ISSN 1673-9418 CODEN JKYTA8
Journal of Frontiers of Computer Science and Technology
1673-9418/2008/02(04)-0389-16
DOI:10.3778/j.issn.1673-9418.2008.04.006

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

Using Abstract Quality Types to Construct High Quality Internetware*

QU Youtian[1+], WU Zejun[2], JIAO Wenpin[3,4], CHEN Tianzhou[5], HE Guolong[1]

1. College of Mathematics, Physics and Information Engineering, Zhejiang Normal University, Jinhua, Zhejiang 321004, China
2. China Construction Bank (Hubei Branch), Wuhan 430015, China
3. Software Institute, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China
4. Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing 100871, China
5. Department of Computer Science, Zhejiang University, Hangzhou 310027, China

+ Corresponding author: E-mail: quyt@zjnu.cn

用抽象质量类型构建高质量网构软件*

瞿有甜[1+],吴泽君[2],焦文品[3,4],陈天洲[5],何国龙[1]

1. 浙江师范大学 数理与信息工程学院,浙江 金华 321004
2. 中国建设银行(湖北省分行),武汉 430015
3. 北京大学 信息科学技术学院 软件研究所,北京 100871
4. 高可信软件技术教育部重点实验室,北京 100871
5. 浙江大学 计算机系,杭州 310027

摘 要:在开发基于因特网的软件系统(文中称为网构软件)时,在开发过程的各个阶段都需要考虑软件系统的多种质量属性,这样网构软件的质量才能在软件系统的规约和设计阶段进行推导和预测,从而在部署和运行阶段进一步进行评估和验证。提出了一种新的抽象概念,即抽象质量类型,将软件实体的数据状态、相关的操作、质量属性以及保证质量得以实现的环境封装在一个统一的语法单元中,从而为建模软件实体以及推导其功能和非功能属性提供了一种严格的手段。探讨了基于软件体系结构及软件交互过程的抽象质量类型的组合,这为

* the National Natural Science Foundation of China under Grant No.60773151 (国家自然科学基金); the National Grand Fundamental Research 973 Program of China under Grant No.2005CB321805 (国家重点基础研究发展规划(973)); the National High-Tech Research and Development Plan of China under Grant No.2006AA01Z175 (国家高技术研究发展计划(863)); the Natural Science Foundation of Zhejiang Province of China under Grant No.M603245,Y106469 (浙江省自然科学基金).
Received 2008-02, Accepted 2008-05.

构建高质量网构软件提供了一定的形式化基础。最后,还描出了一个系统原型来展示如何利用抽象质量类型构造网构软件,并验证和提高网构软件系统的相关质量属性。

关键词:构件;软件质量;抽象质量类型;网构软件

文献标识码:A **中图分类号:**TP301

QU Youtian, WU Zejun, JIAO Wenpin, et al. Using abstract quality types to construct high quality internetware. Journal of Frontiers of Computer Science and Technology, 2008,2(4):389-404.

Abstract: In the development of software systems on the Internet (referred to as Internetware), multiple quality properties should be considered along with the development process so that the quality of Internetware can be inferred and predicted at the specification and design stages and be evaluated and verified at the deployment and execution stages. A new abstraction notion, Abstract Quality Type (AQT), is put forward to encapsulate data types, associated operations, quality properties and the environment guaranteeing the qualities into an uniform syntactical unit, which provides rigorous ways to model software entities and to reason about the functional and non-functional properties. The composition of AQTs is also studied based on the software architecture and the process of interactions, which offers a solid foundation for constructing high quality Internetware. Finally, a prototype is described to illustrate how to construct Internetware by using AQTs as well as how to verify and improve a special kind of quality property of Internetware.

Key words: component; software quality; abstract quality type; Internetware

1 Introduction

The worldwide expansion of the Internet has resulted in the dramatic change of the mode of constructing software systems on the Internet (referred to as Internetware in the paper). Service-oriented architecture[1] has been becoming the foundation of constructing Internetware. Services scattered on the Internet are independent and they can be put together dynamically at run time and possibly across administrative domains. Service providers are autonomous and can decide by themselves whether and how to respond to requests for services, and they even change their behavior modes without notifying the service consumers. Consumers should be assured that they could be served as required. Furthermore, service providers and consumers generally take actions on behalf of diverse interest groups and execute in different administrative domains, and their behaviors must not damage the profit of each other. Therefore, services cannot be invoked in the traditional sense and Internetware may have to be dynamically formed by independent and autonomous software entities. How to construct high quality Internetware has become one of the grand challenges in the era of the Internet.

