ISSN 1673-9418 CODEN JKYTA8
Journal of Frontiers of Computer Science and Technology
1673-9418/2008/02(03)-0274-22
DOI:10.3778/j.issn.1673-9418.2008.03.005

E-mail:fcst@public2.bta.net.cn
http://www.ceaj.org
Tel:+86-10-51616056

Feature-Oriented Software Product Line Design and Implementation Based on Adaptive Component Model*

YANG Yiming, PENG Xin+, ZHAO Wenyun
Department of Computer Science and Engineering, Fudan University, Shanghai 200433, China
+ Corresponding author: E-mail: pengxin@fudan.edu.cn

基于适应性构件模型的软件产品线设计和实现 *

杨益明,彭　鑫+,赵文耘
复旦大学 计算机科学与工程系,上海 200433

摘　要:在当前面向特征的软件产品线开发方法中，需求级的可变性分析、可变点表示以及面向应用的定制已经得到了较好的支持。但是，从需求级的定制和裁剪(特征模型)到实现级(体系结构和构件)的映射仍然存在许多困难。针对这一问题，文章提出了一种基于适应性构件模型的软件产品线开发方法。这种适应性构件模型引入基于特征的领域模型作为构件端口(包括内部端口和外部端口)的语义基础。另一方面，适应性构件模型所具有的微体系结构使得面向特定应用的构件行为定制成为可能。为了实现构件级面向特征的定制，构件内部负责内部和外部协作的控制中心与构件的计算逻辑被分离开来，执行经定制后的构件行为协议和端口语义。构件协作和计算功能的分离使针对构件行为的面向应用的定制更加便利。这样，产品线应用开发中需求级的特征定制就可以映射为体系结构和构件级的结构和行为调整。

关键词:软件产品线;特征;适应性构件;构件模型;定制;产品线实现
文献标识码:A　　**中图分类号:**TP311

YANG Yiming, PENG Xin, ZHAO Wenyun. Feature-oriented software product line design and implementation based on adaptive component model. Journal of Frontiers of Computer Science and Technology, 2008,2(3):274-295.

* the National Natural Science Foundation of China under Grant No.60703092 (国家自然科学基金); the National High-Tech Research and Development Plan of China under Grant No.2006AA01Z189,2007AA01Z125 (国家高技术研究发展计划(863)).
Received 2008-03, Accepted 2008-05.

Abstract: In current feature-oriented methods for Software Product Line (SPL) development, requirement-level variability analysis, representation and application-oriented customization have been well understood and supported. However, it is still difficult to map customization and tailoring on requirement level (feature model) to implementation level (architecture and components). In this paper, a SPL development method based on the feature-oriented adaptive component model proposed in authors' previous work is proposed. The adaptive component model introduces feature-based domain model as the semantic basis of component ports (including internal and external ports). On the other hand, the adaptive component model has a micro control structure within the component, which enables the adaptation of the component behavior, including inter-component interactions, interaction sequence and style. In order to implement the feature-oriented customization on the component level, an in-component control center is separated to enforce the customized behavioral protocol and port semantics for each component according to the mapping specification. This separation of component coordination and computation facilitates the application-oriented customization on component behaviors. Then, in application development, requirement-level feature customization can be mapped to architecture- and component-level adaptations on architectural structure and component behaviors.

Key words: software product line; feature; adaptive component; component model; customization; product line implementation

1 Introduction

Software Product Line (SPL) is a set of software-intensive systems that share a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way[1]. The targeted set of software-intensive applications constitutes a specific domain. SPL has been widely accepted as an effective method to reduce application development cost and time to market, while improve the quality at the same time. This goal is achieved by systematic domain-level analysis, design and implementation (called domain engineering) with comprehensive commonality/variability consideration.

SPL is a systematic reuse-oriented development method for specific domain. The two major engineering processes of SPL are domain engineering and application engineering. The domain engineering process consists of activities for analyzing systems in the domain and creating reference architectures and reusable components based on the analysis results[2]. It is the phase of creating reusable domain assets for reuse. The other major SPL process is the application engineering process, which consists of activities for developing applications using the artifacts created in the domain engineering[2]. It is the phase of reusing domain assets to derivate application systems. Thus, it can be seen that the value of a SPL is finally realized in reuse-based application development.

The ideal mode of product derivation in SPL 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[3]. This ideal mode requests systematic variability consideration in domain engineering. The variability of a product line is usually captured in a feature model during analysis and must consequently be reflected and supported by its architecture and components[4]. In current feature-oriented methods for SPL development, requirement-level variability analysis, representation and application-oriented customization have been well understood and supported. However, their transition to architecture and implementation remains vague[4], making the customization-based application derivation hard to realize on the implementation level. This situation is due to the big gap between problem domain and solution domain, which makes it difficult to map customization and tailoring on requirement level (feature model) to implementation level (architecture and components)[3].

Aimming at the problems, we propose an SPL development method based on the feature-oriented adaptive component model proposed in our previous work[5]. The adaptive component model introduces feature-based domain model as the semantic basis of component ports (including internal and external ports). On the other hand, the adaptive component model has a micro control structure within the component, which enables the adaptation of the component behavior, including inter-component interactions, interaction sequence and style[5]. Static behavioral protocol and port semantics of each business component are specified in domain design and customized in application development. Implementation mappings between feature semantics and program-level parameters are also specified in component specifications. In order to implement the feature-oriented customization on the program level, an in-component control center is separated to enforce the customized behavioral protocol and port semantics for each component according to the mapping specification. This separation of component coordination and computation facilitates the application-oriented customization on component behaviors. Then, in application development, requirement-level feature customization can be mapped to architecture- and component-level adaptations on architectural structure and component behaviors. Therefore, feature model in our method provides both the business view on the requirement level and the customization basis on the architecture and component level, so as to implement feature-oriented application development. And both business-process-level and single-operation-level customization can be supported in our method due to the customizable component behavioral protocol and port semantics.

The remainder of this paper is organized as follows. Section 2 provides overview of the method and some background introduction of feature modeling. A segment of the domain feature model for the online shopping SPL is also introduced in section 2, which is referred throughout the paper to illustrate our method. Section 3 presents the conceptual model of the feature-oriented adaptive component and discusses the reason for adopting the adaptive component model in SPL development. Implementation infrastructure of the adaptive component model is introduced in section 4. Feature-oriented application derivation is presented in section 5, including application architecture customization and component-level behavioral adaptations. Then section 6 introduces the supporting tool and section 7 presents related works and discussions. Finally, section 8 summarizes the paper and provides an outlook of our future work.

2 Method Overview

2.1 Method Process

The process of our SPL development method is presented in Fig.1, involving two major phases of domain engineering and application engineering. In domain engineering, domain requirements are first analyzed and modeled in domain feature model. Then SPL architecture is designed on the system level and component level. On the system structure level, feature implementations are allocated into different business components and basic interaction relationships are also established among components. On component level, semantics of external and internal ports, and static behavioral protocol for the component are designed and specified. External ports are interfaces for inter-component interactions, including provide ports and request ports[5]. Internal ports are internal functional interfaces provided by the component implementation body to fulfill external requests[5]. Usually, feature semantics of internal ports and external request ports are well designed to help fulfill the services provided on provide ports. Component behavioral protocol in our method is the separated coordination logic of the component. The static behavioral protocol is specified in domain design and customized in application development, and then the application-specific behavioral protocols will be enforced by the implementation infrastructure of the adaptive component. In our method, SPL variations are embodied in both behavioral protocol and port semantics of each component. In domain implementation, component developers implement computation logic for each of the internal port according to the component specification, e.g. each internal port is implemented by a method and the parameter conventions are the same with what specified in the component specification. Component developers do not need to implement component coordination logic, since it will be enforced by the implementation infrastructure of our adaptive compo-

