{v} \cdot \mathbf{x} + t = d \tag{3}$$

The margin is then given by $2d/\parallel \mathbf{v} \parallel$, where $\parallel \mathbf{v} \parallel$ is the length of **v**. Since we only care about maximize the margin, we can scale the parameters by dividing them by *d,* so that *we* get a simpler form of above equations as:

$$\mathbf{w} \cdot \mathbf{x} + b = 0 \tag{4}$$
$$\mathbf{w} \cdot \mathbf{x} + b = 1 \tag{5}$$
$$\mathbf{w} \cdot \mathbf{x} + b = -1 \tag{6}$$
$$\text{Margin} = 2/\parallel \mathbf{w} \parallel \tag{7}$$

According to the definition of boundary hyperplanes, all the training data reside on or outside boundary hyperplanes. Therefore for all training data, the following inequality holds:

$$\mathbf{w} \cdot \mathbf{x_i} + b \geq 1, \text{for } y_i = 1 \tag{8}$$
$$\mathbf{w} \cdot \mathbf{x_i} + b \leq -1, \text{for } y_i = -1 \tag{9}$$

We express the inequality in a more compact form as:

$$y_i(\mathbf{w} \cdot \mathbf{x_i} + b) \geq 1 \tag{10}$$

In summary, the simplest case of SVM is given by Equation (7) and (10) and is defined in the following form.

Definition 1 (SVM: Linearly Separable Case). Given a training set of *N* items $\{\mathbf{x_1}, \mathbf{x_2}, \dots, \mathbf{x_N}\}$ that belong to two classes labeled as $y_i \in \{-1, 1\}$, training an SVM is equal to solving the following constrained optimization problem:

$$\max \frac{2}{\parallel \mathbf{w} \parallel}$$
$$\text{subject to} \quad y_i(\mathbf{w} \cdot \mathbf{x_i} + b) \geq 1, i = 1, 2, \dots, N.$$

However, training sets in practice are often non-linear and non-separable. For non-linear cases (i.e. the boundary between classes is not a hyperplane), a function $\Phi(\mathbf{x})$ is used to transform original vectors $\mathbf{x_i}$ into a higher dimensional space (sometimes even an infinite-dimensional space) so that the boundary becomes a hyperplane in that space. For non-separable cases, a few training errors are inevitable, so the penalty of making the errors must be taken into account. As a result, two parameters, C^+ and C^-, are introduced to denote the penalty of making errors on class 1 and -1, respectively. Besides, the margin is slacked by a positive amount ξ_i for each training example $\mathbf{x_i}$ so that it becomes "soft". Formally, the general case of SVM is defined in the following form.

Definition 2 (SVM: General Case). Given a training set of *N* items $\{\mathbf{x_1}, \mathbf{x_2}, \dots, \mathbf{x_N}\}$ that belong to two classes labeled as $y_i \in \{-1, 1\}$, training an SVM is equal to solving the following constrained optimization problem:

$$\min \frac{\parallel \mathbf{w} \parallel}{2} + C^+ \sum_{y_i=1} \xi_i + C^- \sum_{y_i=-1} \xi_i$$
$$\text{subject to} \quad y_i(\mathbf{w} \cdot \Phi(\mathbf{x_i}) + b) \geq 1 - \xi_i,$$
$$\text{where } \xi_i > 0, \quad i = 1, 2, \dots, N.$$

An off-the-shelf classifier such as LIBSVM [3] can compute the **w**, Φ, *b* and ξ_i for us. However, the penalties (or weights) of the classes, C^+ and C^-, must be set manually before using the classifier, and their values significantly affect the effectiveness of the classifier. Our solution is to use an optimization algorithm to find optimized weights of the classes. (We will discuss this later in Section III-E.)

3) Extend SVM to Three-Class Classification

The SVM introduced before only supports classification of two classes. To extend it to three classes, we incorporate a *one-against-one* strategy as follows.

We denote *non-constrained*, *requires,* and *excludes* by y_1, y_2, y_3, respectively. The original SVM classifier runs three times, and each time it distinguishes between two classes (y_i, y_j), for $1 \leq i < j \leq 3$. Training examples that do not belong to either y_i or y_j are ignored when training the classifier on classes (y_i, y_j). For each test example, its class is determined using a voting strategy: when classifying against (y_i, y_j), the corresponding class receives a vote after the feature pair is classified. Then we classify a test example as the class with the most votes. In case that two or more classes have identical votes, we simply select the class with the smaller index. This simple tie-breaking strategy gives good results according to [3].