To model and construct high quality Internetware, we should take into consideration both the functional requirements and the non-functional requirements (or quality properties)[1]. However, cur-

[1] In the literature, non-functional requirements, constraints, goals and qualities are often referred to the same properties of software systems[18]

rent existing approaches usually model those functional and quality properties separately. For example, in the component-based software development, component developers are required to only focus on the business logic of applications whilst the non-functional properties (i.e., quality constraints of the components) are left to the component deployment phase, which will be specified in the deployment information and finally guaranteed by the running support platform of components.

Separating the functional and non-functional requirements in the development of components can bring us many advantages. For instance, when the functional requirements change, we can modify independently the implementation of the component of the rest of the application. When the non-functional requirements change, we even need only to re-describe the deployment information without modifying the component's implementation.

However, the separation of functional and non-functional requirements results in much bewilderment in the process of assembling systems.

First, the composition of quality properties is even impossible or it is possible but not at the same phase as that of functionalities, which makes it difficult to reason about the whole characteristics of the systems according to the properties of the involved components. Quality properties usually cannot be refined, which makes it impossible to decide the quality requirements of components according to the global quality requirements of the systems.

Second, it is a paradox dealing with the functional and quality requirements of components separately. When we are constructing systems by using components, we must be concerned with the system quality properties. However, if we consider the systems as composite components, we should only focus on the business logics of the applications while developing the composite components based on the philosophy of the traditional component-based development approaches.

Thirdly, the outstanding advantage of components is that it is possible to separate the process logics of components from their running environment. However, the qualities of a software system are not only related to the involved components but also dependent on the environment where the system is situated and runs. The separation will make it impossible to reason about the system properties based on the properties of components at the development phases.

Currently, software qualities have become one of the main concerns in the development of software systems. If we cannot reason about the wholeness of the system based on the properties of the involved components, it will be very difficult to construct high quality software systems. In that case, the development of high quality software systems will lack of a rigorous foundation and developers will have to take ad hoc methods to cope with specific quality requirements of software systems.

In this paper, we put forward a new notion to abstract and model software entities as a foundation for constructing high quality Internetware.

Generally, there are two fundamental kinds of abstraction in the development of software systems, i.e., data abstraction and process abstraction. In the early time, these two kinds of abstraction were dealt with separately and until abstract data type and object-orientation became the mainstream techniques in the development of software systems, they were combined into a unit.

In our opinions, there is another kind of abstraction crucial to the development of contemporary

software systems, i.e., quality abstraction. The contributions of this paper are as follows.

First, we propose a new notation to integrate data abstraction, process abstraction and quality abstraction together. As people do while naming abstract data type, we name an unit encapsulating data types, processes (or operations) and qualities as Abstract Quality Type (Abbreviated as AQT).

By encapsulating data types, associated operations and qualities into an unit, it will be possible for us to develop a rigorous way to integrate and reason about non-functional properties of software entities from design-time to run-time.

Second, with the uniformity, we can truly define Internetware recursively from simple components and composite components. Subsequently we can infer the properties of systems from integrated components and inversely select components to integrate systems and improve the quality of the integrated systems.

In the following context, Section 2 describes the formal definition of AQT and gives some examples of AQTs. Section 3 probes into the composition of AQTs to offer a foundation for composing high quality Internetware. Section 4 describes a prototype for verifying and improving the quality of Internetware. The last two sections compare with some related work and make some conclusion remarks on our current work.

2 Abstract Quality Type

Before giving the formal definition of abstract quality type, we will first introduce some preliminary concepts.

2.1 Related Concepts

Environment. While seeing about the qualities of software entities, we should consider all factors that may affect the qualities. However, currently, software systems are often presumed as closed worlds and only pure software factors are taken into consideration.

