A Feature-Oriented Adaptive Component Model for Dynamic Evolution

Xin Peng, Yijian Wu, Wenyun Zhao
Computer Science and Engineering Department, Fudan University, Shanghai, China
{pengxin, wuyijian, wyzhao}@fudan.edu.cn

Abstract

Dynamic adaptation has been an essential requirement for more and more business systems. Some research works have focused on the structural or behavioral changes of adaptive programs. There are also some works on adaptive components, with the emphasis on separation between control flow and basic functions of components. In these works, a business model for the domain is always missing, so a comprehensible business view of adaptations is unavailable for the user. In this paper, we propose a feature-oriented adaptive component model, which introduces the ontology-based feature model proposed in our previous work on feature-based domain modeling to provide both the business view for the user and adaptation basis for the system. Furthermore, the ontology-based model provides unambiguous terminology for both the business view and the component specification, which ensures the consistency between them. This feature-oriented adaptive component model has another characteristic of the micro control flow within the component, which enables the adaptation of the component behavior, including interaction sequence and style. The adaptive component model has been applied in our intelligent connector based framework for dynamic architecture, so a case study on the adaptive version of JPS (Java Pet Store) is illustrated to show the advantages of the component model.

Keywords: component model, adaptive component, feature, dynamic architecture, business semantics, dynamic evolution

1. Introduction

More and more software systems are required to flexibly and quickly adapt to the changing business context and strategies, so as to improve competition of the enterprise. This requirement drives the research on dynamically adaptive systems (e.g. [1], [2], [3]), which are implemented by dynamic architecture in component-based systems. Adaptive systems are generally more difficult to specify, verify, and validate due to their high complexity [3]. Formal specification is one way to support the development of correct and robust dynamic software architectures, which involves the specification of the architectural structure of a system, the architectural reconfiguration of a system, and usually the behavior of a system [1]. However, most of the current works focus on the structural changes of adaptive programs (e.g. the graph-based and ADL-based approaches [3]). Generally, these approaches have taken a macro view of dynamic adaptation: specifying the overall architecture and considering dynamic changes of the structure in various scenarios. In this paper, we focus on a different micro view of adaptations, i.e. adaptive component.

An adaptive component is a component that is able to adapt its behavior to different execution contexts [4]. From the view of internal structure, an adaptive component can be seen as a formal embodiment of an archetype together with rules for combining (sub) component-archetypes into context-specific structures and/or functions [5]. For business components, the rules represent business strategies and are implemented by the controller part of the component behavior, which can be adapted to manipulate the variations of the business. This behavior-centric view has some other advantages for the implementation of dynamic adaptation. One is the finer granularity of control, which makes the adaptation more flexible. Another is that the overall consistency of the dynamically adaptive system can be validated by automated analysis of the behavior specifications of related components. The other is that the separation of business rules makes the adaptation more explicit and amenable to human inspection. Besides the better support for dynamic adaptations, adaptive component also can improve the reusability of the component, since the same component can be used under different business strategies in a configuration-based way.

In current works on adaptive component, the business view of the adaptation is always missing, so the user can only specify the adaptation policies in a physical way. So in this paper, we propose a feature-oriented adaptive component model, which introduces the ontology-based feature model proposed in our previous work on feature-based domain modeling [6] to provide both the business view for the user and adaptation basis for the system. The feature model specifies commonalities, variations and variation-related constraints of the business domain [6]. Commonalities prescribe the business rules which all the business components of the domain must conform to. Variations define the restricted variable space for implementations and adaptations of components, while there are still constraints. Thus, the specifications and adaptations of components can be interpreted and analyzed on the feature model. Furthermore, the ontology-based model provides unambiguous terminology for both the business view and the component specification, which ensures the consistency between them.

The remainder of the paper is organized as follows: Section 2 discusses related works and compares our work with them. Section 3 introduces the feature model and feature semantics. Section 4 gives an overview of the component model from the structural aspect, behavioral aspect and evolution process. Details of adaptations are discussed in section 5, including the behavior protocol, adaptations on protocols, composition analysis and implementation of the protocol adaptations. Section 6 introduces the implementation of the component model in our dynamic architecture framework, and finally Section 7 concludes the paper.

