Ontology-Based Feature Modeling and Application-Oriented Tailoring

Xin Peng, Wenyun Zhao, Yunjiao Xue, and Yijian Wu

Computer Science and Engineering Department, Fudan University, Shanghai 200433, China
{pengxin, wyzhao, yjxue, wuyijian}@fudan.edu.cn

Abstract. Feature models have been widely adopted in domain requirements capturing and specifying. However, there are still difficulties remaining in domain model validating and application-oriented tailoring. These difficulties are partly due to the missing of a strictly defined feature meta-model, which makes it difficult to formally represent the feature models. Aiming at the problem, we propose an ontology-based feature modeling method supporting application-oriented tailoring. In this method features are classified into several categories and are all precisely defined in the OWL-based meta-model. Expression capacity of the feature model can be greatly improved due to the rich types of features. On the other hand the feature model can be easily converted into ontology model and be validated through ontology inference. Application-oriented tailoring can also gain support from the reasoning-based guidance. Finally, advantages of ontology-based feature modeling, especially for component and architecture design, are discussed with our conclusions.

1 Introduction

Domain analysis is an essential activity for successful reuse across applications in the same domain. Based on the domain model, domain assets, including DSSA (Domain-Specific Software Architecture) and domain components can be produced. When implementing a new system, products of domain analysis will be tailored to produce the specification and design of that system [1]. So, it is obvious that domain model is essential for domain and application-specific development.

To provide necessary supports for the domain and application development, the domain model should meet some requirements. First, it should provide sufficient business knowledge for the design of architecture and components. Second, it must offer the means to specify both features and their composition rules [2]. Then validation of the model can be performed on constraints. A domain model is valid if there is at least one set of single-system requirements that can be generated from it and satisfy the constraints [3]. An inconsistent domain model is meaningless for application development and will cause unnecessary investment waste in domain development. Third, it should present an applicable way for application engineers to configure the domain model according to product-specific requirements.

Nowadays, concept of feature model has been widely adopted in domain requirements capturing and specifying, since FODA [1] introduced it into domain engineering. However, a strictly defined feature meta-model is often missing, which makes

M. Morisio (Ed.): ICSR 2006, LNCS 4039, pp. 87 – 100, 2006.

feature models difficult to be formally represented. Thus, validation and application-oriented tailoring of domain models are also difficult due to the missing of a formal basis. Schlick [4] also indicated that a solid theoretical foundation is needed first.

Our research group is concerned with feature-driven domain and application development, including domain analysis, DSSA design, application-oriented tailoring and component-composition based application development. In these fields a strictly-defined modeling basis for features is essential, which should provide a mechanism to connect model elements in various development phases. Ontology related theory is a suitable way to achieve our goals. Ontology is a conceptualization of a domain or subject area typically captured in an abstract model of how people think about things in the domain [5]. Rubén [5] considers domain models as narrow or specialized ontology, and the main difference is that domain models define abstract concepts in an informal way and have no axioms. Because of the facilities for the generalization and specialization of concepts and the unambiguous terminology it provides [6], ontology has been widely used in domain knowledge representation and requirement modeling, reuse and consistency checking. For example, Sugumaran etc. [7] proposed a semantic-based approach to component retrieval, in which ontology and domain models are adopted for capturing application domain specific knowledge to express more pertinent queries for component retrieval. Girardi etc. [6] proposed GRAMO, an ontology-based technique for the specification of domain and user models in Multi-Agent Domain Engineering.

This paper proposes an ontology-based feature modeling method, in which features are divided into several categories (e.g. action, facet, and term) and defined as ontology concepts respectively. In this way, we can provide better support for domain modeling, and succeeding domain design and implementation. First, ontology-based feature model can be formally represented easily and validation of the model can be realized through ontology reasoning. Second, the ontology-based unambiguous terminology and faceted feature description provide precise and detailed semantic knowledge for the domain, so the feature model can also be adopted as the domain business model. The rich business information in it can be used as semantic infrastructure for component description and architecture design. Third, modularity can be supported by feature facets and dependency semantics. Finally, it provides a unified meta-model for both domain model and application model and a stepwise way for customization, which ease the application-oriented tailoring of feature models.

The remainder of this paper is organized as follows. Section 2 presents the ontology-based feature modeling architecture and the meta-model. Section 3 and section 4 discuss model validating and application-oriented tailoring respectively. Finally, we draw our conclusions with discussion of ontology-based feature modeling and future work in section 5.

2 Ontology-Based Feature Model

This section introduces the ontology-based feature modeling architecture and the feature meta-model first. Then some basic rules for feature modeling are introduced in 2.3, followed by an example of feature model in 2.4 for demonstration. Finally representation of complex constraints and the modularity mechanism are specially

discussed in 2.5 and 2.6 respectively. Since OWL [8] is a W3C recommendation for ontology representation, we define the meta-model on OWL.