In fact, the qualities of Internetware are affected not only by the systems' own properties, such as coding and architecture, but also by the environments where the systems are situated. Therefore, while exploring properties of the AQT, we should take the environment factors into consideration as well.

The environment of a software entity provides the living space and resources for the entity to perform actions and achieve goals. A software entity can interact with the environment via perceiving and affecting the environment. Without the environment, an entity cannot exist and run to implement its functionality.

If we view the environment that an AQT is situated as a world formed purely by other AQTs, we can regard the environment of an AQT as a collection of other AQTs that are closely related with the AQT. Those AQTs involved in the environment will contribute more or less to the implementation and run of the AQT.

Suppose that $\mathcal{AQT}$ is the set of AQTs, an AQT's environment (notated as $\mathcal{E}_a$) is a subset of $\mathcal{AQT}$ in which there is a tight relationship between the AQT and each element.

$$\mathcal{E}_a \in P\ \mathcal{AQT} \wedge \forall x \in \mathcal{AQT}(\text{dependOn}(a,x)) \quad (1)$$

$P\ \mathcal{AQT}$ is the power set of $\mathcal{AQT}$ and $\text{dependOn}(x,y)$ is a predicate representing that x depends on y in terms of functionality as well as non-functionality, which will be discussed in detail in the following context.

Quality. Quality is a kind of evaluation to a specific discussed object, which reflects how well the object satisfies the expectation of people (or the

reviewer).

The quality of a software entity is related with its internal states (including data, processes and other qualities) and the external environment.

Suppose that q is one of the quality properties to which that the software entity A is devoted and $\mathcal{E}_a$ is the environment where A is situated, then a quality can be specified as a mapping from its depending factors to an evaluation (i.e., a real value ranged from 0 to 1).

$$q: \mathcal{E}_a \times A \rightarrow [0 \ldots 1] \quad (2)$$

Environment Dependency. Generally speaking, a software entity may rely on others to achieve its design objectives, including its computation and quality. We say that a software entity is dependent on an object if the entity will invoke the object's computation or use the object as a resource when the entity is engaged in the accomplishment of its goals. In the following context, we will not distinguish the different connotations of "invoke" and "use"[2]. Since all of those software entities on which the software entity depends are part of the entity's environment, we refer all dependencies between the entity and others to as environment dependency.

The dependencies between a software entity and the environment can be classified into two categories, i.e., computation and quality dependency. Computation dependency represents that part of the entity's computation must be carried out by other computation entities whilst quality dependency indicates that the entity's quality is impacted by the qualities of other entities involved in the environment. Generally, computation dependencies will definitely affect the quality of the entity, but quality dependencies may not be resulted from computation dependencies. Since functionality can also be considered as a special quality property, we will not purposely differentiate computation dependency from quality dependency in the following context.

When the quality of an AQT is dependent on multiple other objects, the dependency relations can be further specified as concatenated (or linear), parallel (or selective), or hybrid dependency.

Concatenated dependency means that the quality property is simultaneously dependent on several objects. For example, an AQT needs to contiguously invoke several services provided by other AQTs to implement its computation, then the quality property of the AQT (e.g., correctness or performance) is determined by those depended services together. Suppose that A's quality property Q_A is concatenatedly dependent on the related qualities of a list of other AQTs, we can represent this dependency relation as follows.

$$Q_A \ \text{cDependOn}(Q_{A_1}, Q_{A_2}, \cdots, Q_{A_n}) \quad (3)$$

$Q_{A_i}(1 \leqslant i \leqslant n)$ is a specific quality property of AQT A_i. In the dependency relation, Q_{A_i} can be another type of quality different from Q_A[2]. For instance, the A's correctness may be affected by another AQT's reliability.

Parallel dependency means that the quality property is selectively dependent on one or some of a collection of objects. For example, an AQT may select different other AQTs to request for services in different situations and the AQT will depend on

[2] In the following context, we may directly use A dependOn(A') instead of Q_A dependOn($Q_{A'}$) when we do not care about the concrete qualities or the depended and the depending qualities are the same ones.

different AQTs when the selection is varied. A parallel dependency relation can be notated as follows.

$$Q_A \ \text{pDependOn}(Q_{A_1}, Q_{A_2}, \cdots, Q_{A_n}) \quad (4)$$

Hybrid dependency is a mixed case of the above two kinds of dependency and a hybrid dependency relation can be represented as a composition of concatenated and parallel relations.

