Software & System Modeling (2006) 5(2): 172–186
DOI 10.1007/s10270-006-0004-1

SPECIAL ISSUE PAPER

Hong Mei · Wei Zhang · Haiyan Zhao

A metamodel for modeling system features and their refinement, constraint and interaction relationships

Received: 30 April 2005 / Revised: 10 June 2005 / Accepted: 10 June 2005 / Published online: 10 February 2006

Abstract This paper presents a metamodel for modeling system features and relationships between features. The underlying idea of this metamodel is to employ features as first-class entities in the problem space of software and to improve the customization of software by explicitly specifying both static and dynamic dependencies between system features. In this metamodel, features are organized as hierarchy structures by the refinement relationships, static dependencies between features are specified by the constraint relationships, and dynamic dependencies between features are captured by the interaction relationships. A first-order logic based method is proposed to formalize constraints and to verify constraints and customization. This paper also presents a framework for interaction classification, and an informal mapping between interactions and constraints through constraint semantics.

Keywords Feature model · Relationships between features · Refinement · Constraint · Interaction · Customization

1 Introduction

Software development is increasingly under the pressure of variable, various and evolving business requirements, as well as shortened time-to-market, rigorous product quality and growing competition.

One possible approach to resolving this problem is to produce general purpose software artifacts, followed by customization to accommodate different situations [1]. Considering this customization approach generally, elements in software artifacts should be cohesive enough, and dependencies between elements should be specified clearly. Furthermore, traceability between artifact elements at different stages should also be well established.

Software reuse is an instance of the customization approach. In software reuse, customization is limited to specific software domains. Customizable artifacts of specific software domains are produced in the development for reuse phase through commonality and variability analysis, and then customized/reused in the development with reuse phase according to the current reuse context.

Generally, artifacts produced in software development can be classified into two spaces, i.e. the problem space and the solution space. Artifacts in the solution space can be viewed as an implementation of artifacts in the problem space. Based on the above knowledge, two important problems relevant to customization-oriented artifact production can be deduced. One problem is how the problem space of software can be structured in a highly customizable way. The other problem is how this customizable structure of the problem space can be maintained in the solution space. Still another problem connected to the previous two problems is the verification problem of customized artifacts. That is, how the completeness and consistency of those customized artifacts can be checked conveniently and in an automated way.

In the software reuse approach, feature-oriented domain analysis methods [2–5] have been proposed to resolve the customization problem of the problem space. These methods treat features as the basic elements in the problem space, and use features, relationships (i.e. refinements and constraints) between features (called domain feature models) to structure the problem space. In the later development with reuse phase, domain feature models are customized into application feature models according to different reuse contexts. Constraints specified in the domain feature model then provide criteria to verify the completeness and consistency of those application feature models. In such a way, customization of the problem space is operated.

However, most of these feature-oriented methods only use two kinds of binary constraints (i.e. require and exclude) to capture constraints between features. The two kinds of

Communicated by Bernhard Schärtz and Ingolf Krüger

H. Mei (✉) · W. Zhang · H. Zhao
Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China
E-mail: meih@pku.edu.cn, {zhangw, zhhy}@sei.pku.edu.cn

binary constraints are too simple to describe complex constraints involving three or more features.

All these feature-oriented methods also lack effective measures to verify partial customized feature models. A real customization process often includes a series of phases (or binding times), in each of which, certain customizing resolutions are decided and a partial customized feature model is obtained. If these partial results in each phase can not be verified immediately, any error in the current phase will be spread implicitly to the later phases. In this sense, the difficulty of customization increases because of the inability to verify partial customized feature models.

All these feature-oriented domain analysis methods also lack a systematical way to propagate the customizable structure of the problem space to the solution space. Constraints only describe static dependencies between features, but tell little about how these features interact dynamically with each other at run-time. If the interaction context of each feature is not explicitly identified at a high level, the boundary of each feature will still remain unclear, which further causes such a problem that responsibilities of features are not separated clearly from each other in the solution space. The result is that the solution space loses a highly customizable structure corresponding to the structure of feature models.

Besides these feature-oriented methods in software reuse, many researches have brought features into the general software development process. FDD [6] is a feature-driven approach to iterative software development which takes features as basic increment units in and milestones of each iteration process. [7] presents a method to facilitate legacy system evolution by reengineering a feature implementation, which often scatters in source code, into a so-called fine-grained component. For a long time, features (also called services) have been used as the basic units to structure requirements in telecommunication systems [8, 9]. The DFC method [10] models features in telecommunication systems as components, and composes these components into pipe-and-filter architectures to analyze system behavior. Feature Engineering [11] even promotes features as "first-class objects throughout the software life cycle". However, all these methods lack a systematic way to model features and relationships between features at a high level. In most cases, the correct implementation of requirements declared by a feature not only depends on the feature itself, but often depends on the correct interactions with other features. Such a lack reduces, to a high degree, the understandability of the whole behavior of systems at a high level, which is also one of the causes of feature interaction problems [8].

This paper presents a metamodel for modeling system features and relationships between features. The underlying idea of this metamodel is to employ features as first-class entities in the problem space of software and to improve the customization of the problem space and the solution space, by explicitly specifying both static and dynamic dependencies between system features. Based on the idea, this metamodel defines three important relationships between features, namely refinement, constraint and interaction, and clarifies connections between the three relationships. The refinement provides a way to explore various system features from high levels to low-levels of abstraction, and to organize features as hierarchy structures. The constraint provides a way to specify static dependencies between features. And the interaction provides a way to express dynamic dependencies between features. Connections between these relationships offer the capability to identify one kind of relationships based on other kinds of already identified relationships. In addition, this metamodel defines three properties, namely satisfiability, usability and suitability, to verify both the constraints and the partially-customized feature models at any binding phase.

The idea of adopting features as the basic units in interactions is compatible with the "3C model" of reusable software components [12, 13]. In our approach, a feature's requirements correspond to the concept of a reusable component in the 3C model, a feature's interaction context (see Sect. 2.4) corresponds to the conceptual context of a reusable component, and a feature's specification corresponds to the content of a reusable component. The 3C model really points out the importance of feature interaction analysis. That is, the conceptual contexts of features are identified through feature interaction analysis, which further triggers the exploration of the operational contexts and the implementation contexts of features. In this sense, interactions between features can be treated as a bridge between the problem space and the solution space of software.

The main contribution of this paper includes: a metamodel for modeling features and relationships between them; a visual notation for modeling constraints; A first-order logic based method to constraint formalization and to both constraint and customization verification; a framework for binary interaction classification; an informal mapping between constraints and interactions through constraints semantics.

The rest of this paper is organized as follows. Section 2 gives an overview of the metamodel. Sections 3–5 presents the refinement, constraint and interaction relationships between features, respectively. A case study of this metamodel is presented in Sect. 6. Related work is discussed in Sect. 7. Section 8 concludes this paper with a summary and possible future work.

2 The metamodel

In this section, we give an overview of concepts involved in the metamodel, particularly of the two entity concepts (the feature and the resource container) and their attributes. Relationships between entities will be discussed in the following three sections.

2.1 An overview

This metamodel (Fig. 1) is based on four basic concepts: *Entity*, *Refinement*, *Constraint* and *Interaction*. All these

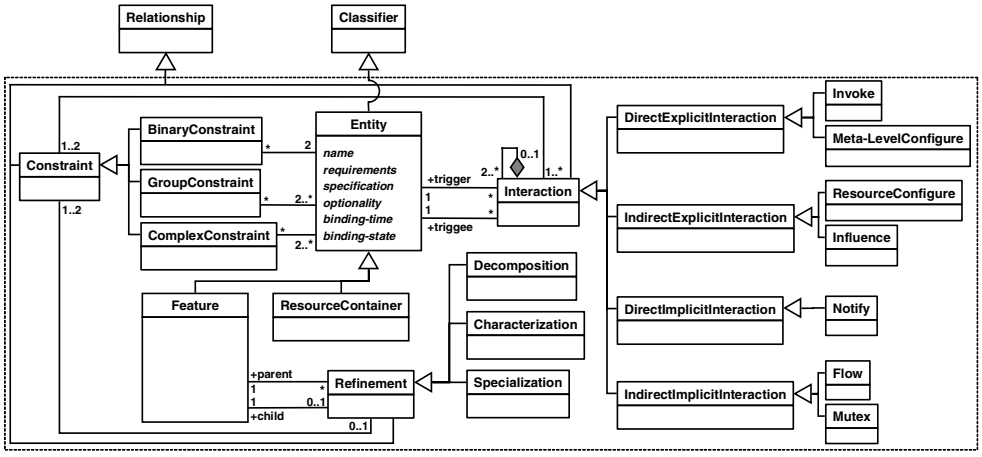


Fig. 1 The metamodel of feature models

concepts are subclasses of the two concepts in UML Core Package, namely Classifier and Relationship.