2.1 Ontology-Based Feature Model Architecture

Our ontology-based feature modeling architecture is presented in figure 1, which is based on a three-layer structure with an ontology level besides the traditional feature model level and meta-model level. The meta-model is strictly defined on OWL, so the feature model can be easily converted into OWL model. Constraints-related rules are also given in the meta-model level and then converted into ontology rules. By ontology reasoning we can perform both domain model validating and tailoring guidance, which are discussed in section 3 and 4 respectively.

The structure is similar to the XML-based three-layer structure in [2], while the main difference is that in our method domain model and application model share the same meta-model, only with different variability policy. Domain model contains domain-level variability, which will be specialized during application engineering to derive final products [4]. Even application model may still contain variability, which will be fixed at runtime by parameters or configuration files.

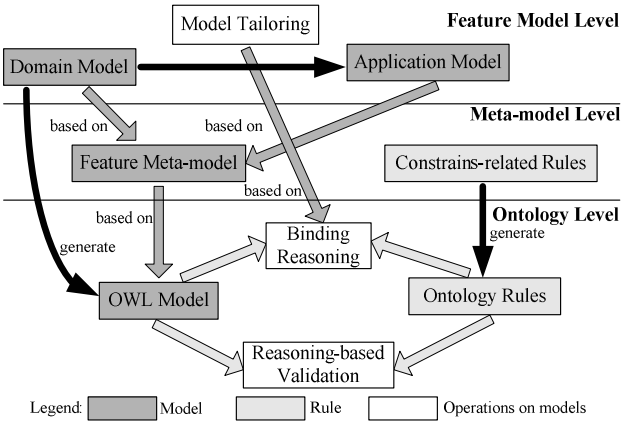

Fig. 1. Ontology-based feature modeling architecture

2.2 Feature Meta-model

Intuitively, a feature is a coherent and identifiable bundle of system functionality that helps characterize the system from the user perspective [9]. So operations with business semantics are basis of the feature model. In feature-oriented domain analysis, aggregation and generalization are applied to capture the commonalities among applications in the domain in terms of abstractions [1]. Differences between applications are captured in the refinements [1], mainly by decomposition and specialization.

In our method, features are divided into several categories. The basis is *Action*, which represents business operations. In order to provide more business details for actions, we introduce the concept of facet for actions, which can be construed as perspectives, viewpoints, or dimensions [5] and is a widely used classification scheme in

library science. *Facet* is defined as dimension of precise description for *Action*. An action can have multiple facets and the facets can be inherited along with generalization relations between actions. Restricted value space for *Facet* is represented by *Term*. By *Facet* and *Term*, we can define a feature in more detail. This mechanism provides the "feature attributes" expressiveness, as proposed in [4].

In the ontology-based feature meta-model (depicted in figure 2) *Action* and *Term* are defined as OWL Class (*owl:Class*). *Facet* is defined as OWL property (*owl:Property*), domain (*rdfs:domain*) and range (*rdfs:range*) of which are *Action* and *Term* respectively. As OWL classes, *Action* and *Term* can form their own specialization tree by the relation *subClassOf*, which represents direct inheritance. All the *subClassOf* instances will be converted into *rdfs:subClassOf* in the OWL model, and the direct *subClassOf* relations will be reserved.

In figure 2, domain and range of OWL properties are represented by aggregation elements with solid and dashed lines respectively. *HasElement* is another important relation, which represents decomposition between certain action and its sub-actions.

Specialized and optional features are two basic mechanisms for variation [10]. In our method, the former is achieved by generalization between actions or terms, while the latter is denoted by the *IfOptional* property defined on *HasElement*. The range of *IfOptional* is the RDF data type *xsd:Boolean* and the value *True* means the sub-action is an optional element for its parent action. It should be noticed that the optional property is not defined on action itself but on the relation with its parent, so an action can have different optional property for different parents. In OWL, a property is also a class, so we can easily define a property on another property.

Three kinds of dependencies are identified in our method, namely *Use*, *Decide* and *ConfigDepend*. *Use* denotes the dependency on other features for its correct functioning or implementation [11], which is somewhat similar to *HasElement*. However, parent action in *HasElement* relationship can be seen as cooperation framework for its sub-actions, while a *Use* client often implements some actual function and assign certain part to the supplier. *Decide* indicates that execution result of an action can determine which variant of a variable action will be bound for its parent action (*HasElement*) or client (*Use*). So, the range of *Decide* includes *HasElement* and *Use*. *Use* and *Decide* embodies direct runtime functional dependencies, while *ConfigDepend* represents configuration constraints, which are static dependencies on binding-states of variable features [12]. Besides *Action*, the meta-model provides more precise expressiveness for elements of configuration dependencies by *FacetValue*. *FacetValue* itself is an OWL property from *Facet* to *Term* or *Boolean*, representing the facet value assumption of certain action.