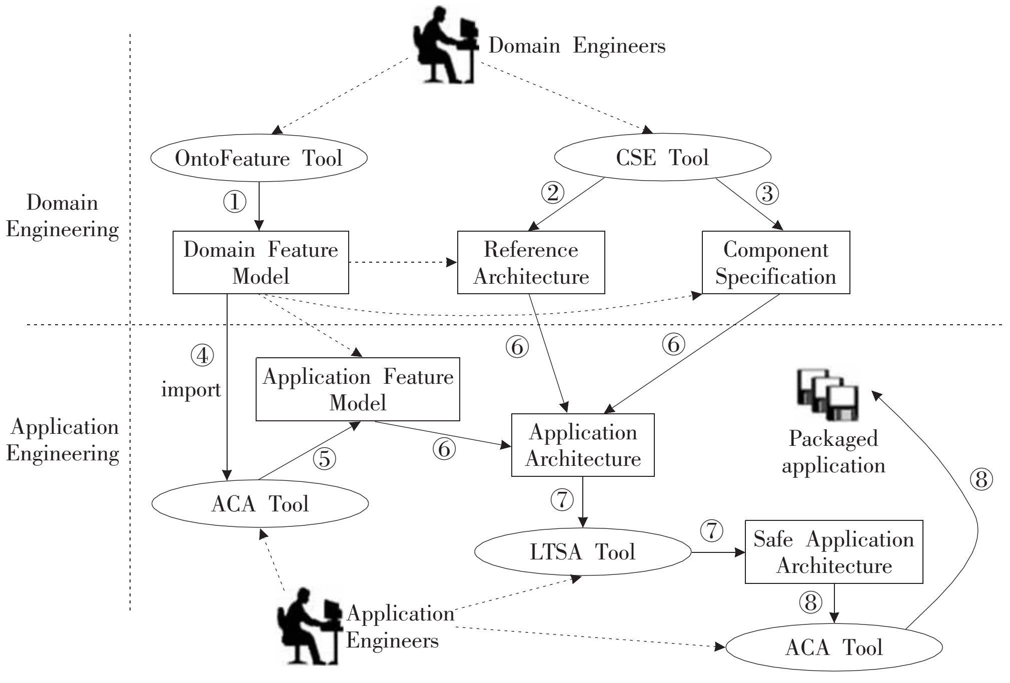


Fig.1 Process of the feature-oriented SPL development based on the adaptive component model

图 1 基于适应性构件模型面向特征的软件产品线开发流程

nent model. Details of the conceptual model and implementation infrastructure of the adaptive component are introduced in section 3 and 4 respectively.

In application engineering, domain feature model will first be customized to represent the application requirement. The customization is based on the variations embodied in the domain feature model, and feature constraints, which are a kind of static dependency among features[6], are evaluated to ensure the consistency. Due to the feature-based semantic specification of component ports and behavioral protocol, feature-level customization can be mapped to semantic adaptations on component ports and behavioral protocol. That is to say, application-specific behavioral protocol and ports semantics will be determined for each component. The adaptations will reflect on the customization of the SPL architecture: optional interaction relationships between components are determined to be bound or removed; inter-component interaction semantics are also refined according to the application-specific requirements. In some cases, application developers may need to develop some application-specific components and integrate them into the customized application architecture. The final determined architecture and component behaviors will be validated to ensure deadlock-free and policy-satisfying. Then the application implementation is ready, and can be packaged and released together with the infrastructure of the adaptive component.

In order to support the feature-oriented development process, we implement a tool to support the feature-oriented architecture and component specification (CSE Tool: Component Specification Editor) and customization (ACA tool: Application Customization assistant), which is integrated with the feature modeling tool (OntoFeature Tool) developed in our previous work[7]. On the other hand, we also implement the infrastructure of the feature-oriented adaptive component, which enables the enforcement of the customized component behaviors. In order to ensure the consistency of the application architecture, all the behavior protocols are validated together to avoid interaction deadlock by the LSTA tool.

2.2 The Feature Model

Feature-based methods have been widely adopted in domain analysis and modeling, e.g. [2,6,7]. A feature model specifies common understanding of a business domain, which can also be taken as the knowledge basis for the component implementation and adaptation[5]. In feature-based domain model, aggregation and generalization are applied to capture the commonalities among applications, while differences between applications are captured in the refinements, mainly by decomposition and specialization. In our previous work on domain analysis and modeling[7], we propose an ontology-based feature model, which adopts the W3C recommended ontology language OWL[8] as formal foundation of the feature meta-model. In the model, features and inter-feature relations are subdivided into several categories which are all defined by OWL, including:

Action: An action represents a business operation (or function) of various granularities with domain-specific semantics.

Facet: Facets are defined as perspectives, viewpoints, or dimensions of precise descriptions for certain action, providing details of business semantics. An action can have multiple facets and corresponding value for each facet. Facets can be inherited along with generalization relations between actions.

Term: Terms are restricted value space for facets.

subClassOf: It is the self-defined ontology property between two actions, two terms or other ontology concepts representing direct specialization relationship between them. There is also the reflexive and transitive property of *rdfs*: *subClassOf* in OWL, so *subClassOf* is a sub-property of *rdfs*: *subClassOf*.

IfOptional: This relation denotes whether a *HasElement* dependency between two actions is optional or not, and the value *True* means the element action is optional for its parent action.

ConfigDepend: It represents configuration constraints, which are static dependencies on binding-states of variable features. Both the source and target of it can be an action or specific facet value of an action (denoted by the expression of *FacetValue*).

BusinessObject: They are objects of business operations (*Action*) and usually represent certain business entities in the domain.

In the ontology-based meta-model *Action* and *Term* are defined as OWL classes (*owl*: *Class*). Facet is defined as OWL property (*owl*: *Property*), domain (*rdfs*: *domain*) and range (*rdfs*: *range*) of which are *Action* and *Term* respectively. As OWL classes, *Action* and *Term* can form their own specialization tree by the relation *subClassOf*. There are also some other elements in the feature meta-model, including those for binding time and other dependencies. Readers can refer to [7] for detailed descriptions for the ontology-based feature model.

2.3 Segment of the Online Shopping Domain Feature Model

With all the feature elements introduced in section 2.2, we can describe the feature model of a domain by defining all the business concepts and relationships among them[5]. A segment of the feature model for the online shopping domain is presented in Fig.2. We can see *OnlineShopping* mainly in-

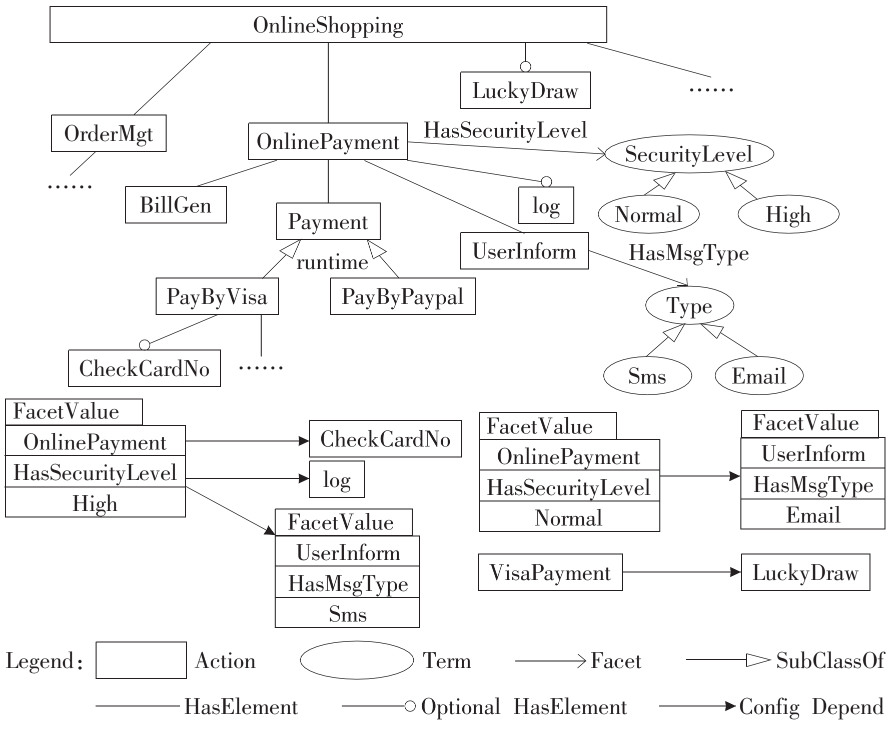


Fig.2 Segment of the feature model for the online shopping domain

图 2 在线购物领域特征模型片段

cludes *UserMgt*, *OrderMgt*, *OnlinePayment* and the optional *LuckyDraw*. *OnlinePayment*, an important part in online shopping, is analyzed in detail and presented by a series of sub-features. It is decomposed into *BillGen* (generate the bill to be paid), *Payment* (execute payment through certain channel), *Log* (record payment log) and *UserInform* (inform payment result to the user), in which *Log* are identified to be optional.