Entity is a subclass of Classifier and has two subclasses: *Feature* and *Resource Container*.

Refinement is a subclass of Relationship and has three subclasses: *Decomposition*, *Characterization* and *Specialization*. Through refinements, features at different levels of abstraction form hierarchy structures.

A constraint describes static dependencies between features and/or resource containers. This metamodel defines three kinds of constraint, namely *binary constraints*, *group constraints* and *complex constraints*.

An interaction describes dynamic dependencies between features and/or resource containers. This metamodel defines an interaction classification framework. All interactions are classified into four general kinds: *direct explicit interactions*, *direct implicit interactions*, *indirect explicit interactions* and *indirect implicit interactions*. Based on this classification, this metamodel further defines seven kinds of more specific interaction.

This metamodel also identifies connections between the three relationships (i.e. refinement, constraint and interaction). A refinement between features implies one or two constraints on features. Every interaction between entities also associates with one or two constraints. These associations between different relationships provide useful guidelines on feature modeling. That is, the identification of one kind of relationships will further trigger the exploration of other kinds of relationships.

2.2 Feature

Generally, the definition of a concept can be considered from two aspects: intension and extension. The intension describes the intrinsic qualities of a concept, while the extension characterizes the external embodiment. Many researches have given their definitions of features from either of the two aspects. For example, [7, 11, 14] focus much on the intension aspect, defining a feature as a set of related requirements, while [2, 15] emphasize the extension aspect, stating that a feature is a software-characteristic in the user or customer view.

In this paper, we do not introduce any novel idea about features, but just combine these two aspects and give the following definition of features.

> In intension, a feature is a cohesive set of individual requirements. In extension, a feature is a user/customer-visible characteristic of a software system.

2.3 Resource container

Resource containers contain resources which are produced, consumed or accessed by features in their execution. The essentials of resource containers are a way to structure resources used by features. With respect to the visibility of users, resource containers are classified into two categories: user-visible and user-invisible. One example of user-visible resource containers is the various email boxes in email-client software, i.e. received-box, copy-box, sending-box, and other user-defined boxes. User-visible resource containers are directly related to a set of user requirements, that is, the capabilities to classify, view or retrieve resources. In this sense, user-visible resource containers can be viewed as another kind of first-class objects in feature models.

User-invisible resource containers store resources which are invisible to users. This kind of resource container is not directly related to user requirements. Their existence depends on features. That is, if a user-invisible resource container is not accessed by any feature, this resource container will be useless in software. User-invisible resource containers can be viewed as second-class objects in feature models, since they are noteworthy only at the specification level. Separating resource containers from specification of features makes these features' specification more concentrating on the behavior description of themselves. Generally, we can see features as a kind of computation units, while resource containers as storage units.

In addition, resource containers often play as the medium of interaction between features. For instance, by employing resource containers as buffers, two features can form a producer-consumer interaction pattern.

The responsibilities assigned to resource containers is to passively accept features/users' requests for resource storing, querying or retrieving. Thus, whether a resource container can fulfill these responsibilities is independent of features.

2.4 Basic attributes of feature and resource container

Feature and Resource container also contain a set of attributes: *name*, *requirements*, *specification*, *optionality*, *binding-time* and *binding-state*.

Name is a short character string. Its main purpose is to facilitate the communication between stakeholders. All names in a feature model form a vocabulary of communication.

Requirements are a description of entities in user/customer view. For those user-invisible resource containers, the value of this attribute is null.

Specification is a description of entities' inner-behavior. Particularly, an entity's specification describes its running logic and its interaction with other entities. This metamodel doesn't restrict the format of requirements and specification. Informal, semi-formal (for instance, UML diagrams) or formal techniques can be employed according to the real context.

Optionality describes whether an entity has the chance to be removed when its parent entity (if has) has been bound. This attribute has two values: *mandatory* and *optional*. Removing entities from feature models should not violate constraints on entities (see Sect. 2.3). For example, if an entity is still depended by other entities which are not removed from the feature model, then the entity should not be removed.

Binding-time is an attribute related to optional entities. It describes a phase in the software life-cycle when an optional entity should either be bound or removed. Binding-time reflects the demand for system flexibility, and thus influences the complexity of system architectures. Typical binding-times include *reuse-time*, *compile-time*, *deploy-time*, *load-time*, and *run-time*. There may be different binding-time sets related to different kinds of software.

The attribute *binding-state* has three values: *bound*, *removed* and *undecided*. A *bound* feature means that if its trigger conditions (for instance, users' requests or user indirect-caused events) and pre-conditions are satisfied, its requirements will be satisfied. Obviously, if a feature is in the bound state, all the resource containers it depends on will also be in the bound state. A *bound* resource container means that other entities can correctly access it. A *removed* entity means that it will never be bound again. If an entity is removed, all entities depended on it will also never be in the bound state. An *undecided* entity means that it is currently not in the bound state, but still has the chance to be bound or removed in later binding-times.

According to the three values of the binding-state, entities in a feature model can be partitioned into three sets: *BFSet*, *UFSet* and *RFSet*. The definition of these three set is:

- $BFSet = \{e \mid e.binding - state = bound\}$;
- $UFSet = \{e \mid e.binding - state = undecided\}$;
- $RFSet = \{e \mid e.binding - state = removed\}$;

For an undecided entity, it will be either bound or removed in certain later binding time. That is, this entity will finally be moved from *UFSet* to *BFSet* or *RFSet*.

3 Refinement

Refinements are a kind of binary relationships between features, which integrate features at different levels of abstraction into hierarchy structures. Hierarchy structures provide an effective way to describe complex systems, since it is easier to understand a complex system from general to specific and from high levels to low levels.

Refinements can further be classified into three more concrete subclasses: *decomposition*, *characterization*, and *specialization*.

Refining a feature into its constituent features is called decomposition [12]. For example, the feature *edit* in many software applications is often decomposed into three sub-features: *copy*, *paste* and *delete* (see Fig. 2).

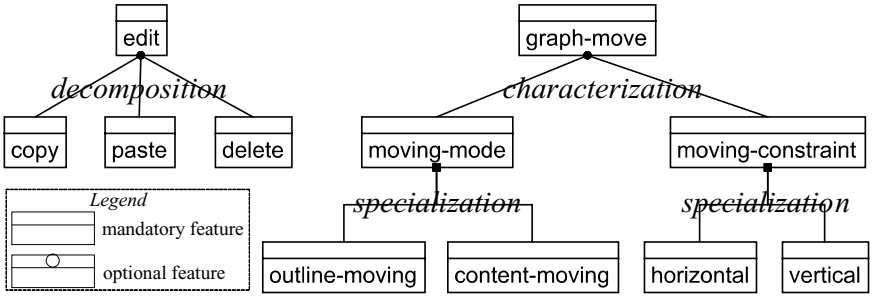


Fig. 2 Example of refinements

Table 1 Roles in refinements

Refinement	Parent-Role	Child-Role
Decomposition	Whole	Part
Characterization	Entity	Attribute
Specialization	General-Entity	Specialized-Entity

Refining a feature by identifying its attribute features is called characterization. For example, in graph-editor applications, feature *graph-move* can be characterized by two attribute features: *moving-mode* and *moving-constraint*.

Refining a general feature into a feature incorporating further details is called specialization [12]. For instance, feature *moving-mode* can be specialized by more concrete features: *outline-moving* and *content-moving*. Specialization is often used to represent a set of variants of a general feature. A general feature is also called a variation point feature (*vp*-feature) in [4].

The three kinds of refinement can be differentiated by roles of features involved in them. Suppose a and b are two features involved in a refinement, we call the feature at a higher level of abstraction the parent, and the other feature the child. Table 1 shows the different roles played by parents and children in different kinds of refinement.

Features with different levels of abstraction form hierarchy structures through refinement relationships. More strictly, this metamodel limits this hierarchy structure to be one or more feature trees. That is, a feature is either a root feature or refined exactly from one feature. This limitation contributes to the simplicity and understandability of the refinement view of feature models. [16] and [17] present guidelines on maintaining tree structures in feature modeling process and transforming general structures into tree structures.

