A Comprehensive Feature-Oriented Traceability Model for Software Product Line Development

Liwei Shen, Xin Peng and Wenyun Zhao
School of Computer Science, Fudan University,Shanghai,China
{061021062, pengxin, wyzhao}@fudan.edu.cn

Abstract

Feature-oriented traceability is essential for efficient Software Product Line (SPL) development, including product derivation and SPL evolution. Widely-used feature based method has been proved to be effective in domain analysis and modeling. However, it cannot support the traceability naturally due to the big gap between the problem space and the solution space. In this paper, we propose a comprehensive feature-oriented traceability model for SPL development, which provides mechanisms for various features and implementation types throughout the four levels of goal model, feature model, feature implementation model and program implementations. In it, the feature implementation model is introduced as the intermediate level between features and implementation artefacts. The feature interactions are captured in the finer role level, and they help to clarify the complex mapping between features and program implementations. The traceability meta-model for SPL development is introduced and a practical case study on the library management domain is demonstrated.

1. Introduction

The purpose of SPL method is rapidly producing cost-efficient and high-quality application products with domain core assets, including domain model, domain-specific architecture and domain components, etc. The ideal mode of product derivation is constructing the final product by configuring and tailoring of core assets, following a prescribed process, and complemented by application-specific implementation of some parts [4]. Widely-used feature based method has been proved to be effective in domain analysis and modeling. In these feature-based methods, proper mechanism of feature-oriented traceability throughout domain analysis, design and implementation is expected to be established to ease feature-based core assets selection, configuration and composition.

Feature-oriented traceability is the embodiment of requirement traceability in SPL development, which has been widely studied in several areas in software engineering, such as requirement management, software evolution, program comprehension, etc. Gotel et al. [2] define requirement traceability as “the ability to describe and follow the life of a requirement, in both a forwards and backwards direction”. In SPL development, requirement traceability is to identify and explicitly represent the deriving relationships among different domain artefacts, usually from domain feature model to design decisions, and further down to program implementations.

In SPL development, feature-oriented traceability is expected to map features to SPL designs and implementation elements, including commonalities and variations, to enable feature-oriented product derivation and SPL evolution, etc. However, existing feature-based methods do not provide the traceability naturally due to the big gap between the problem space and the solution space, which is generally called the feature tangling and scattering. Usually, there may be several components contributing to a single feature. A single component may also contain implementations for several features. As stated in [11], feature tangling and scattering have negative impacts on the system's maintainability. On the other hand, there exist various kinds of implementation artefacts for user-visible features. Component is the most acceptable that a system can be constructed by several components working together. Other implementation types such as code fragment, configuration file, data structure also make sense in the SPL development. Usually they cannot be encapsulated like component but they are used to collaborate with components to provide additional functions and controls. Thus, the traceability is especially difficult in SPL development due to the inherent demand of variability analysis, design and implementation.

In this paper, we propose a comprehensive feature-oriented traceability model for SPL development, which provides mechanisms for various feature types, considering commonality/variability, diverse binding time and differences among composite, generalization and atomic features, etc. The model focuses on the low-end user's perspective of modeling requirement dependencies, allocation of requirements to model and implementation elements which are included in the work by Ramesh and Jarke [13]. It provides visualized and comprehensive traceability representations for SPL development, throughout the four levels of goal model, feature model, feature implementation model and program implementations. In the model, feature implementation model is introduced as the intermediate level between features and implementation artefacts. The feature implementation level captures feature interactions (including cross-cutting interactions) in the finer role level, and helps to clarify the complex mapping between features and program implementations including components and other kinds of program units [4]. The traceability model involves trace links both within a level and between different levels. Based on this model, developers can understand how the requirements are realized and it also helps to guide the product derivation and SPL evolution phases.

The remainder of this paper is organized as follows. Section 2 introduces related works on feature-oriented traceability in SPL development. Section 3 presents the meta-model of traceability, including the four levels and diverse traceability links. A practical case study on the library management domain is demonstrated in section 4. Then in section 5, some discussions related to the feature-based traceability model is placed. Finally, section 6 concludes the whole paper and discusses our future works.

2. Related work

There is much seminal work in the area of traceability representation for SPL. They utilize model-based approaches which introduce several hierarchical layers to visually model the product line artefacts as well as the traceability links among the artefacts. For example, Lago et al. define a product-oriented model to represent features, design decisions and implementation assets at both the Product Family Level and the Product Level in [6]. The model supports cross-level traceability and has been used to extend a tool to support traceability in product families. In [5], Riebisch introduces traceability links among software development artifacts grouped and categorized in different abstraction levels including feature level, architecture component level, class level and implementation artifact level. He also introduces feature model as an intermediate element for linking requirements to design models in [7]. The model is useful to eliminate the huge difference between requirements and design elements, and features are regarded to offer advantages such as the reduction of links' number, easier verification, maintenance and comprehension. Similar with Riebisch's work, we follow the idea of representing various traceability links between artefacts in different abstraction levels. The difference is that we extend the semantic of the feature model as well as the implementation types. Furthermore, we introduce an innovative feature implementation model as the bridge between features and implementation artefacts. It represents the developers' design decisions which assign the requirements (by features) to the implementations in a finer granularity level. It also helps to reduce the complex relationships between the problem space and the solution space, making the requirement traceability comprehensive and natural.