The phase in the software life-cycle to decide whether a variation feature should be bound to or removed is called binding-time of features [13]. And the binding times will influence the design of system architectures [10]. So *HasBindTime* is defined on *HasElement* or *Use* relationships where the sub-action or use supplier action has variations (e.g. optional *HasElement* instance or abstract action with several variants). Typical binding times include reuse-time, compile-time, load-time, and run-time [1, 10]. However, only two binding times of *BuildTime* and *RunTime* are identified, since the main purpose of feature model is to support the tow development phases of domain engineering and application engineering. Intuitively, *BuildTime* binding variations should be implemented as abstract components, which are replaced by

application components in product development, while *RunTime* binding variations should be implemented as domain components with variant-selecting proxy or interface parameters, which can determine the final semantics of the invoked service at runtime.

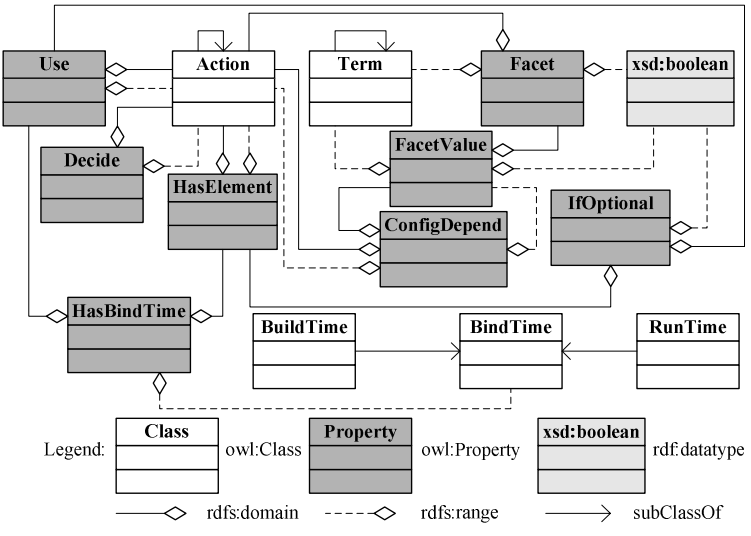

Fig. 2. Ontology-based feature meta-model

2.3 Basic Rules

There exist some basic rules for feature modeling. They act on the feature model directly and can be easily validated in modeling process. Because of the limited space, only three of them are listed here.

Definition 1. Facet set and facet value
$\forall action \in Action$, *FacetSet(action)* denotes all the effective facets for action. And for $\forall facet \in FacetSet(action)$, *action.facet* represents the value of *action* taken on *facet*. For example, there is *TextualDisplay.NeedProtocol=Telnet* in figure 3.

Property 1: Facet value restriction on sub-actions
$\forall a1,a2 \in Action$, $f \in FacetSet(a1)$, lets (*a2 subClassOf a1*), then $f \in FacetSet(a2)$ and (*a2.f subClassOf a1.f*).

Note that in OWL a concept is a subclass of itself.

Property 2: Decomposition consistency
$\forall A1,A2 \in Action$, (*A2 subClassOf A1*), then for $\forall a1 \in Action$ and *A1-a1*=(*A1 HasElement a1*):

If (*A1-a1 IfOptional False*), then $\exists a2 \in Action$, (*a2 subClassOf a1*), *A2-a2*=(*A2 HasElement a2*) and (*A2-a2 IfOptional False*).

This property states that all the mandatory elements of an action should be reserved in its variants. For the optional elements, the variants can have their own policies (removed, or reserved to be mandatory or optional).

Property 3: Decide binding time

$\forall a1,a2 \in Action$, *a1-a2*=(*a1 HasElement a2*) or *a1-a2*=(*a1 Use a2*), if $\exists a3 \in Action$ and (*a3 Decide a1-a2*), then (*a1-a2 HasBindTime Runtime*).

This property limited the target of *Decide* to have the binding time of *Runtime*: The source of Decide will execute at runtime to determine the variant of the target.

2.4 Feature Model Example

A segment of EBBS domain feature model is presented in figure 3. The top feature *BBSService* is decomposed into *UserLogin*, *MailService*, *BoadService*, and *MsgDisplay*, in which *MailService* is identified as optional element. Facet *HasFilePolicy* is defined between *BoardService* and *FilePolicy*, which represents whether file-upload is supported when posting and the allowed file types. According to the concept inheritance semantics in OWL, facets defined on an action are inherited by its variants. Furthermore, values of the facets are restricted by the super-action. For example, *NeedProtocol* is defined on *MsgDisplay*, so the facet values of *NeedProtocol* on *GraphicalDisplay* and *TextualDisplay* are restricted to be subclass of Protocol (*Http* and *Telnet* in figure 3). Besides abstract and optional actions (denoted by *subClassOf* and *IfOptional*), *Facet* also reserves variations for corresponding *Action* feature. For instance, *Facet HasStorageWay* defined on *FileTransfer* enables application engineers to make the decision of storing uploaded files in database or file system.