Security is an important property of online payment, with different sensibility levels in different applications. Therefore, *HasSecurityLevel* is identified as a facet of *OnlinePayment*, and can be High or Normal. *Payment* is a variable feature, which has two variants of *PayByVisa* and *PayByPaypal*. This is a runtime variability, that means choice of the payment mode is determined at runtime by the user. Feature constraints are presented in the figure as configuration dependency relationships. From the constraints, we can know when the security level of *OnlinePayment* is determined to be high, *Log* and *UserInform(SMS)* must be involved in the application.

3 Conceptual Model of the Adaptive Component

In order to realize feature-oriented customization on the architecture and component level, the SPL implementation technology should satisfy two requirements: flexible enough to support customization on system structure and behavior; well established feature mapping to reflect feature customization on the implementation level. The feature-oriented adaptive component model proposed in our previous work[5] provides both the flexibility for structure/behavior adaptation and feature-oriented implementation mapping. Therefore, we introduce the adaptive component model in our SPL development method.

3.1 Structural Model of the Adaptive Component

Structural model of the adaptive component is presented in Fig.3. It has a microstructure in which the control flow is separated from the component implementation, specified and performed on all the internal and external ports[5]. It can be seen from Fig.3 that a component is composed of the implementation body, several internal/external ports and a control center. The implementation body encapsulates computation logic of the component and exposes some internal ports for the control center. The control center is separated from the component implementation and controls the interactions on all the ports. Ports represent the interfacing points for internal and external interactions. Three kinds of ports are identified[5]:

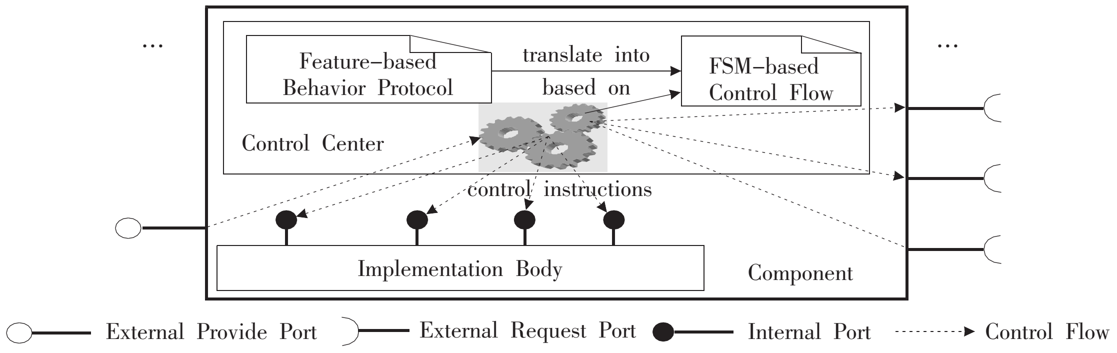


Fig.3 Feature-oriented adaptive component model

图3 面向特征的适应性构件模型

- **External provide ports**. They are the provide ports of the component and entries for other components to startup the control flow. Usually, an external request received through the provide ports will cause a series of interactions on the internal ports or request ports of the component.

- **External request ports**. They are the request ports of the component and entries for the control flow to involve external service providers. Usually, the control center may request external services through the request ports to complete a service-providing process.

- **Internal ports**. They are internal functional interfaces provided by the implementation body to fulfill external requests. Usually, after an external request is received, one or more internal ports will be involved in an execution of the control flow to complete the whole service process.

In the adaptive component model, control center is separated from the component implementation to enforce component-level coordination on its external and internal ports according to the behavioral protocol, which will be translated into runtime control flow described by state machine. At runtime, usually, the control center will be activated by requests of other components on a provide port, then it will perform a series of interactions on internal ports and external request ports on the runtime control flow.

Behavioral protocol in our method forms the basis of feature-oriented behavior customization. Corresponding to the two processes of domain engineering and application engineering, each domain component has a static behavioral protocol and multiple application protocols for different applications. Static behavioral protocol is the compl ete description of the component behaviors by interaction events on all its external and internal ports, including interaction process and feature semantics of each event. In our method, we adopt a CSP[1]-like process language with feature semantics in the behavioral protocol. Each event in the static protocol is annotated with feature semantics based on the domain feature model. Then, requirement-level feature customization can be mapped to the adaptation of component behaviors.

Static behavioral protocol is specified in domain engineering, denoting the capability of the component. It embodies domain-level variations on both interaction process and events. These variations will be resolved in application engineering. Application protocol is application-specific behavioral protocol. In application engineering, static behavioral protocol is customized to application behavioral protocol along with requirement-level feature customization. Then, application architecture can be derived after all the component behavioral protocols are resolved: inter-component interaction structure is determined since each optional interaction is resolved to be bound or removed; interaction process and semantics are determined with the component behavioral protocols.

Fig.4 gives the example of the *OnlinePayment* component and its interactions with other components. The component provides the service of *OnlinePayment* via its external provide port. The service is implemented by a process involving two internal ports of *Log* and *BillGen* and three external request ports of *Payment*, *UserInform* and *LuckyDraw*. External provide port *OnlinePayment* is activated by the request message from *OrderMgt*'s ex-

[1] CSP:Communicating Sequential Processes,it was first described in a paper by C. A. R. Hoare in 1978.

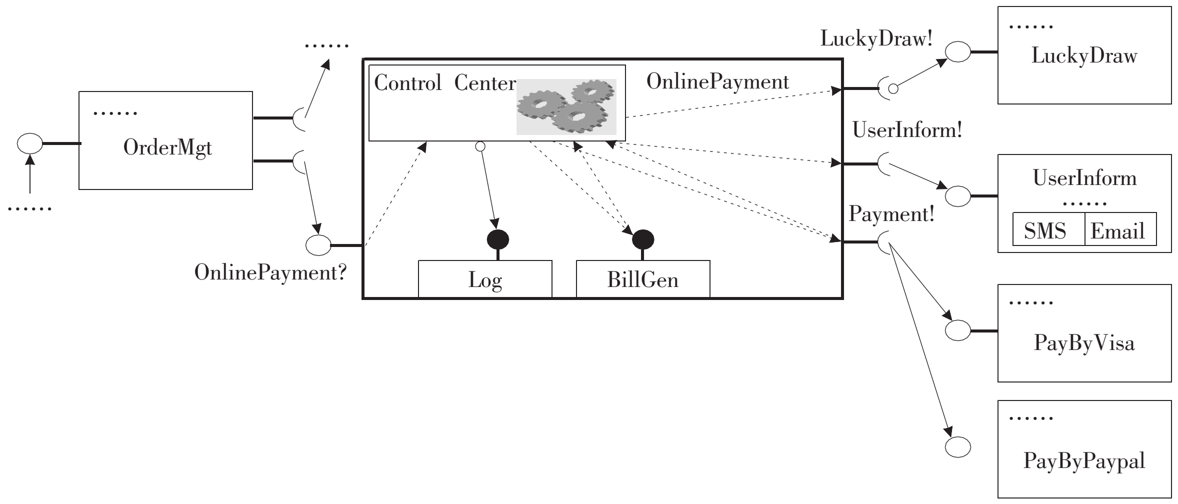


Fig.4 Example of interaction between components

图 4 构件交互示例

ternal request port. Control center in component *OnlinePayment* will coordinate the interactions on all the internal and external ports according to behavioral protocol. Fig.4 is a static view of the SPL architecture and component behaviors, which will be customized in application development. For example, interaction with the *LuckyDraw* component is optional, which can be bound or removed in specific application. Therefore, SPL architecture in our method can be represented by the external connections involving variations between components.