3. Traceability meta-model

3.1. Overview

The feature-oriented traceability meta-model for SPL is presented in figure 1. It consists of four hierarchical layers with different stakeholder concerns, including domain requirement goals, feature model, feature implementation model and program implementations. Artefacts and trace links within each level are modeled: goal level involves functional and non-functional requirements; feature level includes different kinds of features such as Action, Facet as well as the related relationships [3]; feature implementation level consists of roles, inter-role relationships and interactions; program level includes domain/application components and other kinds of implementation units for variant features, such as configuration file items, code segments, data structures, etc. Traceability links are divided into explicit ones and implicit ones as stated in [12]. The former are visible in the model, represented by the lines with solid arrows with annotations. Contrarily, the latter are invisible and they are derived from the explicit traceability links.

3.1.1 Goal level. Goals capture stakeholder intentions [16] by means of high-level functional and non-functional requirements (FRs and NFRs). In our model, they are described in natural language. FRs are desired operations or abilities for the realization of the SPL, and an example can be "accomplish the functions of borrow and return in a library system". NFRs are

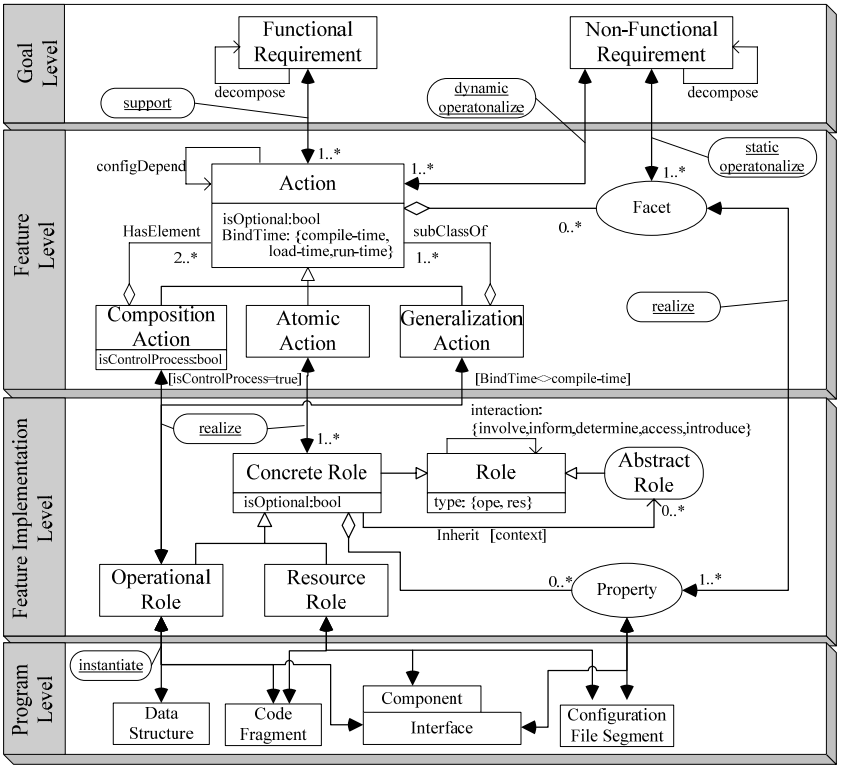


Figure 1. Feature-oriented traceability meta-model

usually on quality aspects of the SPL and those can be satisfied by system functions are included. Example for them can be "have good traceability in book circulation" and "ensure the reliability of borrow&return of valued books" in the library management domain. Both FRs and NFRs can be refined into sub-goals by decomposition and specialization. For example, FRs can usually be decomposed according to the expected menu hiberarchy while the NFRs can be continually decomposed until the requirement engineers consider the soft-goals can be satisfied [8].

3.1.2 Feature level. Domain feature model formally structures domain-specific requirements by various kinds of features and inter-feature relationships. It is the result of domain analysis with careful considerations on commonalities among applications and differences between applications in the domain. Feature level discussed in this paper is represented by the ontology-based feature model proposed in our previous work on feature modeling [3]. In the model, features and inter-feature relationships are subdivided into several categories. Their concepts are exemplified by the function *BorrowBook* on the library management domain whose feature model is illustrated in figure 2.

Action: An action represents a business operation (or function) of various granularities with domain-specific semantics. In figure 2, actions are represented by rectangles.

IfOptional: This attribute denotes whether an action is optional or not, and the value True means the element action is optional for its parent action. The optional actions in figure 2 are blocks with a circle.

HasElement: It is a type of specialization relationship and it divides the function of a parent-feature into the functions of its sub-features.

Composition Action: It is a kind of action which can be decomposed into sub-actions and has *HasElement* relation with its partitions. In particular, there is a new attribute *isControlProcess* that is introduced to determine whether the composition action should manage the execution sequence of its sub-actions. For example, the root action *BorrowBook* is a composition action but not controls process (*ifControlProcess=false*, not labeled). Contrarily, the *Borrow* action specifies the execution sequence of its sons (*ifControlProcess=true*) so that the operation can be performed in a correct behavior.

subClassOf: It is the self-defined ontology property between two actions or other ontology concepts representing direct specialization relationship between them.

