DOI: 10.1007/s11432-006-2027-1

A software architecture centric engineering approach for Internetware

MEI Hong, HUANG Gang, ZHAO Haiyan & JIAO Wenpin

School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China
Correspondence should be addressed to Mei Hong (email: meih@pku.edu.cn)
Received April 30, 2006; accepted September 4, 2006

Abstract As a new software paradigm evolved by the Internet, Internetware brings many challenges for the traditional software development methods and techniques. Though architecture-based component composition (ABC) approach is originated in the traditional software paradigm, it supports the engineering of Internetware effectively due to its philosophy, rationales and mechanisms. ABC has three major contributions to the engineering of Internetware in detail. First, the feature oriented domain modeling method can structure the "disordered" "software entities" to "ordered Internetware" bottom-up in the problem space. Second, the architecture centric design and analysis method can support the development of self-adaptive Internetware. Third, the component operating platform is a reflective and self-adaptive middleware that not only provides Internetware with a powerful and flexible runtime infrastructure but also enables the self-adaptation of the structure and individual entities of Internetware.

Keywords: internetware, component, software architecture, feature model, reflective middleware, autonomous component.

The Internet development brings new challenges to the information technology which requires innovation of the present technologies, meanwhile, it produces multi-hotspot fields about information technology research and practice. For example, the grid management discusses the future application and construction model for the network systems from the perspective of resource sharing and management; from the perspective of humancomputer interaction, the pervasive computing discusses how future network applications will be used and operated ubiquitously; the service computing emphasizes the idea of "software as a service" and proposes a new software paradigm that pays special attention to the coordination and dynamism of services; the model-driven development is based on domain-specific code generation and focuses on middleware-based development methods and techniques. Almost all of the work can be considered as attempts to review, rethink and evolve the information technology from some new perspectives. Similarly, Internetware grounds itself in the open, dynamic and ever-changing Internet

and focuses on such new software paradigm that will be autonomous, evolutionary, cooperative, polymorphic, and context-aware[1]. Since traditional software engineering methods and techniques are originated from and more suited for a static and closed environment, they are not appropriate for open, dynamic, and ever-changing Internetware, which requires innovations of the traditional software development methods and techniques.

Technically, Internetware forms a Software Web[1] on Internet, similar with the present information Web. The software entities constituting Internetware, supported by software components and other technologies, will be distributed over the Internet openly and autonomously. And they will be published in an open environment, cooperating with each other in various manners. Due to the open, dynamic, and ever-changing Internet, as well as the various user preferences, Internetware keeps evolving after it has been developed and deployed. When an Internetware is published, it is capable of perceiving the dynamic changes of its environment and evolves according to its functionality, performance and trustworthiness, etc. so that it not only satisfies users' requirements but also improves experiences. Besides, the variation of user preferences and return of investment usually lead to a long lived and ever-evolving Internetware. The engineering of such Internetware as well as its supporting technologies is quite different from the traditional ones.

Traditional software development processes are more suited for the relatively close, static and stable platforms. Most of them adopt a common top-down approach, scoping the system border and using divide-and-conquer principles to make the whole process under control. However, the platform of Internetware has abundant resources and it is always open, dynamic, and ever-changing. The development over this platform can be seen as the composition of various "disordered" resources into "ordered" software systems. As time elapses, changes of resources and environments may "disorder" the existing software systems again, which will become "ordered" sooner or later. The iterative transformation between "ordered" and "disordered" Internetware implies a bottom-up, inside-out and spiral development process. Besides, the traditional software lifecycle emphasizes the importance of the whole development process; the software evolution is only managed by the phase of "software maintenance". This is appropriate for the development under a static and closed environment; however it is not suitable for Internetware, because 1) the present software composes the new software entities. And all of them are relatively independent and there is no central control for them, so it is difficult to guarantee that the composed Internetware satisfies the functionalities and qualities as planned unless it starts to run; 2) the open, dynamic, and ever-changing environment requires the Internetwares and their cooperation may face many changes. No matter whether these changes can be predicted or not exactly, the running system has to do the durative adjustability regulation. As a result, software maintenance would become a relatively important phase; 3) Internetware provides service for the worldwide users. Furthermore, an Internetware is usually composed of software entities which are distributed over the Internet and then has no chance to be shut down completely after it is deployed and starts to run. This implies that all maintenance activities, including debugging, optimizing and

upgrading have to perform online. All these activities also have the phases of analysis, design, implementation, testing and deployment, which cannot be controlled well by the concepts and techniques in the traditional software maintenance.

In the traditional software development methods, less attention is paid to the autonomy, cooperation, context-awareness, evolution and polymorphism of Internetware. First, the Internetware's autonomy means a software entity which is relatively independent. It can perform operations as it will and adapt itself when necessary. Internetware is autonomous because it is usually developed and managed as well as runs independently on distributed nodes. The goal and the services of an Internetware are determined by its owner; therefore, the Internetware behaves according to not only the composition or deployment strategies but also the owner's goal. Actually, an Internetware may collect information on environment change and adapt itself according to the preset strategies in order to accommodate the ever-changing environment. The cooperation of an Internetware means that software entities constituting an Internetware may cooperate with each other in static or dynamic manners. Comparatively speaking, the traditional software systems usually adopt a single and static connection mechanism. However, the connection mechanisms adopted by Internetware may also be changed if necessary. For example, communication protocols of different Internetware may be switched, the security level can be increased or decreased at runtime, and the availability of the passing messages may vary according to the changing environment. The context-awareness means that an Internetware is capable of perceiving its runtime context, which includes both the underlying operating platform and other Internetware. Consequently, both the Internetware and the operating platform should expose their runtime states and behavior. Evolution refers to that the structure of an Internetware may change dynamically according to the requirements as well as its environment. Possible evolutions include the number of its constituents, the adjustment of its topologies, as well as dynamic configurations. This requires that the software architecture model for an Internetware should be able to change dynamically. Polymorphism means that an Internetware may incarnate several compatible goals. Based on some basic cooperation principles, an Internetware may satisfy different but compatible goals in a dynamic environment. This requires the modeling of compatible goals. Besides, an Internetware should be able to determine its goal dynamically when the environment changes. In short, most characteristics an Internetware should have are related to its self-adaptability, including the self-adaptability of single software entity as well as the software topology. A self-adaptable Internetware is capable of adapting itself according to specific changes at the right time, in the right place, such that it can satisfy the desired functionalities and qualities. Naturally, the degree of human intervention determines the degree of self-adaptability. Therefore, a real challenge for the engineering of Internetware is how to develop a self-adaptable Internetware, such that it can behave properly with as little human intervention as possible.

Traditional software development tools are usually targeted at one or several phases before the software is delivered; after delivery, the software is maintained by various management tools. As shown above, the emphasis of Internetware development is shifted from pre-delivery to the runtime. The runtime adaptation of Internetware (no matter

whether by hand or automatically) usually depends on activities and artifacts produced in different development phases. Therefore, the software development tools for Internetware should not only cover the whole software lifecycle, but also be integrated with the runtime platform. Conversely, the above characteristics incarnated in Internetware challenge the runtime platform in various ways. First, the runtime platform not only is able to present its states and behavior at runtime, but also facilitates the Internetware to present its runtime information. Second, the runtime platform should provide mechanisms to adapt to the constituents of an Internetware as well as its topologies; otherwise, the Internetware cannot become self-adaptive.

In summary, the engineering of an Internetware is quite different from that of the traditional software. It follows a specific bottom-up process which incoperates "disordered" software entities into "ordered" software systems, and it pays special attention to the self-adaptability of the constituents and the structure of an Internetware. The supporting tools cover the whole lifecycle and are usually integrated with the operating platform, which meets the requirements to implement the self-adaptation of an Internetware. In this paper, we propose a software architecture centric engineering approach for Internetware, which is a natural extension of the Architecture Based Component Composition (ABC) approach[2]. ABC uses software architectures as blueprints to develop middleware-based software systems by assembling reusable components. We think that the philosophy, rationales and mechanisms of ABC are compliant with the open, dynamic and ever-changing characteristics of Internetware, in particular, the above-mentioned three major issues of the engineering of Internetware can be efficiently handled by ABC.

The rest organization of this paper is as follows: Section 1 introduces the core idea, design rationales as well as the process model of ABC; section 2 discusses how to use feature models to organize and manage "disordered" Internetware resources; section 3 presents how to design a self-adaptive software architecture for an Internetware; section 4 introduces a reflective middleware that supports self-adaptation of software entities as well as software structures; section 5 concludes this paper and identifies the work in the future.

1 Overview of ABC methodology

As originated in 1998 and formally proposed in 2000[2], ABC is a combination of software architecture (SA) and component-based software development (CBSD) for supporting software reuse. SA provides a top-down approach to realize component-based reuse. This approach uses architecture description languages (ADL) to abstract components, connectors as well as their interactions, which constitute the whole architecture model. However, SA does not pay enough attention to the refinement and implementation of the architectural descriptions, thus not fully able to automate the transformation or composition which results in an executable system. On the other hand, CBSD provides a bottom-up way by using the existing middleware infrastructures. It emphasizes how to reuse pre-fabricated components to build large-scale software systems. However, this method is mainly restricted to binary components (e.g., COM, CORBA, EJB); besides, it

is not able to systematically guide the CBSD process, especially the component composition at higher abstract levels. In our opinion, SA and CBSD are complementary and able to be combined to realize effective component-based reuse.