2. Related Works

Adaptive system attracts more and more attentions from software engineering research and practice. There have been some works on adaptive component and dynamic architecture. Zhang et al [3] propose a model-driven process to the development of dynamically adaptive programs, which separates the model specifying the non-adaptive behavior from the one specifying the adaptive behavior. Kim et al [7] propose to extend component property to business rules so that the business strategies implemented in the component can be redefined and executed at runtime. A rule component is introduced to separate and extract variable parts from business components so that the system can adapt flexibly to the changing business strategies. Boinot et al [4] present a declarative approach to program adaptation, which separates the adaptation declaration from the basic implementation of the main program. Lau et al [8] describe an approach which separates control flow from component code and supports the runtime generation of the control flow.

Similar to us, Mencl et al [9] also focus on the microstructure of the component. They introduce a minimalist component model to capture the structure of the controller part, coining the term *microcomponent* for the controller part elements. Zhu et al [2] implement self-adaptation by dynamically configuring the workflow of composite components. These approaches separate and specify the adaptive behavior from basic functions, supporting dynamic adaptation of component behaviors. However, the adaptation is directly performed on physical model of the component and a business view for the adaptation is missing. So the user can not view and specify the adaptations in a natural way and business-related constraints in adaptations are not taken into account.

Behavior protocol is the basis of our component model. There are several formal methods which have been adopted in behavior specification, such as the Petri-Net in [3], the widely used FSM (Finite State Machine), CSP (Communicating Sequential Processes) [10] in Wright [11] and other process algebras, etc. However, in these works business semantics of the interactions, including domain-specific relationships and variability-related constraints, are missing. So, business-oriented adaptations can not be supported on the protocol-level.

3. Feature Model and Feature Semantics

Feature models have been widely adopted in domain requirements capturing and specifying [6], which employ the element of “feature” to construct structural requirement models. 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.

3.1. Ontology-Based Feature Model

A feature is a coherent and identifiable bundle of system functionality, so operations with business semantics are basis of the feature model [6]. 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. Aiming at the missing of a formal foundation for feature modeling and validating, we propose an ontology-based feature model in our previous domain engineering work [6], which adopts the W3C recommended ontology language OWL [12] 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 [6] for detailed descriptions for the ontology-based feature model.

With these feature elements, we can describe the business model of a domain by defining all the business concepts and relationships among them. A segment of the feature model for the online-shopping domain is presented in figure 1. In the model, *OrderProcess* is described to be composed by *OrderTreatment*, *Inform*, *Log* and *Delivery*. The latter two are optional elements, which mean an execution of *OrderProcess* may or may not involve the operations of *Inform* and *Log*. *OrderTreatment* is defined as a variable feature with two sub-actions of *Distribution* and *ProductBooking*. The former means the mode of direct sale and distribution, while the latter means to accept the booking of the product and distribute later. The facet *HasSaleMode* is defined on *OrderProcess* to denote the sale mode. Constraints among features are represented by instances of *ConfigDepend*. For example, constraints on *Distribution* show that if *OrderProcess* takes the value *DirectSale* on the facet *OrderProcess* then *Distribution* should be bound. Furthermore, the binding of *Distribution* demands *Inform* to take the value *SMS* on the facet *HasMsgType* and depends on the binding of *Delivery*.

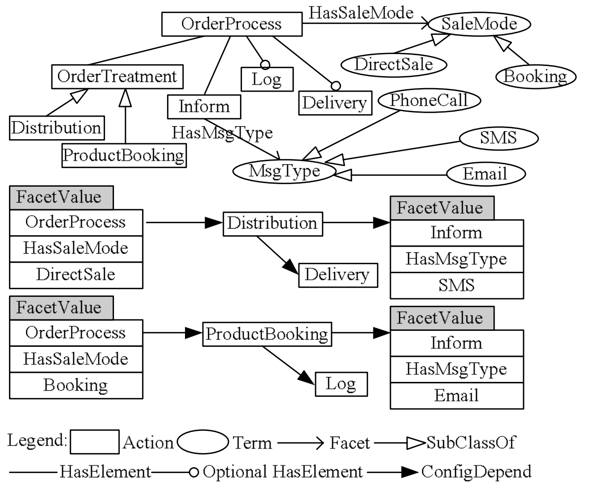


Figure 1. Fragment of the feature ontology of the online-shopping domain

3.2. Action Semantics

It can be seen from figure 1 that actions constitute the skeleton of the feature model, the semantics of which are depicted by their facets and sub-actions. The feature model specifies the general semantics for each action with restricted variations, which can be gradually refined in specific component implementation and runtime adaptation. For example, the facet *HasMsgType* reserves the variability of message types of result informing, which can be refined in runtime component adaptations.