Generalization Action: It is another kind of action whose sub-actions are specialized from it and have *subClassOf* relation with it. In figure 2, the action *RewardPointCaculation* is a generalization action. It has two concrete calculation types which can both take effect in products.

Atomic Action: It is the leaf of a feature model and could not be decomposed further.

Facet: Facets are defined as perspectives, viewpoints, or dimensions of precise descriptions for certain action, providing details of business semantics. For example, the action *BorrowControl* in figure 2 has a facet *Minimal Storage*, which acts as a parameter in one of the action's dimensions to decide the behavior of controlling the book process.

ConfigDepend: It represents configuration constraints, which are static dependencies on binding-states of variable features. Usually it is shown as *require* and *exclude* between two actions.

BindTime: It represents the time when the binding status of an optional feature will be decided. Three kinds of *BindTime* are distinguished: the first is *compile-time*, which means to make decision at program compiling phase; the second is *load-time*, which indicates the decision is made when system starts up; the last is *run-time*, which means whether the function is used depends on the runtime situation. In the example, the bind time of action *RewardPointCaculation* is compile-time, which means only one of its sub-action is included when deriving a product.

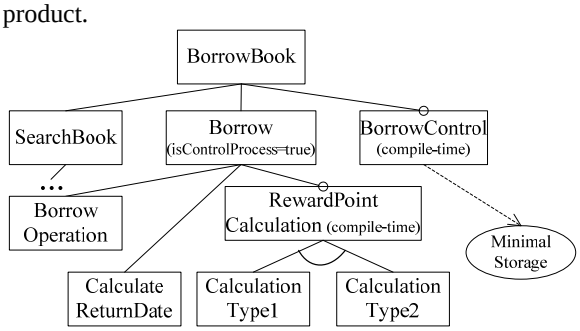


Figure 2. Ontology-based feature model example

Readers can refer to [3] for detailed descriptions for the feature model.

3.1.3 Feature implementation level. In this level, feature implementation model is introduced as an intermediate level between features and program implementations.

Feature implementation model can also be called role model. Role is the basic element. It is a logical unit, a responsibility which should be taken by program fragment for feature implementation. Compared with features, roles reserve the knowledge of features as well as endow features with the implementation aspects, so it can support the mapping from features to program implementations in a more natural way. The concept of role here is similar to the *responsibility* in [10] and the *role* in [9].

A role can be specialized into a ***concrete role*** or an ***abstract role***. An action in feature model can be realized by one or more concrete roles while abstract role does not refer to any feature but it is inherited by other concrete roles. The introduction of abstract role makes sense because it represents a set of concrete roles that share common characteristics so that interactions between a role and one in that set can be reduced, i.e. to provide expression for crosscutting interactions. For example, *borrow* and *return* are roles that will both change the storage of a book, so interactions between a role doing log and these two roles can be simplified as one interaction between the role *log* and an abstract role *bookChange* generalized from them.

In addition, two kinds of concrete roles are distinguished: one is *operational role*, and the other is *resource role*. An operational role is a functional segment while a resource role represents specific internal or external entity necessary for the implementation of a feature. A concrete role can also be optional to represent variability. Some are identified by inheriting from the feature model. Others are found as the so-called internal variability [1] which is necessary for technical reasons. Besides it, *property* is introduced to denote the perspective of a specific role, which is similar to the definition of *Facet*.

The interactions between roles are important because they indicate how roles cooperate with each other to implement a feature's function (action in our feature level) together, and they are also used to guide the composition of corresponding implementation artefacts. The interactions include those between roles from the same feature, as well as those between roles from different features. In our method, five kinds of role interactions are identified in table 1.

Table 1. Interaction types between roles

Interaction	Description
Involve	An operational role activates another operational role in a synchronous mode and makes it a part of the host operation.
Inform	An operational role informs another operational role to activate in an asynchronous mode.
Determine	Execution result of an operational role can determine the execution of another operational role, including whether execute or not and choosing a variant from several choices.
Access	An operational role reads or writes a resource role, or both, to fulfill its responsibility in specific feature implementation.
Introduce	A resource role is introduced into implementation unit of another operational role to be a sub-element.

3.1.4 Program level. Components and other kinds of implementation artefacts are modeled in this level. They come from asset depository and are reused to derive software applications. In the SPL context, a product line has a mandatory part which is regarded as the base programs while the variable part is implemented to weave into the base programs.

In our mechanism, we assume that all the commonalities for a SPL (base programs) are constituted by the components which collaborate with each other. On the other hand, the variations for a SPL are supposed to be implemented by all kinds of program artefacts which can be component, code fragment, configuration file, or data-structure. Components are black-box entities whose interfaces are exposed. Thus, an application is derived based on component composition through interfaces and other implementation artefacts which complement the system functions. These program implementations are attached to the base programs by means of specific weaving techniques such as AOP, if the corresponding variant is bound. Among them, configuration file which usually in the form of text-file or ini-file, contains formalized information as well as different parameter values organized in specific items. Data-structure denotes the database schema supporting the implementation of specific functions. For instance, a table for storing log information should be created if a function doing log takes effect.