In a dependency relation, an AQT's quality can be dependent on the same or different quality properties of other AQTs. For instance, an AQT's availability may be dependent on another AQT's security in the case that the depended AQT's security threshold is so high that it often does not provide services, which will result in a low availability of the depending AQT.

The dependency between AQTs can be qualitative as well as quantitative. Further, qualitative dependency can be positive or negative. Positive (or negative) dependency means that the higher the depended AQT's quality is, the higher (or lower) the depending AQT's quality is.

Q_A +cDependOn(Q_{A_1}) or Q_A −cDependOn(Q_{A_1})

The sign (+/−) indicates the dependency relation is positive or negative.

For quantitative dependency, the measurement of an AQT's quality can be calculated from those depended qualities. For different dependency relations, the computations of qualities may be distinct. For example, suppose that A's qualities are dependent on the qualities of $A_1, A_2, \cdots, A_n$.

For the concatenated dependency, A's reliability can be computed as $R_A = \prod_{i=1}^{n} R_i$, where R_i is A_i's reliability. Differently, for the parallel dependency, if the depended AQTs are used to realize a redundancy-based fault-tolerant mechanism, A's reliability can be calculated as follows.

$$R_A = \prod_{i=1}^{n} R_i - \sum_{i_1, i_2=1; i_1 \neq i_2}^{n} R_{i_1} \times R_{i_2} + \cdots + (-1)^{n-1} \sum_{i_1, \cdots, i_n=1; i_1 \neq i_2 \neq \cdots \neq i_n}^{n} \prod_{j=1}^{n} R_{i_j}$$

If the depended AQTs are accessed randomly to provide services, A's reliability is $R_{AC} = \sum_{i=1}^{n} p_i \times R_i$ assumed that the probabilities of selecting and accessing those AQTs' services are $p_i (1 \leqslant i \leqslant n)$.

Environment Requirement. The environment dependency accounts for how the existing environment restricts and affects the behavior of AQTs while they are realizing their objectives. Differently, the environment requirements specify the prerequisite that an AQT could realize its goals, which describes the environment conditions required for the AQT to run and implement its functionality. However, even though the environment requirements of the AQT are satisfied, the AQT is not assured of realizing its goals if the environment dependencies are damaged. For instance, an AQT needs to invoke another AQT's service to implement its own service. Then the AQT's environment is required to situate the other AQT for the invocation. Nevertheless, if the other AQT is unreliable, the AQT depending on it will be unreliable, either.

The environment requirements of an AQT can be divided into two classes, i.e., requirements for implementing the AQT and requirements for running the AQT. The implementation requirements denote the assistance, mainly including those invoked services, which are provided by other AQTs for the AQT to implement its functionality; and that the

running requirements are the demands on the runtime environment after the AQT is deployed and executed. For instance, in a client/server-based system, the implementation of the AQT at the client end relies on the services provided by the server, while the execution of the AQT lies on the stable connection to the server.

However, because an AQT may be integrated into different Internetware and executed in different running environments (for example, in a client/server-based or layered software system), its runtime environment requirements cannot be determined and fixed at the specification and design phases of the AQT. Therefore, while discussing the environment requirements of AQTs, we will only specify the implementation requirements in the specifications of AQTs whilst the runtime environment requirements and dependency will be discussed when we investigate the composition of AQTs in the following section.

Only when the environment meets the requirements can the AQT provide services properly; and only under the environment satisfying the AQT's requirement can the AQT's qualities be computed according to the environment dependency relations.

2.2 Abstract Quality Type and Example

Similarly as ADT (Abstract Data Type), an AQT encapsulates the data types, associated operations, and expected quality properties of a software entity in an uniform syntactical unit and meanwhile uses axioms to formally specify the relationships among them, especially the dependency relations between the quality properties and the environment. An AQT's specification is independent of any particular implementations of its data types and the operations.