Fixing stages of all these variations are indicated by *BindTime*. For example, *BindTime* of the *HasElement* instance between *BBSService* and *MailService* is BuildTime, so the supported file policy of *BoardService* should be decided in application development and be fixed at runtime.

Two variants of *UserLogin* are identified in the model. *NormalLogin* means accepting and treating the request normally, while *RejectLogin* means rejecting the request. The binding time of them is *RumTime* and the binding is determined by *LoadEvaluate*. That means evaluating the system load and rejecting the login request when the load is heavy. *Decide* is similar to the *Modification* dependency in [11], which is interpreted as that the behavior of a feature can be modified by another feature. We consider *Decide* is clearer for dependency description in that the modifier is just a determining factor for variant-selecting of modifiee.

Both *BoardService* and *MailService* have *Use* dependencies on *FileTransfer*, and they are both optional. It can be seen from the *ConfigDepend* on *FileTransfer* that facet values of *BoardService* and *MailService* can determine the binding of *FileTransfer*. The *FacetValue* on *BoardService* can be interpreted as the situation of "*BoardService* having the value of *FileSupport* on the facet of *HasFilePolicy*". The appearance of that situation depends on the binding of *FileTransfer* and *GraphicalDisplay*.

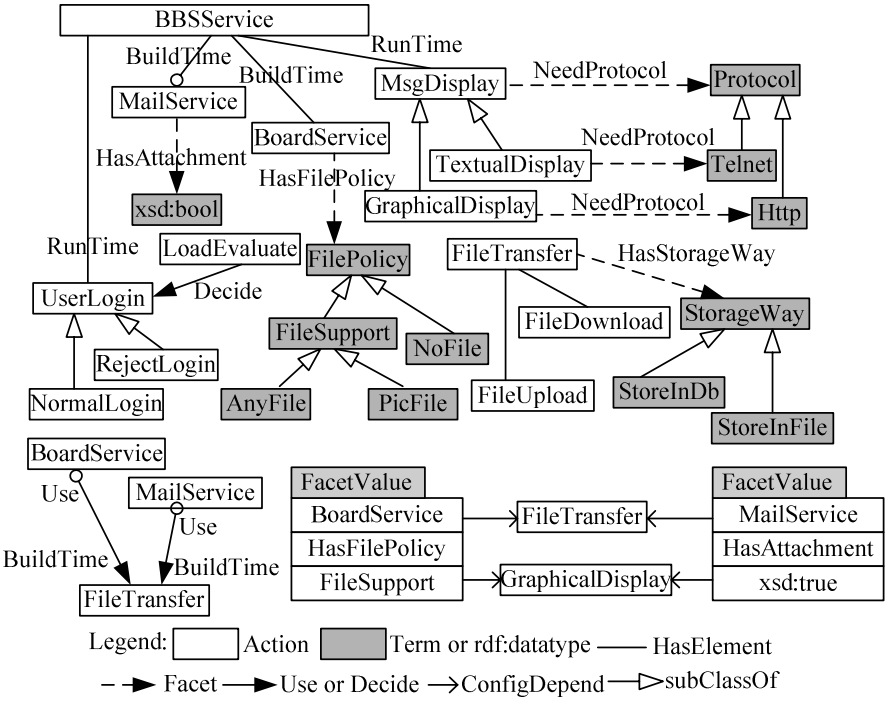

Fig. 3. Example of EBBS domain feature model

2.5 Complex Constraints

Constraints are a kind of static dependencies among binding-states of features [12], which provide a way to verify the results of requirements customization [13]. Three constraint categories, namely binary constraints, group constraints, and complex constraints, are identified in [12].