Detailed introductions on the implementation of the adaptive component are presented in section 4, including the grammar of behavioral protocol and implementation infrastructure of the control center, etc.

3.2 Why Introduce the Adaptive Component Model in SPL Development

The micro control structure and feature-oriented behavioral protocol are the two major characteristics of the adaptive component model. They provide both the flexibility for structure/behavior adaptation and feature-oriented implementation mapping, so facilitate the feature-oriented application development.

The micro structure of the adaptive component distinguishes internal por ts from traditional recognized provide/request ports. Internal ports represent internal functional implementations that may be adapted to feature customization but are only part of the external-visible component service. For example, the internal port *Log* in Fig.4 is implemented inside the component *OnlinePayment* and is invisible to external components. It is an optional part of the *OnlinePayment* service, but the binding decision is hidden to outside. From the feature constraints in Fig.2, we can see binding state of *Log* can be determined by the semantics of the external provide port *OnlinePayment* according to the security level. It is similar to the internal and external variability discussed in [9]: external variability is the variability of domain artefacts that is visible to customers; internal variability is the variability of domain artefacts that is hidden from customers. Hiding variability from the customer (internal variability) leads to reduced complexity to be considered in application engineering[9] and ensure the consistency of customization.

On the other hand, the microstructure eases the synchronized and asynchronous int eractions between components[5]. In traditional components, a single method may contain a synchronized request to other components. Then the request port is hard to be separated from the implementation and composed with other components. While in our component model, the component implementation is divided into several independent fragments, and none of them contains external requests. So the control center can perform the control logic and coordinate synchronized interactions on the explicit level of external and internal ports.

In the adaptive component model, coordination is separated from computation and implemented in the control center. In order to support the behavioral customization in application development, we encode the variability-involved component behaviors in static the behavioral protocols, which are processes of action semantics based on the domain feature model[5]. Based on static behavioral protocol, application-specific component behavioral can be derived following the feature customizations by process reorganization and event semantics adaptations. Furthermore, the behavioral protocol is formal enough to support automatic consistency validation of the overall behaviors.

3.3 Component Specification

In our method, component specification (including port semantic, port implementation) is described in XML. The XML schema for the component specification is presented in Fig.5. Elements in

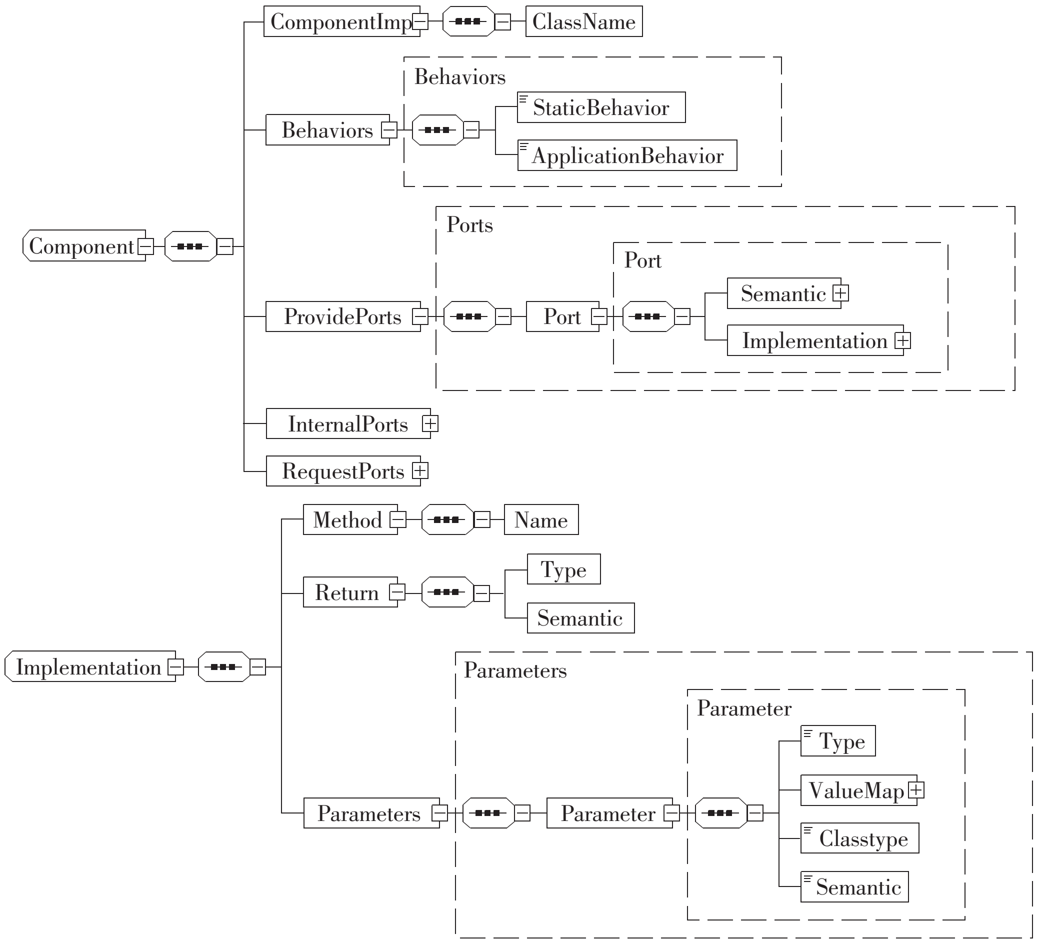


Fig.5 XML schema of component specification

图 5 构件规约的 XML Schema

the specification are descried as below.

ComponentImpl. This element defines the mapping between conceptual component and its physical implementation (Jar or Class).

Behavior. Component behavioral protocol in our method is the separated coordination logic of the component. There two types of behavioral protocol, static behavioral protocol and application-specified behavioral protocol. The static behavioral protocol is specified in domain design and customized in application development, and then the application-specific behavioral protocol will be enforced by the implementation infrastructure of the adaptive component. Behavioral protocol is written by a CSP-like language which will be described in detail in next section.

Port. A port in component specification consists of two parts: port semantics and its implementation description.

Port Semantics. In component definition, each port is annotated with feature semantics corresponding to action in feature model. The port semantics describe the service ability or request requirement of component ports. The port semantics are depicted by their facets and sub-actions. A port corresponding to action in feature model can have multiple facets and corresponding value for each facet. During application behavior computation, port semantics are required to decide which actual action according to component port would be involved. More detailed definition for action semantic could be referred to our pervious works on Ontology-Based Feature modeling[7].

Port Implementation Info. Not only port semantics should be defined to perform behavior composition but also implementation information should be denoted for component execution and adaptation. The elements (method, return value and parameters) of port implementation description are mapping to their corresponding Java elements. There are two kinds of parameters in our port description: data parameter and control parameter. Data parameter is mapping to a Java data object (JavaBean). How data parameter is consumed inside component is transparent to developers. Control parameter determines the action semantic of the port. When customization is done (runtime or buildtime), control parameters will be replaced by a real instant value according to port semantics. So a mapping between port semantic and control parameter should also be provided.

Fig.6 is a segment of the specification of port *UserInform*. In this specification we could know that *UserInform* is able to provide two functions: Inform users by Email or SMS. Which function will be provided is decided by the control parameter's value. When the parameter *type* is appointed as *false*, the port will provide *SMSInform* function.

4 Implementation Infrastructure of the Adaptive Component

In order to implement the final application derivation on the program level, implementation infrastructure of the adaptive component model must be provided. This section introduces implementation infrastructure of the adaptive component, including component implementation model, behavioral protocol specification and translation, and control center implementation.