These variations (denoted by the action semantics) construct the basis for component adaptations. We first define the facet set and facet value, and then give the definition of action semantics.

Definition 1 (Facet set and facet value): $\forall$*action*$\in$*Action*, *FacetSet(action)* denotes all the effective facets for action. And for $\forall$*facet*$\in$*FacetSet(action)*, *action.facet* represents the value of *action* taken on *facet*.

Definition 2 (Action semantics): An action semantics is denoted by a 2-tuple <*action*, *FValue*>, in which *action*$\in$*Action* and *FValue* is the function from

FacetSet(action) to *Term* representing the specific implementation decisions on the port. And for each *action*∈*Action*, we define a special facet value function *fm_value(action)* to denote the original facet values of *action* in the feature ontology.

In the feature model, abstract actions are refined by their sub-actions. Besides, the action semantics can also be further refined in component implementation or dynamic adaptations. This relationship of refinement can be defined as follow.

Definition 3 (Refinement of action semantics): The action semantics of $as_1=<a_1,fv_1>$ is the refinement of another action semantics $as_2=<a_2,fv_2>$, iff (a_1 *rdfs:subClassOf* a_2) $\wedge$ $\forall f\in FacetSet(a_2)\bullet(fv_1(f)$ *rdfs:subClassOf* $fv_2(f))$. The refinement relation can be depicted by $as_1\sqsubseteq as_2$.

In the feature model, variability is embodied by the optional *HasElement* relations between two actions and the abstract action with undetermined facet values or several sub-actions. *HasBindTime* is defined on *HasElement* or *Use* relationships to denote the binding time of the variation, which is the phase to decide the binding of a variation feature [6]. Typical binding times are *BuildTime* and *RunTime*. *BuildTime* denotes the phase of specific application development. *RunTime* embodies the dynamic binding of variations, which can be caused by the runtime operations of users or the dynamic adaptation of the system according to the business and physical context.

4. Component Model Overview

The component model of is presented in figure 2. From the aspect of structure, 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. From the aspect of adaptation, it provides the feature-oriented capability of dynamic evolution with the feature-based adaptive behavior protocol.

4.1. Structural Aspect

It can be seen from figure 2 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:

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.

These ports are also control points for coordination and adaptation, which are implemented by the control center. The control center maintains the control flow and performs the coordination according to it, which is similar to the exogenous connector discussed in [8] and enables the adaptation on component behaviors.

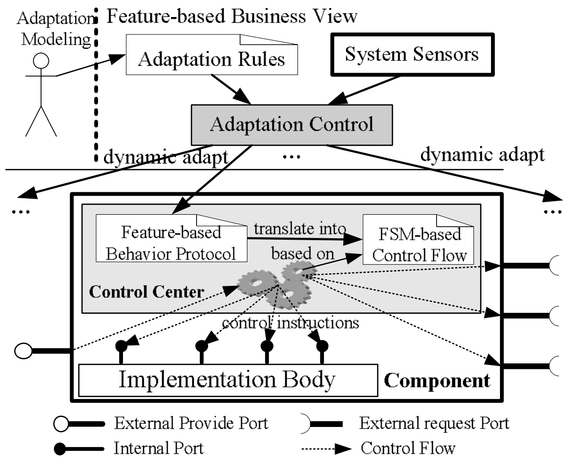


Figure 2. The component model

The microstructure also eases the synchronized and asynchronous interactions between components. 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 be assembled 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.

Figure 3 presents the structure model of an example component *OrderProcess*, which contains two external ports and three internal ports. The collective service of the component is *OrderProcess* provided on the provide port, which needs the external service requested on the request ports.