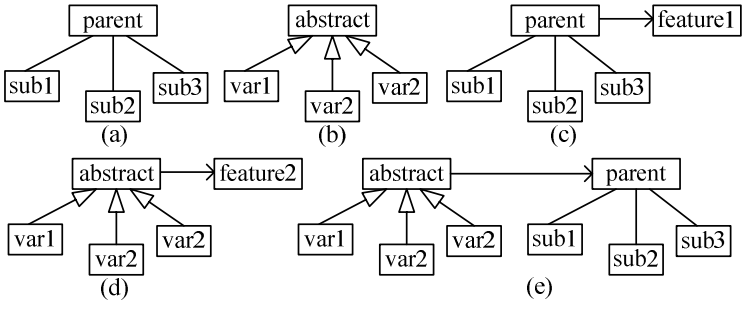

Fig. 4. Expression of complex constraints

In our method, constraints are represented by *ConfigDepend*, which is a kind of binary constraints defined between *Action* and *FacetValue*. By decomposition, specialization and binary constraints, we can also intuitively express most group constraints and complex constraints referred in [12]. Figure 4 shows the most usual cases. Figure 4a shows the common case of *all-bound* group constraints in that mandatory elements *sub1*, *sub2*… should be bound for *parent*. Figure 4b indicates that one and only one of the variants var1, var2… of *abstract* should be bound for it. Figure 4c-4e present common expressions of complex constraints. For example, 4e denotes that the binding of *abstract* (any of its variants) requires the binding of *parent* (all of its elements).

2.6 Modularity Mechanism

An applicable modularity mechanism is essential for feature modeling, especially when the model is large and complex. For example, the *feature macro* mechanism is proposed in [2] to split a large model into independent modules. A feature macro contains a feature node with all its sub-nodes, and can be extended in different instances by adding new features to its sub-tree. But this parameterized modularity mechanism has some shortages in flexibility and encapsulation.

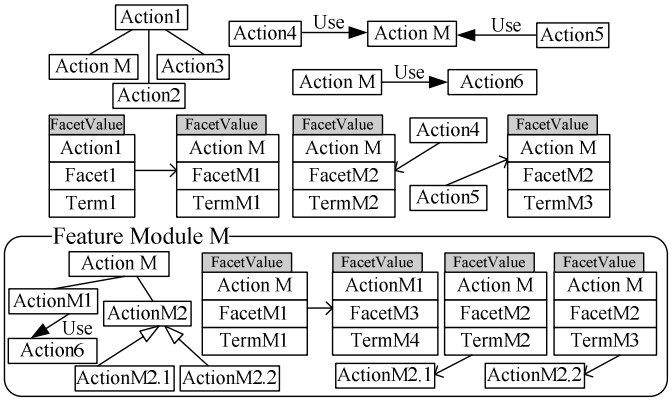

Fig. 5. Facet-based feature modularity mechanism

In our method, a module is a block box with a top feature and all its sub-features, which can be referred at different points of upper feature diagrams in different manners. The inner structure of the module is defined in its module feature diagram. Thus, a large feature model can be easily decomposed into modules at various layers. Figure 5 demonstrates the facet-based feature modularity mechanism, in which *Action M* is referred as whole by three actions and the inner structure of the module is defined in *Feature Module M*. Action1, Action4 and Action5 reuse *Action M* by the relationships of *HasElement* and *Use* with different parameters, which are expressed by those *ConfigDepend* instances on *Action M*. For example, *Action4* demands the facet *FacetM2* of *Action M* to be *TermM2*. Thus *Feature M* can be reused for different detailed demands as a black box, only its facets can be referred as "feature interface". Inner model of Feature M can be defined independently. In *Module M* (depicted in figure 5), *Action*

M is further described by its sub-features. Realization of various semantics of *Action M* depends on its sub-features. The dependencies are represented by the *ConfigDepend* instances between *Action M* and its sub-features. So we can see how various outer demands can be implemented by *module M* clearly. Besides, the sub-features can also employ those outer features provided by the top feature. For example, *Action M* has *Use* relationship with *Action6*, so *Action6* can be referred in *module M* by *ActionM1*. Thus, when developing the feature model of a module, one need only to consider how to meet various facet semantics of the top feature with outer resources provided by the top feature.

3 OWL-Based Formal Representation and Validation

Based on the meta-model over OWL we can transform a feature model into OWL model easily. Then reasoning-based validation can be performed on the ontology by inference engine. Jena [14] is a widely used Java framework for building semantic web applications, which provides a programmatic environment for RDF, RDFS and OWL, including a rule-based inference engine. In our implementation, it is adopted to generate the OWL file and reason on it.

3.1 OWL Representation of Feature Model

A segment of the OWL model for the EBBS domain feature model (depicted in figure 3) is presented in figure 6. *Action* features (e.g. *BBSService*, *MsgDisplay*) are defined as subclasses of *Action*. Facets (e.g. *NeedProtocol*) are defined as subproperties of *Facet* with various facets and terms. The decomposition relationship between *BBSService* and *MailService* is denoted by *BBSService-MailService*, which is a subproperty of *HasElement* with the *IfOptional* value "true". The OWL model can be generated by Jena API and be further processed with inference rules.

3.2 Constraints-Related Rules

This subsection introduces constraints-related rules for feature modeling. Constraints on variation features capture the binding relations with other features [10]. Validation of these rules is not as direct as the basic rules (see 3.1). So some methods are proposed to capture and validate on the constraints, such as propositional logic [10, 13], first-order logic [3] and XSL [2]. In our method, constraints are captured by inference rules and validated by ontology reasoning. So the constraints-related rules are described as Jena OWL rules here. A Jena rule has the format of:

[uncle: (?a fatheris ?b), (?b brotheris ?c) -> (?a uncleis ?c)].