4.1 Implementation Model

Fig.7 illustrates the implementation model of the adaptive component. It is composed of a behavior interpreter, a container storing state machine, a control center, a parameter adaptor and several internal ports with implementation bodies. The core part of the implementation model is the control center. At runtime, usually, the control center will be activated by requests of other components on a provide port, then it will perform a series of interactions on internal ports and external request ports on the runtime control flow. When control center is activated, component will be initialized. During initial process, behavior interpreter will get text-based application-specific behavioral protocol from component specification files and translate the customized

```
<ProvidePort name="UserInform">
                  ……
    <Implementation>
        <Method name="inform"/>
        <Return semantic="" type="void"/>
        <Parameters>
        <Parameter name="order" classtype="edu.fudan.se.sample.Order" semantic="Order" type="data"/>
            <Parameter name="type" classtype="bool" facet="HasMsgType" type="control">
                <ValueMap>
                        <Value facetvalue="sms" instantvalue="false"/>
                        <Value facetvalue="email" instantvalue="true"/>
                </ValueMap>
            </Parameter>
        </Parameters>
    </Implementation>
</ProvidePort >
```

Fig.6 A segment of port inform

图 6 端口信息片段

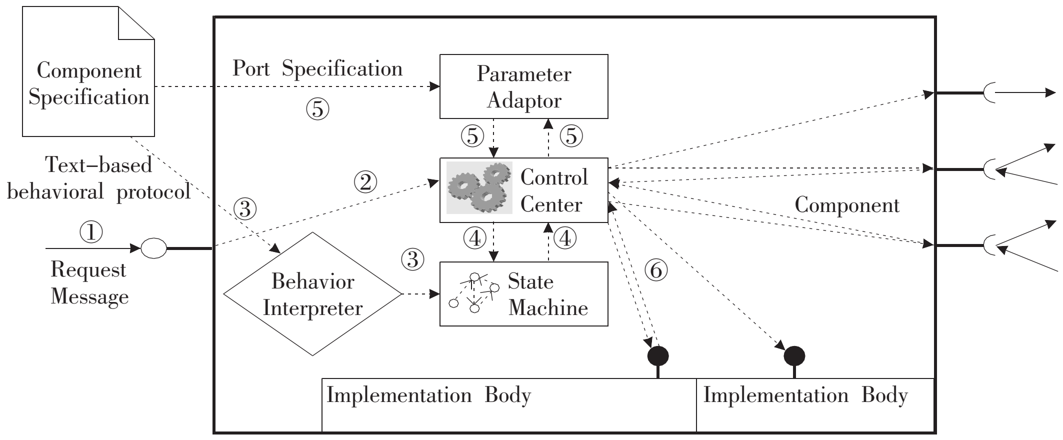


Fig.7 Implementation model of the adaptive component

图 7 适应性构件的实现模型

application-specific behavioral protocol to state machine. Our behavior interpreter is implemented by JavaCC[2] that will be explained in detail in section 4.2. Then according to state machine, control center decides which action should be done on current state. This means control center will employ corresponding port to fulfill certain function. Before port interaction, parameter adaptor will appoint the instance value of control parameter by the parameter mapping definition in component specification.

4.2 Behavioral Protocol Specification and Translation

In our method, static behavioral protocol of each component is specified in domain engineering and customized in application development. We define a CSP-like behavioral language for behavior specification with feature-based action semantics, which facilitates feature-oriented behavioral customization. The CFG (Context Free Grammar) of the behavioral language and its corresponding JavaCC Implementation are defined in Fig.8, and major elements are introduced as follows.

- **Action**: The basic unit of the behavior specification is an action. An action represents an atomic interaction on external or internal ports. The notation "!" or "?" following each action represents the direction of message sending, in which "!" denotes message sending and "?" denotes message receiving. For example, "billGen?" represents that an input message *billGen* is expected to be received on certain port.

- **<ECHOICE>** is the external choice. It means the choice of which action will be happened is determined by external influence such as values of parameter.

- **<NEXT>** represents the transition between actions corresponding to '->' in test-based behavioral protocol description.

- **Process**: Process is the execution flow of actions. Processes are described by combining actions and other simpler processes. The simplest process is NULL, the process that represents an action does nothing.

The terminals used in our CFG and its corresponding user-written behavioral protocol symbol, JavaCC symbol and their semantics meanings are listed in Table 1.

CFG	A->S? A->S! P->PNP P->A P->ANA P->PN(P)%%(P)

JavaCC Grammar	TOKEN: { <NEXT: "->"> | <INPUT: "?"> | <OUTPUT: "! " >
	| <ECHOICE: "%%"> | <LPAREN: "("> | <RPAREN: ")"> |
	<STRING : (["A"-"Z", "a"-"z", "0"-"9"])+> }
	Action( ) { (<STRING>) (<INPUT> | <OUTPUT>) }
	Process( ) { Process( ) <NEXT> Process( ) }
	Process( ) { (Action( ) (<NEXT> (Action( ) | (<LPAREN>Process( )<RPAREN>
	<ECHOICE> <LPAREN>Process( )<RPAREN>)))*) }

Fig.8 CSP-like behavioral protocol grammar

图 8 类 CSP 行为协议语法

[2] https://javacc.dev.java.net/

Table 1 Terminal used in CFG and its corresponding user-written behavioral protocol symbol, JavaCC symbol and their semantics meanings

表 1 上下文无关语法中使用的符号及其对应的行为协议符号、JavaCC 符号和语义

Terminal in CFG	user-written behavioral protocol symbol	JavaCC symbol	Semantics
S	["A"-"Z", "a"-"z","0"-"9"]	<STRING>	operation semantics
?	?	<INPUT>	message sending
!	!	<OUTPUT>	message receiving
N	->	<NEXT>	action transition
(	(	<LPAREN>	
)	)	<RPAREN>	
E	%%	<ECHOICE>	external choice

Our control center is based on state machine because state machine is more easily to handle relative to text-based behavioral protocol. A behavioral protocol interpreter is provided to translate text-based behavioral protocol to state machine. Our interpreter is based on JavaCC. JavaCC stands for "the Java Compiler Compiler"; it is a parser generator and lexical analyzer generator. JavaCC will read a description of a language and generate code, written in Java, that will read and analyze that language. JavaCC is particularly useful when you have to write code to deal with an input language which has a complex structure; in that case, hand-crafting an input module without the help of a parser generator can be a difficult job. This technology originated to make programming language implementation easier -hence the term "compiler compiler"- but make no mistake that JavaCC is of use only to programming language implementors.

An example behavioral protocol of the *Online-Payment* component in the online shopping domain is depicted in Fig.9. It is the static protocol which can be specified in domain design.

After translation, state machine information is stored in a container inside component. A state machine is composed of a set of states. A state corresponding to action of behavioral protocol represents the status in the process of interactions mediation. A state has some transitions. When a transition is fired, state machine changes current state from one state to another. If a state has two or more than two outgoing transitions, each transition from the state should have a guard condition to help event management decide which event would happen. This guard condition is defined by external choice mentioned above. A triggered event will cause the firing of the transition.

```
OnlinePayment=OnlinPayment?->BillGen! ->BillGen?->Payment! ->Payment?->
                    ->LuckyDraw! ->LuckyDraw?->log! ->UserInform!
Static Behavior
```