3.2. Explicit traceability in the meta-model

Explicit traceability links represent the developers' knowledge about requirements and their realization in the developed system [12]. They are usually manually specified. In our meta-model, we separate the explicit traceability links into two categories: ***intra-level traceability*** and ***inter-level traceability***. Next we first briefly introduce the intra-level traceability.

Intra-level traceability is the associations between elements in the same level: the *decompose* relation between FRs or between NFRs in the goal model; the *HasElement*, *subClassOf* and *configDepend* relations in the feature model; five kinds of interactions between roles in the feature implementation model. Usually, in each level, abstract and composite elements may first be refined, and then the atomic elements can generally be mapped to elements in another level. On the other hand, interactions between roles are modeled to clarify how a user-visible feature is implemented by several logical sides and to guide the program-level customization and composition.

Inter-level traceability is more complex for it is the associations between elements of different abstraction levels. In our mechanism, we assume that the explicit inter-level traceability links only exit between adjoining levels. From figure 1, we can see diverse kinds of traceability links, distinguished with different names and semantics.

3.2.1 Traceability between the goal level and the feature level. We define the traceability between FRs and actions as ***support***. Usually, we analyze a textual requirement specification and map it to a set of actions. Some of the actions may be optional because of the variability in the requirements. On the other hand, the *BindTime* attribute of each optional action depends on how the requirements express. For example, function *X* takes effect if the physical memory is bigger than *Y*. The *BindTime* of the corresponding action should be set to *load-time*, that is to decide the binding status of the action according to the computer's physical configuration as soon as the system starts.

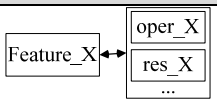
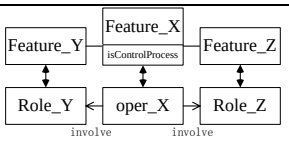
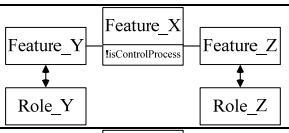
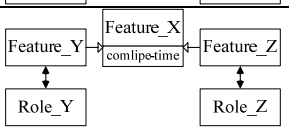
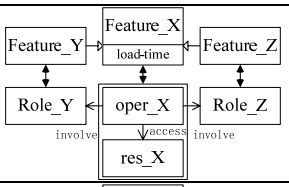
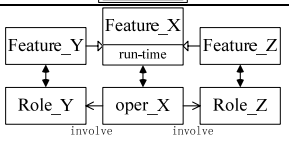
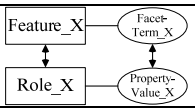
The traceability related to NFRs follows the mechanism in [8], which identifies the traceability from NFRs to actions as ***dynamic-operationalize***, and traceability from NFRs to facet as ***static-operationalize*** respectively. Dynamic-operationalize means that such kinds of NFRs require actual operations to accomplish them while static-operationalize means perspectives of an action which can be adjusted to satisfy the specific quality requirements.

3.2.2 Traceability between the feature level and the feature implementation level. This kind of traceability is called ***realize***. It denotes the associations from the elements in feature model to the elements in role model (table 2).

An atomic action can be realized by a set of concrete roles. These roles may include operational roles as well as resource roles which divide the action's function into finer logic units. The variability of these roles does not follow the action all along, i.e. some of them are the newly defined internal variations which are always technique-related and unconcerned by customers. Besides roles, interactions (table 1) between the roles are also modeled. These roles cooperate with each other according to the interaction rules to realize the action's function correctly.

The traceability from a composition action is different. It exits only if the *isControlProcess* attribute of the composition action is true, i.e. it managers the execution sequence of its partitions. Under the situation, the functional part is decomposed and delegated by its sub-elements, but the part relating to process controlling still remains to be traced. In our mechanism, we map such a composition action to an operational role which takes the responsibility of controlling the execution flow of the other related roles. On the contrary, if the composition action doesn't involve the

Table 2. The realize traceability

Origination	Precondition	Target	General Realization Units	Illustration
atomic action	N/A	operational, resource role(s)	component interface, code fragment, data structure, component…	
composition action	*isControlProcess* =true	operational role	component interface that controls the execution flow	
	isControlProcess =false	N/A	N/A	
generalization action	*BindTime*= compile-time	N/A	makefile (in C++) build.xml (Ant for Java) …	
	BindTime= load-time	operational role, resource role	configuration file, system parameter, method that readers the parameters	
	BindTime= run-time	operational role	runtime logic user choice	
facet	N/A	property	component interface accessing property, configuration file	

execution sequence, the traceability makes no sense and it can be ignored.