This rule denotes a simple inference that a brother of one's father is his uncle.

For a variation feature, the binding-state can be *Bound*, *Removed* or *Undecided* [13]. Features with the state *Bound* or *Removed* are state-decided, while *Undecided* features remain to be variable. In our ontology-based feature model, the *Action* feature also has these three binding-states. However, there is still another binding type, i.e. specialization of *Facet* values. For example, the *Action FileTransfer* can be specialized to have the value *StoreInFile* in the application model. In order to describe

the binding-state of Action features, we introduce a new ontology property *HasState* for *Action* with two possible value of *Bound*, *Removed* and *Conflict* (denotes the inconsistent state). And the existing property *Facet* can represent current state of *Facet* value binding. For example, in figure 3, when the *Facet HasFilePolicy* takes the value *FileSupport*, we will know *BoardService* is chosen to support file and *NoFile* has been negated. However, the binding of *AnyFile* or *PicFile* is still undecided.

For feature constraints, the basic relationships are *Require* and *Mutex*. They have the following semantics: (1) (*a Require b*) denotes that binding of feature *a* depends on the binding of feature *b*; (2) (*a Mutex b*) denotes feature *a* and *b* can not be bound at the same time. All constraints will be converted to these two basic relations and then be validated.

Property 4: Decomposition constraints
Description: Binding of an *Action* depends on the binding of its mandatory elements and *Use* suppliers.
Ontology Rule:
[(?a ?h ?b), (?h rdfs:subPropertyOf HasElement), (?h IfOptional false) -> (?a Require ?b)].
[(?a ?u ?b), (?u rdfs:subPropertyOf Use), (?u IfOptional false) -> (?a Require ?b)].

Property 5: Action specialization constraints
Description: If one and only one variant of an Action is bound, it is bound itself. Note that *subClassOf* denotes the direct inheritance defined in the meta-model (not the transitive *rdfs:subClassOf* in OWL).
Ontology Rule:
[(?a rdfs:subClassOf Action), (?a subClassOf ?b) -> (?a Require ?b)];
[(?c rdfs:subClassOf Action), (?a subClassOf ?c), (?b subClassOf ?c) -> (?a Mutex ?b)].

Property 6: Facet value constraints
Description: Binding of a *FacetValue* feature depends on the binding of the corresponding *Action* feature and those *FacetValue* features with more general value.
Ontology Rule:
[(?f rdfs:subPropertyOf Facet), (?a ?f ?t), (?t subClassOf ?m) -> (?a ?f ?m)];
[(?f rdfs:subPropertyOf Facet), (?a ?f ?t), (?v rdfs:subPropertyOf FacetValue), (?f ?v ?t) -> (?a Require v?)];
[(?v rdfs:subPropertyOf FacetValue), (?f ?v ?t), (?a ?f ?m) -> (?v Require ?a)].

Property 7: Configuration dependency constraints
Ontology Rule:
[(?b Decide ?a) -> (?a Require ?b)];
[(?a ConfigDepend ?b) -> (?a Require ?b)].

Property 8: Binding constraints
Ontology Rule:
[(?a Require ?b), (?a HasState Bound) -> (?b HasState Bound)];
[(?v rdfs:subPropertyOf FacetValue), (?f ?v ?t), (?a ?f ?m) -> (?a ?f ?t)].

Property 9: Removing constraints
Ontology Rule:
[(?a Require ?b), (?b HasState Removed) -> (?a HasState Removed)];
[(?a Mutex ?b), (?a HasState Bound) -> (?b HasState Removed)];
[(?a Require ?b), (?a Mutex ?b) -> (?a HasState Removed)].

```
……
<rdf:Property rdf:about="#HasElement">
  <rdfs:range rdf:resource="#Action"/>
  <rdfs:domain rdf:resource="#Action"/>
</rdf:Property>
<owl:DatatypeProperty rdf:ID="ifOptional">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#boolean"/>
  <rdfs:domain rdf:resource="#Use"/>
  <rdfs:domain rdf:resource="#HasElement"/>
</owl:DatatypeProperty>
<owl:Class rdf:ID="BBSService">
  <EBBS:BBSService-MsgDisplay rdf:resource="#UserLogin"/>
  <EBBS:BBSService-MsgDisplay rdf:resource="#MsgDisplay"/>
  <EBBS:BBSService-BoardService rdf:resource="#BoardService"/>
  <EBBS:BBSService-MailService rdf:resource="#MailService"/>
  <rdfs:subClassOf rdf:resource="#Action"/>
</owl:Class>
<rdf:Property rdf:ID="BBSService-MailService">
  <rdfs:subPropertyOf rdf:resource="#HasElement"/>
  <EBBS:IfOptional>true</EBBS:IfOptional>
</rdf:Property>
<rdf:Property rdf:ID="Facet">
  <rdfs:range rdf:resource="#Term"/>
  <rdfs:domain rdf:resource="#Action"/>
</rdf:Property>
<rdf:Property rdf:ID="NeedProtocol">
  <rdfs:subPropertyOf rdf:resource="#Facet"/>
</rdf:Property>
<owl:Class rdf:about="#MsgDisplay">
  <EBBS:NeedProtocol rdf:resource="#Protocol"/>
  <rdfs:subClassOf rdf:resource="#Action"/>
</owl:Class>
……
```