Fig.9 Static behavioral protocol of OnlinePayment in sample e-business system

图 9 OnlinePayment 的静态行为协议

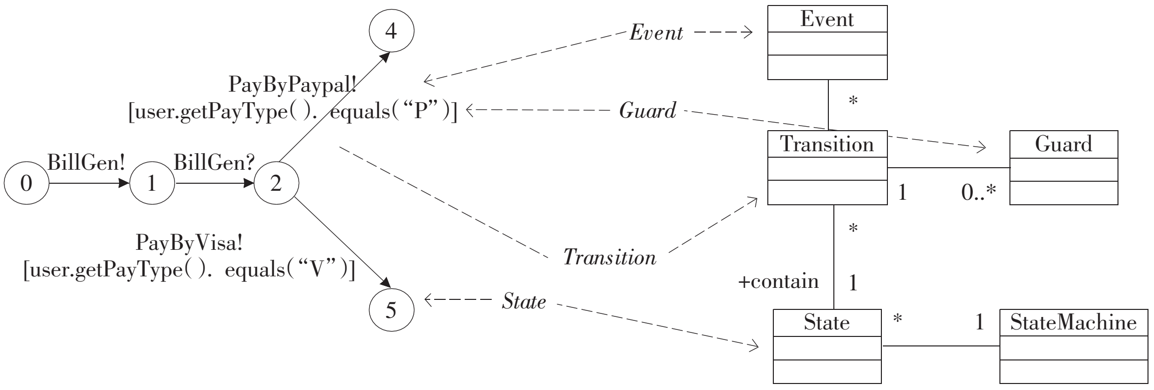


Fig.10 Part of the meta model of state machine

图 10 状态机元模型的片段

4.3 Implementation of the Control Center

Beyond behavior interpreter and parameter adaptor, we also have to care for the implementation of control center which drives the component execution, i.e. the method that makes state machine progress by selecting which transition must be triggered according to events and the current state. Our state machine progression model is illustrated in Fig.11. Class *ControlCenter* is the facade of control center. Method *step*( ) is a run-to-completion[10] procedure. In our approach the default time model is synchronous. But it is easily extended to be asynchronous if we apply an observer pattern. In the model, each event is either a send-request procedure or a receive-result procedure. *ControlCenter* call *step*( ) to progress. In synchronous situation, if the following event is receive-result event corresponding to current send-request event, these two events would be merged to a method call with return value. In asynchronous situation, when an receive-result event occurs, *ControlCenter* will be notified and method *step*( ) will be called to progress. Send-request event will directly execute.

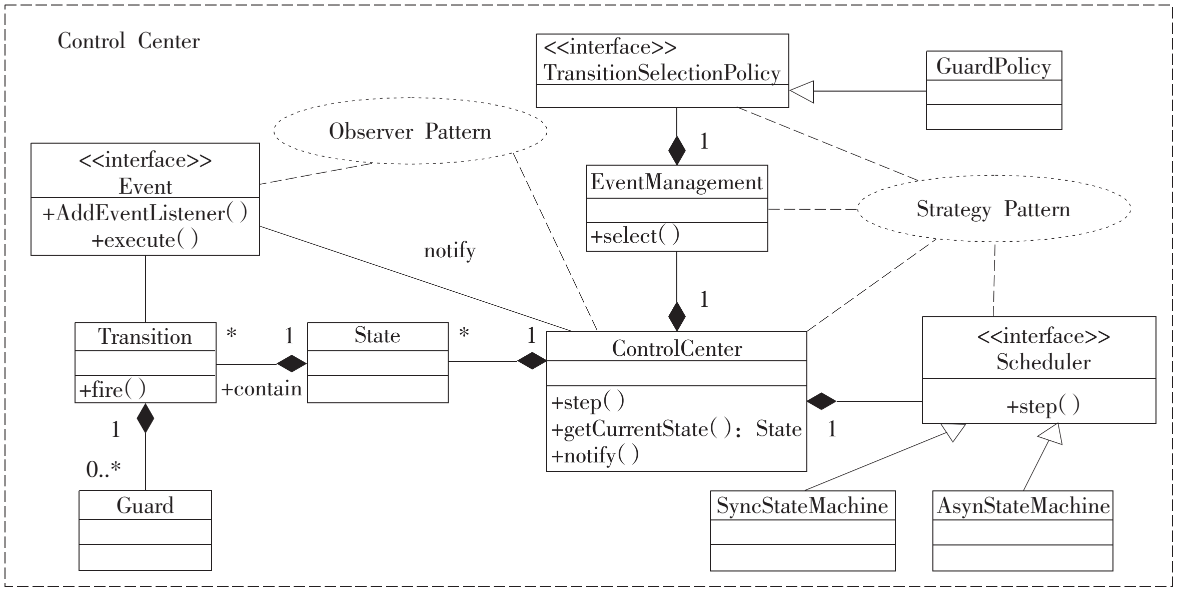


Fig.11 State machine progression

图 11 状态机的行进

5 Feature–Oriented Application Derivation

5.1 Application Behavioral Protocol Computation

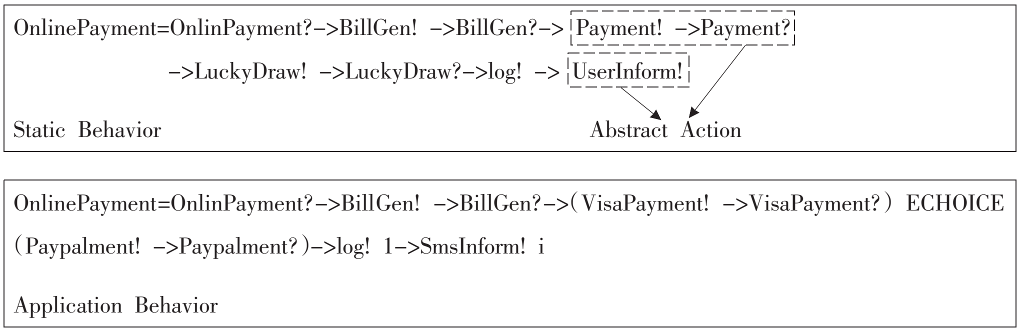
The domain model needs to be customized in application engineering. In domain engineering we develop reusable component assets which have static behavioral protocol inside. Static behavioral protocol is the complete description of the functions implemented by component and its execution path. It stands for the capacity of component self. In static behavioral protocol, optional feature are directly defined by actual actions corresponding to certain component ports in execute path. Variant actions are represented by abstract actions in static behavioral protocol. These actions may be replaced by an actual action in application behavioral protocol if the binding time is buildtime after tailoring. Or they would exist as an external choice of candidates in application behavioral protocol and be decided in runtime by parameters. The external choice may be explicitly decided by control parameter or implicitly decided by data parameter. For example, the choice of Payment could be decided by control parameter *payType*. It may also be decided by computation result of the data parameter *userinfo* if we suppose that certain customers are enforced to use one payment type. The explicit external choice decision can be resolved by control parameter mapping. But the implicit external choice decision should be manually written as a Java expression by developers. How to ease this implicit situation definition is one of our feature works. We provide an algorithm to compute application behavioral protocol as follow:

```
// a: action, S: static behavioral protocol
INPUT: feature model, static behavioral protocol
OUTPUT: application behavioral protocol
for all $a \in S$ do
begin
    if the type of $a$ is "optional" then
    if the bind time of $a$ is "buildtime" then
      Add $a$ to application behavior;
      else if the bind time of $a$ is "runtime" then
      Add $a$ with an external choice to application;
      Define condition of the external choice;
      end if
      else if the type $a$ is "specialized" then
      if the bind time of $a$ is "buildtime" then
        Add corresponding actual action to application
        behavior;
      else if the bind time of $a$ is "runtime" then
      Add corresponding actual action with an external
      choice;
      Define condition of the external choice;
      end if
    else
      Add current action to application behavior;
    end if
end for
```

Consider the static behavioral protocol and application behavioral protocol shown in Fig.12. The abstract action Payment is decided by facet *PayType* at runtime. So it is replaced by the external choice of two corresponding actual action. In our application, we assume the security level is customized as high at buildtime. So the optional behavior *Log* is selected to be represented in execution path and the action semantics of *UserInform* is specified as *SmsInform*.

5.2 Behavioral Protocol Validating

Application behaviors of components involved are assembled together to form the ap plication ar–



```
OnlinePayment=OnlinPayment?->BillGen! ->BillGen?->Payment! ->Payment?
              ->LuckyDraw! ->LuckyDraw?->log! ->UserInform!
```

```
OnlinePayment=OnlinPayment?->BillGen! ->BillGen?->(VisaPayment! ->VisaPayment?) ECHOICE
(Paypalment! ->Paypalment?)->log! 1->SmsInform! i
```

Fig.12 Customized application-specific behavioral protocol

图 12 定制后的应用行为协议

chitecture of components. This architecture is automatically computed by static behavioral protocol and feature model. Therefore to insure deadlock-free and policy-satisfying, validation mechanism should be provided. We integrated LTSA[3] into our tool because the model after behavior translation in our method has the same semantic foundation as LTSA tool. LTSA (Labelled Transition System Analyser) is a verification tool for concurrent systems. It mechanically checks that the specification of a concurrent system satisfies the properties required of its behavior. In addition, LTSA supports specification animation to facilitate interactive exploration of system behavior. A system in LTSA is modeled as a set of interacting finite state machines. The properties required of the system are also modeled as state machines. LTSA performs compositional reachability analysis to exhaustively search for violations of the desired properties.