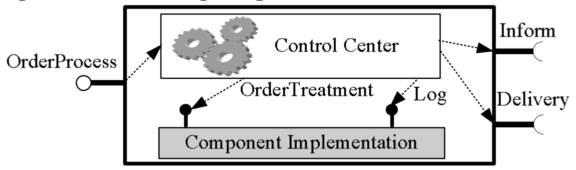


Figure 3. Structure model of component *OrderProcess*

4.2. Behavioral Aspect

The behavior protocol is the basis for behavioral adaptation and the control center also needs it to build the control flow (depicted in figure 2). So it should meet some requirements. First, it is formal enough to support the automated analysis of the overall behavioral consistency. Second, the protocol and its dynamic adaptation can be easily interpreted and implemented by the control center. Third, it can involve the feature semantics to enable feature-oriented adaptations.

In our component model, we adopt the CSP process notations with feature semantics for the behavior protocol. In the protocol, each event is annotated with feature semantics, so feature-oriented adaptation can be performed on the behavior protocol based on the feature binding analysis. The capability of composition analysis (as showed in Wright [11]) still remains and feature-related consistency rules are added to enable the behavior composition and analysis with business semantics. The behavior protocol still needs to be interpreted and executed by the component, which is implemented by the control center (see section 5.4).

4.3. Evolution Process

Adaptive components constitute a gray-box, strategy-black when their properties fit the context as, otherwise white [5]. In our component model, the behavior protocol and control flow of the component provide the gray-box like view for dynamic adaptations. The adaptation process is presented in figure 2, performed on the adaptive component. First, the business user can specify adaptation rules for the whole system on the feature-based business view. The rules can also be modified at runtime. Then system sensors monitor the runtime business status and report specific events concerned by certain rules. After that, the adaptation control module can analyze the events and related rules and decide necessary evolutions. The system-level evolutions are embodied by adaptations on behavior protocols of related components. Finally, the control center validates the new protocol, translates it into control flow and performs interaction coordination according to the new control flow.

In the whole process of evolution, adaptation and validation on behavior protocol is the key problem, which is discussed in section 5.2 and 5.3 How to implement the protocol adaptation on the component is another problem, which is touched in section 5.4.

5. Feature-oriented Dynamic Adaptation

5.1. Feature-based Behavior Protocol

In Wright [11], behaviors of components and connectors are modeled as CSP processes, which can be composed and analyzed according to CSP rules. However, interactions engaged in process expressions are general actions (e.g. “read”, “write”, “close”, etc), and domain-specific business semantics are missing. In order to enable the feature-oriented adaptations, we introduce action semantics (Definition 2) into CSP-based protocols. First, we define the port semantics.

Definition 4 (Port semantics): The semantics of each external or internal port of a component is a 2-tuple ps=<*action*,*FValue*> satisfying $action \in Action \wedge ps \subseteq$ <*action, fm_value(action)*>.We use the expression of “*facet*=*term*” to denote facet values in *FValue*. A special facet value “*optional*=*true*/*false*” is introduced to denote optional actions (e.g. *Delivery* in figure 5).

Port semantics is a kind of static semantics, which is the general function of the port and determined when development. In specific interaction context (e.g. embodied by the execution path on the control flow and the interaction parameters), a port may show a refined behavior of the port semantics, which is depicted by the operational semantics of the port.

Definition 5 (Operational semantics of ports): For a port with static semantics *ps*, semantics of each operational channel on it is represented by an action semantics *ocs*=<*a*, *fv*>, with the constraint of $ocs \subseteq ps$.

When a service port is connected with a request port by a connector as in figure 4, an interaction channel is established between them. This channel is called physical channel and several operational channels can be identified on it. For internal ports, the connection is established between an internal port (Service Port) and the control center (Request Port).

Definition 6 (Physical channels and operational channels): If a request/service port of a component is denoted by *p*, then the input channel and output channel are denoted by *p_in* and *p_out* respectively. Similarly, the input and output sub-channel of a operation channel *c* are denoted by *c_in* and *c_out* respectively. Suppose request/service port *p* is assembled with the service/request port *q* of another component, and a operational channel *pc* on *p* and a operational channel *qc* on *q* are identified to be connected, then *qc_out*/*pc_in* and *pc_out*/*qc_in* form two shared CSP channels respectively.