4 Constraint

Constraints are a kind of static dependencies among features, and more strictly among binding-states of features. It provides a way to verify the results of requirements customization and release planning [18, 19]. Only those results that do not violate constraints on features can be treated as candidates of valid requirements subsets or releases.

4.1 Formal definition of constraint

We have identified three important constraint categories, namely *binary constraints*, *group constraints*, and *complex constraints*.

Binary constraints are constraints on the binding-states of two features. Table 2 shows three kinds of binary constraint and their formal definitions.

Group constraints are constraints on a group of features. These constraints are extensions of binary constraints.

Table 2 Binary constraints

Binary constraint	Definition
requires (a, b: Feature)	bound (a) $\rightarrow$ bound (b)
m-requires (a, b: Feature)	requires (a, b) $\wedge$ requires (b, a)
excludes (a, b: Feature)	$\neg$(bound (a) $\wedge$ bound (b))
Where: bound (a: Feature)	$=_{\text{def}}$ (a. binding-state = bound);

Table 3 Group constraints

Group Constraint	Definition
mutex-group (P: set Feature)	$\forall\, a \in P, b \in P, a \neq b \bullet$ excludes (a, b)
all-group (P: set Feature)	$\forall\, a \in P, b \in P \bullet$ m-requires (a, b)
none-group (P: set Feature)	true

Table 4 Group constraints

Group Predicate	Definition
single-bound (P: set Feature)	$\exists_{\text{one}}\, a \in P \bullet$ bound (a)
all-bound (P: set Feature)	$\forall\, a \in P \bullet$ bound (a)
multi-bound (P: set Feature)	$\exists_{\text{some}}\, a \in P \bullet$ bound (a)
no-bound (P: set Feature)	$\forall\, a \in P \bullet \neg$bound ($a$)

Table 5 Complex constraints

Complex Constraint	Definition
requires (x, y: Group-Predicate)	$x \rightarrow y$
m-requires (x, y: Group-Predicate)	$(x \rightarrow y) \wedge (y \rightarrow x)$
excludes (x, y: Group-Predicate)	$\neg(x \wedge y)$

Table 3 shows three kinds of group constraint and their formal definitions.

Before talking about complex constraints, we first introduce four important predicates on a group of features, namely *group predicates*, which extend the parameter of the predicate *bound*(*a*: *Feature*) (see Table 2) to a feature set. Their formal definitions are given in Table 4.

Complex constraints are constraints between two feature sets, which extend the parameters of binary constraints to group predicates. Table 5 shows complex constraints and their formal definitions.

For convenience of constraint modeling, we develop a visual constraint notation (Fig. 3) to represent all aforementioned constraints in a graphical manner. This notation will be used in the remainder of this paper to visualize constraints on features.

4.2 Refinement-imposed constraints

Refinements implicitly impose constraints on features. We call these constraints refinement-imposed constraints. These constraints are caused by the following rules when binding or unbinding features:

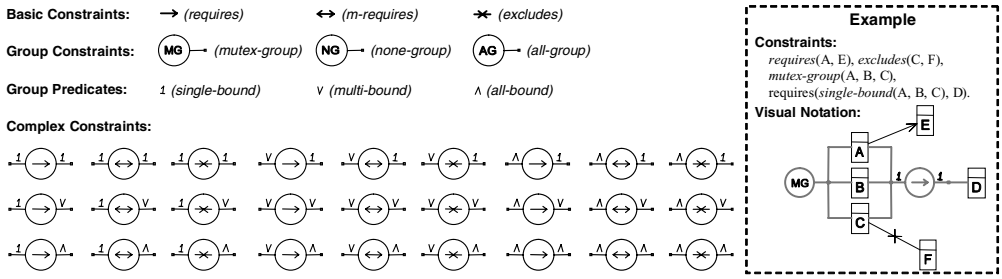
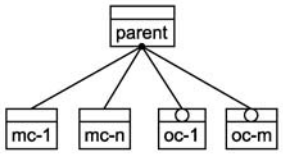
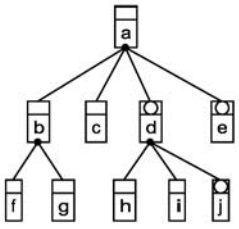


Fig. 3 Visual constraint notation

Table 6 Refinement imposed constraints

Refinement Scenario	Implied Constraints
	*all-group*(parent, mc-1, mc-n), *multi-bound*(oc-1, oc-m) *requires* parent.
	*all-group*(a, b, c, f, g), *all-group*(d, h, i), *multi-bound*(d, e) *requires* a, j *requires* d.

A. A pre-condition of binding a feature is that its parent must have been bound.
B. A post-condition of binding a feature is that all its mandatory children must also be bound.
C. A pre-condition of unbinding a feature is that all its children must have been unbound.
D. A post-condition of unbinding a mandatory feature is that its parent must also be unbound.

These rules essentially reflect constraints imposed by refinements. Table 6 shows two examples of refinement scenarios, and their implied constraints on features. By tool support, those refinement imposed constraints can be automatically extracted for further analysis of them.

4.3 Constraint and customization verification

Constraint verification here means to check the consistency of constraints. Customization verification means to verify binding resolutions in customization. Customization is a process of changing *undecided* entities' binding-states to *bound* or *removed*, which may occur at each binding time.

We propose three properties (i.e. *satisfiability*, *usability* and *suitability*) to solve both the constraint and the customization verification problem.

Suppose $C1, C2, \ldots, Cn$ is the set of logic sentences capturing all constraints in a feature model and *CRSet* denotes the set of all possible customizing resolutions to features in *UFSet*. After customization at each binding time, entities in *UFSet* should satisfy the following three properties to ensure the rationality of binding resolutions in the current customization.

– *Satisfiability:*

$$\exists I \subset CRSet,\ I| = \cap_{i=1,\ldots,n} C_i$$

There exists at least one set of customizing resolutions to all features in *UFSet*, which can satisfy all constraints in $C1, C2, \ldots, Cn$.

– *Usability*:

$$\forall f \in UFSet,\ \exists I \subset CRSet,\ I| = (\cap_{i=1,\ldots,n} C_i) \cap f$$

To every feature f in *UFSet*, suppose f is bound, there exists at least one set of customizing resolutions to all other features in *UFSet*, which can satisfy all constraints in $C1, C2, \ldots, Cn$.

– *Suitability*:

$$\forall f \in UFSet,\ \exists I \subset CRSet,\ I| = (\cap_{i=1,\ldots,n} C_i) \cap (\neg f)$$

To every feature f in *UFSet*, suppose f is removed, there exists at least one set of customizing resolutions to all other features in *UFSet*, which can satisfy all constraints in $C1, C2, \ldots, Cn$.

The *satisfiability* ensures the consistency of constraints and the consistency of binding resolutions in the current customization. If this property is not satisfied, constraints on entities or binding resolutions in the current customization should be revised to eliminate inconsistencies.

For example, suppose a, b and c are three features in a partially-customized feature model. The current feature partition is:

$BFSet = \{a\}, UFSet = \{b, c\}, RFSet = \{\}.$

Constraints on the three features are:

$C1$: *a require b*; $C2$: *a require c*; *and* $C3$: *b exclude c*.

We can find that such a partially-customized feature model violates the *satisfiability*, because there exists no customizing resolution on b and c, which can satisfy $C1$, $C2$ and $C3$ at the same time. The possible cause of the violation may be that $C1$ or $C2$ or $C3$ should not exist.

The *usability* ensures that every entity in *UFSet* has the possibility of being bound in some future binding time. If this property is not satisfied, it means that there are one or more entities that will never have chances to be bound after the current binding time. These violations can be eliminated by putting these entities into *RFSet*, or by revising those binding resolutions in the current customization.

For example, suppose d and e are two features in a partially-customized feature model. The current feature partition is:

$$BFSet = \{\}, UFSet = \{d\}, RFSet = \{e\}.$$

A constraint on the two features is:

$$C4 : d \ require \ e.$$

We can find that such a partially-customized feature model violates the *usability*, because d has no chance to be bound; otherwise, the $C4$ will not be satisfied. The possible cause of the violation may be that $C4$ should not exist, or the reuser forgot to remove d, or e should not be removed.

The *suitability* ensures that every entity in *UFSet* has the possibility of being removed in some future binding time. If this property is not satisfied, it means that there are one or more entities that will not have the chances to be removed after the current binding time. These violations can be eliminated by putting these entities into *BFSet*, or by revising those binding resolutions in the current customization.