6 The Supporting Tool

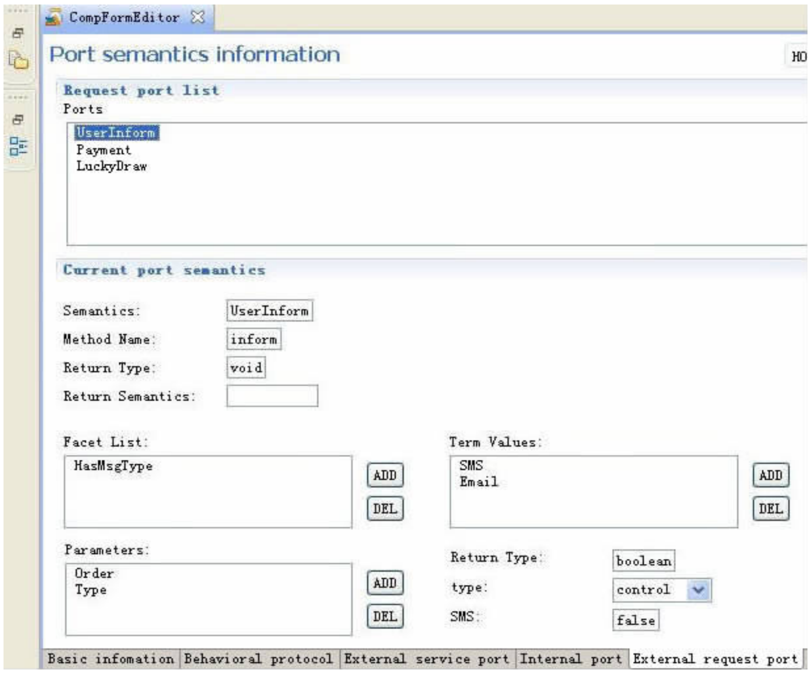
A prototype of the feature-oriented and adaptive-component-based SPL design and application development tool has been implemented to support our method. It is a RCP (Rich Client Platform) application based on Eclipse Platform. In order to acquire the domain feature model, it is integrated with OntoFeature, the ontology-based feature modeling tool developed in our previous work[7], by importing XML-based feature model description. The tool provides supports for both domain architecture/component design and feature-oriented application derivation.

Based on the domain feature model, the tool allows users to define specifications for all the domain components by the component specification editor, including port semantics, static behavioral protocol and implementation information (see Fig.13). In our method, each domain component should inherit the base Class of *Component* we provide. Component developers should implement computation logic for each internal port using a standard Java method. These information of implementation class for each component and implementation method for each internal port should be specified in component implementation information.

In application development, customization will first be performed on the feature level. After that, behavioral protocols of all the components are adapted

[3] http://www.doc.ic.ac.uk/ltsa/

correspondingly. Then, all the involved component implementation and component specification with customized behavioral protocols can be packaged and released. This process of application derivation is supported by the application customization assistant in the supporting tool (see Fig.14). In the derivation process, the user is requested to resolve the application-development-time feature variations, e.g. determine the facet value for variable facets or determine which variant is bound for variable feature.



Ports:
UserInform
Payment
LuckyDraw

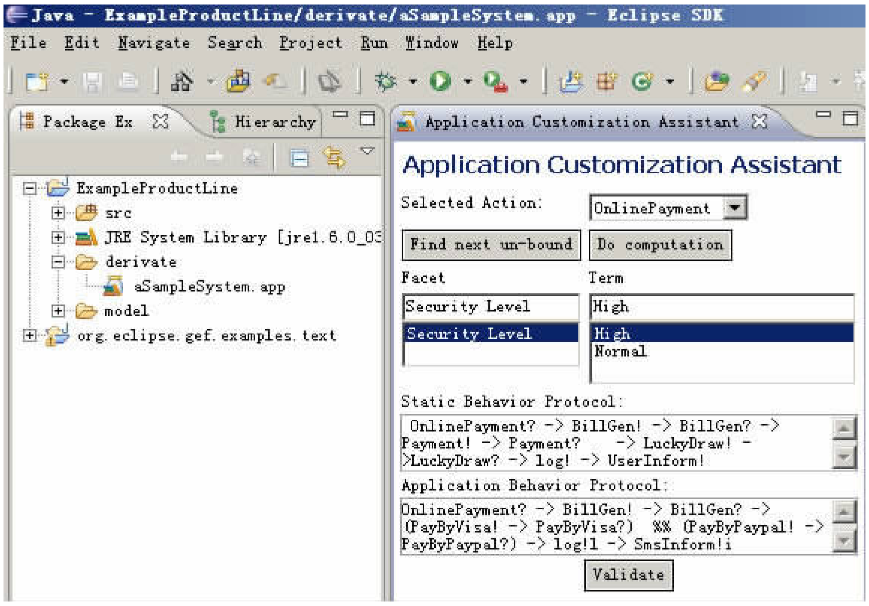
Semantics: UserInform
Method Name: inform
Return Type: void
Return Semantics:

Facet List: HasMsgType
Term Values: SMS Email

Parameters: Order Type
Return Type: boolean
type: control
SMS: false

Fig.13 A snapshot of component specification editor

图 13 构件规约编辑器的截图


Java - ExampleProductLine/derivate/aSampleSystem.app - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Package Ex
Hierarchy
Application Customization Assistant
Application Customization Assistant
Find next un-bound
Do computation
Validate

```
ExampleProductLine
  src
  JRE System Library [jre1.6.0_03
  derivate
    aSampleSystem.app
  model
org.eclipse.gef.examples.text
```

Selected Action: OnlinePayment

Facet	Term
Security Level	High
Security Level	High
	Normal

Static Behavior Protocol:

```
 OnlinePayment? -> BillGen! -> BillGen? ->
Payment! -> Payment?    -> LuckyDraw! -
>LuckyDraw? -> log! -> UserInform!
```

Application Behavior Protocol:

```
OnlinePayment? -> BillGen! -> BillGen? ->
(PayByVisa! -> PayByVisa?)  %% (PayByPaypal! ->
PayByPaypal?) -> log!l -> SmsInform!i
```

Fig.14 A snapshot of application customization assistant

图 14 应用定制辅助器的截图

After feature-level customization, application-specific protocol can be generated for each component, and inter-component interaction structure and semantics are also determined.

In order to ensure the consistency of the application architecture, structure-level and syntax-level validations will be performed. On the structure level, all the behavior protocols are validated together to avoid interaction deadlock by the LSTA tool. On the syntax level, all the application-specific behavioral protocols are checked by JavaCC to avoid syntax errors. Finally, the application implementation is ready, and can be packaged and released together with the infrastructure of the adaptive component.

7 Related Works and Discussions

This section describes related research and discusses some benefits and shortcomings of our approach.

Markus Voelter and Iris Groher proposed an Aspect-Oriented and Model-Driven method to implement product line[11]. The approach described in their paper facilitates variability implementation, management, and tracing from architectural modeling to implementation of product lines by integrating both model-driven (MDSD) and aspect-oriented software development (AOSD). The works of us have the same goal (construct the final product by configuring and tailoring a high-level model) but different approaches. Their focus is on static structure of SPL. But they do not describe business-process-level customization.

Klaus Pohl[9] proposed a framework for software product line engineering. In their approach, an orthogonal variability model is used to bind variability in domain artefacts consistently in the entire application. They provided a guideline for manual variability binding; but no automatic mechanism or development tool is given.