For specifying the quality properties formally and the dependency relations between quality properties and the environment in an appropriate way, we also include the environment requirements of the AQT in the specification. Then an AQT can be defined in a quintuple.

$$AQT=<DT,OP,Q,Env\text{-}Req,Ax> \quad (5)$$

DT is the data types of variables encapsulated in the AQT, OP is the set of operations exported by the AQT, and Q is the quality properties expected by the AQT.

$Env\text{-}Req$ is the environment requirements on which the AQT depends to perform its operations with expected quality guarantees.

Ax is the set of axioms for specifying the relationships among the elements of the AQT, especially including the relationships among operations and dependency relations between quality properties and the environment. Since the AQT's environment must satisfy its environment requirements, the environment dependency relations can be defined on the environment requirements.

For the environment dependency specified in Ax, it can include qualitative dependency relations as well as quantitative relations. For a quantitative quality, its computation can also be specified as an axiom.

To illustrate how to use AQTs to specify software entities, we give an example.

First, stack is a well-known data type and it can be specified as an ADT, in which the relationships among the data and the operations are constrained through the axioms.

Now, we can extend the ADT into an AQT. Assumed that the stack is regarded as a closed world and we are only interested in the reliability of

the stack. The reliability of the stack will depend on the correctness of the executions of its associated operations, the availability of the memory allocated for the stack, and the exclusive or concurrent accessibility to the stack. The corresponding AQT of the stack can be specified as following.

In the extension, both the qualitative and quantitative dependency relations are specified in the axioms section.

R denotes the measurement of the reliability of the stack, which depends on its computation, its memory allocation and the access type. Among the dependency relations, the reliability negatively depends on the number of concurrent access if the access type is concurrence enabled.

Then, the computation depends parallel on the reliabilities of pushing and popping data in/out the stack, which further concatenatedly lie on the concrete operations *push*(), *pop*() and the operations checking the states of the stack, respectively.

Meanwhile, the reliability of the stack is also quantitatively dependent on the operations and the memory allocation. In the quantitative dependency relation, R_x denotes the reliability of the operation x and f, g, and h are the functions of the reliabilities of *push*() and *full*(), and *pop*() and *empty*(), and the availability of the memory, respectively.

Stack

element: Integer

push(), *pop*(), *top*(), *empty*(), *full*(), *init*()

$R:Q$

$EnvReq:\mathcal{E}nv$

.../*axioms*

$EnvReq=\{Memory, AccType, ConcNo\}$

$AccType=\text{Concurrent}\Rightarrow R-\text{DependOn}(AccType, ConcNo)$

$AccType=\text{Exclusive}\Rightarrow R+\text{DependOn}(AccType)$

$R \quad +\text{cDependOn}(A_{mem}, R_{comp})$

$R_{comp} \quad +\text{pDependOn}(R_{in}, R_{out})$

$R_{\text{in}} \quad +\text{cDependOn}(R_{push}, R_{full})$

$R_{\text{out}} \quad +\text{cDependOn}(R_{pop}, R_{empty})$

$R=h((A_{mem}, AccType, f(R_{push}, R_{full})+g(R_{pop}, R_{empty}))/2)$

3 AQT-based Internetware

The wholeness of Internetware lies on not only the behaviors of individual software entities (e.g., AQTs) involved in the systems but also the interactions among the software entities. To some extent, we can say that the wholeness of Internetware is synthetically emerged from the interactions among AQTs.

3.1 AQT-based Composition

The interactions among AQTs can be characterized by the interconnection relations and the process of the interactions.

First, the AQT model is independent of the implementation so it is not concerned about how an AQT is connected with the depended AQTs. Because an AQT may be assembled into different Internetware running in varied environments, the interconnections between the AQT and the depended AQTs can only be determined in the assembly. The interconnections among AQTs can be considered as first order software entities[3].

Suppose A_i is a depended AQT of A, the dependency relation between A and A_i can be either concatenated or parallel, and c is the connector linking A and A_i. Let w simulate the wrapper to wrap c and A_i into an unit, the connection between A and A_i is specified by the predicate

$$isConnectedVia(A, A_i, c)$$

Then A directly depends on w and further w concatenatedly depends on c and A_i. The composite of A and A_i can primarily be described as following.