The traceability from a generalization action differs a lot due to the *BindTime* attribute. Usually, variability lies in it because it determines the different binding and realization mechanisms. Suppose that a generalization action *X* is specialized into action *Y* and *Z* (last column in table 2), we will give three traceability situations based on the different *BindTime* settings. Firstly, if the *BindTime* of *X* is *compile-time*, only one of the sub-actions in *Y* and *Z* will be chosen to compose the final product. In such circumstances, *X* is modeled as a placeholder so that it will be replaced by one of its concrete actions in product derivation. Sometimes the replacement can be realized by means of *makefile*(in C++) or *build.xml*(Ant for Java), which configures the program units. However, they are not software implementation artefacts in the common sense, thus not contained in our meta-model. Secondly, if the *BindTime* of *X* is *load-time*, the binding decision depends on the actual environment when the system is starting. It is usually decided by reading a configuration file containing startup settings or reading from system parameters. So we can map *X* to an operational role and a resource role. The former makes the decision through accessing the latter. Thirdly, if the *BindTime* of *X* is run-time, the binding decision is undetermined and it may change when system is running. The choice can be made by user or by program logics. Therefore, *X* can be regarded as a proxy and further realized by an operational role that is able to accept user choice or take effect by prescribed logics.

On the other hand, *realize* also resides in traceability from facet in feature model to property in role model. Since facet can be regarded as a kind of variability towards a feature, we reserve the variability on implementation level and it will ultimately be implemented by specific program units such as component property (accessed through interface) and configuration file's segment.

3.2.3 Traceability between the feature implementation level and the program level. ***Instantiate*** denotes the traceability from concrete roles to program artefacts.

Operational roles, representing the business logics, are usually instantiated by component interfaces, code fragments or data structures. Interfaces of components offer the business operations. Code fragments always provide operational roles with additional implementations which are combined to the base programs. Data structures are necessary if an operational role requires tables to store information in a database. In our model, the mandatory roles are always mapped to the component interfaces which will be composed to derive the common part of a SPL.

Resource roles are usually instantiated by components, code fragments and segments in configuration files. For example, if a resource role denotes a visible widget, it should be instantiated by a widget component and some code fragments to initialize it in the UI. If the resource role represents a parameter, it should be mapped to the relevant segment in a configuration file.

Property denotes the perspectives of a role. It can be accessed from outer files or coded in programs. Thus, we usually instantiate them as the configuration file segments or the components properties which are actually accessed through the interfaces.

3.3. Implicit traceability in the meta-model

Implicit traceability links are automatically derived using explicit links to gain a more complete insight into a system by showing relationships between artefacts that the developer may not have been aware of [12]. They are not visually represented in the meta-model. Using criteria like transitivity, we can derive implicit traceability links which will bring benefits. For example, the traceability between requirements and the implementation artefacts is derived by combining the inter-level traceability links to check if all the requirements are implemented as well as to ease the comprehension of the SPL. Similarly, elements in the same level can also have implicit traceability if they refer to the same program units so that the traceability is helpful when performing change impact analysis.

4. Case study

Represented by the model above, a practical case study on library management domain is illustrated in figure 3. The top layer is the goal level representing FRs and NFRs of the domain. The next layer is the feature level containing the ontology-based feature model. In it, white rectangles represent mandatory features while grey-filled ones represent variable features (for the sake of clarity, we user color instead of circle). Role model is described in next level. Similarly, variable roles are represented by grey-filled rectangles and roles surrounded by a pane are traced from a same feature. In the bottom level, components and other kinds of implementations such as configuration file and data structure are illustrated. For the sake of clearness, each program implementation has a prefix whose legend is on the corner.

In the goal level, FRs and NFRs are expressed in natural language specifications. *Reader Management*, *Book Management* and *Borrow&Return Operations* are separated functional requirements. *Storage Safety* (the storage of a book cannot be less than a minimal value to ensure the book is available in library) and *Operation Traceability* (record borrow&return operations of readers to avoid influence by system failure) are non-functional requirements decomposed from the goal of *System Safety*.

The FRs are supported by actions, e.g. *Borrow&Return* is supported by *BorrowBook* and *ReturnBook*. Next we discuss *BorrowBook* in detail. *BorrowBook* is decomposed into *SearchBook*, *Borrow* and *Borrow-Control* (control book borrowing by prescribed policy) by *HasElement* relationship. The first two are mandatory actions possessed by each application while the third one is dispensable. *SearchBook* has an optional action *BookPicShow* (show the book picture when browsing) whose *BindTime* is compile-time. *Borrow*, as a composition action controlling execution sequence (*isControlProcess =true*), is further decomposed into *Borrow Operation*, *Calculate ReturnDate* and *RewardPoint Calculation*. *RewardPoint Calculation* here is a generalization action whose *BindTime* is also compile-time which indicates only one calculation type will be bound and used in an application.

As for the NFRs, *Operation Log* is an action dynamic operationalized from *Operation Traceability*, and another NFR *Storage Safety* is static operationalized to a facet *Minimal Storage* belonging to *BorrowControl* to specify the parameter.

In the next layer, an action is realized by a single role or a set of roles surrounded by a pane. For example, the action *SearchBook* refers to the role *BookSearch*, while the action *BorrowOperation* is realized by two roles *SetBookAmt* (update the storage of a book) and *AddReaderRec* (add a book to the reader's borrowing list). In particular, the action *Borrow* is realized by a role *BorrowProcess* because its *isControlPrcess* attribute is true. And the action *RewardPoint Calculation* doesn't trace further because it acts as a placeholder (compile-time binding).