Operational channels on the same port represent all the possible operations which can be distinguished by current component and provided to or required from other components. In our method, data messages passed on operational channels are also annotated on feature ontology. Then events can be defined as data messages passed on operation channels.

Definition 7 (Communication data and events): Data messages passed on operational channels can be represented by a set $msg=\{O_1,O_2,...,O_n\}$, in which $O_i \in BusinessObject$ ($1 \leq i \leq n$). Then sending and receiving data message *msg* on operational channel *c* are denoted by *c*!*msg* and *c*?*msg* respectively.

Finally we can define the behavior protocol of components by operational channel and data message.

Definition 8 (Behavior protocol of components): The behavior protocol of a component *com* is a CSP process P satisfying $\alpha P=\{c.v|c$ is a input or output (physical or operational) channel of *com* and *v* is a data message passed on $c\}$.

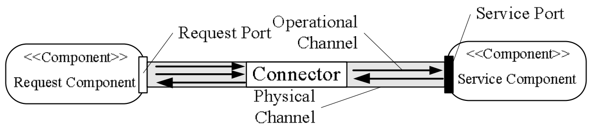


Figure 4. Physical channel and operation channel

The behavior protocol of *OrderProcess* is given in figure 5. It is the static protocol of *OrderProcess*, which is specified on physical channels and can be refined and adapted at runtime (see section 5.2). From the protocol, we can see *Inform* is an asynchronous interaction without response.

Abbreviation of port semantics:

OP=<OrderProcess,()>,OT=<OrderTreatment,()>,LOG=<Log,()>,
INF=<Inform,()>,DEL=<Delivery,(optional=true)>
Component OrderProcess =
OP_in?Order→OT_out!Order→OT_in?Bool→INF_out!Order
→LOG_out!Order→LOG_in?Bool→DEL_out!Order
→DEL_in?Bool→OP_out!Bool §

Figure 5. Static behavior protocols of *OrderProcess*

5.2. Feature-oriented Protocol Adaptation

As showed in figure 2, adaptations on are first performed on behavior protocols, which provides a business-oriented evolution view for the user. The adaptation originates from the potential variables in the protocol. In our model, the variables are feature variables, including optional and variable features. For example, in the static protocol showed in figure 5, *Delivery* is an optional element and *Inform* is variable in the type of message according to figure 1 (e.g. by *SMS* or *Email*). These variables can be determined in various phases, which is called binding time (typically *BuildTime* and *RunTime*) [6].

In domain engineering and product line, feature model is adopted to model the domain-specific requirements and guide the customization of application-specific requirements and architecture [6]. So feature binding is primarily discussed in application development in the domain. In this paper, feature model is adopted as the business-oriented basis for dynamic evolutions of components. So runtime binding of variable features is concerned to provide the business-oriented mechanism for runtime adaptations.

The business orientation is first embodied by the adaptation instruction in the adaptation rules. Some binding demands of action semantics are directly specified in adaptation rules. When certain conditions are met, these action semantics in behavior protocols of related components will change according to the rule. Then binding inference will be performed on related protocols according to the constraints defined in the feature model, and then other parts of the protocols will be adapted. That is another embodiment of the business orientation.

Direct binding demands are usually specified on high-level features, representing business viewpoint of the user. For example, the user may demand that when the storage is insufficient the sale mode should change into *Booking* (see table 1). This may cause a series of binding changes of other action semantics, which is binding inference. The binding inference here is similar to that introduced in our feature modeling work of [6], in which binding inference is implemented by ontology inference (implemented by Jena in [6]) since the feature meta-model is defined on OWL. Here binding inference is performed at runtime, so the only difference is that the inference should be performed incrementally to make the efficiency feasible for runtime adaptations. The ontology-based inference is based on constraints (various dependencies on variable features) defined in the feature model and independent rules defined for feature inference. Readers can refer to [6] for details of feature constraints, inference rules and ontology-based binding inference.