E. Step 4: Optimize the Classifier

We utilize LIBSVM to implement the classifier. It has three parameters (see Table II) that need to be optimized on each given training set. Each parameter has an initial value and a valid range in which we try to find the optimized value.

For the weights of classes, we use the *non-constrained* class as a baseline and always set its weight to 1. Then we compute two ratios:

$$R_1 = \frac{\#\, non\ constrianed\ pairs}{\#\, requires\ pairs}$$

$$R_2 = \frac{\#\, non\ constrained\ pairs}{\#\, excludes\ pairs}$$

TABLE II. CLASSIFIER PARAMETERS TO BE OPTIMIZED

Parameter	*Meaning*	*Initial Value*	*Range*	*Step*
C_{req}	The weight of the *requires* class.	R_1	$(1, 10R_1)$ or $(1 / 10R_1, 1)$	0.5
C_{exc}	The weight of the *excludes* class.	R_2	$(1, 10R_2)$ or $(1 / 10R_2, 1)$	0.5
γ	A parameter in the function Φ (see Definition 2) used by LIBSVM.	$\frac{1}{4}$	$(\frac{1}{40}, 2.5)$	0.01

We use them as initial values for the *requires* and *excludes* class, respectively. The rationale is that a correct classification of a rare class should have greater value than a correct classification of a majority class. Therefore classes can be weighted according to their size (i.e. the number of instances in them).

A parameter γ is used in the function Φ mentioned in Definition 2. LIBSVM suggests that the default value of γ should be set to (1 / number of attributes), which is 1 / 4 here. It also suggests that γ needs to be optimized in practice.

We set the ranges based on a factor of 10, that is, start from 1 / 10 of a parameter's initial value, and end with 10 times larger than it. The weight is an exception. It is always larger than 1 (if $R_i > 1$) or smaller than 1 (if $R_i < 1$).

Before we optimize the classifier, we need to know how to evaluate the classifier on a given set of training examples. A standard method in the field of classification is known as *k-fold cross-validation,* defined as follows. First, we divide the training set into k equally sized subsets. Then we run the classifier k times, and each time a distinct subset is chosen for testing and the other $k - 1$ subsets are used for training. Therefore in the end each instance is tested exactly once. We evaluate the performance of the classifier by computing its *error rate* during the cross-validation:

$$Error\ Rate = \frac{\#\ wrongly\ classified\ instances}{\#\ instances\ in\ total} \times 100\%$$

We incorporate a *genetic algorithm* to implement the optimization (see Algorithm 1). A *solution* is a tuple of the three parameters, i.e. (C_{req}, C_{exc}, γ). The main operations in a genetic algorithm are *mutation* and *crossover*. A *mutation* takes a solution and makes a little change on a random number (1 to 3) of its parameters. The little change here means randomly increases or decreases the parameter by a predefined step (see Table II). A *crossover* takes two solutions and combines a random number of parameters from one solution with the rest parameters from another solution. Both operations produce a new solution.

The first step of the algorithm is to generate an initial set of solutions. This is done by performing mutations on a *seed solution* consisting of the initial values of the parameters.

The algorithm then repeats an evolution step for a given number of times. In each evolution step, a fresh classifier is trained with parameters in each solution and evaluated by a k-fold cross-validation. A certain amount of solutions with the lowest error rate (known as the *elites*) are kept, and the rest solutions are eliminated. New solutions are produced by randomly performing mutation or crossover on randomly selected elites. These new solutions are added to the solution set until the set is full again.

Finally, the overall optimized solution is the best one in the last solution set. We use the parameters in this solution to train a fresh classifier to get an optimized classifier.