The interactions between roles are also modeled. In the figure, *BookPicShow* is realized by three roles: *PicShowControl*, *ImageFetch* and *ImageContainer*. *ImageContainer* is an image container for picture visualization, *ImageFetch* is to fetch image data, and *PicShowControl* takes the responsibility of controlling image fetching and showing. So, we can see that *ImageFetch* is involved by *PicShowControl*, which means the former is activated by the latter, and *PicShowControl* accesses *ImageContainer* to set the image. They are the interactions between roles from a same feature. On the other hand, *PicShowControl* is involved by *BookSearch* to take effect. It's the interaction between roles from different features. In particular, the role *BorrowProcess* controls the execution process, so it involves all the roles mapped from the sub-actions in the composition action *Borrow*.

We also model the role interactions related to abstract roles (see in section 3.1.3). This kind of interactions usually crosscut multiple parts of a system. For example, in figure 3, *Log* will be informed to activate by all the events of readers, e.g. *AddReaderRec*, *DelReaderRec*, etc. So, abstract role *ReaderEvent* is modeled to generalize the two roles and to simplify the interactions towards them.

Roles are instantiated by different kinds of assets in the bottom level. For instance, the role *SetBookAmt* is instantiated by the *setAmount* interface in the component *Book*. The role *Penalty Param* corresponds to the penalty segment in a configuration file which is assessed by the *Calculate Penalty* role. The role *ImageContainer*, as a resource role, is instantiated by a component *Canvas* as well as a code fragment to be used for its initialization in UI. The two *RewardPoint Calculation Type* roles are optional roles and they are realized by two methods which are not included in any component but only one of the methods will be composed into the base programs. The property *Minimal Storage* can be instantiated into an interface accessing the related variable in component *Book*. The role *log* is instantiated as a *log* method to perform its function as well as a schema to create the log table in database for storing operation records.