ABC introduces software architectures into each phase of software life cycle, takes SA as the blueprint of system development, deployment and management, shortens the gap between high-level design and implementation by supporting tools and mapping mechanisms, and realizes the automated system composition. The process model for ABC method is shown in Fig. 1, including the following phases:

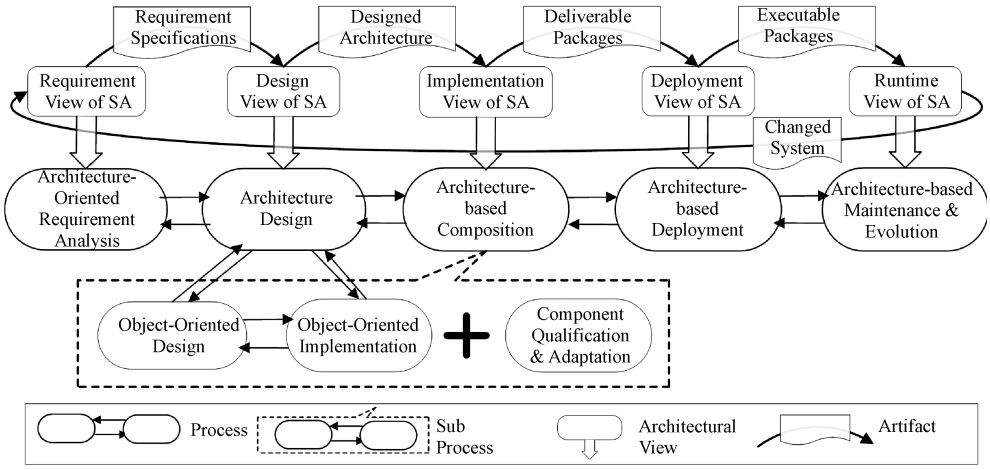

Fig. 1. Process and artifacts of ABC.

● Requirement Analysis (Requirement View): SA is introduced into the requirement analysis phase to guide component composition at a high level. In this phase, the problem space and the requirement specification are structured in the way similar to SA[3]. ABC uses features to represent software requirements and relationships among features to represent relationships among requirements. In a word, feature is treated as a first class entity in the problem space. Features and static or dynamic relationships among features are organized as feature models. In particular, for the purpose of reuse, the feature model in ABC method uses variability mechanism to capture the commonality and variability of a set of similar requirements. The relationships among requirements are modeled by four relations in the feature model, including refinement, constraint, influence and interaction[3,4]. These relations can be used to validate the completeness and consistency of a feature model[5]. Based on feature models, an initial SA model can be designed by identifying responsibilities for each component and analyzing dependencies among requirements[6]. The initial SA model is the input for the later phase of architecting, composition and maintenance.

● Architecting Phase (Design View): A complete SA model comes into being in this phase. The architect determines the global design decisions based on the requirement specification. In this phase, components and connectors in the initial SA model would be refined. New components and connectors are created if necessary; different views (in-

cluding type view, instance view and process view) may also be created. The relationships between the requirement specifications and SA models are set up in this phase[2,7]. Since reuse is emphasized in ABC method, the architects should take into account reusable components and connectors during the design. It should be noted that ABC method is not specific to a certain development paradigm. For example, object-oriented design (OOD) can also be adopted in this phase. The high-level OOD model can also be treated as an SA model, as long as the modeling elements in OOD model are encapsulated into those in the SA model (e.g., a set of classes may be encapsulated into a component according to their interaction frequency, the whole-part relation, the general-special relation, etc.).

● Composition Phase (Implementation View): in ABC method, the SA-based composition implements a software system. In this phase, component implementations are qualified, selected or adapted according to the SA model. After all necessary component implementations are integrated, a deliverable software package is produced[2,8]. However, in practice, there may be some components that have no reusable implementations. In that case, a detailed design model (e.g., UML model) or some C++ or Java programming frameworks for those components would be generated in an automated manner[2]. These newly implemented components can be finally assembled into the target system as well.

● Deployment Phase (Deployment View): component-based software systems are usually specific to a certain middleware, such as Common Object Request Broker Architecture/CORBA Component Model (CORBA/CCM), Java 2 Platform Enterprise Edition/ Enterprise JavaBeans (J2EE/EJB), Component Object Model (COM), Web Services, and so on. These software systems start to run only after they are deployed properly. Deployment-related information is usually given by hand, which is tedious and error-prone because such information is voluminous and trivial. In practice, most deployment-related information can be deduced from former views, including the design view and the implementation view. Therefore, an explicit deployment view is introduced in ABC. This view presents most information that is deduced from other views. It also supports intuitive operations on deployment-related information. The information on resources and workloads of the target environment is shown in real time as well. With the support of this deployment view, a component-based software system can be deployed in an automated manner[9].

● Maintenance and Evolution Phase (Runtime View): ABC can be seen as iterative refinement, mapping and transformation between different SA views. Each time the SA model is refined or transformed, it becomes more precise and integrated. During the maintenance and evolution phase, the runtime view is used to depict the runtime states and behavior of the software system. It is the view that has the most precise and complete information on the target system. Based on the support of reflective middleware, the runtime software architecture (RSA) embodied in the runtime view reflects the target system at runtime. The target system can then be maintained or updated at runtime by operations to the RSA[10, 11].

To facilitate the above process, a set of tools are provided by ABC, including the feature modeling tool, SA modeling tool and middleware as component operating platform.

In particular, the SA modeling tool supports visualized SA design, component composition, deployment, as well as online maintenance and evolution. To date, ABC has been applied to the development of several real applications experimentally, including the modeling of information system for Beijing Olympic 2008 and a loan management system in some commercial banks.

Though ABC is originated from traditional software systems, it can support the engineering of Internetware effectively because Internetware is an evolution of the traditional software and ABC takes into account the support for the main characteristics brought by Internetware since 2002. Details are as follows: 1) Internetware shapes up from "disordered" resources to "ordered" software systems. This process embodies a typical (and even ideal) paradigm for reuse. However, it still requires support for related technologies, which are also keys to ABC and have been already well supported by ABC. Definitely, there are some significant differences between Internetware and the traditional software. For example, reusable assets are distributed and decentralized, software entities are autonomous. 2) Though the autonomy of software entities is very typical in Internetware, it is under control in a certain degree. Otherwise, it is hard to form an "ordered" Internetware. Therefore, an explicit SA is still necessary to perform the global and loose control over Internetware. Compared to the traditional software, some entities, as well as their connections, in the SA model for Internetware may be undetermined before runtime or changed continuously at runtime. 3) Different from the traditional software, Internetware emphasizes the ever-evolution after delivery. As a result, it is difficult to split the development of Internetware from its execution. The development tools should then be integrated with the runtime platform. In ABC, SA model plays a central role in integrating tools for design, implementation, deployment, maintenance and evolution. In a word, the idea and process of ABC method are well suited to Internetware, and ABC mechanisms can support the engineering of Internetware. In particular, ABC pays special attention to the following issues brought by the unique characteristics of Internetware. As to the engineering process, ABC cares about how to use feature-oriented requirement modeling to support the bottom-up development process as well as the organization and management of "disordered" resources; after that, how SA models can be used to integrate the design, implementation, deployment, maintenance and evolution of Internetware. As to the engineering method, ABC cares about how to design a self-adaptive SA model for Internetware, especially when various qualities are taken into account. As to the supporting techniques, ABC cares about how to strengthen the existing platforms that support EJB, Web Services and other main stream component models. The enhanced platform provides a reflective framework to support the monitoring and controlling on Internetware as well as the platform itself. It also provides mechanisms for rule-based autonomous components.

2 Feature-oriented requirement modeling for Internetware

Based on a platform with rich sets of software assets, the engineering of Internetware is usually in a bottom-up fashion since new applications can be built by selecting and

composing these existing assets according to users' requirements. However, the platform is an open, dynamic and ever-changing framework and most of the existing assets in it are distributed, decentralized and heterogeneous. In that sense, the platform manifests itself "disordered" from a global perspective. Therefore, one challenge for the engineering of Internetware is how to conform these disordered assets to ordered and controllable ones, so as to allow the developers to employ mature software development methods, such as the traditional top-down, stepwise refinement method in their construction of an Internetware.

As a systematic way to produce the reusable artifacts in a particular problem domain, domain engineering addresses the creation of domain models and architectures that abstract and represent a set of reusable assets within a domain through domain scoping, commonality and variability analysis and adaptable design construction based upon the study of the existing systems, knowledge from domain experts, and emerging technology within a domain, taking the possible requirement changes, technology evolution, economic benefits and some limitation into consideration. Here, domain analysis refers to the process of identifying, collecting, organizing and representing the relevant information in a domain. In a sense, the domain analysis is a process in the bottom-up fashion coincided with the engineering of Internetware.

Therefore, ABC adopts the methods and techniques of domain engineering to coordinate the underlying resources, on which the engineering of Internetware is based, making the resources at the bottom sites into a set of ordered components, building the Internerware that fulfills some specific business goals.

As shown in Fig. 2, we first structure or organize the disordered resources distributed over the Internet into a domain model with variability representation mechanism, which embodies high-level business goals for a bundle of Internetware, by domain scoping and analysis; and then we build a new application by tailoring and extending the domain model according to the application-specific requirements. When time elapses, the new application may be scattered somewhere on the Internet as a service and then becomes a new disorder resource. In turn, these new disordered resources can be added into the domain model by further analysis, and therefore form the iterative process of disordered resources to ordered ones. The feature-oriented domain modeling method (abbr. to FODM[3−6]) in ABC provides an effective means to conform the building-block resources for Internetware.

Regarding features as basic elements in the problem space, ABC uses features and relationships (i.e. refinements and constraints) between features (called domain feature models) to structure the problem space, i.e., use features to group and organize the requirements which support modeling of domain requirements systematically. The relationship between features includes the refinement, constraint, influence and interaction. The former two are the static dependencies between features which are significant to the commonality and variability modeling, while the latter two are dynamic ones which are important for the domain design based on feature models. Speaking concretely, 1) refinements are a kind of binary relationships between features. They integrate features at different levels of abstraction into hierarchical structures which provide an effective way

to describe complex systems. 2) Constraints are a kind of static dependencies among features which provide a way to verify the results of requirement customization and release planning. 3) An influence between two features means one feature imposes additional responsibilities on the other, which depicts the dependencies between feature in the specification level. 4) Interactions reflect how features interact with each other at runtime. Fig. 3 illustrates a concrete form of a feature model in ABC. Besides recording all the service features, function features, behavior characteristics features and use case features a system has, this model records the system's quality features and the constraint and interaction relationships between features explicitly. Features with different abstract levels and granularities (service, function and behavior characteristics) form a hierarchical structure via refinement relationships between them. The features in Use-Case Section are related to the service, function and behavior characteristics features through the dependencies between the features; while the quality features record the service, function and behavior characteristics features they may affect.