IV. Experimental Evaluation

We evaluate our approach by a series of experiments. Input data are constructed by third parties so that they are completely independent to our approach. We evaluate the performance of the classifier in different scenarios, including:

Algorithm 1: Find optimized parameters for the classifier

```
optimize (seed: Solution, m: int, e: double, p: double): Solution
  solution_set ← {seed}
  repeat
    Add a mutation of seed to solution_set
  until solution_set is full

  repeat m times
    For each solution, train a classifier and do cross-validation
    elites ← {The best $e \times 100\%$ solutions in solution_set}
    solution_set ← elites
    repeat
      r ← a random number between 0 and 1
      if (r < p)
        x ← a randomly selected solution from elites
        Add a mutation of x to solution_set
      else
        ($x_1$, $x_2$) ← two randomly selected solutions from elites
        Add a crossover of ($x_1$, $x_2$) to solution_set
      end if
    until solution_set is full
  end repeat

  return the best solution in solution_set
end
```

- Training data and test data come from the same or different domains. These experiments check that whether a classifier trained by a feature model of some domain can be applied to a feature model of some totally unrelated domain. (See Section IV-C.)
- The training is supervised or semi-supervised. These experiments evaluate the effect of different training strategies. (See Section IV-D.)
- Feedback is enabled or disabled during testing. In practice, human analysts may use our approach to get constraint candidates and then give feedback on correctness of the candidates. The procedure may repeat several times until all constraints are found. We also design experiments to simulate the above scenario and show the effect of human feedback on the classifier. (See Section IV-E.)

A. Data Preparation

We use two feature models from the SPLOT repository[1] for the experiments. Table III shows the basic information about the feature models. The authors of the two feature models are experts in this field (Don Batory and the pure-systems Corp., respectively), so they can be trusted inputs for our experiments.

A major problem is that feature models available online or in publications do not contain feature descriptions. Our solution is to search the features in Wikipedia and copy the first paragraph of their definitions as their descriptions. The rationale is that most features in the two feature models are

[1] http://www.splot-research.org

TABLE III. THE FEATURE MODELS FOR EXPERIMENTS

Name	*Features*	*Feature Pairs*	*Constraint Pairs*
Weather Station	22	196	6 *requires* 5 *excludes*
Graph Product Line	15	91	8 *requires* 5 *excludes*

Weighted Graph A weighted graph associates a label (weight) with every edge in the graph. Weights are usually real numbers. They may be restricted to rational numbers or integers.

Freeze Point The freeze point of a liquid is the temperature at which it changes state from liquid to solid.

Figure 6. Example features in the experiments.

domain terminologies that can be clearly defined, and by convention, the first paragraph in a Wikipedia page is a term's abstract that is close to a real description that would appear in a feature model. Figure 6 illustrates some features and their descriptions completed in this way. A few features are not defined in Wikipedia, and in such cases, we leave their descriptions as blank.

B. Measure the Performance of the Classifier

The error rate introduced in Section III-E is a standard measurement of the overall performance of a classifier. In addition, since we focus on finding constraints, we compute a *confusion matrix* for *requires* and *excludes* class, and calculate their *precision, recall* and *F_2-measure.* A confusion matrix shows the number of instances predicted correctly or incorrectly (see Table IV). The counts tabulated in a confusion matrix are known as True Positive (TP), False Negative (FN), False Positive (FP), and True Negative (TN). The precision, recall and F_2-measure are then computed as:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F_\beta = \frac{(\beta^2 + 1) \times Recall \times Precision}{Recall + \beta^2 \times Precision}$$

The F_β-measure receives a high score if both precision and recall are reasonably good. The parameter β is a positive integer which means recall is β times more important than precision, and it is often set to 1 or 2 (i.e. F_1- or F_2-measure). We use F_2-measure because for human analysts, finding a missing constraint (false negative) is much harder than identifying a false positive; in other words, the classifier should strive for high recall on binary constraints.

C. Compare Different Data Set Selection Strategies

The first group of experiments is designed to compare the effect of different data set (i.e. training set and test set) selection strategy. Given two feature models, one of them is known as the training feature model, and the other is known as the test feature model, there are three strategies:

TABLE IV. A CONFUSION MATRIX

	Predicted Positive	*Predicted Negative*
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

- Cross-Domain Strategy. The training set is the training feature model, and the test set is the test feature model. If this strategy wins, it means that in practice, analysts may directly apply a trained classifier to their work, and the accumulated knowledge from several domains may further benefit the classifier.
- Inner-Domain Strategy. Only the test feature model is used, that is, the training set is a fraction of the test feature model, and the test set is the rest of the feature model. If this strategy wins, it means that in practice, analysts have to manually classify a fraction of feature pairs in advance, and then incorporate a fresh classifier to work.
- Hybrid Strategy. The training set is the training feature model plus a fraction of the test feature model, and the test set is the rest of the test feature model.