For example, suppose f and g are two features in a partially-customized feature model. The current feature partition is:

$$BFSet = \{f\}, UFSet = \{g\}, RFSet = \{\}.$$

A constraint on the two features is:

$$C5 : f \ require \ g.$$

We can find that such a partially-customized feature model violates the *suitability*, because g has no chance to be removed; otherwise, the $C5$ will not be satisfied. The possible cause of the violation may be that $C5$ should not exist, or the reuser forgot to bind g, or f should not be bound.

As we have seen, the most important value of the three properties is to help locate possible mistakes in constraints and in customizing resolutions. Currently, we are employing the model checker SMV [20] to automate the checking of these three properties and to further locate possible errors.

4.4 Constraint semantics

Constraints provide necessary criteria, by exposing customization rules, to help retrieve information from domain feature models. However, what we are interested in is the origins of constraints which are called *constraint semantics* in this paper. We think that the constraint semantics reveals the nature of constraints, and provides more valuable information than constraints themselves to guide the further analysis of dynamic relationships between system features. In this subsection, we give a more detailed classification to the basic constraints *require* and *exclude* according to their different semantics.

This metamodel defines four kinds of different semantics that a concrete require constraint may manifest: *strong necessity require*, *weak necessity require*, *availability require*, and *usability require*.

Strong Necessity Require ($require_{ns}$) For any two entities A and B, A $require_{ns}$ B means that in any case for A to be correctly executed, it will depend on whether B can be correctly executed. The $require_{ns}$ constraint often exists on features and their accessible resource containers. One example of such constraint in the email client software is "*email-filter* $require_{ns}$ *filtering-rules*", in which the feature *email-filter* depends on the resource container *filtering-rules* to provide the necessary data for its correct execution.

Weak Necessity Require ($require_{nw}$) For any two entities A and B, A $require_{nw}$ B means that in some cases for A to be correctly executed, it will depend on whether B can be correctly executed, while in other cases, A's correct execution may have no relation with B. One example of such constraint is "*email-filter* $require_{nw}$ *email-decryption*", in which the feature *email-filter* depends on the feature *email-decryption* only when a received email is encrypted and *email-filter* needs to filter this email according to its content.

Availability Require ($require_{a}$) For any two entities A and B, A $require_{a}$ B means that whether A can be executed depends on whether B have been executed. That is, if B is not bound to the software, A will never be available to users. Those features whose execution is triggered by other features often involve such constraints. One example is the constraint "*email-copy-auto-saving* $require_{a}$ *email-sending*", in which the feature *email-copy-auto-saving* depends on the feature *email-sending* triggering its execution when an email has been sent out. After the execution of *email-copy-auto-saving*, the copy of a sent-out email will be saved to the *copy-box*.

Usability Require ($require_{u}$) For any two entities A and B, A $require_{u}$ B means that A's usability depends on B's execution. That is, if B is not bound to the software, then A's execution is useless to users although it can be successfully executed. One example of such constraint is "*email-filter-configurator* $require_{u}$ *email-filter*", in which the feature *email-filter-configurator* modifies filter rules stored in the resource container *filtering-rules*, while the

feature *email-filter* read rules from it when executing. Although *email-filter-configurator* may still behave correctly without *email-filter*, its execution is useless in this case.

For the exclude constraint, this metamodel defines two kinds of different semantics that a concrete exclude constraint may manifest: s*ingle-value exclude* and *conflict exclude*.

Single-Value Exclude (*exclude*$_s$) This kind of semantics often exists on features which are specialized from a same general feature. If we treat a general feature as a variation dimension and its specialized features as values in this dimension, then the exclude constraint on any two values in this dimension means that at most one value can be in the bound state at any time. The semantics of an exclude constraint on two features in a same dimension is called *single-value exclude* in this paper. For instance, the two features *outline-moving* and *content-moving* on the *moving-mode* dimension have the *single-value exclude* semantics (i.e. *outline-moving exclude*$_s$ *content-moving*) (Fig. 2). Single value dimensions are also referred to as *single adaptors* in [21].

Conflict Exclude (*exclude*$_c$) There are also exclude constraints that involve features not in a same variation dimension or even not being values in a dimension. The semantics of these exclude constraints are called *conflict exclude* in this paper. Binding two conflict exclude features in a same context will cause undesirable feature interactions. An example of conflict exclude exists between the two features *caller-ID* and *block-caller-ID* in the telecommunication software [22]. If the recipient binds *caller-ID*, and the originator binds *block-caller-ID*, there will be no software behavior that could satisfy both of the two features.

It should be pointed out that the phrase "*an entity's correct execution*" which we have used above have particular meanings especially when this entity denotes a feature. The correct execution of a feature means that this feature's behavior complies with both its requirements and its specification. A feature's behavior complying with its specification does not exactly mean its behavior complying with its declared requirements, because for a feature to satisfy its requirements, it often has to depend on other features or resource containers to behave correctly. For example, requirements of the feature *email-copy-auto-saving* may be that "*after an email is sent out, a copy of this email will be saved in the copy-box*", and its specification may be that "*when it receives a message, which indicates that an email has been sent out, from the feature email-sending, a copy of the email will be put into the copy-box*". Then, even if *email-sending* sends a wrong message to it, *email-copy-auto-saving*'s behavior will still comply with its specification. But in that case its behavior does not comply with its requirements. Vice versa, a feature's behavior complying with its requirements also doesn't exactly mean that its behavior complying with its specification, because sometimes not all parts of a feature's specification relate to its requirements. That is, some parts of its specification only relate to other features' requirements. For instance, one part of *email-sending*'s specification may be that "*after an email is sent out, send a message to the listeners* (e.g. *email-copy-auto-saving*)". Then, even if this part is not correctly implemented, *email-sending*'s behavior will still comply with its requirements.

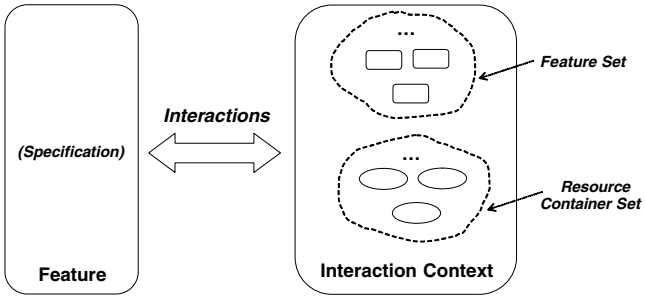


Fig. 4 Feature and its interaction context

The reason for defining features' correct execution according to the two dimensions of complying with both its requirements and its specification is that such a definition provides a suitable view to identify and analyze interactions between features. The scattering nature of requirements is the origin of interactions, and specification is the way to implement requirements.

5 Interaction

Speaking generally, any interaction involves at least two entities. Those interactions that involve three or more entities can often be decomposed into a set of interactions between two entities. This metamodel only focuses on binary interactions. In the view of an individual entity, all entities that interact with it constitute its interaction context. In this metamodel, a feature's interaction context consists of a set of features and a set of resource containers (Fig. 4).

This metamodel distinguishes different roles played by entities involved in an interaction. The entity which triggers the interaction is called *trigger* and the other entity is *triggee*. For example, in the interaction between *email-sending* and *email-copy-auto-saving*, the former is the trigger since it is the sender of messages, and the latter is the triggee. It should be noticed that trigger and triggee are only roles played by entities. An entity may play as trigger in one interaction while playing as triggee in other interactions.

This metamodel classifies interactions according to two dimensions. The first dimension is whether the trigger interacts with the trigger directly. There are two values in this dimension: *direct* and *indirect*. The *indirect* value means two features interact with each through a resource container. The *direct* value means two features interact with each other without any resource container involved. The second dimension is whether there is a constraint *requires(trigger, triggee)* between the trigger and the triggee. There are also two values in this dimension: *explicit* and *implicit*. The *explicit* value means there is such a constraint, and the *implicit*

value means the reverse. Consequently, there are totally four different combinations of values in the two dimensions: *direct explicit interactions*, *direct implicit interactions*, *indirect explicit interactions* and *indirect implicit interactions*. This metamodel employs these four combinations as an *interaction classification framework*.

If only considering the factor of who plays the trigger role in an interaction, there are theoretically four kinds of interaction in each aforementioned interaction class:

1. Interactions only between features.
2. Interactions between features and resource containers (features play the trigger roles).
3. Interactions between features and resource containers (resource containers play the trigger roles).
4. Interactions only between resource containers.