Fig. 6. Segment of OWL representation for the EBBS domain model

3.4 Reasoning-Based Validation

Rules defined in 3.3 capture restrictions among features. In order to derive the validation conclusions from the model, we define the final rule for *conflict* as:

[(?a HasState Bound), (?a HasState Removed) -> (?a HasState Conflict)].

Besides the feature model, users should also specify the mandatory features in the model. These mandatory features (usually the top service features) will be represented by the relation of (*feature HasState Bound*). Then we can convert the OWL model into RDF statements and store them in a database (such as MySQL in our implementation) with Jena [14]. The conversion is executed along with the reasoning based on the inference rules, including basic rules (e.g. RDFS and OWL axioms) and user-defined rules.

After reasoning and RDF statements storage, we can get the validation result directly from the database by the RDF triples of (*a HasState Conflict*). If there is no such statement, then the model is valid. Otherwise, the model is inconsistent denoting an invalid original domain model or tailoring decisions. However, we can see in section 4 that the tailoring process can be guided by reasoning to avoid invalid decisions.

Besides *consistency*, *nonredundancy* and *necessity* are also identified as properties of well-formed domain feature models in [10]. These two properties can also be

validated by reasoning: violation of *nonredundancy* means some variation features are set to be *Removed*, and violation of *necessity* means some variation features are set to be *Bound* (mandatory).

4 Application-Oriented Tailoring

The domain model needs to be customized in application engineering. In the instantiation process generic assets provided by the domain are configured to build a particular application [2]. After that, reuse decisions for domain assets and development of product-specific assets can be made. So application-oriented tailoring of the domain model is essential for successful product development. Two methods of requirement customization, free selection and discriminant-based selection, are identified in [15]. The former allows engineers to select requirements from the product line model freely, while the latter uses constraints and permits choices to be made only at discriminant points. In our opinion, free selection is likely to be used for immature domains, while discriminant-based selection looks fit for mature, steady and thoroughly-analyzed domains. Discriminant-based selection is chosen to be supported in our method, since it can ease the tailoring and strictly ensure the consistency between domain and application feature models.

Domain models in practice are often large and complex, so two mechanisms are provided to ease the tailoring. First, the model can be customized gradually with the hierarchical modules. When tailoring is executed in a hierarchy, only features or modules (see 2.5. Modularity Mechanism) in the same hierarchy are involved. After that, tailoring process is executed within each module involved in the upper hierarchy. It is obvious that well-designed hierarchy and modularity is essential for the effect of this principle. Second, the tailoring can be guided by identification of free variable for customization. A free variable means a variation feature on which no other variation feature depends for binding. Identification of active points is obvious: those variation features do not appear as the *object* of *Require* statements in the ontology. Also, in a well-designed model, free variables usually appear in upper hierarchies relative to passive variables.

5 Evaluation and Conclusion

A strictly-defined formal basis is essential for applicable feature modeling. In this paper, ontology is introduced as the definition foundation of the feature meta-model. Ontology has been widely adopted in domain knowledge modeling and has corresponding modeling language, such as OWL. So the feature model can be easily converted into formal ontology model. Furthermore, rule-based reasoning can be performed on the ontology model for model validating and tailoring guidance.

Establishing a mapping between domain model and the architecture is the objective of domain engineering [16]. However, there is a large gap between the problem space and the solution space. We can reduce the gap by establishing a smooth transition from elements in the domain model (i.e. features) to elements in the architecture model (i.e. components). In our research, domain ontology (i.e. the ontology-based feature model) is also representation basis for component semantics. We can first map

the feature model to conceptual architecture, which defines business function assignments among components and semantic interactions between them. Then the conceptual architecture can be converted into the artifact architecture by combining technical factors (e.g. the component model and platform).

The ontology-based method has been implemented in OntoFeature (depicted in figure 7), an ontology-based feature modeling tool. It supports multi-view feature modeling and generation of the integrative feature diagram. Generation and validation of ontology model are also implemented by Jena APIs integrated in it. In the process of application-oriented tailoring, the tool provides a stepwise guidance through active-variations analysis.