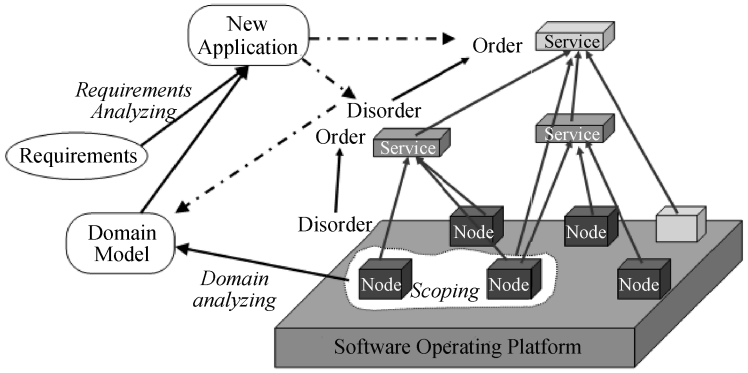

Fig. 2. Conforming the bottom resources by domain engineering.

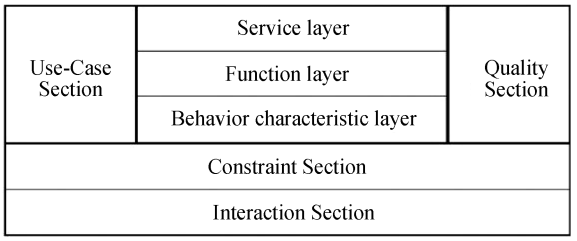

Fig. 3. A concrete form for a feature model.

ABC's feature model is in nature a way to partition and organize the requirements since the features and their relationships depict the essential elements of the problem space. In the view of requirement's intension, a feature embodies a kind of capabilities or characteristics the system possesses, which reflect the requirement-enticer's understanding of the system; in the view of requirement types, a feature can be a functional requirement, a quality requirement, or some kind of environment constraints to the system.

During the process of conforming and structuring the bottom resources on which to coordinate the Internetware, ABC uses features to represent and organize these resources,

and relationships between features to depict the combination relationships between these resources. Consequently, to a set of Internetware with common requirements and certain variable requirements, ABC's feature model can depict their functions and services to be revealed, the system goals they need to achieve, and their adaptable requirements to environment. Furthermore, by identifying the responsibilities that features possess and analyzing the mutual dependencies between features, developers can get a high-level abstract architecture for this set of Internetware, which can be used as the basis for discriminating and filtering the bottom resources, and the guidelines for the later phases of design, composition and maintenance. If a set of Internetware with common and certain variable requirements is regarded as an application domain, the disordered bottom resources they depend on can be coordinated into ordered and controllable ones at a high level through the feature-oriented domain modeling method.

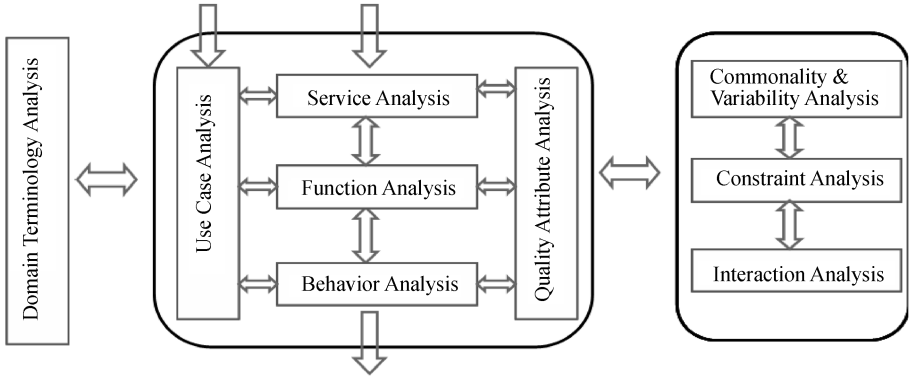

Fig. 4. The modeling process of a feature model.

Fig. 4 presents the feature modeling process of ABC, in which the activity of Service Analysis identifies the service features that consist of Internertware to define their capabilities revealed to users or customers. The activity of Function Analysis identifies the functional features that a service possesses, which can derive what functions should be included to fulfill a specific service. The activity of Behavior Analysis is to identify the behavior characteristics of a function, such as the pre and post-conditions of a function to be executed, and the control flow after executing a function. Crossing the activities of service-function-behavior analysis, the activities of Domain Terminology analysis, commonality/variability analysis and Interaction analysis, Quality attribute analysis can be conducted simultaneously. That is, through analyzing the domain terminology, we can find the services it implies through analyzing the domain terminology; through analyzing the commonality and variability at the service level, we can find the constraints between services; for each service, function and behavior, we can analyze their relevant use cases and the commonalities and variabilities in use cases; and we can also find the quality attributes a service may have. The main task of the Use Case Analysis is to discover the features existing in the interaction between users and the Internetware systems, and by identifying the business process related with services, we can extract the commonalities and use cases that reflect the domain characteristics. The activity of Quality Attribute

Analysis focuses on identifying the requirement for the quality attributes that should satisfy this set of Internetware, and further make clear the system goals to achieve.

One practical approach to requirement reuse is the domain-specific and customization-based reuse, that is, when the feature model in a specific domain has been constructed, the following activity is how to reuse this feature model. One effective way to reuse the model is to accommodate different applications in this domain by customization. That is, we can get a set of features that interest the current application by tailoring the domain feature model. Since there are many dependencies between features, how to ensure the tailoring result to be consistent and integrated becomes very important. ABC proposes a tailoring process for feature models based on the concept of atomic set, and the verification criteria to check the rationality of the tailoring result depend on the constraint relationship between features.

To make the designer easily discriminate and filter the suitable components from the bottom resources to build the new Internetware application, it is necessary to establish the corresponding mechanism between the features and their relationships identified in the phase of requirement analysis with the bottom resources and their relationships. Through the identification and assignment to the system responsibilities, ABC gives an approach to transforming a feature model to a high-level abstract software architecture (just acts as a draft model for architects).

There are two fundamental problems to be addressed for the model transformation. One is the traceability between the source model and the target model, which is the foundation of model transformation. The other is construction of the target model, which is the core of model transformation. The embodiment of these two problems in ABC is as follows: the traceability between features and components; the software architecture construction based on feature model.

There exist *n*-*n* relations between features and components in nature. To trace this kind of complex relations, ABC introduces the concept of responsibility as the connector between features and components. A responsibility is a cohesive set of program specifications, and can be used as a basic unit for task assignment to software developers. Via the connection of responsibility, the complex *n*-*n* relation between features and components can be decoupled into two 1-*n* relations, that is, a feature can be operationalized into a set of responsibilities, and one component can implement multiple responsibilities. On this basis, the traceability between features and components can be established in two steps: operationalizing features to responsibilities, and assigning responsibilities to components.

An overview of the transformation from feature models to software architectures is depicted in Fig. 5. The concepts involved in the transformation can be divided into two levels. One is the Requirement Level of Internetware, at which the requirements can be structured as feature models. The other is the Specification Level. At this level, program specifications are first organized as a set of responsibilities, a set of resource containers, and interactions between them; and then responsibilities and resource containers are clustered into conceptual components, and the interactions between responsibilities or resource containers form the interactions between components by filtering and clustering.

In fact, components can be considered as a kind of responsibility containers, so it can be constructed by responsibility clustering. The identification of interactions between components is guided by the following assumption: if two responsibilities are assigned to two different components, then any interaction between these two responsibilities will be developed into an interaction between components.

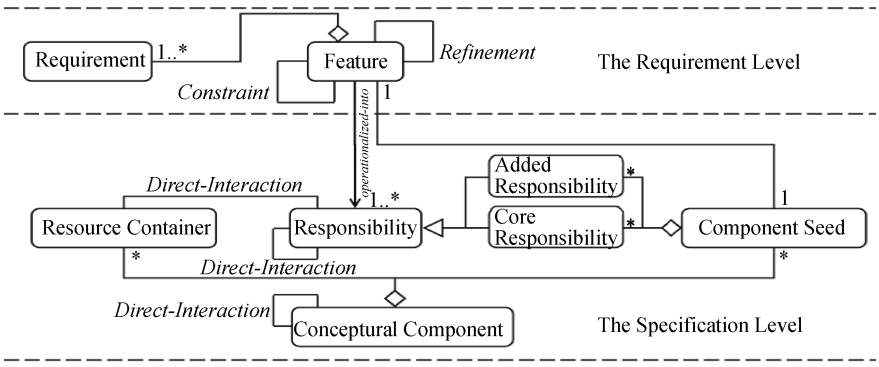

Fig. 5. An overview of transformation from a feature model to software architecture.

Using the forementioned ABC feature modeling method, we can model the Internetware's system goals and constituent capabilities and responsibilities naturally and clearly, so as to guide the discrimination of the bottom resources.

To better support the feature-oriented requirement modeling, ABC has developed a corresponding graphical supporting tool called FMTool, which provides means to model and edit the feature model easily. In addition to serving the traditional domain engineering and software product line, it is applicable to modeling complex software applications like Internetware.