The last kind of interaction essentially reflects the way to structure information in software systems, such as those methods of database design. Because our interests mainly focus on interactions involving features, this kind of interaction will not be discussed in the remainder of this paper. However, not all the other three kinds of interaction make sense or are of enough value in each interaction class. In the rest of this subsection, we will address seven typical interactions (see Fig. 1) which we have identified according to the interaction classification introduced above.

5.1 Direct explicit interactions

In a direct explicit interaction, the trigger interacts with the trigger directly, and depends on triggee.

Invoke interactions One commonly occurred instance of this class is the *invoke interactions*, in which the invokers play the trigger role.

Figure 5 shows an invoke interaction between *address-auto-retrieving* and *address-adding* in the email client software, and the constraints involving these two features (corresponding constraint semantics is attached to the lower of the two end-points of constraint notation). The requirements of *address-auto-retrieving* is that "*after an email is sent out, the receiver's address of this email will be added to the address book*", and the requirements of *address-adding* is that "*add the address that user requires to the address book*". From this example, we can see that in an invoke interaction, not only does the trigger have the *necessity require semantics* to the triggee, but the triggee has the *availability require semantics* to the trigger.

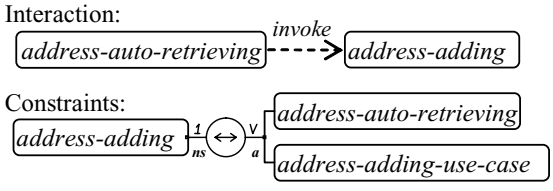


Fig. 5 Invoke interactions in the email client domain

Those invoke interactions between features and resource containers, in which features play the trigger roles, essentially reflect the features' control on resources. By such interactions, two features can interact indirectly with each other. Generally, these interactions can be further classified into *modify invoke* and *read invoke* according to whether the states of resource containers are changed by features, or *produce invoke* and *consume invoke* according to the producer or consumer role played by features. An example of such interactions can be found in Fig. 8 (see details in Sect. 5.2).

It seems that those invoke interactions between features and resource containers, in which resource containers play the trigger roles, are of little value in practice. Since a resource container' main responsibilities are passively accepting features' requests for resource storing, querying and retrieving, they have no reason to rely on features to achieve these responsibilities. However, if resource containers are assigned with more responsibilities than their nature, such interactions may also exist.

Meta-level configure interactions Another instance of direct explicit interactions we have identified is the *meta-level configure interactions* between features. Complex software systems often have the capability to modify their behavior ([23] calls such systems *self-modifying systems*). From this view, [9] makes a distinction between core features and modulating features. One kind of modulating feature is those features that can change other features' binding states. Interactions between features in which the triggees' binding states are changed by the triggers are called meta-level configure interactions in this paper. One example of meta-level configure interactions is interactions between features *meeting-state-modulator*, *ringer* and *vibrator* in the mobile-phone client software (Fig. 6), in which when users trigger *meeting-state-modulator*, it will configure *vibrator* to the bound state and *ringer* to the undecided state. From this example, we can identify that in a meta-level configure interaction, the trigger has the *availability semantics* to the triggee.

5.2 Direct implicit interactions

In a direct implicit interaction, the trigger interacts with the trigger directly, but doesn't depend on triggee.

Notify interactions One instance of this class is the *notify interactions*. The feature which is the source of notification

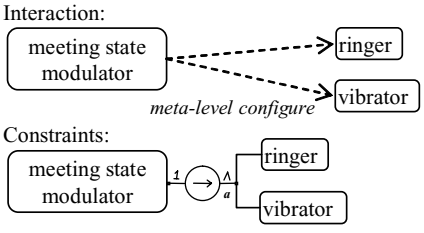


Fig. 6 Meta-level configure interactions in the mobile-phone client domain

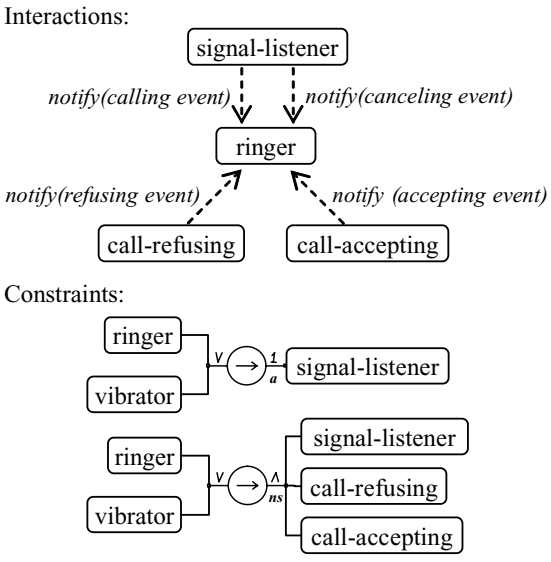


Fig. 7 Notify interactions in the mobile-phone client domain

plays the trigger role in a notify interaction, and the triggee depend on the trigger to notify proper information.

Figure 7 shows an example of notify interactions between features in the mobile-phone client software. In interactions between these features, *ringer* is triggered by the notification of the *calling event* from *signal-listener*, hence *ringer* has the *availability require semantics* to the *signal-listener*. After *ringer* is triggered, it can only shut itself down when one of the three events (*canceling event*, *refusing event* and *accepting event*) happens. For *ringer* to behave correctly, all of the three features *signal-listener*, *call-refusing* and *call-accepting* should behave correctly, that is, they should not notify *ringer* of wrong events. Therefore, *ringer* has the *strong necessity require semantics* to all of the three features. Interactions and constraints involving *vibrator* can also be identified in the similar way.

There are also notify interactions between features and resource containers in which resource containers play the trigger roles. In these interactions, resource containers notify features that their states have satisfied certain conditions. Figure 8 shows a well-known example of such interactions between *model* and *views* in the MVC pattern, in which *model* notifies *views* when its state has changed. In the MVC pattern, *controller* can be treated as one or more features that provide controlling capability to users, *model* as a resource container that stores data, and *views* as features that provide data viewing capability with different styles to users. From this example, we can also see that *controller* and *views* interact indirectly with each other through a set of direct interactions between features and resource containers.

Those notify interactions between features and resource containers in which features play the trigger roles seem unnecessary in the situation that features' specification and resource containers' specification are clearly separated. The reason is just as we have explained in Sect. 5.1: a pure resource container needn't depend on features to implement its specification.

5.3 Indirect explicit interactions

In an indirect explicit interaction, the trigger interacts with the triggee through a resource container, and depends on the triggee.

Resource configure interactions One instance of this interaction class is the *resource configure interactions* between features, in which the triggers configure the triggees' behavior by modifying resources accessed by the triggees. An example of resource configure interactions is the interaction between *email-filter* and *email-filter-configurator* in the email client software, in which *email-filter-configurator* modifies the resource container *filtering-rules*, and *email-filter* read information from *filtering-rules* (see Fig. 9). From this example, we can see that a resource configure interaction is actually implemented by a set of direct interactions between features and resource containers, and that the trigger has the *usability require semantics* to the triggee.

Influence interactions Another instance of indirect explicit interactions is the *influence interactions*, in which triggers influence triggees' behavior by imposing additional responsibilities on them, and these additional responsibilities further cause indirect interactions between triggers and triggees. An example of such interactions can be found between the three features *copy*, *paste*, *delete* and the feature *un*/*re-do* in many software systems (Fig. 10). The additional responsibility imposed by *un*/*re-do* is that each of the three editing features should record the un/re-doing information of their execution to a resource container. For *un*/*re-do* behaving correctly, these responsibilities must be correctly completed. Hence, *un*/*re-do* has the *strong necessity require semantics* to these editing features. Furthermore, *un*/*re-do* also