Chong-Mok Park[12] divided the type of variability in components into external and internal. An external variability is a variability that is externally visible at the level of components and their connections, whereas an internal variability is one that exists internally within the implementations of the components and not visible at the level of components. They proposed a component model which supports both decomposition and composition of CE software product lines. Both of our method supports the separation of multiple levels of abstraction. But their method addressed issues for supporting decomposition and composition of consumer electronics software product lines.

Our approach takes implementation mappings between requirement-level feature model and program-level component as the key driver for developing application system from core assets. But in practice people may argue about how to deal with legacy component or the fussy development in domain engineering. And we also have found some benefits that we have not taken into account before. We will discuss about them in this section as follows:

– Deal with Legacy Component and COTS Component

Besides built in-house core assets, many components, interfaces, and other software assets in product line are not created anew. Instead, they are derived from the legacy systems. Another kind of components is COTS components available in the market. So people may argue that how to introduce these kinds of components to our component model. Aimming at this question, we have two solutions: using them as a particular case without internal ports or refactoring them. Obviously the first solution is suitable to COTS components. COTS components can easily wrapped and their behavioral protocol can be defined as a simply receive action followed by a send action (stand for the service port) without internal port actions. If the second solution is used, costs should be considered. But we think refactoring is an effective way to improve the design of existing components. And how to help refactoring legacy components is one of our feature works.

– Dynamic product reconfiguration

Dynamic product reconfiguration refers to making changes to a deployed product configuration while a system is running. Recently, there have been increasing demands for dynamic product reconfiguration in various application areas (e.g., ubiquitous computing, self-healing systems, etc.)[13]. in our method, components are be organized by their behavioral protocols and driven to provide service according to state machine inside. So it is easy to introduce dynamic mechanism supporting for runtime dynamic evolution and reconfiguration if we give control center of the component the ability to change its application behavioral protocol in runtime.

– Development Process

In our method, most steps in application engineering are automatically handled such as application architecture computation. These automatic computation is based on a precisely defined feature model and component specification. Compared with traditional approach, obviously our method needs more works to be done in domain engineering. Integrating behavioral protocol into component model brings flexibility to change the business process. But it also demands for more challenges in developing component. How to refactor legacy component will also be a problem. Nevertheless, we think it is worthwhile to spend more energy in domain engineering because it can support to construct the final product by configuring and tailoring a high-level model.

– Implementation mappings

In our method, the customization is based on the semantic information in the feature model, together with the adaptive component model. We assume that SPL developers follow certain naming conventions when defining the component model and feature model (semantics name in these two models should be same). Furthermore, we assume that the parameters in internal ports should be subset of the parameters in service port. Due to this restriction, implementation mapping is a little coarse-grained. Our approach can't handle the situation that the parameter's value of an internal port is the output of another internal computation although this situation is conventional. An ideal solution is to construct a precise ontology model to describe all entities of the domain. Implementation mapping should be computed through ontology reasoning. But in the real world to construct this kind of ontology model is difficult. This would be another research work.

8 Conclusions and Future Works

This paper proposes a SPL design and implementation method based on the feature-oriented adaptive component model. The adaptive component model has the micro-structure of control center, implementation body (embodied by internal ports) and external ports. Coordination within component is separated from computation and enforced by the control center. This flexible micro-structure facilitates the component behavioral customization in application development. On the other hand, the component model has the feature-based action semantics with domain-level variability. Therefore, the component behavioral adaptation can be naturally related to feature-level customization. These two characteristics enable the feature-oriented implementation customization in application development. In order to help the SPL development practice, we provide implementation infrastructure for the adaptive component model and develop a supporting tool for component specification and application derivation.

The adoption of our adaptive component model in SPL development requests that all the business components are implemented on our infrastructure. This limits the application of COTS components and other legacy components in SPL development. How to integrate COTS in SPL development and how to help refactor legacy components are still to be further researched. On the other hand, the current tool can only provide basic support for SPL development. Configuration management and evolution management are expected to be integrated to form a systematical tool set for SPL development.

References:

[1] Clements, Paul, Northrop L. Software product lines practices and patterns[M]. New York: Addison Wesley, 2002: 1-13.

[2] Kang K C, Kim S, Lee J, et al. A feature-oriented reuse method with domain-specific reference architectures[J]. Annals of Software Engineering, 1998,5:143-168.

[3] Peng Xin, Shen Liwei, Zhao Wenyun. Feature implementation modeling based product derivation in software product line[C]//the 10th International Conference on Software Reuse, Beijing, China, 2008.

[4] Thiel S, Hein A. Systematic integration of variability into product line architecture design[C]//the 6th International Software Product Lines Conference, San Diego, USA, 2002: 130-153.

[5] Peng Xin, Wu Yijian, Zhao Wenyun. A feature-oriented adaptive component model for dynamic evolution[C]//the 11th European Conference on Software Maintenance and Reengineering, Amsterdam, Netherlands, 2007:49-57.

[6] Zhang Wei, Mei Hong, Zhao Haiyan. Feature-driven requirement dependency analysis and high-level software design[J]. Requirements Engineering, 2006,11(3):205-220.

[7] Peng Xin, Zhao Wenyun, Xue Yunjiao, et al. Ontology-based feature modeling and application-oriented tailoring[C]//the 9th International Conference on Software Reuse, Torino, Italy, 2006:87-100.

[8] Bechhofer S. Owl web ontology language reference[EB/OL]. [2004]. http://www.w3.org/TR/owl-ref/.

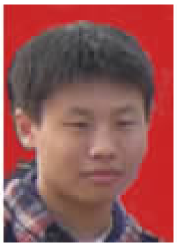
[9] Pohl K, Bockle G, van der Linden F. Software product line engineering foundations, principles and techniques[M]. Berlin Heidelberg: Springer-Verlag, 2005:68-70.

[10] David H, Amnon N. The STATEMATE semantics of statecharts[J]. ACM Transactions on Software Engineering and Methodology, 1996,5(4):293-333.

[11] Voelter M, Groher I. Product line implementation using aspect-oriented and model-driven software development[C]// the 11th International Software Product Lines Conference, Kyoto, Japan, 2007:233-242.

[12] Park Chong-Mok, Hong S, Son Kyoung-Ho, et al. A component model supporting decomposition and composition of consumer electronics software product lines[C]//the 11th International Software Product Lines Conference, Kyoto, Japan, 2007:181-190.

[13] Lee J, Kang K C. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering[C]//the 10th International Software Product Lines Conference, Baltimore, Maryland, USA, 2006:131-140.



YANG Yiming received the B.S. degree from Harbin Institute of Technology in 2005. He is a Ph.D. candidate at Fudan University. His research interests include software product line and software reengineering, etc.
杨益明(1983-),男,上海市人,复旦大学计算机科学与工程系博士研究生,主要研究方向包括软件产品线、软件再工程等。

PENG Xin was born in 1979. He received the B.S. and Ph.D. degrees in Computer Science from Fudan University, China, in 2001 and 2006 respectively. He is current an assistant professor at the Department of Computer Science and Engineering of Fudan University and a CCF member. His research interests include software product line, software architecture and software reengineering, etc.
彭鑫(1979-),男,博士,湖北黄冈人,复旦大学计算机科学与工程系讲师,CCF会员,2006年在复旦大学获得博士学位,主要研究方向包括软件产品线、软件再工程、软件体系结构与构件技术等。

ZHAO Wenyun was born in 1964. He received his B.S. degree in 1984 from Fudan University, then became a faculty member of Fudan University and received his M.S. degree in 1989, also from Fudan University, China. Now he is a professor and doctoral supervisor at the Department of Computer Science and Engineering of Fudan University and a senior CCF member. His main research area is software engineering. Currently, his research interests include software engineering and software development method. He leads the Software Engineering Lab of Fudan University.
赵文耘(1964-),男,江苏常熟人,复旦大学计算机科学与工程系教授、博士生导师,CCF高级会员,现任复旦大学计算机科学与工程系软件工程实验室主任,主要研究方向包括软件工程和软件开发方法。