To demonstrate this point we use the Internet shops as the case study. There exist a lot of E-shopping sites today, and their portal software systems are generally disordered as "each does things in its own way". Here, by analyzing the requirements of this kind of software using feature modeling method, we can abstract their requirements as eight types of service, in which the customer registration service, customer logging service, commodity information searching and browsing service, commodity ordering service are provided to the customers; while the order form manual processing service, commodity information management service, and automatic order form processing configuration service are provided to the shop assistants; the automatic order form processing service is provided as an extra intelligent service. Fig. 6 presents the refinement view of the feature model for a class of Internet shop software systems.

The above feature model can be reused to get the feature model for a specific E-shop. For example, the feature model for an Internet Pet Shop[1)] shown in Fig. 7 is a tailoring

1) Standard JPS is a traditional Web application. To illustrate ABC for the engineering of Internetware, we modify JPS a certain degree, such as making it to satisfy some additional variable requirements. However, the source codes almost remain unchanged, because the additional requirements are mainly satisfied by the self-adaptation of Internetware supported by middleware.

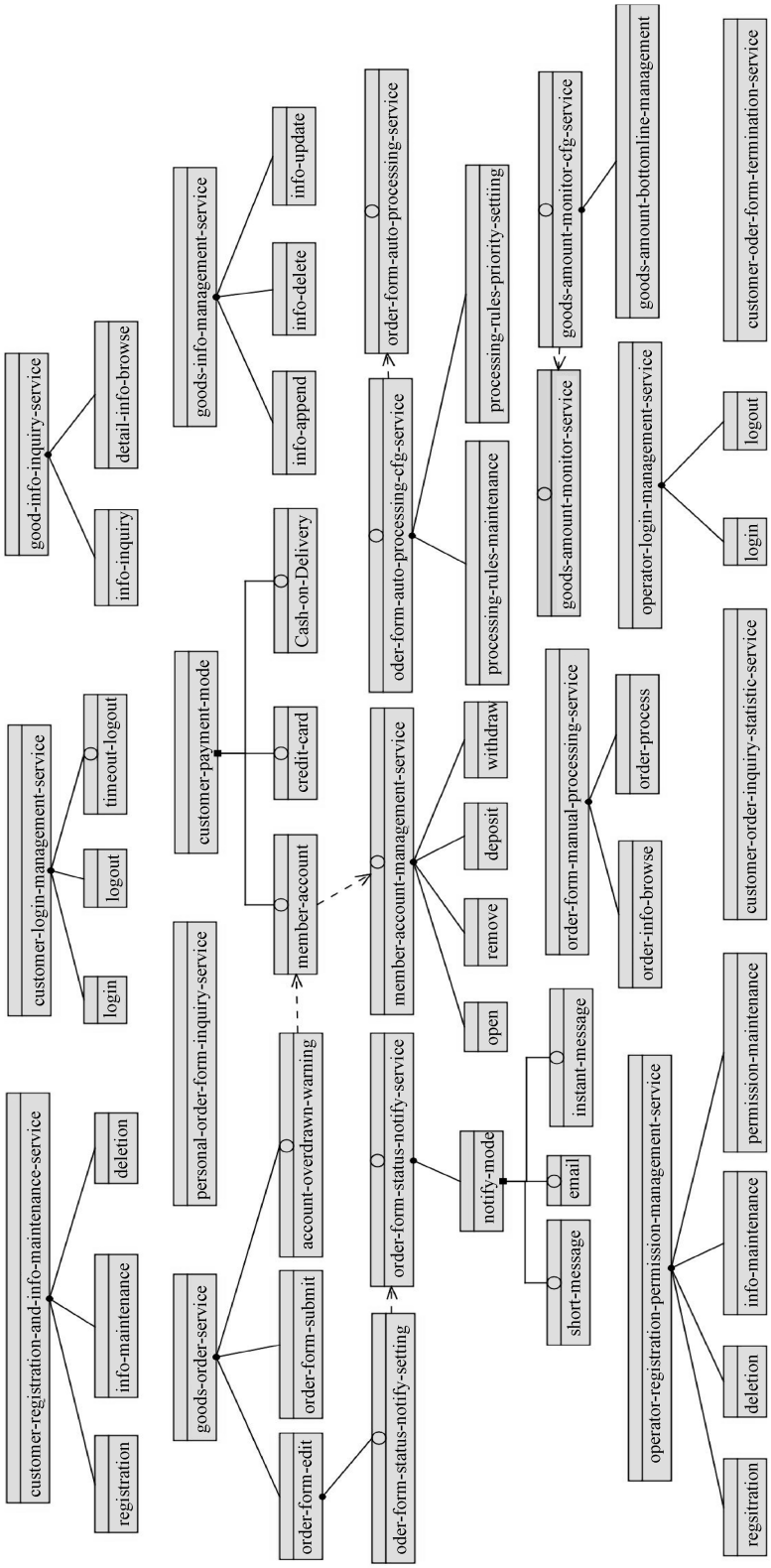

Fig. 6. Domain feature model for E-shops.

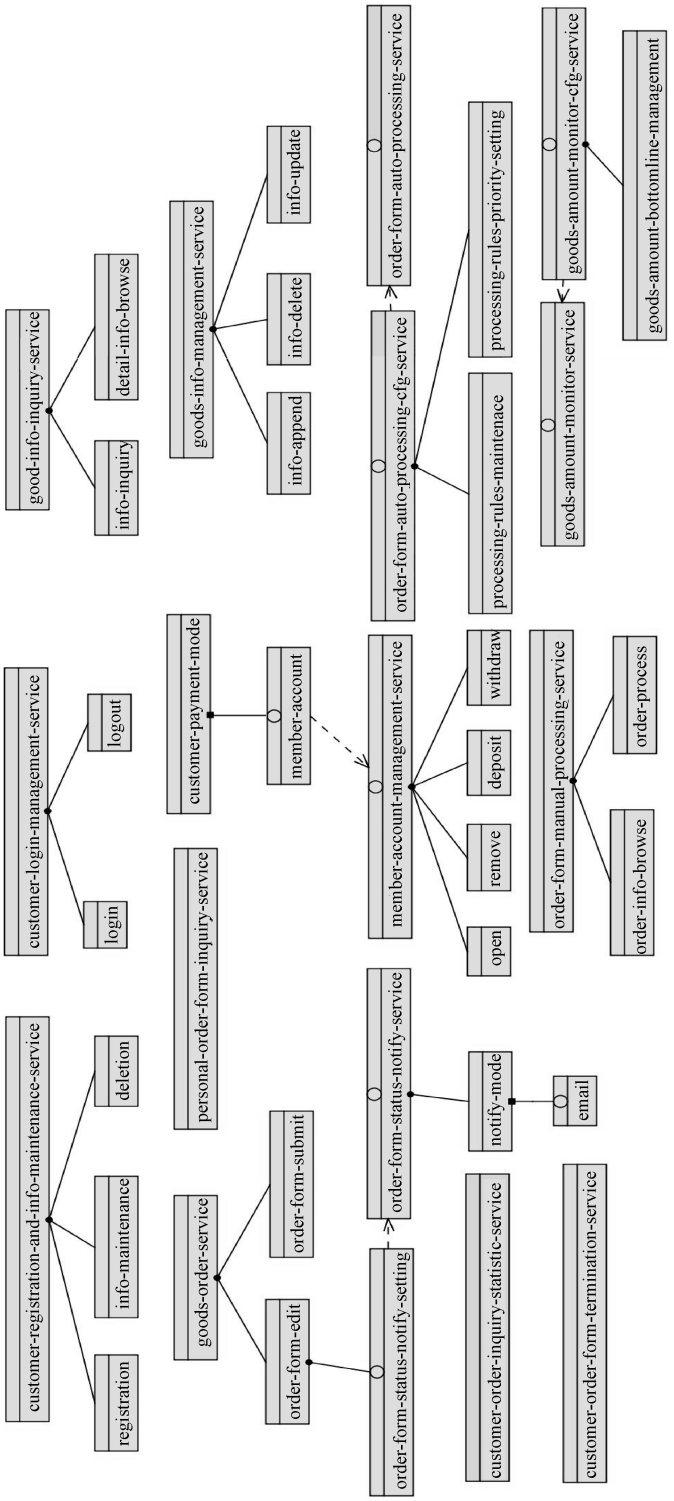

Fig. 7. A feature model for JPS.