Weather Station and Graph Product Line are used as the training and the test feature model, respectively. Their roles are then exchanged. Therefore there are 2×3=6 classification settings. Common parameters for the settings are shown in Table V. For the inner-domain and hybrid strategies, 1 / 5 of the test feature model is used for training (see the last row in Table V), so we first randomly divide the feature pairs in test feature model into 5 equally sized subsets. Then we do the classification 5 times, and each time we use a distinct subset for training and optimizing and others for testing. We call the above process a 5-fold train-optimize-classify process. For the cross-domain strategy, the training set and test set do not change so we run a 1-fold train-optimize-classy process. For each setting, each strategy is repeated 20 times, and the average results of optimization and classification are recorded as follows.

TABLE V. COMMON EXPERIMENT PARAMETERS

Parameter	*Value*
The fold of cross-validation	10
The size of solution set	100
The proportion of elite solutions	20%
The probability of mutation (crossover)	30% (70%)
The number of iterations	200
The fraction used for training in the inner-domain and hybrid strategy	1 / 5

TABLE VI. INITIAL AND OPTIMIZED AVERAGE ERROR RATE ON TRAINING SET (10-FOLD CROSS-VALIDATION)

Feature Model	*Training = WS, Test = GPL*			*Training = GPL, Test = WS*		
Strategy	Cross	Inner	Hybrid	Cross	Inner	Hybrid
Avg. Error % (Initial)	18.2	72.89	2.89	16.17	64.68	12.97
Avg. Error % (Optimized)	0.82	12.95	2.40	8.83	4.70	11.01

1) *Results of Optimization*

We use a 10-fold cross-validation to estimate the effectiveness of classifier during the optimization. The number 10 is recommended in most data mining tasks. Table V shows the parameters of the genetic algorithm (row 2 to row 5). The average error rate before and after the optimization is shown in Table VI.

Table VI clearly reflects the need and benefit of optimization. The average error rate of initial parameters is somewhat random, ranging from 2.89% to about 73%. The optimization effectively decreases the error rate, especially for cross-domain and inner-domain strategies. The worst optimized error rate is only 12.95%.

2) *Results of Classification*

The average performance of classification is shown in Table VII, under the columns marked "L" (for Labeled training, Section IV-D will give further explanations).

The first observation is that the cross-domain strategy fails to find any *excludes* constraints. By looking at the features involved in *excludes* in both feature models, we find that descriptions of these features follow totally different patterns in different feature models. In the weather station feature model, the rationale of *excludes* is often beyond the description of the features. For example, an *excludes* states that sending weather report via text message (the feature *Text Message*) does not support XML format (the feature *XML*). Regarding to the four numeric attributes of feature pairs, the two features seem to be totally unrelated. Their similarity is very low, and they do not share any objects or target on others' names. By contrast, the rationale of *excludes* can be deduced from descriptions in the graph product line feature model. For example, two algorithms (each one is a feature) have the same effect on the same type of graphs, so that users must choose between them and therefore they exclude with each other. In this case, the two features are similar to each other; particularly they share many objects that indicate there is interference between their functional areas. The recall of *excludes* is improved by inner-domain and hybrid strategy, due to the training data from the part of test feature model.

The mining of *requires* receives a high recall for each strategy. The reason is that most *requires*-constrained feature pairs follow a similar pattern: one feature targets another to some extent. For example, a certain kind of weather report (e.g. *Storm*) *requires* specific sensors (e.g. *Temperature* and *Pressure*); an algorithm (e.g. *Prim*) only applies to a specific kind of graph (e.g. *Connected* and *Weighted*). In summary, the occurrences of most *requires* can be deduced from the feature descriptions, which benefits the mining of *requires* in our experiments.

The precision of mining binary constraints is not stable in our experiments, and is highly dependent on the test feature models. However, there are still some interesting patterns. First, the cross-domain strategy gets the lowest score again, so it may not be a preferable strategy in practice. Second, a strategy that gives a higher precision in *requires* will give a lower precision in *excludes*. This phenomenon again conforms to the different patterns of the two kinds of constraints as described before. Therefore a classifier that is good at recognizing one kind of pattern is somewhat bad at recognizing another. Finally, a larger test feature model (i.e. the weather station) tends to give higher precisions. The reason is that there are more training data from the test feature model for inner-domain and hybrid strategies, therefore the classifier can be better trained.