Currently, our work is focusing on feature-driven DSSA design and architecture customization. Besides domain modeling and design, we are also interested in feature-oriented component composition, requirements trace and change management. Our final goal is to provide all-life-long support for feature-oriented domain and application development, including methods and corresponding tool set.

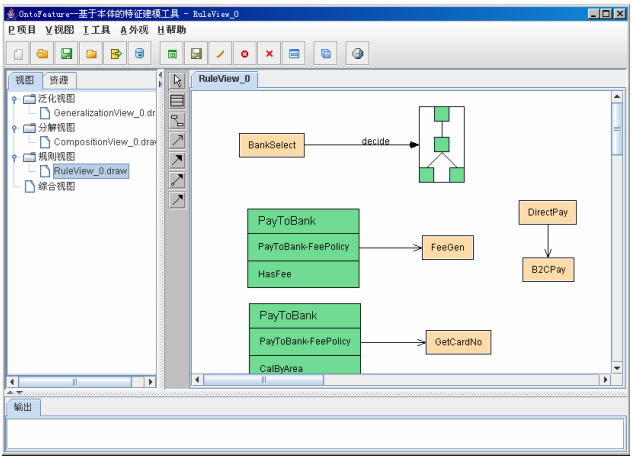

Fig. 7. Snapshot of OntoFeature tool

Acknowledgments. This work is supported by the National High Technology Development 863 Program of China under Grant No. 2004AA113030, 2005AA113120, the National Natural Science Foundation of China under Grant No. 60473061, the Science Technology Committee of Shanghai under Grant No. 04DZ15022. We would like to thank the members of the Feature Engineering group of our Software Engineering Lab, especially Jianhua Gu, Liwei Shen, Yiming Liu, etc.

References

1. Kang, Kyo C., Sholom G. Cohen, James A Hess, William E. Novak, and A. Spencer Peterson: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

2. V. Cechticky, A. Pasetti, O. Rohlik, and W. Schaufelberger: XML-Based Feature Modelling. ICSR 2004, LNCS 3107, p.101–114, 2004. Springer-Verlag.
3. Mike Mannion: Using First-Order Logic for Product Line Model Validation. SPLC2 2002, LNCS 2379, p.176–187, 2002. Springer-Verlag.
4. Schlick, M., and Hein, A.: Knowledge Engineering in Software Product Lines. European Conference on Artificial Intelligence (ECAI 2000), Workshop on Knowledge-Based Systems for Model-Based Engineering, August 22, 2000, Berlin, Germany.
5. Rubén Prieto-Díaz. A faceted approach to building ontologies. Proceedings of IEEE International Conference on Information Reuse and Integration (IRI 2003). 2003: 458~465.
6. Rosario Girardi, Carla Gomes de Faria. An ontology-based technique for the specification of domain and user models in multi-agent domain. CLEI electronic journal, Vol.7(1), 2004.
7. Vijayan Sugumaran, Veda C. Storey. A semantic-based approach to component retrieval. ACM SIGMIS Database, Vol.34:pages 8-24, 2003.
8. Sean Bechhofer, et al. Owl Web Ontology Language Reference", http://www.w3.org/TR/owl-ref/, 2004-02-10.
9. Carlton Reid Turner, Alfonso Fuggetta, Luigi Lavazza, Alexander L. Wolf: A conceptual basis for feature engineering. Journal of Systems and Software 49(1): 3-15 (1999).
10. Hong Mei, Wei Zhang, Fang Gu: A Feature Oriented Approach to Modeling and Reusing Requirements of Software Product Lines. Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC'03).
11. Kwanwoo Lee and Kyo C. Kang: Feature Dependency Analysis for Product Line Component Design. ICSR 2004, LNCS 3107, p. 69–85, 2004. Springer-Verlag.
12. 12.Wei Zhang, Hong Mei, Haiyan Zhao: A Feature-Oriented Approach to Modeling Requirements Dependencies. Proceedings of the 2005 13th IEEE International Conference on Requirements Engineering (RE'05).
13. Wei Zhang, Haiyan Zhao, Hong Mei: A Propositional Logic-Based Method for Verification of Feature Models. ICFEM 2004, LNCS 3308, p. 115–130, 2004. Springer-Verlag Berlin Heidelberg 2004.
14. Jena home. http://jena.sourceforge.net.
15. M.Mannion, H. Kaindl, J. Wheadon: Reusing Single System Requirements from Application Family Requirements. Proceedings of the 1999 International Conference on Software Engineering, ICSE1999.
16. Kyo C Kang , Sajoong Kim , Jaejoon Lee , et al. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. Annals of Software Engineering, 1998,5 :143～168.