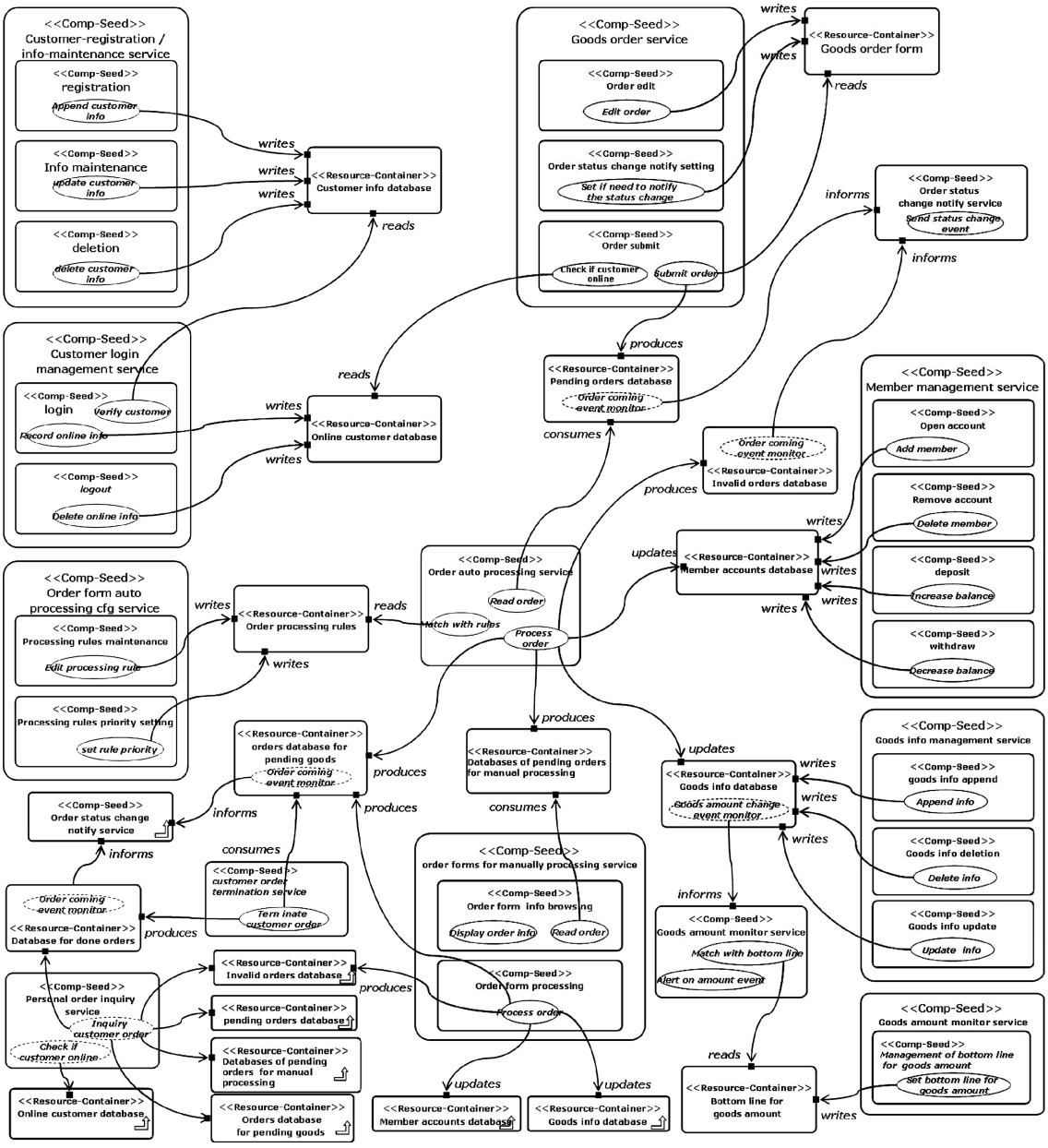

Fig. 8. High-level architecture for JPS.

result of the model given in Fig. 6. In the process of customization, according to the JPS's specific requirements, some variation points in Fig. 6 can be bound or removed, such as removing the feature of "short message" refined from the feature "notify mode".

On this basis, the responsibilities for each feature in the model can be further identified and assigned to the corresponding components so as to establish the traceability between the features in the problem space with the components in the solution space. At the same time, the interactions between components can be found and established by analyzing the interactions between features and the interactions between responsibilities. Finally, the high-level abstract architecture of JPS, shown in Fig. 8, has been constructed.

3 Architecture modeling of self-adaptive internetware

Early research on self-adaptation concentrates on enabling the self-adaptability of a software system "physically". Most of these research leverages technologies that support self-adaptability, such as agents, design patterns and middleware. Recently, it has been well recognized that the key to self-adaptation is to enable the self-adaptability of a software system "logically", that is, how to position the part that should be self-adaptive, how to determine the self-adaptation policies, and how to evaluate the self-adaptation. Considering that most self-adaptation is dedicated to the qualities of a software system, we argue that software qualities should play a central role in analyzing, designing and evaluating the self-adaptability. Meanwhile, SA is not only a blueprint for a software system, but also a carrier of system qualities. Most qualities are evaluated according to SA. The design decisions embodied in SA are almost restricted by quality constraints. Usually, a single design decision may influence several qualities; and in that sense, the design of SA can be seen as tradeoffs between different qualities. Furthermore, SA is also an important artifact for the management of runtime system changes, e.g., dynamic SA (DSA) records allowable system changes explicitly to guide the self-adaptation of a software system at runtime.

Although the existing research provides direct support to the self-adaptability of software systems, there is still much room for improvement. For example, most SA design and evaluation methods remain at a high level, not taking other development phases into account; therefore, the qualities can only be simulated or analyzed in a static way. For such qualities that are related to runtime (e.g., performance, availability), it is difficult to ensure the correctness and precision of their evaluation results. Some adaptation strategies for desired qualities may not be optimal or even wrong. Research on DSA introduces mechanisms to model allowable changes, such that the SA model can be used to guide the system maintenance and evolution at runtime. However, most DSA research concentrates on system evolution. It cares about how to add, delete or modify the system's functions, paying little attention to the system's qualities. Therefore, maintainers are required to adjust the qualities by hand and the system is not self-adaptive. Some techniques that support self-adaptation (e.g., reflective middleware) can be applied to implement self-adaptation policies; however, they have no clue for "why", "when" and "what" to do to achieve the desired qualities.

As discussed before, self-adaptability is one of the most important capabilities of Internetware. And for enabling Internetware self-adaptable, ABC leverages existing efforts on self-adaptation in a systematic manner. First, SA models are used to analyze expected qualities and the part of SA models that should be self-adapted is located. Second, DSA records what should be done at runtime to achieve the desired qualities. Finally, proper self-adaptation mechanisms, like reflective middleware, will implement or execute the designed adaptation at runtime. ABC has no special restriction on the design of SA model according to requirement specifications. For example, a possible way to design an SA model is as follows. Developers can use the feature model-based architecture derivation mentioned above, that is, organize features in a feature model into responsibilities, aggregate semantically-related responsibilities into a component, and produce a draft SA model. If OO analysis and design are adopted, the class diagram for the target system is designed first; after that, the classes can be encapsulated into coarser grained components guided by some principles; finally, an initial SA can also be derived from OOD artifacts. Before the initial SA is implemented, we adopt the process as shown in Fig. 9 to make the SA model self-adaptive.

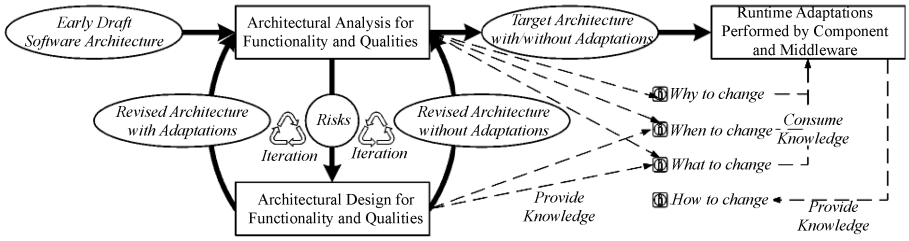

Fig. 9. SA-centric modeling of self-adaptive Internetware.

First, the SA model should be analyzed to ensure the desired functionalities and qualities; otherwise, the SA model should be modified or refined before starting the self-adaptation modeling. During the modification, if the architects find that it is difficult to design a static SA that can achieve the desired functionalities or qualities, it implies the necessity of self-adaptation modeling. The self-adaptation modeling may or may not change the SA model violently. If components involved can be designed as autonomous component which can fulfill the requirements, it is unnecessary to modify the topology of the SA model. For example, if some autonomous components is capable of processing requests according to their priorities, surely it can also satisfy such requirement that the response time should be varied according to the priorities of requests. However, if the component involved cannot be designed as autonomous, or the autonomy is not enough to achieve the desired functionalities or qualities, then it is necessary to figure out a proper self-adaptation policy. Usually, there are two ways to design the self-adaptation policy. One is to clarify what should be done for adaptation, including the triggering conditions and the addition or deletion of specified components or connectors. This is appropriate when only part of SA needs to be adapted. The other is to prepare several SA candidates, each of which satisfies certain functionalities or qualities. These SA candi-

dates are switched at runtime according to the desired functionalities or qualities. The above two ways may be combined to produce a more complex self-adaptive SA. Once the SA model satisfies all desired functionalities and qualities (otherwise, the requirements may be modified), it can be implemented by component composition and deployed into a proper execution platform. The execution platform for self-adaptive Internetware should have sufficient self-adaptation mechanisms, such that it can monitor the runtime information and adapt the runtime SA accordingly. In short, ABC locates the part of Internetware that requires self-adaptation through SA analysis, determines when and what should be done for self-adaptation by SA design, records strategies for self-adaptation in DSA, and interprets these strategies by means of autonomous components or reflective middleware.

The modeling process in Fig. 9 is independent of specific techniques. To validate its feasibility and effectiveness, ABC currently adopts ATAM[12] to analyze the qualities of an SA model, ADD[13] to design an SA model in terms of given qualities, ABC/ADL[7] to describe DSA, and a reflective middleware, PKUAS[14], as well as the autonomous component model[15] to execute architectural adaptations. The process is exemplified as follows.

ATAM is a systematic SA evaluation technique, which is proposed by CMU/SEI and put into practice for several years. All stakeholders are involved in ATAM to investigate whether a specified SA model does satisfy desired quality attributes. In ATAM, scenarios are used to capture requirements on qualities. These scenarios are prioritized and organized by a utility tree. Each quality attribute scenario uses six elements to identify a specific scenario: a source (some entity outside the system) generated a stimulus (a condition to be considered when it arrives at a system) to some artifact (the stimulated artifact in the system) in a specific environment (the condition of the system when the stimulus occurs) and the artifact responses (the activity undertaken after the arrival of the stimulus) to the stimulus by some response measure (the response should be measurable in some fashion so that the requirement can be validated). After a careful analysis, sensitivity points and tradeoff points are identified, which are used to trade off different quality attributes. The ATAM in ABC is regulated to fit our needs; particularly, the semantics of some artifacts are specific to ABC, e.g., the quality attribute scenario is used to analyze the time when self-adaptation is necessary, the source and stimulus are used to record the external conditions or events that trigger self-adaptation, all involved elements are recorded in environment and artifact, and the threshold of quality attributes is recorded by response and response measure.