In summary, the cross-domain strategy may not be preferable in practice, and there is no significant difference between the inner-domain and hybrid strategies. The classifier gives a reasonably high recall on both kinds of constraints, although the precision depends on the feature model being tested. However in practice, finding a missing constraint in a large feature model is often much harder than reviewing a constraint candidate, therefore the high recall shows that our approach might be promising in practical use.

TABLE VII. COMPARISON OF DATA SET SELECTING STRATEGIES AND TRAINING STRATEGIES

	Requires						Excludes					
	Precision %		Recall %		F_2-Measure		Precision %		Recall %		F_2-Measure	
Strategy	L	LU	L	LU	L	LU	L	LU	L	LU	L	LU
	Training FM = Weather Station, Test FM = Graph Product Line											
Cross-Domain	7.5	**17.53**	**100**	94.44	0.288	**0.503**	N/A	N/A	0	0	N/A	N/A
Inner-Domain	**14.95**	12.14	84.67	**93**	**0.438**	0.399	100	100	100	100	1	1
Hybrid	**23.41**	20.42	84	**84.67**	**0.553**	0.52	14.17	**20.46**	100	100	0.452	**0.563**
	Training FM = Graph Product Line, Test FM = Weather Station											
Cross-Domain	**66.67**	50	100	100	**0.909**	0.833	N/A	N/A	0	0	N/A	N/A
Inner-Domain	**92.67**	86	**100**	94.67	**0.984**	0.928	**22.14**	2.68	80	**100**	**0.525**	0.121
Hybrid	73.06	**74.07**	93.33	**100**	0.884	**0.935**	**35.14**	22.17	66.67	**80**	**0.565**	0.526

D. Compare Different Training Strategies

In the previous sub-section, the classifiers are trained with labeled data (i.e. data in the training set) only. It is also known as the supervised training strategy. Another type of widely adopted training strategy, known as the semi-supervised training, is to train a classifier with both labeled and unlabeled (i.e. unclassified data in the test set) data. The rationale is that although the class of unlabeled data is unknown, unlabeled data still contain a lot of information (e.g. parameter distribution) that could benefit the classifier. A well-known procedure of the labeled/unlabeled (LU) training is sketched as Algorithm 2. First, an initial classifier is trained by the labeled data. The initial classifier performs an initial classification on the unlabeled data. The classified unlabeled data and the labeled data are then used together to re-train the classifier, and perform the classification again. The procedure repeats until the difference of error rate of two consecutive iterations is less than 1%.

Algorithm 2: Labeled/Unlabeled (LU) Training

```
LU_Training (L: Training_Set, U: Test_Set)
   // A supervised, initial training
   Train the classifier by L only
   Classify U
   U′ ← The result of classification

   // Iterated semi-supervised training
   repeat
      Re-train the classifier by L ∪ U′
      Do classification on U with the re-trained classifier
      U′ ← The result of classification
   until the result of classification is stable enough
end
```

We change the training strategy to LU training and keep other things unchanged, and redo previous experiments. The results are shown in Table VII under the columns marked “LU”. We can observe that the LU training strategy slightly improves the recall of *excludes*, however the precision is still not stable. It even decreases the precision to a great extent in many cases.

In summary, the LU training strategy does not improve the performance significantly in our experiments, and considering the additional training time it takes, we prefer the traditional labeled training strategy.

E. The Effect of Limited Feedback

In practice, human analysts may use the classifier to get some results, and then provide feedback on the correctness of the results, and in turn, re-train the classifier with the feedback. Figure 7 explains such scenarios. We call it limited feedback since the number of results presented to human analysts must be small enough to keep their workload as low as possible.

We incorporate a *constraint-first and internal-similarity-first* strategy to choose results for feedback. The internal similarity of a feature pair is the similarity between the descriptions of the paired features. First, all test feature pairs are sorted by descend order of their internal similarity. Then the constraint candidates with the highest internal similarities are selected. If there are not enough constraint candidates, the *non-constrained* candidates with the highest internal similarities are selected to fill the blank.