Condition	Adaptation
storage>100	<OrderProcess,(HasSaleMode=DirectSale)> is bound for OrderProcess
storage<=100	<OrderProcess,(HasSaleMode=Booking)> is bound for OrderProcess

Table1. Adaptation rules for *OrderProcess*

Figure 6 presents an example of protocol adaptation on the rules defined in table 1. The scenario is the storage shortage of the pet store. Initially, the system performs the order process in the direct sale mode. When the storage decreases to a prescribed level, the system will be adapted into the booking mode. After the adaptation control module detects status of storage shortage from a storage sensor, it performs corresponding policies, and then the facet value of *OrderProcess* on the facet *HasSaleMode* will change to *Booking*. After that, binding inference can be performed to determine binding results of all the features. According to the constraints defined in figure 1, the action *OrderTreatment* should be refined to *ProductBooking*. The optional action *Log* is bound and the message type of *Inform* changes to *Email*, because of the dependencies defined on *ProductBooking*. Accordingly, optional action *Delivery* is removed.

Behavior Protocol before Adaptation
Abbreviation of port semantics: OP=<OrderProcess,(HasSaleMode=DirectSale)>, OT=<Distribution,()>,INF=<Inform,(HasMsgType=SMS)>, DEL=<Delivery,()> **Component** OrderProcess = OP_in?Order→OT_out!Order→OT_in?Bool→INF_out!Order →DEL_out!Order→DEL_in?Bool→OP_out!Bool §
Behavior Protocol after Adaptation
Abbreviation of port semantics: OP=<OrderProcess,(HasSaleMode=Booking)>, LOG=<Log,()>, OT=<ProductBooking,()>,INF=<Inform,(HasMsgType=Email)> **Component** OrderProcess = OP_in?Order→OT_out!Order→OT_in?Bool→INF_out!Order →LOG_out!Order→LOG_in?Bool→OP_out!Bool §

Figure 6. Protocol adaptation for storage shortage

5.3. Composition Analysis of Protocols

CSP provides operator of parallel composition for processes (denoted by "‖") [10]. So composition analysis can be performed for CSP-based process expressions, as discussed in Wright [11]. In our feature-based behavior protocols, the additional issue is to unify the event notations when composition. Since all the channels in the events are represented by action semantics (Definition 8), event notations can be unified on the feature ontology. Here is the definition of event semantics negotiation in composition.

Definition 9 (Event semantics negotiation): If an external port *R* is connected with an external port *S* of another component, *r_out*!*u* and *s_in*?*v* are events on *S* and *R* respectively, then *s_in*?*v* can be refined to *r_out*!*u* if $r \subseteq s \wedge (u\ rdfs{:}subClassOf\ v)$.

This rule specifies that events of on an external port with general semantics can be refined when it is composed with external port of other components. Then consistent events of two connected ports can be unified on feature semantics. After that, further feature binding inference and adaptation are performed on the component involving the refined port. So this kind of protocol adaptation is driven by component compositions.

After event semantics negotiation, behavior protocols of two composed components can be composed by CSP composition operators and the composition can be repeatedly performed with other components till all related components are involved. General behavior consistencies (e.g. deadlock free) can be validated when composition. Furthermore, as the dynamic customized instance of domain model fragment, satisfying feature constraints is another basic requirement for the runtime protocol. This is specified in feature consistency on traces.

Definition 10 (Feature consistency on traces): For each $tr \in traces(sys)$ of a composed system *sys*, there is: any two operation semantics as_1=<a_1,fv_1> and as_2=<a_2,fv_2> must satisfy all the feature dependencies between a_1 and a_2 specified in the feature model.

This rule specifies that each execution path of the system protocol must not violate the feature consistency, which can ensure the consistency of runtime adaptations together with traditional composition analysis in CSP.

5.4. Mapping to Component Execution

In the protocol interaction behavior of component is specified, adapted and validated on an abstract level. It should also be interpreted and executed by the component, which is implemented by the control center in our component model. In our model, the protocol will first be translated into FSM-based control flow. For example, figure 7 shows the contro flow of *OrderProcess* before and after adaptation, which is represented by a UML activity diagram. The starting event denotes requests received on external ports and each activity denotes a call to internal or external request port (depicted by gray ellipses)

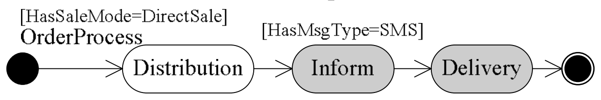
On the protocol level, adaptations are embodied by changes of action semantics and binding/remove of optional actions. After mapping to the control flow, these adaptations can be interpreted as:

Changes of external service semantics: They are changes of service semantics on external service ports (e.g. semantic adaptation of *OrderProcess* in figure 7).