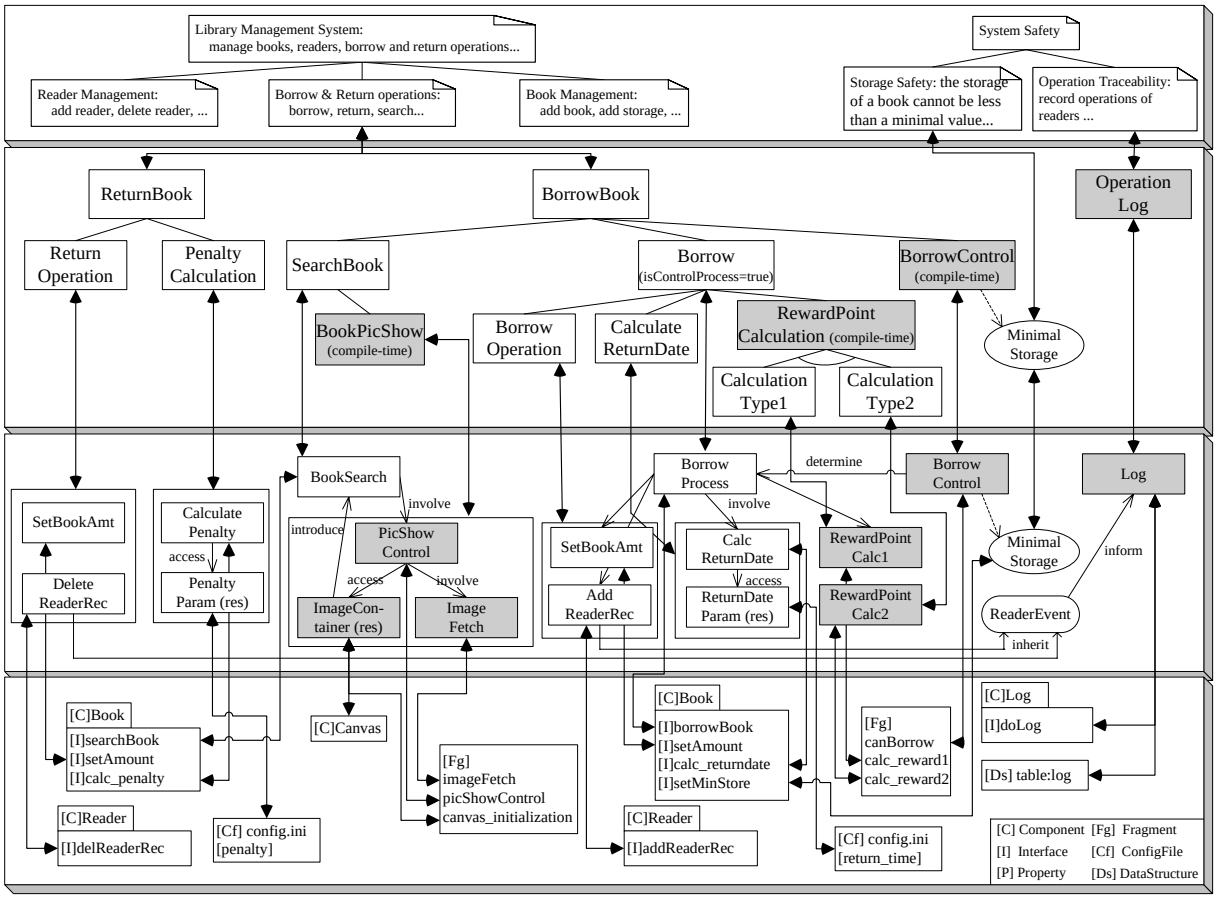


Figure 3. The traceability model on library management domain

5. Discussion about the feature-oriented traceability model

In this section, we will briefly discuss about the feature-oriented traceability model: why we adopt the feature implementation model and ignore the architecture, how the model directs the product derivation and SPL evolution phases. On the other hand, related tools are also presented.

5.1. Feature implementation model instead of SPL architecture

A product line architecture (PLA) specifies the architecture for a set of closely related software products [14], usually in terms of components, connectors and configurations [15]. A PLA focuses on modeling the variability for a domain and promotes the reuse in the SPL development. However, in the traceability context, feature tangling and scattering still exist between feature model and PLA, i.e. several components may contribute to a single feature and a single component may contain implementations for several features. Therefore, it is difficult to explain how the functions of features are splitted as well as what is the intrinsic semantic of the feature interaction.

In our model, the feature implementation model is introduced to replace the complex many-to-many traceability between features and implementation artefacts with two sets of clear trace links. Roles decompose features and the inter-relationships between the feature parts (role interactions) are also defined. Role model provides the design decisions from the viewpoint of system designers, which are not concerned by the requirement analysers. It also captures the inner structure of a feature and records the semantic of feature implementation (reason for splitting feature functions) and feature interactions (the intrinsic relationships between features) in a finer level. Furthermore, roles are explicitly assigned to different implementations artefacts including components and other forms of implementations. Thus, the semantic for traceability from requirements to implementations is complemented and extended.

5.2. Traceability-based product derivation and SPL evolution

Traceability is the basis of product derivation because we need to find out the variability-related program implementations according to the variable requirements and combine them with the base implementations [9]. Usually, the derivation process is divided into feature-driven customization and composition. In our mechanism, we extend it with traceability-based role-level customization and program-level composition.

The role-level customization decides whether the optional roles will be included in the final product. Some of the result comes from the feature-level customization if the roles are homogeneous with the features which can be decided by requirements directly. However, internal variability-related roles are unconcerned by clients and they should be customized by developers in role-level. For example, we suppose that role *ImageFetch* in figure 3 is modified as a variable role and it has two sub-roles which are to fetch image from database or from file system. In our method, the concrete fetching style of the role is determined only by the developers because clients don't care about it.

After role-level customization, variability-related implementation artefacts can be selected and configured according to the role instantiation traceability. Then role interaction, as an important kind of traceability, guides the program-level composition, which is to instruct what kinds of program implementations should be composed and how they can be composed. In our mechanism, the component composition is instructed because the traceability between roles and component interfaces as well as the role interactions are determined. Developers are also guided how to weave the variable program units into the component based programs by means of different variability implementation approaches such as AOP, which is adopted in our tools.

Traceability also plays an important role in SPL evolution. We are able to locate the features, roles and program implementations that are involved in the evolution through traceability and then make decisions on the evolution of the models, the implementations, as well as the traceability links. Usually evolution is driven by two factors. One is the requirement changes, the other is bug fixing. Our mechanism is useful in these perspectives for it assists change impact analysis to identify the involved part of a SPL driven by a specific evolution request.

5.3. Tool support

The tools *OntoFeature*[3] (figure 4) and *FDAPD* (feature-driven and aspect-based product derivation tool) [4] (figure 5) are developed to support the manually traceability capturing and visualizing. They are now separated and are used by different developers (requirement analyzer, system designer) in the SPL development process.

OntoFeature is used to accomplish the ontology-based feature modeling which includes actions, facets

and the relationships between these elements. FDAPD is integrated with OntoFeature by importing and capturing the feature list and dependencies. It provides editing space for each feature, where the roles, interactions and the corresponding implementation artefacts are identified. In practice, the two tools are cooperated, i.e. OntoFeature for representing requirements and FDAPD for further design and implementation.

FDAPD is also developed to support the role customization and program-level composition by invoking the *AOP* mechanism i.e. the variability-related implementations are selected as aspects to be woven into the base programs according to the interaction types. The detailed composition process can be referred to [4].