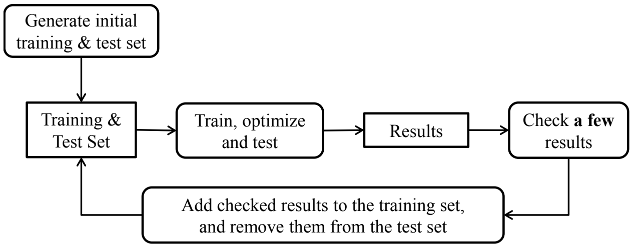
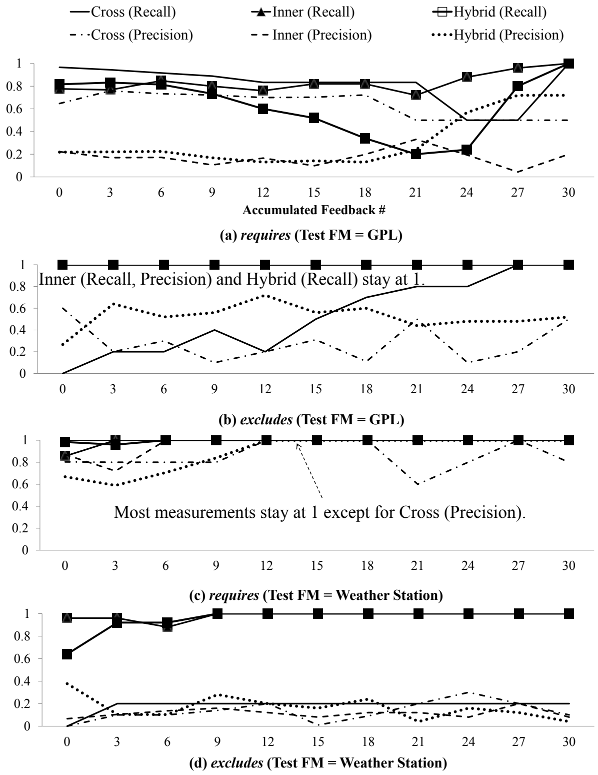

Generate initial training & test set
Training & Test Set
Train, optimize and test
Results
Check **a few** results
Add checked results to the training set, and remove them from the test set

Figure 7. The process of limited feedback.

We experiment limited feedback with supervised training strategy. After each train-optimize-classify process, 3 pairs (i.e. about 2% to 5% of the whole test set) are selected for feedback. The process is looped for 10 times, therefore in the end human analysts need to check about 20% to 50% of test data. We repeat the above experiment for 5 times, and the changes of average precision and recall over accumulated number of feedback are shown in Figure 8.

One observation is that the recall will increase eventually, after a certain number of feedback loops. In other words, the recall at the end is not less than the recall at the beginning in every situation. It is best illustrated by the recall of excludes of cross-domain strategy in Figure 8(b), where the recall increases from 0 to 1. However, the recall fluctuates within a range in some cases, and sometimes the range is wide, e.g.



(a) *requires* (Test FM = GPL)

(b) *excludes* (Test FM = GPL)

(c) *requires* (Test FM = Weather Station)

(d) *excludes* (Test FM = Weather Station)

Figure 8. The effect of limited feedback.

the recall of hybrid strategy in Figure 8(a). The reason is that since we provide feedback on constraint candidates first, the proportion of real constraints left in the test set continuously decreases after each feedback loop, and in turn, the difficulty of mining the rest constraints rises. Therefore the recall may decrease during a period of time. At some point, when the classifier has learned enough amount of feedback (i.e. the training set become large enough), it is again able to find the rest constraints, so that the recall eventually increases.

Another observation is that the precision is still unstable during the feedback loop. It changes a lot (about 40% to 50%) when the test feature model is graph product line, but only fluctuates a little (less than 20%) in another feature model. It is also not guaranteed to increase eventually.

The most useful benefit brought by limited feedback is the improvement in mining *excludes*. The cross-domain strategy fails to find any *excludes* before. With the help of feedback, it eventually finds all *excludes* with a precision around 30% in the GPL feature model (Figure 8(b)), and finds some *excludes* with a precision around 10% in another feature model (Figure 8(d)).

F. *Summary*

Our experiments show that our approach successfully finds binary constraints at a high recall (near 100% in most cases). The precision is unstable and dependent on the test feature models. In most cases the *requires* constraints are better mined than the *excludes* constraints; a possible reason is that the rationale behind *excludes* is often beyond feature descriptions.

Optimization significantly improves the score of cross-validation of classifiers, where the cross-validation is a standard procedure for estimating the effectiveness of classifiers in data mining research.