Composite AQT

CompA:$\mathcal{AQT}$

Q-list:$\mathcal{Q}$

EnvDep:*AQTs*+*Connections*+*Wrappers*

$CompA=A$

$AQTs=\{A_1,\cdots,A_n\}$

$\forall q\in Q\text{-list}(q\ \ hDependOn(AQTs,Connections,Wrappers))$

$\forall a\in AQTs\ \exists c\in Connections\ \exists w\in Wrappers$

$(A\ \ dependOn(a))\wedge isConnectedVia(A,a,c)\Rightarrow$

$(A\ \ dependOn(w\ \ cDependOn(c,a)))$

Q-list is the set of quality requirements of the composite AQT. *EnvDep* is the environment on which the composite AQT depends. In the previous section, the environment information specified in an atomic AQT only focuses on the implementation requirements. Differently, the environment of a composite AQT is concerned with those AQTs involved in the composition and the connections among them as well.

Second, when there is not a dependency relation between two interconnected AQTs (e.g., A_i and A_j) participating in the composite, the quality of the composite will concatenatedly depend on the two AQTs and the connector between them.

Composite AQT

CompA:$\mathcal{AQT}$

Q-list:$\mathcal{Q}$

EnvDep:*Connections*

$CompA=\{A_i,A_j\}$

$\forall q\in Q\text{-list}(q\ \ hDependOn(Connections))$

$\forall a,b\in CompA\ \exists c\in Connections$

$(CompA\ \ cDependOn(a,b))\wedge$

$isConnectedVia(a,b,c)\Rightarrow$

$(CompA\ \ cDependOn(a,c,b))$

Thirdly, the process of interactions among the AQTs involved in the system specifies how interconnected AQTs act and interact to realize the system objectives. In general, the interaction process among AQTs can be recursively specified as the compositions of sequential, branched and iterative activities[4]. When the interaction among components is sequential, the composite entity will concatenatedly depend on those components involved in the interaction. When the interaction is a branch, the dependency will be parallel. When the interaction is a loop, the dependency will be hybrid, in which the composite concatenatedly depends on the iteration body in the prescribed number of times.

Suppose the composite AQT is composed of components $A_1,A_2,\cdots,A_n$, the process of interactions among them can be formally described like a finite automaton.

$$P=<\mathcal{AQT},\ \mathcal{ACT},\ \mathcal{T\chi}> \tag{6}$$

$\mathcal{AQT}=\{A_1,A_2,\cdots,A_n\}$. $\mathcal{ACT}$ is the set of activities performed by the AQTs involved in the composition. $\mathcal{T\chi}$ is the transitions among activities in the process, i.e., $\mathcal{T\chi}\subseteq\mathcal{ACT}\times\mathcal{ACT}$.

- When one activity (e.g., a') is the subsequence of another activity (e.g., a) and suppose the AQTs that perform the two activities is composed into a new AQT (e.g., A'), A' will concatenatedly depend on the two AQTs.
- When several activities (for example, a' is one of them) are branches after an activity (e.g., a), the composite AQT will parallelly depend on the AQTs performing the activities.
- When some transitions form a loop, the whole loop can be considered as a single activity[3] and the dependency relation between the composite and the loop body is similar as a relation between the composite and a simple activity.

[3] If the process is well structured, any loop has only one entry.

Then the composite can be enhanced and described as follows.

Composite AQT

$CompA:\mathcal{AQT}$

$Q\text{-list}:Q$

$EnvDep:Connections+Wrappers+Pr$

$$CompA=\{A_1,A_2,\cdots,A_n\}$$

$$\forall q\in Q\text{-list}(q\ \ hDependOn(Connections,Wrappers,Pr))$$

...//Connection Dependencies

$$\exists q\in Q\text{-list}(\exists a,a'\in\mathcal{P}r\exists A_i,A_j,A'\in\mathcal{AQT}($$
$$(Perform(A_i,a)\wedge Perform(A_j,a')\wedge isSubsequenceOf(a,a')\wedge$$
$$A'=\{A_i,A_j\}\wedge q\ \ dependOn(A_i,A_j)\Rightarrow$$
$$q\ \ dependOn(A'\ \ cDependOn(A_i,A_j)))\vee$$
$$(Perform(A_i,a)\wedge Perform(A_j,a')\wedge isBranchOf(a,a')\wedge$$
$$q\ \ dependOn(A_i,A_j)\Rightarrow q\ \ pDependOn(A_j))\vee$$
$$(Perform(A_i,a)\wedge isLoopBody(a)\Rightarrow q\ \ hDependOn(A_i))))$$