We illustrate the ATAM analysis by Java Pet Store (JPS). JPS is divided into four parts, namely the pet store (PS), order processing center (OPC), administration and supplier. The PS part interacts with end users directly, and sends order forms to OPC. OPC is responsible for the processing of order forms and for sending order requests to the administration. The administrators approve (charging fees accordingly from the user's credit card) or reject (if the credit card has no enough money) the order forms through the administration part. The results are then sent back to OPC. For approved order forms, OPC forwards the order forms to the supplier, who ships the ordered pets to the end user and

sends according invoices to OPC; for rejected order forms, OPC sends a notification email to the end user. Table 1 shows part of quality attribute scenarios. These scenarios are analyzed with the utility tree, which prioritizes quality attribute scenarios according to the category of qualities. The importance as well as the difficulty to the support of each scenario is also determined. An important output of ATAM analysis is sensitivity points and tradeoff points derived from the quality attribute scenarios. A sensitivity point is a property of the architecture that is critical for the achievement of a particular quality attribute (e.g., using encryption to achieve confidentiality). A tradeoff point is a sensitivity point that is sensitive for multiple quality attributes (e.g., encryption improves security but increases latency). From ABC's perspective of self-adaptive analysis and design, the quality attribute scenarios imply when self-adaptation is necessary; the according sensitivity and tradeoff points show which part of the SA model should be adapted when some response measures are not satisfied. Based on quality attribute scenarios as shown in Table 1, we can find that the first 5 scenarios are concerned with performance. After evaluating the SA model of JPS, we can find that most components involved in these scenarios access databases. A performance sensitivity point of database access is then deduced. There are usually two ways to access a database. One is accessing databases whenever data is read or written (in this case, less memories are required at the cost of longer response time), and the other is buffering data and accessing databases only when data is written (in this case, shorter response time is acquired at the cost of more memories). As a result, components that access databases should compromise between performance and memory usage.

After the adaptation point is located, the corresponding adaptation policy should be designed. There are three possible design decisions for the above example. One is to modify the SA model for JPS. For components (including Customer, Order, Catalog) that access databases, they are assigned two implementations (one uses data buffering, while the other does not). Both implementations are composed into the target system. When there are enough memories, the implementation using data buffering is used for better performance; when available memories become limited, the other implementation will be used to ensure the stability of the whole software system. The switching between implementations can be implemented by the connection or disconnection to corresponding connectors. Another design decision is based on the self-adaptability of middleware. If those components leverage the data access services provided by middleware (e.g., entity EJB), the time when data buffering should be used is left to middleware. What middleware needs to know is the switching rules (for example, a possible rule may be that data buffering should not be used when more than 80% of memories have been consumed). The rules are part of configurations for the target software system. The last decision is concerned with the self-adaptability of software entities. When involved components are autonomous, the switching of implementation is left to components themselves. What designers need to do then is the determination of the switching rules.

Similar to the above analysis, we can also find[16] that 1) the process control in OPC is a tradeoff point for performance and security; accordingly, the design of process control should be revised to enable adaptability; 2) the encryption of order forms is also a trade-

Table 1 Quality attribute scenarios for JPS

No.	Source	Stimulus	Environment	Artifact	Response	Response Measure
Scenario 1	End User	Detail pet information in the catalog	Runtime	Web site	The according pet information, including pictures	Response time is less than 5 sec
Scenario 2	End User	Query of pets in the catalog	Runtime	Web site	Query result	Response time is less than 2 sec
Scenario 3	Orders to be processed	Sending orders to OPC	Runtime with normal load	OPC	Order processing	More than 10 orders are processed in 1 min
Scenario 4	End User	Order submission	Runtime with normal load	OPC	Notification to the end user (rejected or approved)	Response time is less than 5 min
Scenario 5	End User	Order submission	Runtime with heavy load	OPC	Notification to the end user (rejected or approved)	Response time is less than 6 min
Scenario 6	End User	Order submission	Runtime	The whole system	Detail information on orders	All shopping records
Scenario 7	Orders to be processed	Sending orders to OPC	Runtime	OPC	Order reception	The order should be encrypted
Scenario 8	Maintainer	Transporting to other databases	After delivery	The whole system	The transportation can be done easily	The transportation can be done within 12 h

off point for performance and security; we can design proper connectors to make it adaptable. All these designs for adaptability can be implemented, more details of which are discussed in the next section.

4 Reflective middleware as internetware operating platform

As shown before, the operating platform for Internetware should not only possess most capabilities in main stream platforms, such as interoperability, concurrency, security, transaction, persistency; but also provide mechanisms for self-adaptability, including that of runtime monitoring and management, rule-based reasoning. The operating platform for ABC is a software middleware, named PKUAS[17,10], which has been successfully applied in such fields as finance, communication, education and governance. It is one of the core products of Orientware, a middleware suite sponsored by National 863 High-Tech Program. PKUAS is a J2EE-compliant application server which is the platform including J2SE, common services and one or both of Web Container and EJB Container. It provides all functionalities required by J2EE v1.3 and EJB v2.0 in its componentized structure, as shown in Fig. 10. The characteristics of PKUAS include:

● Micro kernel based-componentized platform: The design of PKUAS embodies the idea of componentization. It provides a registry and an invocation framework for the above platform components and other management entities, like class loading, relation, timer and monitor, thus presenting a componentized architecture. This architecture is based on a set of fundamental functions, which forms the micro kernel. Other plat-

form-related functions are encapsulated into independent modules (called system components). They can be customized or extended according to specific domains. When PKUAS starts up, the micro kernel is responsible for the organization of a domain-specific component operating platform. In this way, PKUAS has great flexibility and extensibility.

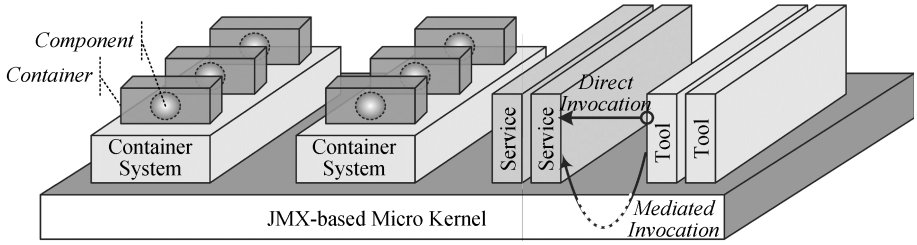

Fig. 10. Componentized structure in PKUAS.

● Container system supporting online evolution: A container provides a runtime space for the components in the deployed applications with lifecycle management and contract enforcement. PKUAS implements standard EJB containers for stateless session beans, stateful session beans, bean-managed entity beans, container-managed entity beans and message-driven beans. One instance of a container holds all instances of one EJB. And a container system consists of the instances of the containers holding all EJBs in a single application. The container system supports online evolution by adding, deleting, or replacing components at runtime. In this way, the applications can be debugged, upgraded, or optimized at runtime.

● Open interoperability framework: PKUAS supports most main-stream interoperability protocols, including IIOP, JRMP, SOAP, and EJB Local. It even allows user-defined interoperability protocols. The interoperability protocols used are determined at deploy time; therefore, the choice of protocols used is transparent to the application developers.

● Standard and extensible service: PKUAS provides common services as specified in J2EE, including data, communication, security, and transaction. With the support of PKUAS, developers need to care about only business logics. Besides, PKUAS supports extended services, such as logging, clustering, and concurrency. Specifically, users are allowed to define specific services. For example, PKUAS supports a special data integration service that is targeted to software project management systems only.

● Rich tool support: PKUAS has provided a set of development, deployment and management tools for J2EE applications. For example, the deployment tool facilitates the composition, deployment, redeployment and un-deployment of J2EE applications at development and testing phases. It is also capable of modifying deployment descriptors or environment properties dynamically. Another example, a web-based management tool is capable of monitoring and managing J2EE applications as well as J2EE servers at runtime.

Based on the above characteristics, PKUAS supports SA-based reflection and the autonomous component, which can be used to support the self-adaptability of Internetware application structure and entities, respectively.

ware application structure and entities, respectively.

4.1 SA-based reflection

Being one of the hot topics in the researches and practices on next generation middleware, reflective middleware is considered as the fundamental approach to adaptable middleware. The users are allowed to access and operate in the runtime states and behavior of middleware in a restrict way by the mechanism of reflection. By reflection, we mean that a system can provide a self-representation on its states and behavior. The self-representation is always consistent with the runtime states and behavior, that is, changes to the self-representation apply to the runtime states and behavior immediately, and *vice versa*. PKUAS implements an SA-based reflection, whose self-representation is SA and then middleware as well as its applications can be observed, reasoned, and manipulated from the perspective of SA[10].

As shown in Fig. 11, the states and behavior of middleware platform and applications can be observed and adapted from the perspectives of the platform RSA and application RSA respectively. The platform RSA represents the implementation of middleware platform as components and connectors. Middleware applications are invisible or represented as the attributes of some components. For example, J2EE application server consists of containers and services and the J2EE application consists of EJBs or Servlets. In the platform RSA, the containers and services are represented as components; their interactions or dependencies are represented as connectors; and the EJBs or Servlets are represented as the attributes of the containers. For reflective middleware, the platform RSA is the representation of a set of platform-specific meta entities, which are responsible for the reflection of base entities in the reflective middleware. On the other hand, the application RSA represents middleware application as components and connectors. Middleware platform details are typically represented as constraints or attributes of components and connectors. For example, J2EE security and transaction services are represented as the security and transaction constraints on the EJBs or Servlets. For reflective middleware, the application RSA is the representation of a set of application-specific meta entities, which are responsible for acquiring and maintaining the RSA of the application. In our implementation, the platform RSA is implemented as a set of meta entities, which collect the structural information on PKUAS componentized platform and monitor its states and behavior. The platform will be adjusted as soon as its meta data changes. Similarly, the application RSA is implemented as a set of application-specific meta entities, which are built on top of meta entities of platform RSA. They are responsible for maintaining the application RSA as well as rich semantics from the SA in design time. The modification to application RSA is done by the meta entities of platform RSA indirectly. Users are allowed to access or manipulate the platform and application RSA through reflective API.