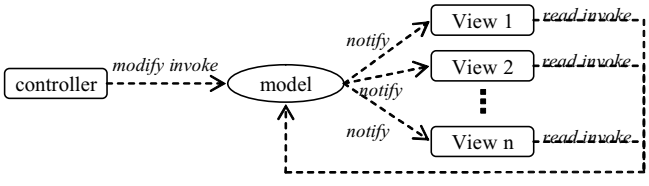


Fig. 8 Interactions in the MVC pattern

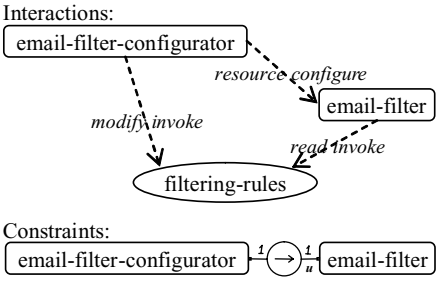


Fig. 9 Resource configure interaction in the email client domain

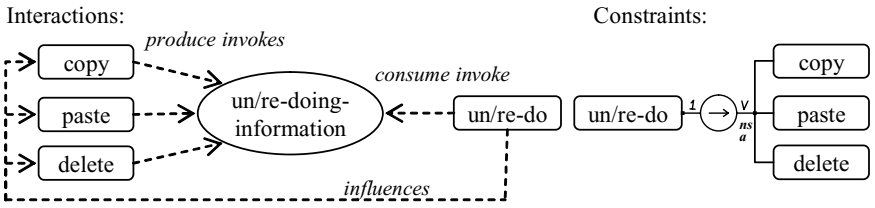


Fig. 10 Influence interactions

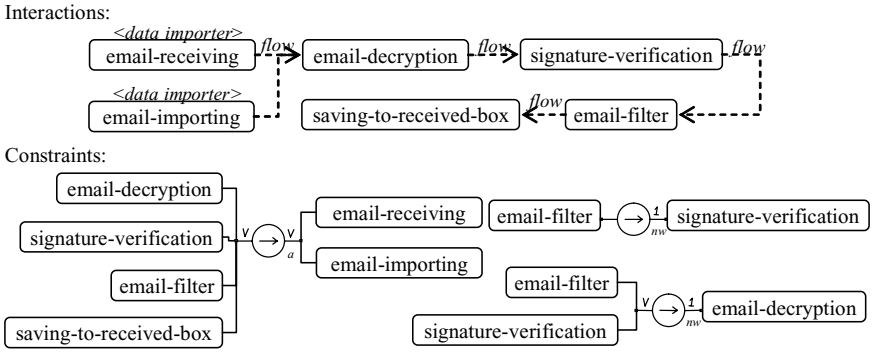


Fig. 11 Feature flow in the email client domain

has the *availability require semantics* to these editing features, since if all editing features are removed, *un/re-do* will have no chance to be executed.

5.4 Indirect implicit interactions

In an indirect implicit interaction, the trigger interacts with the triggee through a resource container, but doesn't depend on the triggee.

Flow interactions One instance of this interaction class is the *flow interactions* between features, in which data are processed by the triggers and then flow to the triggees for further processing. Flow interactions reflect the possible data flows in software. The *require* constraint on features in a flow interaction often determines which feature plays the trigger role. Suppose that the constraint "A *require* B" exists, in which A and B are two features involved in a flow interaction. Three scenarios may occur in this flow interaction:

1. A depends on B's processing of data to a proper format so that these data can be further processed by A. In this scenario, the constraint semantics between A and B is "A $require_{nw}$ B" or "A $require_{ns}$ B".
2. B is a data importer. That is, the requirements declared by B are to receive data from the environment of software. In this scenario, the constraint semantics between A and B is "A $require_{a}$ B", because if B is removed, A will never have the chance to be executed.
3. A is a data exporter. That is, the requirements declared by A are to send data to the environment of software. In this scenario, the constraint semantics between A and B is "B $require_{u}$ A", since if A is removed, previous processing on data will become useless.

Obviously, the only possible role assignment in these three scenarios is B plays the trigger role and A the triggee role.

Two independent features may incidentally exhibit a data flow relation because of implementation reasons, but we don't treat this as a flow interaction, since these two features are concurrent in their nature, although this concurrency may be sequentially implemented in the final software.

As trigger or triggee is only a role played by features, a feature which plays the triggee role in one flow interaction may play the trigger role in another one. By this way, a set of related flow interactions can form a pipe-and-filter architectural style [24], which we called a *feature flow*. As in a flow interaction, the *require* constraint often plays the *partial order* on features in a feature flow. Thus, by exploring constraints among a set of features in which data flows may exist, the sequence or concurrency in feature flows can be systematically identified or verified. Figure 11 shows a feature flow existing in the email client domain in which a set of features are composed by certain order to collaboratively process received/imported emails. More information about the pipe-and-filter architectural style or examples of feature flows can be found in [10, 24, 25].

Mutex interactions Another instance of indirect implicit interactions is the interaction between two excluded features which is called a *mutex interaction* in this paper. For software to behavior correctly, there must be a third party that plays the medium role between any two run-time excluded features. For example, a single value variation dimension may be accompanied by a configurator which ensures that at most one value in this dimension can be in the bound state, or a coordinator may exist between two *conflict excluded* features to resolve the conflict between them.

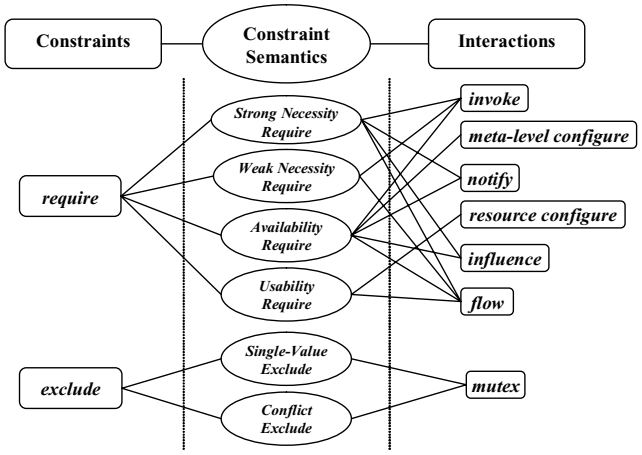


Fig. 12 Mapping between constraints and interactions through constraints semantics

5.5 Mapping between constraints and interactions

Based on the analysis of interactions and their connected constraints described above, we can thus create a informal mapping between interactions and constraints. Figure 12 shows the mapping between two basic constraints and seven typical interactions through six kinds of constraint semantics.

This mapping provides guidelines for the identification of possible interactions between features according to constraints on features. After the problem space of a software domain is systematically explored and the feature model is created as in most feature-oriented domain analysis methods (feature models in these methods only contain information about refinements and constraints), reusable assets producers can then analyze the semantics implied by constraints, and select suitable interaction types which can best embody the constraint semantics.

Furthermore, this mapping also helps elicit constraints on features. In some cases, interactions between features are more obvious than constraints on features (for example, in our experience, we often first identify the *notify* interaction between two features, and then realize the *require* constraint on them). Then, analysts can use these identified interactions as the input to find possible constraints through constraint semantics analysis. This further improves the customization of feature models.

6 Case study

As a case study, we applied our approach to the email client software, which is a simple and mature software domain.

Figures 13–15 show part of the modeling result. Information about features and refinements between features is shown in Fig. 13, information about constraints on features is shown in Fig. 14, and information about interactions between features is shown in Fig. 15. Because of space limitation, some attributes of features/resource containers, such as *requirements*, *specification*, *binding-time*, are not given in this paper. Optional features are marked with cycle symbols. In Fig. 14, only those *analyst-imposed constraints* on features are given, since those *refinement-imposed constraints* can be automatically retrieved from refinements between features.

From this case study, we can see that by explicitly modeling system features and their refinement, constraint and interaction relationships, software can be specified at a high abstract level in an easily understandable manner for both customers/users and software developers. Customers/users can know what the software provides for them from the feature refinement view, and then customize the software according to their real requirements under the guidance of the feature constraint view. While software developers can know how these features interact with each others at run-time from the feature interaction view, and further implement these features and enable interactions between features. Since software is implemented in a feature-oriented way, artifacts in the solution space maintain a same customizable structure