The inner-domain and hybrid data set selecting strategies perform better than the cross-domain strategy. It conforms to the intuition that the differences between domains are significant. However, when trained by a small number of known binary constraints in a feature model, our classifier can be effectively find binary constraints left in the feature model, no matter which domain it belongs to.

The more complicated and time-consuming training strategy, i.e. the semi-supervised training, does not show significant improvement in the experiments. The traditional supervised training is enough for us.

Continuous feedback from human analysts benefits the mining process, especially for mining excludes constraints. Therefore in practice, our classifier should be used in an interactive way, that is, human analysts check only a few constraint candidates after each turn of mining, and then the classifier repeats the train-optimize-test process again.

V. THREATS TO VALIDITY

A. *External Validity*

A major threat to the external validity of our experiments is that we did not carry out the experiments on real feature models. By “real” we mean that the feature model is created from real applications by experienced domain analysts, such as the Linux and eCos [12] feature models. We did not use these real feature models because they do not contain feature descriptions. However, the two feature models used in our experiments share some characteristics with real feature models, especially regarding to our approach:

First, the abstract of a definition in Wikipedia and the description of a feature are similar in the way of wording and expression: they both strive for clear and concise, and they both tend to use domain terminologies. It means that there are few unnecessary words disturbing the computation of the four numeric attributes for feature pairs.

Second, some features do not need descriptions in real feature models. For example, some common terminologies, such as File or Play Music, can be understood without further explanation. The feature models we used also contain some non-described features, and our classifier still works.

Another threat is that the size of data set in our experiment is smaller than most data sets used in the field of data mining, so the performance of the classifier may be affected. Our solution is to optimize the classifier before it is used for classification. Researchers find that such an optimization even works for small data set (300 to 400 data) [8], and our data set has about 300 data so it is still suitable for a classifier to work. In fact, the error rate before and after optimization reported in [8] follows a similar pattern to ours.

B. *Internal Validity*

We strive for high recall of the binary constraints, because finding a missing constraint in a large feature model is much harder than reviewing constraint candidates. However, the relatively low precision may still impose heavy workload on human analysts, because people have to review a large number of candidates to get all constraints. Improving the precision is a part of our future work.

The size of our data set is unfair to semi-supervised training strategy, because the strategy performs best when the test set is much larger than the training set, which is not the case in our experiment. In a large feature model that contains thousands of features and maybe hundreds of thousands of feature pairs, the semi-supervised training strategy may perform better.

VI. RELATED WORK

Since a feature can be treated as “a cohesive set of individual requirements” [15], many researchers [2][4][14] have proposed a *requirements-clustering*-based approach to help identify features and construct feature trees. Their ideas are similar. First, they perform a hierarchical clustering on requirements documents of a single product to get a feature tree of the product. They repeat this procedure on a family of products so that they get a family of feature trees. Finally, they merge the feature trees into a domain feature model, and particularly, determine the variability of each feature. Their major drawback is the lack of retrieving constraints. In contrast, our work takes a feature tree as input and finds cross-tree binary constraints. Therefore it is possible that the two kinds of approaches are used together to build the entire feature model from requirements in a semi-automated way.

Acher et al. [1] proposed an approach to build a feature model from a family of products described in a set of tables. The header row of each table indicates top-level features, and each body row represents a single product in which each cell may further indicate low-level features. In this method, users extract feature trees via specifying rules in a language called VariCell. Binary constraints can be extracted by analyzing the tabular data. For example, if two features never (resp. always) appear in the same row, there is an *excludes* (resp. *requires*) relation between them. A potential problem of this idea is that it only reflects the truth of a *sub*-domain that is depicted by the input products. In contrast, the descriptions input to our method are written from a domain's perspective and contain more domain knowledge than tabular data, so that we may find constraints that really exist in the domain.

She et al. [12] proposed an approach to recover a feature model from a given set of features and constraints. Their main purpose is to recover the refinements (also known as implicit constraints) hidden in the constraint set. They also compute similarity between features to recommend possible parent features for a given feature. The full constraint set is given as an input, which is the major difference from our work. Similarly, Czarnecki [5] proposed a method to build a feature model from a constraint set (given as propositional logic formulas), however they do not utilize feature names and descriptions and it leads to an unsatisfying feature tree.

Many approaches have been proposed to find relations between software requirements. We would like to highlight the research of Dag et al. [6]. Although their purpose was to find duplicate requirements using the cosine (and other) similarity measurement, they also reported that the similarity between requirements seems to significantly relate to other constraints (similar to *requires* or *excludes*) between them. They did not go further, however their findings give us more confidence to find constraints based on measuring similarity.