In our opinion, different structural adaptations require different reflection mechanisms. For example, the addition or deletion of components requires the mechanism of hot deployment, the replacement of components requires the mechanism of online evolution, the adjustment of connectors usually depends on the interoperability framework, various

constraints on components, connectors as well as SA (e.g., transaction, security, persistency, availability, etc.) require various services. More details can be found in refs. [10, 11, 14, 16, 18, 19]. For the space limit, only the detail for component switching is discussed for illustrating the above adaptation sample of JPS.

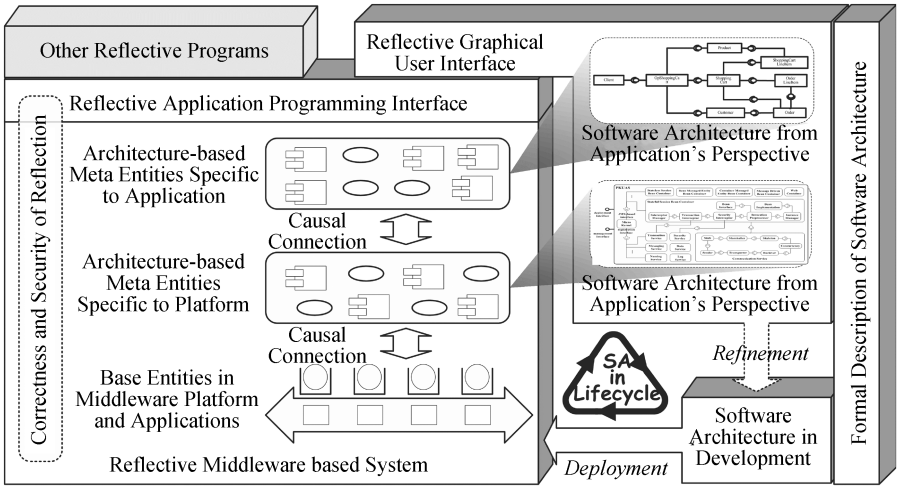

Fig. 11. Framework of SA-based reflective middleware.

As Fig. 12 (a) shows, in a typical EJB container, the component interface, implementation, as well as its context are independent of each other. Each interface has an implementation, which may have several instances at runtime. Each instance has its own context, which is used to maintain the instance's information on transaction, security, session, and so on. To support dynamic switching of component implementations, the above structure should be modified. As shown in Fig. 12 (b), in PKUAS, the interface level presents the EJB managed by the container. The interface instances level presents multiple EJB instances for dealing with the concurrency. The implementation instances level presents the implementation instances in an EJB array. In the three-level management, an EJB instance still keeps an EJBContext, but the implementation instance is separated from the EJB instance into the implementation instance level. This management makes it possible that an EJB keeps multiple implementations in the container without breaking

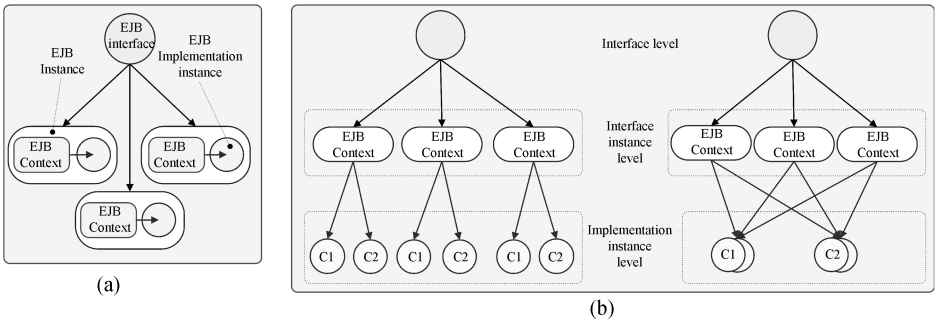

Fig. 12. Automatic switching of component implementation in PKUAS. (a) Management of EJB instances in standard J2EE; (b) management of EJB instances in component array.

the concurrency. If a component implementation is isomorphic, the according implementation instance can be shared by several interfaces; otherwise, an interface instance has its own implementation instance exclusively. The dynamic switching of component implementations is done by forwarding requests to different implementation instances according to preset switching rules. The switching rules are defined in a similar way to those for autonomous components, which will be discussed in the next section.

4.2 Autonomous components

Autonomous components retain the features and characteristics of traditional components on the one hand, and they have a conjunction with agents in possessing autonomous behavioral abilities on the other hand. However, autonomous components are neither traditional components nor agents. Like components, autonomous components have relatively independent functionalities. They provide specified services for outside use, and they can be reused and assembled into software systems. Nevertheless, autonomous components are software entities with autonomy. They are driven by their own goals, and they provide services because doing so can help them to achieve their goals. Even when they are achieving their goals, they can adjust their behavior in responding to the changes of their environment via changing the time and way of providing services.

Specifically, an autonomous component can be formally described from five aspects[14]. First, its behavior is goal-driven. Next, it also provides services to the outside though it is no longer passive and can even autonomously decide whether or not to provide its services. Third, the use contract describes how users can access the services in an appropriate way. Fourth, it is situated in the environment. It can perceive the changes of the environment and adjust its behavior to adapt to the changing environment. Finally, the dependency relationship between its goals and the environment determines how it can reason about and decide its behavior to achieve its goals.

Because an autonomous component is specified mainly via its goals, the services, and the environment, it is crucial to reflect the interaction relationships among the goals, the services, and the environment in the implementation of the autonomous component. Therefore, we use the goal-driven rules and plan to relate the goals, the environment and the services together.

When the environment comes upon the state which allows the autonomous component to achieve a goal, the autonomous component will trigger corresponding rules to activate the process of achieving the goal.

$$f(E) \rightarrow \text{Activate}(G),$$

where f is a function of the environment and G is one of the goals of the autonomous component.

After the process of achieving a goal is activated, the autonomous component will plan the achievement of the goal, i.e., determine how to take actions to achieve the goal. Among the actions that the autonomous component takes to pursue the achievement of its goal, there may involve the activities of requesting or providing services.

$$\text{Plan: } G \rightarrow S^*,$$

where a plan is a mapping from a goal to a sequence of actions (probably including ac-

tivities of providing services).

When the autonomous component perceives a request for service from others, it will reason about its behavior to check out whether the request will activate the process of achieving a specific goal. If yes, it will respond to the request while achieving its goal; otherwise, it will reject or ignore the request. Even for the same request for service, the autonomous component possibly activates different goals, which may lead to different ways of responding to the request, under different states of the environment.

Although an autonomous component has some extends of autonomy, it must expose its interface and provide normal functionalities, like those traditional components, so that it could be assembled into software systems. Therefore, what makes an autonomous component different from traditional components is that the autonomous component takes a different way to implement its interface, i.e., it may adjust its behavior to provide services and implement its interface according to the states of the environment. However, users always use the component and request its services through its interface and they care about neither how the autonomous component implements its interface nor how it adapts its behavior while providing services. We can say that the true difference between an autonomous component and a traditional component is the implementation structure and the runtime behavior. The implementation structure of an autonomous component is shown in Fig. 13.

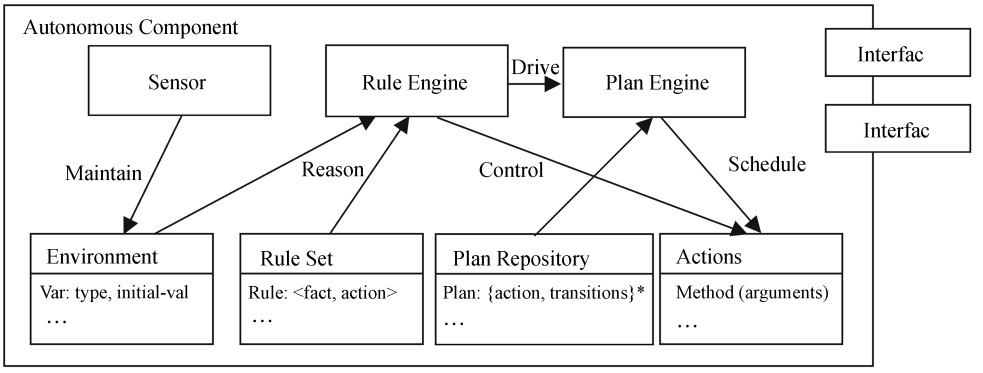

Fig. 13. The implementation structure of the autonomous component.

In Fig. 13, the environment information includes those environmental variables or data that the autonomous component cares and can perceive. The sensor is used to capture the states and changes of the environment and it is also responsible for maintaining the environment information according to the perceived information. The rule set specifies those rules driving and controlling the behavior of the autonomous component. A plan in the plan repository specifies the sequence of actions for the autonomous component to achieve a goal or implement a service. In order to simplify the implementation, we currently adopt an off-line way to define plans for the autonomous component.

The autonomous component will not take any actions unless a specific rule is triggered. The rule engine is responsible for deducing and triggering rules based on the environment information and the rule set. The plan engine will be actuated by some rules and is

responsible for selecting appropriate plans to achieve goals.

The behavior mode that an autonomous component responds to requests and provides services can be described as follows. 1) When the autonomous component receives a request for service, its rule engine will reason about its behavior based on the current environment and the rule set. If there is no such rule being able to be triggered, the autonomous component may not take any actions or directly rejects the request for service; otherwise, it may activate the plan engine to select and execute an appropriate plan to carry out the service. 2) In the execution of the selected plan, if all actions involved in the plan can be performed correctly and their executions do not violate any environment constraints, the requested service will be provided successfully; otherwise the autonomous component will report an exception of service failure. 3) Even if there is no outside request for service, the rule engine may also trigger rules to carry out some actions according to the current environment states.