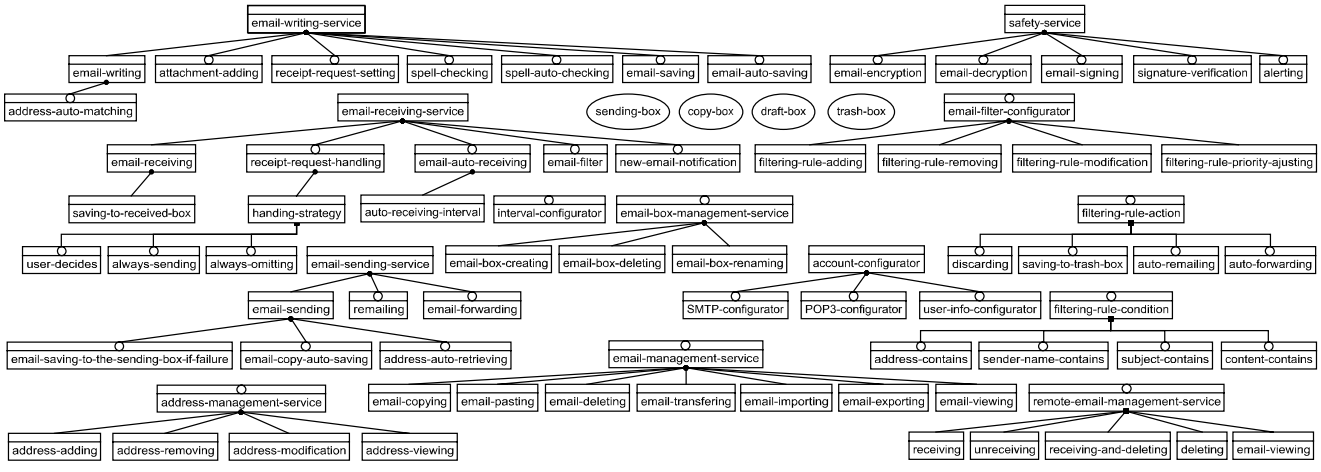


Fig. 13 Feature refinement view of the email client software

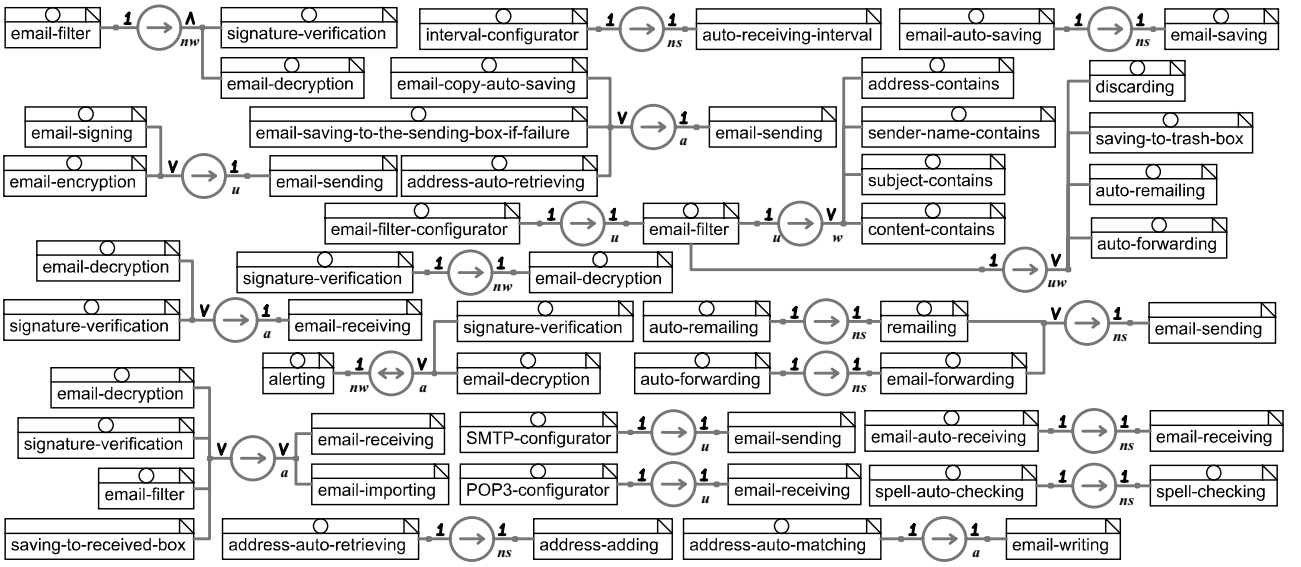


Fig. 14 Feature constraint view of the email client software

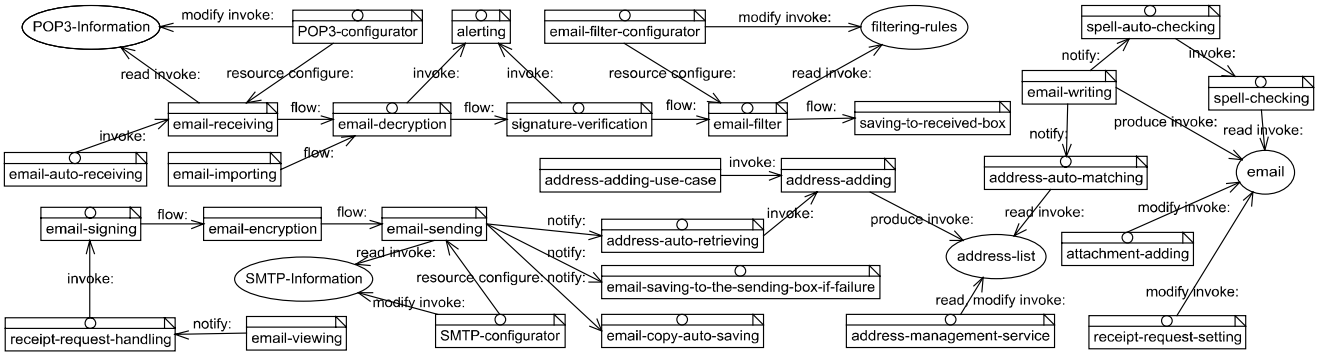


Fig. 15 Feature interaction view of the email client software

as artifacts in the problem space (i.e. feature models), which improve the alignment between the problem space and the solution space of software.

7 Related work

Our work on constraint based feature interaction analysis was inspired in part by [17], which focuses on extracting feature interactions and constraints from legacy systems. While [17] tries to separate interactions and constraints from feature refinement structures (the feature tree view) and model them in the so-called *feature interaction and dependency view*, our approach further makes a clear distinction between interactions and constraints. Thus, there are three types of view in our approach: *feature tree view*, *feature constraint view* and *feature interaction view*. It should be noticed those feature models used in current feature-oriented methods [2–5] only cover information in the former two views of our approach. The reason why we distinguish between constraints and interactions in our approach is motivated by the following two considerations. The main consideration is that the two kinds of concepts have different characteristics and functions in software development: constraints are static and can be employed to resolve the artifact retrieval problem of software reuse, or facilitate customization of software artifacts; while interactions are dynamic in their nature and facilitate the analysis of features' running logic, and thus provide a starting point to specify and implement features in the solution space. Another consideration is that most current feature-oriented methods have contained information about constraints in their feature models, and thus we want to make our approach as a compatible extension to these methods.

Biddle et al. [26] present a dependency-based conceptual model for understanding the nature of software reusability at the language level. The concept *dependency* in this model is similar to the constraint in feature models. While this model tries to reveal the relation between reusability and those dependencies caused by language constructs, our work focus on analyzing constraints imposed by domain nature at the requirement level. This model can be viewed as the result of a "bottom-up" approach to software reuse, while our

work follows a "top-town" approach. However, there is no conflict between this model and our approach. They can be two complementary approaches in practice, i.e. employing this model to guide the way to implement features so that only those requirement level constraints are embodied in final software.

Sutcliffe et al. [27] describe a different approach to requirement reuse from the domain analysis approach (e.g. most current feature-oriented domain analysis methods). Unlike the domain analysis approach of that only concerned reusable assets for specific software domains, this approach implicitly supposes that there is a finite set of *domain abstractions* that covers the nature of requirements of most domains, and thus aims to identify and specify these domain abstractions and automatically retrieve them when engineering requirements for individual applications. Another instance of this approach is proposed by Jackson [28], which tries to decompose the problem space of software into a set of commonly occurred *problem frames*. This kind of approach often bases on humans' analogy capability to identify and reuse those domain abstractions or problem frames, and tries to resolve the software reuse problem at a more abstract level than the domain analysis approach. One question we are exploring is whether these domain abstractions or problem frames can be modeled in the form of interactions between features, since in our work we have identified a set of commonly occurred feature interaction abstractions. This may be a possible combination of these two distinct kinds of approach. Besides the difference, a similarity between [27, 28] and our work is that all of them agree on the idea of separating information entities from system behavior specifications, i.e. the *information system models* in [27], the *lexical domains* in [28] and the *resource containers* in our work are proposed respectively to encapsulate information entities.

Jackson and Zave [10] proposes the DFC method for feature specification and composition in telecommunication systems. This method follows the idea of implementing features as individual containers. DFC mainly explores the problem of using the pipe-and-filter architectural style to compose features. Resource containers accessed by features are abstracted into the *underlying architectural substrate* in this method.