The support vector machine classifier, LIBSVM, used in our experiments is implemented by [3]. The need of optimizing a classifier is explained in [8]. The basic idea of genetic algorithm is learned from [7], and we implemented the algorithm by ourselves.

VII. CONCLUSIONS

In this paper, we propose an automated approach to mining binary constraints from a partially constructed feature models. We first construct feature pairs from the input, and then train a classifier with previous known binary constraints. A genetic algorithm is utilized to optimize the classifier. The trained and optimized classifier is then check feature pairs with unknown class to find possible binary constraints. We conduct a series of experiments on two feature models in the SPLOT repository. The descriptions of the features are extracted from Wikipedia. We evaluate different data set selecting strategies and training strategies. We also examine the effect of continuous feedback on predicted constraint candidates. Results show that we can mine binary constraints at a high recall (near 100% in most cases).

Our future work will focus on improving the precision of finding constraints and also perform user experiments and case study to evaluate the approach in practical use.

ACKNOWLEDGMENT

The authors want to thank the anonymous reviewers for their valuable comments and suggestions. This work is supported by the National Basic Research Program of China (973) (Grant No. 2009CB320701), Key Project of National Natural Science Foundation of China (Grant No. 90818026), National Natural Science Foundation of China (Grant No. 60873059), and the National Key Technology R&D Program (Grant No.2008BAH32B02).

REFERENCES

[1] M. Acher, A. Cleve, G. Perrouin, P. Heymans, C. Vanbeneden, P. Collet, P. Lahire. On extracting feature models from product descriptions. In Proc. 6th International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS 12), 45-54.

[2] V. Alves, C. Schwanninger, L. Barbosa, A. Rashid, P. Sawyer, P. Rayson, C. Pohl, A. Rummler. An Exploratory Study of Information Retrieval Techniques in Domain Analysis. SPLC 2008, 67-76.

[3] C. C. Chang and C. J. Lin, LIBSVM : a library for support vector machines. ACM Transactions on Intelligent Systems and Technology, 2:27:1--27:27, 2011.

[4] K. Chen, W. Zhang, H. Zhao, H. Mei. "An approach to constructing feature models based on requirements clustering," In Proc. IEEE International Conference on Requirements Engineering, 2005: 31- 40.

[5] K. Czarnecki, A. Wasowski. Feature Diagrams and Logics: There and Back Again. Software Product Line Conference, 2007. SPLC 2007. 11th International , vol., no., pp.23-34, 10-14 Sept. 2007.

[6] J. N. Dag, B. Regnell, P. Carlshamre, M. Andersson and J. Karlsson. A Feasibility Study of Automated Natural Language Requirements Analysis in Market-Driven Development. Requirements Engineering, 2002 (7): 20-33.

[7] D.E Goldberg. Genetic algorithms in search, optimization and machine learning. Addison-Wesley, 1989.

[8] C. W. Hsu, C. C. Chang, C. J. Lin. A practical guide to support vector classification. http://www.csie.ntu.edu.tw/~cjlin/papers/guide/

[9] 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.

[10] D. Klein, C. D. Manning. Accurate Unlexicalized Parsing. Proceedings of the 41st Meeting of the Association for Computational Linguistics, 2003, pp: 423-430.

[11] K. Nigam, A. K. Mccallum, S. Thrun and T. Mitchell. Text Classification from Labeled and Unlabeled Documents using EM. Machine Learning, vol. 39, no. 2-3, 103-134, 2000.

[12] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. Reverse engineering feature models. In Proceedings of the 33rd International Conference on Software Engineering (ICSE '11). ACM, New York, NY, USA, 461-470.

[13] J.A.K. Suykens and J. Vandewalle. Least Squares Support Vector Machine Classifiers. Neural Processing Letters, vol. 9, 293-300, 1999.

[14] N. Weston, R. Chitchyan, A. Rashid. A framework for constructing semantically composable feature models from natural language requirements. In Proc. 13th International Software Product Line Conference (SPLC '09), pp. 211-220.

[15] W. Zhang, H. Mei, H. Zhao, "A feature-oriented approach to modeling requirements dependencies," in Proc. of the 13th IEEE Intl. Conf. on Requirments Engineering (RE 05), 2005, pp. 273–282.