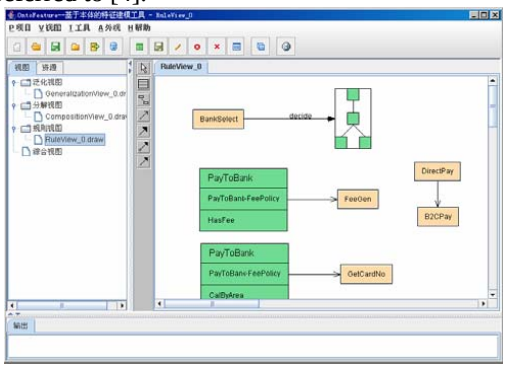


Figure 4. OntoFeature

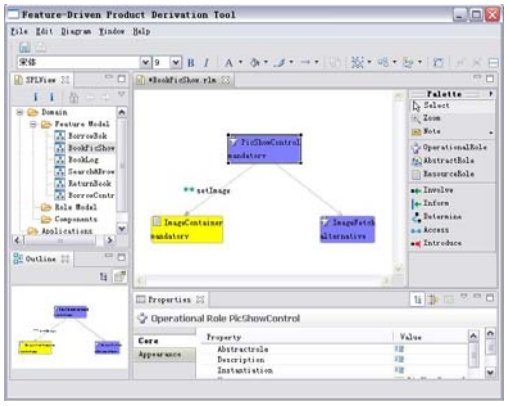


Figure 5. FDAPD

6. Conclusion and future work

In this paper, we focus on the visualized representation of the traceability in a software product line, and introduce a comprehensive feature-oriented traceability model. The model explicitly represents the product line artefacts in different abstraction levels. It also contains various kinds of traceability links (explicit/implicit) among the artefacts. Based on it, the traceability information from requirements to implementations is extended, described in finer-grained levels. In the whole model, the feature implementation model is innovative that it integrates the knowledge of both business logics and implementation techniques. We have discussed that it is helpful to reduce the big gap between the problem space and the solution space in the traceability context, which is the reason of ignoring the product line architecture. We also discuss the role traceability plays in product derivation and SPL evolution phases.

However, the method we propose is far from mature since we only provide an informal representation model for SPL traceability. The model elements and the traceability information can be manually captured and recorded by mean of the developed tools, but we still need a formal representation mechanism to support the automatic traceability management, deduction and validation. Thus we will try to give a formal notation for SPL traceability in the future work. On the other hand, the meta-model is not complete that the traceability can be extended to other SPL development perspectives. For example, traceability related to testing is necessary in many development processes. Thus, we will complement the current traceability on design and implementation with the trace to testing artefacts in the future work.

Acknowledgments. This work is supported by National Natural Science Foundation of China under Grant No. 60703092, and National High Technology Development 863 Program of China under Grant No. 2007AA01Z125.

7. References

[1] D. M. Weiss and C. T. R. Lai, “Software Product Line Engineering: A Family-Based Software Development Process”, Addison-Wesley, 1999.

[2] O.Gotel and A.Finkelstein, “An Analysis of the Requirements Traceability Problem”, in Proceeding of 1st International Conference on Requirement Engineering, 1994.

[3] Xin Peng, Wenyun Zhao, Yunjiao Xue and Yijian Wu, “Ontology-Based Feature Modeling and Application-Oriented Tailoring”, in Proceedings of International Conference on Software Reuse (ICSR2006), pp.87-100 .

[4] Xin Peng, Liwei Shen, Wenyun Zhao, “Feature Implementation Modeling based Product Derivation in Software Product Line”, in Proceedings of International Conference on Software Reuse(ICSR2008).

[5] M.Riebisch, R.Brcina, “Optimizing Design for Variability Using Traceability Links”, in Proceedings of International Conference on Engineering of Computer Based Systems (ECBS2008), pp.235-244.

[6] P.Lago, E.Niemela, H.Van Vliet, “Tool Support for Traceable Product Evolution”, in Proceedings of the Eighth

European Conference on Software Maintenance and Reengineering (CSMR2004), pp.261-269.
[7] M.Riebisch, “Supporting Evolutionary Development by Feature Models and Traceability Links”, in Proceedings of IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS2004), pp. 370-377.
[8] L.Cysneiros, J.Leite, “Nonfunctional Requirements: From Elicitation to Conceptual Models”, in IEEE Transactions on Software Engineering, Vol. 30, No. 5, May, 2004.
[9] A.G.J. Jansen, R. Smedinga, J. van Gurp and J. Bosch, “First class feature abstractions for product derivation”, in IEE Proc.-Softw., Vol. 151, No. 4, August 2004.
[10] Wei Zhang, Hong Mei, Haiyan Zhao,“Feature-driven requirement dependency analysis and high-level software design”, in Requirements Eng (2006) Vol.11, pp: 205–220.
[12] M.Aleksy, T.Hilenbrand, C.Obergfell, M.Schwind, “A Pragmatic Approach to Traceability in Model-Driven Development”, in Proceedings of the Multikonferenz Wirtschaftsinformatik 2008 (MKWI2008).
[13] B.Ramesh, M.Jarke, “Towards reference models for requirements traceability”, in IEEE Transactions on Software Engineering, 2001, 27(1):58.
[14] J.Bosch, “Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach”, Pearson Education (Addison-Wesley & ACM Press), May 2000.
[15] N. Medvidovic, R.N. Taylor, “A Classification and Comparison Framework for Software Architecture Description Languages”, in IEEE Transactions on Software Engineering, 2000. 26(1): p. 70-93.
[16] Yijun Yu, Yiqiao Wang, J.Mylopoulos, et al, “Reverse Engineering Goal Models from Legacy Code”, in Proceedings of IEEE International Conference on Requirements Engineering (RE2005), pp.363-372.