In [29], a logic based variability validation method was proposed. However, this method does not consider the *binding-time* attribute of features, that is, there is no concept of *undecided* features in it. For this reason, this method can only apply well when all undecided features are decided (*bound* or *removed*). While our constraint and customization verification method (Sect. 2.3.2) integrates the logical verification with binding times, it can thus apply to any binding phase without any limitation to features' binding states. For example, suppose features A and B are two run-time binding features with an *exclude* constraint on them, and after customization, neither of them is removed from the feature model. Such a customization result will be considered to be "invalid" in [29], although the program logic can automatically maintain this *exclude* constraint at run-time. While in our method, these two features can coexist in any customized feature model, as long as one or both of them are in the *undecided* state at run-time.

8 Conclusion and future work

In this paper, we mainly present a metamodel to support the systematical modeling of system features and their refinement, constraint and interaction relationships. Particularly, three kinds of refinement, four kinds of constraint and seven kinds of binary interaction are defined in this metamodel. A visual notation for modeling constraints and a propositional logic based method to constraint formalization and to both constraint and customization verification is proposed. An approach to mapping between constraints and interactions through constraint semantics is also described.

Our current work presented in this paper can not fully resolve the problem about how to propagate the customizable structure (i.e. feature models) of the problem space to the solution space. But we do think that our current work provide the foundation for this problem, since we have systematically explored one of the most important relationships between features when constructing software, that is, the interactions between features, and have proposed a way to build traceability between constraints and interactions. Once constraints and interactions are identified and their traceability is build, the next work needs to do is to implement features and their interactions in a highly customizable way.

Our future work will focus on the feature implementation problem. A possible approach is to keep features as first-class objects in the solution space. Hence, we will explore the feasibility of such an approach by encapsulating each feature's specification as one individual component, and we will further analyze the *operational contexts* and the *implementation contexts* of features, following the 3C model of reusable software components [12, 13].

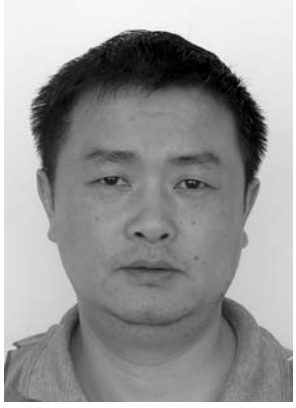
Acknowledgements This work is supported by the National Grand Fundamental Research 973 Program of China under Grant No. 2002CB312003, the National Natural Science Foundation of China under Grant No. 60233010, 60125206 and 90412011, and the Beijing Natural Science Foundation under Grant No. 4052018.

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions.

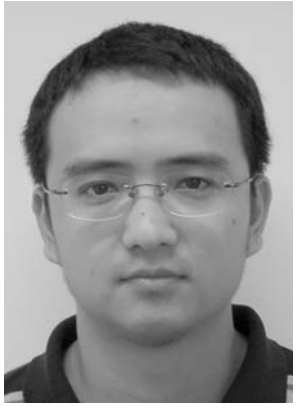
References

1. Barstow, D., Arango, G.: Designing software for customization and evolution. In: Proceedings of the Sixth International Workshop on Software Specification and Design, pp. 250–255 (1991)
2. Kang, K.C. et al.: Feature-oriented domain analysis feasibility study. SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (November 1990)
3. Kang, K.C. et al.: FORM: A feature-oriented reuse method with domain-specific architecture. Annals of Software Engineering **5**, 143–168 (1998)

4. Griss, M.L., Favaro, J., d'Alessandro, M.: Integrating feature modeling with the RSEB. In: Proceedings of Fifth International Conference on Software Reuse, pp.76–85. IEEE Computer Society, Canada (1998)
5. Chastek G. et al.: Product line analysis: A practical introduction. (CMU/SEI-2001-TR-001), Software Engineering Institute, Carnegie Mellon University (2001)
6. Palmer, S.R., Felsing, J.M.: A Practical Guide to Feature-Driven Development. Prentice Hall PTR (2002)
7. Mehta, A., Heineman, G.T.: Evolving legacy system features into fine-grained components. In: Proceedings of the 24th International Conference on Software Engineering. Orlando, Florida (2002)
8. Keck, D.O., Kuehn, P.J.: The feature and service interaction problem in telecommunications systems: A survey. IEEE Transactions on Software Engineering **10**(24), 779–796 (1998)
9. Antón, A.I., Potts, C.: Functional paleontology: The evolution of user-visible system services. IEEE Transactions on Software Engineering **29**(2) (2003)
10. Jackson, M., Zave, P.: Distributed feature composition: A virtual rchitecture for telecommunications services. Ieee Transactions on Software Engineering **24**(10) (1998)
11. Turner, C.R., Fuggetta, A., Lavazza, L., Wolf, A.L.: A conceptual basis for feature engineering. Journal of Systems and Software **49**(1) (1999)
12. Tracz, W.: The 3 cons of software reuse. In: Proceedings of the Third Annual Workshop: Methods and Tools for Reuse (1990)
13. Edwards, S.H.: The 3C model of reusable software components. In: Proceedings of the Third Annual Workshop: Methods and Tools for Reuse (1990)
14. Wiegers, K.E.: Software Requirements. Microsoft Press (1999)
15. Griss, M.L.: Implementing product-line features with component reuse. In: Proceedings of Sixth International Conference on Software Reuse, pp. 137–152. LNCS 1844,Vienna (2000)
16. Kang, K.C., Lee, K., Lee, J., Kim, S.: Feature oriented product line software engineering: principles and guidelines. A chapter in "Domain Oriented Systems Development—Practices and Perspectives", UK, Gordon Breach Science Publishers (2002)
17. Ferber, S., Haag J., Savolainen, J.: Feature interaction and dependencies: modeling features for reengineering a legacy product line. In: The Second Software Product Line Conference 2002, LNCS 2379, pp. 235–256 (2002)
18. Carlshamre, P., Sandahl, K., Lindvall, M., Regnell, B.: Natt och Dag, J.L An industrial survey of requirements interdependencies in software product release planning. In Proceedings of Fifth IEEE International Symposium on Requirements Engineering, pp. 84–91 IEEE Computer Society (2001)
19. Karlsson, J., Olsson, S., Ryan, K.: Improved practical support for large-scale requirements prioritizing. Requirements Engineering Journal **2**(1), 51–60 (1997)
20. SMV, Model Checking @CMU, The SMV System", http://www-2.cs.cmu.edu/∼modelcheck/smv.html
21. Mannion, M., Kaindl, H., Wheadon, J., Keepence, B.: Reusing single system requirements from application family requirements. In: Proceedings of the 21st International Conference on Software Engineering, pp. 453–462 (1999)
22. Fife, L.D.: Feature interaction: How it works in telecommunication software. IEEE (1996)
23. Buhr: Use case maps as architectural entities for complex systems. IEEE Transactions on Software Engineering **24**(12) (1998)
24. Garlan, D., Shaw, M.: An introduction to software architecture. In: Advances in Software Engineering and Knowledge Engineering, vol. 1. World Scientific (1993)
25. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Inc. (1996)
26. Biddle, R.L., Tempero, E.D.: Understanding the impact of language features on reusability. In: Proceedings Fourth International Conference on Software Reuse, pp. 52–61 (1996)
27. Sutcliffe, A., Maiden, N.: The domain theory for requirements engineering. IEEE Transactions on Software Engineering **24**(3) (1998)
28. Jackson, M.: Problem Frames: Analysing and Structuring Software Development Problems. Addison-Wesley (2001)
29. Mannion, M.: Using first-order logic for product line model validation. The Second Software Product Line Conference 2002, LNCS 2379, pp. 176–187 (2002)



Hong Mei received the BSc and MSc degrees in computer science from the Nanjing University of Aeronautics and Astronautics (NUAA), China, in 1984 and 1987, respectively, and the PhD degree in computer science from the Shanghai Jiao Tong University in 1992. He is currently a professor of Computer Science at the Peking University, China. His current research interests include Software Engineering and Software Engineering Environment, Software Reuse and Software Component Technology, Distributed Object Technology, and Programming Language. He has published more than 100 technical papers.



Wei Zhang received the BSc in Engineering Thermophysics and the MSc in Computer Science from the Nanjing University of Aeronautics and Astronautics (NUAA), China, in 1999 and 2002, respectively. He is currently a PhD student at the School of Electronics Engineering and Computer Science of the Peking University, China. His research interests include feature-oriented requirements modeling, feature-driven software architecture design and feature-oriented software reuse.

Haiyan Zhao received both the BSc and the MSc degree in Computer Science from the Peking Univeristy, China, and the Ph.D degree in Information Engineering from the University of Tokyo, Japan. She is currently an associate professor of Computer Science at the Peking University, China. Her research interests include Software Reuse, Domain Engineering, Domain Specific Languange and Program Transformation.