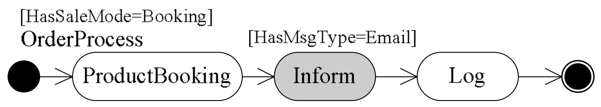
Changes of internal operational semantics: Internal ports are interfacing points for business functions implemented by current component. So, changes on internal operational semantics denote the change of invocation details, such as invocation or configuration parameters. For example, the binding of *ProductBooking* for *OrderTreatment* (see the example showed in figure 6, 7) will make the control center use another parameter in the invocation to the internal port.

Changes of internal process: They are changes of internal control flow on interaction sequences. From the aspect of feature, adaptations on internal process usually exhibit as rebound and removal of optional actions provided by internal ports or external components (e.g. *Delivery* and *Log* in figure 7).

Changes of external component connections: They can be caused by change of internal process and embodied by connection and disconnection with other components (e.g. *Delivery* in figure 7). They can also be caused by semantic adaptations on external request ports. After that, the control center will find new service providers from the service register center according to the new request semantics, which can also cause the reconnection of components.



(a) The control flow before adaptation



(b) The control flow after adaptation

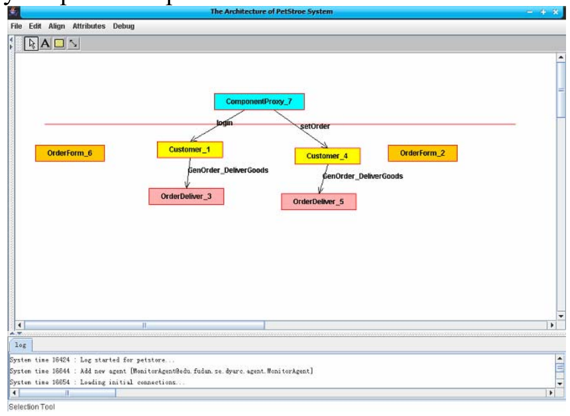
Figure 7. Control flows before and after adaptation

6. Implementation

We have applied the adaptive component model in our implemented framework for dynamic architecture [13] and implemented the administration tool for the framework. Self-defined sensors monitoring the runtime environment or the business status can be plugged into the framework and adaptation rules can be defined on corresponding sensors.

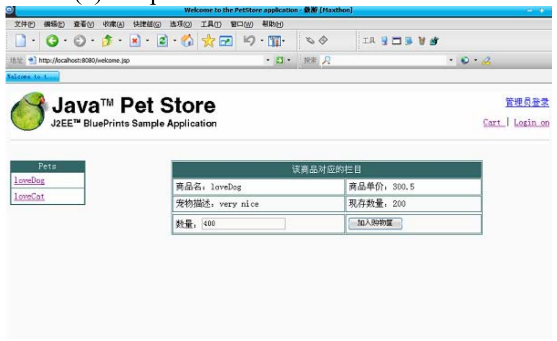
Figure 8 (a) shows the monitoring interface for the runtime architecture, which is designed for Java based web applications. Elements above the line denote the service proxies, each corresponding to a servlet session. Elements below the line represent all the business components, which are adaptive and shared among different sessions. Figure 8 (b) shows a snapshot of the adaptive JPS. We modified the original JPS according to the adaptive component and the feature model presented in figure 1. A sensor for the storage is implemented to fire the adaptation, which query current storage at regular intervals. In our experiment, when we buy a large number of pets on the web page, the storage will decrease to a low level and the runtime architecture will change.