Since the usage of an autonomous component is like that of a traditional component, we add a new container into PKUAS to support the run of autonomous components. In the implementation, we integrated an open-source rule engine called Drools and developed a plan engine to supply the capabilities of rule-driving and planning for autonomous components, as shown in Fig. 14. Just like the EJB containers in PKUAS, the autonomous component container provides the running space for the instance of an autonomous component, managing the life cycle of the autonomous component and the communications between the autonomous component and other components. The interceptor in the container implements the sensor of the autonomous component. The interceptor is responsible for intercepting communications between autonomous components and supports the interoperability of autonomous components. On the other hand, it captures the state information about the environment and maintains the environment information, which will be used by the rule engine to infer and trigger rules.

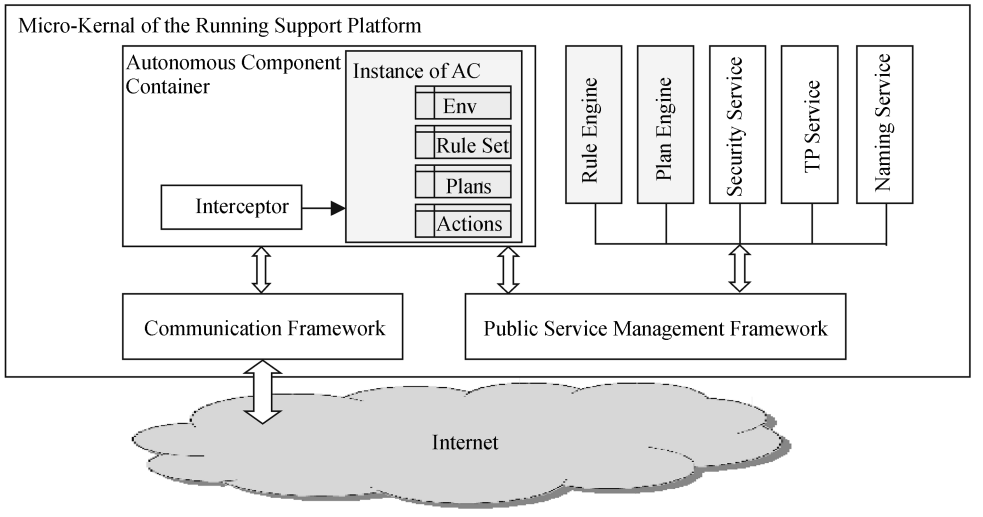

Fig. 14. The runtime support platform for autonomous components based on PKUAS.

Considering that every instance of the autonomous component has a rule engine and a plan engine, we implement the rule engine and the plan engine as public services and put them into the public service management framework in the middleware. The autonomous component container will call those public services to control the behavior of the autonomous component. When the autonomous component container receives a request for service, it will transfer the request to the interceptor, and then the interceptor will activate the rule engine and the plan engine to schedule the executions of the actions of the autonomous component.

For example, as mentioned above, when a JPS entity component (e.g., Customer) becomes the performance bottleneck of the system, it possibly needs to change the structure or the implementation of the entire system. In order to improve the structural and behavioral adaptabilities of the system, we can implement these components as autonomous components. We can customize specific behavior rules for the autonomous components to enable them to switch automatically between different service providing modes according to the runtime states of the environment and the system.

```
<rule name="normalCustomer">
    <parameter identifier="customer">
        <class>Customer</class>
    </parameter>
    <java:condition>
        memUsage/memAllocated < 0.8
    </java:condition>
    <java:consequence>
        loadNormalCustomer ();
    </java:consequence>
</rule>
```

```
<rule name="cacheCustomer">
    <parameter identifier="customer">
        <class>Customer</class>
    </parameter>
    <java:condition>
        memUsage/memAllocated >= 0.8
    </java:condition>
    <java:consequence>
        loadCachedCustomer();
    </java:consequence>
</rule>
```

Fig. 15. Rules for the autonomous component to adjust its behaviors.

For example, for the Customer, it can provide its services in different ways when it uses different modes to access the database. Thus, we can specify behavior rules for the Customer to adopt different ways to perform its service as follows (Fig. 15). These rules specify that, when the ratio between the amounts of occupied memory and the allocated memory is greater than 0.8, which implies that the request for the Customer's services has already become the performance bottleneck, the Customer will start the implementation version that uses the cache memory, otherwise it will start the normal version.

This kind of adaptation may influence the interaction relationships inside or among autonomous components. Under a specific circumstance, the changes of the interaction relationships among autonomous components may result in the reconfiguration of the software architecture.

5 Conclusion

In the past several decades, software technologies have experienced a series of development. The main line for this development is clear: the granularity of constituent software entities grows coarser; the software model fits our thinking better and better; as the

power of software execution platform keeps increasing, more and more underlying complexities as well as those relating to software development are hidden; software techniques have been applied to more and more real world areas. As the Internet keeps rapid and continuous development, the existing software technologies may confront more and more requirements and challenges. Typically, due to their static and close nature, the traditional software technologies are not adequate and suited for the open, dynamic and ever-changing Internet; instead, a new software paradigm, Internetware, appears naturally. It accommodates the open, dynamic and ever-changing Internet in a better way, manifesting itself as flexible, continually reactive software with multiple compatible goals. The engineering of Internetware requires innovations on traditional software development methods and techniques. In this paper, we introduce a component-oriented, architecture-centric and middleware-based approach, called ABC, to support the engineering of Internetware. ABC spans over all phases of software lifecycle, including the analysis, design, implementation, deployment, maintenance and evolution of Internetware. In particular, feature models are used to model, organize and manage "disordered" software resources; adaptive SA models are used to design a self-adaptive Internetware; reflective middleware is used to enforce the self-adaptation. Whatever, there are many open issues to be addressed, such as improving the automation of design for self-adaptability, incarnating cooperation among Internetware, refining component models dedicated to Internetware, and so on.

Acknowledgements This work was supported by the National Basic Research Program of China (973) (Grant No. 2002CB312003); the National Natural Science Foundation of China (Grant Nos. 60233010, 90612011, 90412011, 60403030, 60303004), and the Natural Science Foundation of Beijing (Grant No. 4052018).

References

1 Yang F Q, Mei H, Lu J, et al. Some thoughts on the development of software technologies. Acta Electronica Sinica (in Chinese), 2003, 26(9): 1104－1115

2 Mei H, Chang J C, Yang F Q. Software component composition based on ADL and middleware. Sci China Ser F-Inf Sci, 2001, 44(2): 136－151

3 Zhang W, Mei H. A feature-oriented domain model and its modeling process. Journal of Software (in Chinese), 14(8): 1345－1356

4 Zhang W, Mei H, Zhao H Y. A feature-oriented approach to modeling requirements dependencies. In: Proceedings of 13th IEEE International Requirements Engineering Conference (ICRE), La Sorbonne, France, August 29-September 2, 2005. 273－282

5 Zhang W, Zhao H Y, Mei H. A propositional logic-based method for verification of feature models. In: Proceedings of Sixth International Conference on Formal Engineering Methods (ICFEM). Lecture Notes in Computer Science Series (LNCS 3308), Berlin: Springer, 2004. 115－130

6 Zhang W, Mei H, Zhao H Y, et al. Transformation from CIM to PIM: Feature-oriented component-based approach. 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2005), Montego Bay, Jamaica, October 2-7, 2005, Proceedings. LNCS 3713: 248－263

7 Mei H, Chen F, Wang Q X, et al. ABC/ADL: An ADL supporting component composition. In: George C, Miao H K, eds. Formal Methods and Software Engineering, LNCS 2495. Heidelberg/New York: Springer-Verlag. Proceedings of 4th International Conference on Formal Engineering Methods, ICFEM2002, Shanghai, China, Oct. 2002, 38－47

8 Yang J, Huang G, Chen X P, et al. Consistency assurance in flattening hierarchical architectural models. Journal of Software (in Chinese), 2006, 17(6): 1391－1400
9 Lan L, Huang G, Ma L Y, et al. Architecture based deployment of large-scale component based systems: The tool and principles. 8th International SIGSOFT Symposium on Component-based Software Engineering (CBSE), USA, 15－16 May 2005, 123－138
10 Huang G, Mei H, Yang F Q. Runtime software architecture based on reflective middleware. Sci China Ser F-Inf Sci, 2004, 47(5): 555－576
11 Huang G, Mei H, Yang F Q. Runtime recovery and manipulation of software architecture of component-based systems. Inter J Auto Software Eng, 2006, 13(2): 257－281
12 Kazman R, Klein M, Clements P. ATAM: Method for architecture evaluation. Technical Report. Software Engineering Institute, Carnegie Mellon University, 2000
13 Bass L, Clements P, Kazman R. Software Architecture in Practice. 2nd ed. Boston: Addision-Wesley, Apr 9, 2003
14 Mei H, Huang G. PKUAS: An architecture-based reflective component operating platform. 10th IEEE International Workshop on Future Trends of Distributed Computing Systems, 2004, Suzhou, China. 163－169
15 Jiao W, Zhu P P, Mei H. Modeling internet-based software systems using autonomous components. Chinese J Electronics, 2006, 15(4): 593－598
16 Shen J R, Sun X, Huang G, et al. Towards a unified formal model for supporting mechanisms of dynamic component update. The Fifth Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE'05), Lisbon, Portugal, September 5-9, 2005. 80－89
17 Zhu Y, Huang G, Mei H. Modeling diverse and complex interactions enabled by Middleware as connectors in software architectures. 10th IEEE International Conference on the Engineering of Complex Computer Systems (ICECCS2005), Shanghai, China, 16-20 June 2005. 37－46
18 Teng T, Huang G, Li R C, et al. Feature interactions induced by data dependencies among entity components. 8th International Conference on Feature Interactions in Telecommunications and Software Systems (ICFI05), 28th June to 30th June, 2005, Leicester, UK. 252－269
19 Liu T, Huang G, Fan G, et al. The coordinated recovery of data service and transaction service in J2EE. In: Proceedings of 29th Annual International Computer Software and Applications Conference (COMPSAC05), Edinburgh, Scotland, July 2005. 485－490