From the implementation, we also realize the limitation on runtime efficiency. It is caused mainly by the efficiency of runtime ontology inference is still limited. However, it is sure that the component model can be applied in some small-scale systems. Now we are developing an online fee-charging system for university students. The dynamic switch of different online payment services (according to the dealing amount of each bank service) and runtime login control (according to current load of server) are implemented by adaptive components.



```
System time 16424 : Log started for petstore...
System time 16644 : Add new agent [MonitorAgent@edu.fudan.se.dyarc.agent.MonitorAgent]
System time 16654 : Loading initial connections...
```

(a) Snapshot of the administration tool



该商品对应的栏目	
商品名：loveDog	商品单价：300.5
宠物描述：very nice	现存数量：200
数量：400	

(b) Snapshot of the adaptive JPS

Figure 8. Implementation of the JPS case study

7. Conclusions and Future Work

Adaptive components are essential to implement dynamic evolution of business systems. Existing approaches on adaptive components always lack the corresponding business model to specify and perform adaptations in a business-oriented way. In this paper, we integrate the researches on feature-based domain modeling and adaptive components into a feature-oriented adaptive component model. The feature model represents the common understanding of the domain, specifies possible variations together with constraints, providing a shared business view for all the users, component developers and application engineers of the domain. So it can provide both the business view for the user and adaptation basis for the system. Furthermore, the ontology-based feature model provides unambiguous terminology for both the business view and the component specification, which ensures the consistency between them. The feature-oriented adaptive component model has another characteristic of the micro control flow within the component, which enables adaptation of component behaviors, including interaction sequence and style.

All of the “how, why, when and what to do” in adaptations should be answered in approaches of adaptive systems [2]. Our component provides a feature-oriented way for adaptation implementations (how). It also enables the feature-oriented specification of adaptation policies and dynamic decisions on adaptations schemas (why, when and what). In our future work, we will study the event and context model of dynamic adaptations. This model is the foundation of context monitoring, analysis and adaptation decision.

Acknowledgments. This work is supported by the National High Technology Development 863 Program of China under Grant No.2005AA113120, the National Natural Science Foundation of China under Grant No. 60473061 and 60473062.

8. References

[1] J. Bradbury, J. Cordy, J. Dingel, and M. Wermelinger. A survey of self management in dynamic software architecture specifications. Proceeding of the ACM SIGSOFT International Workshop on Self-Managed Systems (WOSS’04).

[2] Yali Zhu, Gang Huang, Hong Mei. Quality attribute scenario based architectural modeling for self-adaptation supported by architecture-based reflective middleware. Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC’04).

[3] Ji Zhang and Betty H.C. Cheng. Model-Based Development of Dynamically Adaptive Software. ICSE’06.

[4] Philippe Boinot, Renaud Marlet, Jacques Noye’, Gilles Muller and Charles Consel. A Declarative Approach for Designing and Developing Adaptive Components. ASE2000.

[5] Paul G. Bassett. The Theory and Pratice of Adaptive Component. GCSE 2000, LNCS 2177, pp.1-14, 2001.

[6] Xin Peng, Wenyun Zhao, Yunjiao Xue, Yijian Wu. Ontology-Based Feature Modeling and Application-Oriented Tailoring. Proceedings of the 9th International Conference on Soft-ware Reuse (ICSR2006), LNCS 4039.

[7] Jeong Ah Kim, YoungTaek Jin, and SunMyung Hwang. A Business Component Approach for Supporting the Variability of the Business Strategies and Rules. ICCSA 2005, LNCS 3482, pp. 846 – 857, 2005.

[8] Kung-Kiu Lau and Vladyslav Ukis. Automatic Control Flow Generation from Software Architectures. The 5th International Symposium on Software Composition, SC06.

[9] Vladimir Mencl, Tomas Bures. Microcomponent-Based Component Controllers: A Foundation for Component Aspects. Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC’05).

[10] C.A.R. Hoare. Communicating Sequential Processes. Prentice Hall, 1985.

[11] R Allen, D Garlan. A formal basis for architectural connection. ACM Trans on Software Engineering and Methodology, 1997, 6(3): 213-249.

[12] Sean Bechhofer, et al. Owl Web Ontology Language Reference”, http://www.w3.org/TR/owl-ref/, 2004-02-10.

[13] Xin Peng, Wenyun Zhao, Liang Zhang, Yijian Wu. An Intelligent Connector Based Framework for Dynamic Architecture. Proceeding of the 5th International Conference on Computer and Information Technology (CIT2005). IEEE Computer Society Press.