In the enhanced specification, the process is added in the environment as a new dependency factor.

The above specification just describes the qualitative dependency between the composite and its environment. We can also specify quantitative dependency for the composite on the components.

3.2 Inference and Improvement of the Quality

Based on the specifications of AQTs involved in the integration and the integrated Internetware itself, we can infer and improve the quality of the Internetware at different stages of the system's lifecycle.

First, at the specification and design stages, the qualitative and quantitative dependency relations offer the basis for reasoning about and predicting the quality of the system.

Second, at the integration stage, those positive or negative dependencies can facilitate the decision-making on the selections of components.

Thirdly, at the runtime, the dependency relations can be used to dynamically select components (or service providers) to improve the quality of the executing system.

In the next section, we will describe a prototype system for quantitatively inferring and improving the quality of Internetware.

4 Prototype for High Quality Internetware

4.1 Implementation Structure of AQTs

In the implementation of an AQT, the dependency relations are no longer solely used for inferring the quality of Internetware in which the AQT involved; instead, they are mainly used for the AQT to improve the quality of Internetware. Therefore, the dependency is specified via behavior rules concerned with how the AQT reacts to the environment to improve its performance.

In the prototype, an AQT is implemented with the following structure, in which the environment dependency of the AQT is defined in the environment model.

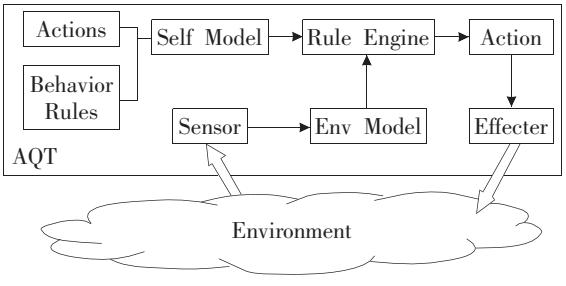

Fig.1 The simplified architecture of AQTs

图1 AQTs 简化体系结构

In an AQT, the self-model stores the data specifying the state of the AQT, the actions performed by the AQT and the rules controlling the AQT's behavior. The rule engine will reason about the AQT behavior on the AQT's internal and exter-

nal states (i.e., its data and the environment) and then trigger the performance of the actions. In the environment model, the depended resources or services are defined as the friends (or acquaintances) of the AQT and the AQT has some beliefs on the qualities of their friends. Each belief is on one special quality property and it is expressed as an equation. For example, expression $Bel(A, F, S, Reliability)=0.8$ means that A believes that the reliability of its friend F to provide the required service S is about 80 percent. The percentage will be referred to as trust value in the following sub-sections.

4.2 Experimental Prototype

The current prototype mainly focuses on the reliability of Internetware. On the Internet, for each service requested by the system, there may be more than one service provider and then the system may parallelly depend on those different service providers with varied reliabilities.

- In the system, every service provider is wrapped into an AQT, which will request for (or invoke) some services to realize its functionality. The providers of those depended services are modeled as the AQT's friends.
- When the AQT commits itself to a task, it will decompose the task into parts and delegate the sub-tasks to its friends according to the dependency relations.
- When an AQT's friend enters the environment, it will declare its reliability to the AQT. The AQT may not completely trust its friend's declaration since the declaration has not been tested or verified via real invocations. Nevertheless, the AQT can assign an initial trust value to the friend based on the declared quality value.
- The AQT will designate a selection probability to each friend according to the trust value and when the AQT requests for service, it will select a service provider from its friends according to the specified probability distribution. The AQT can revise the trust values according to the successes or failures of requests. The AQT can also modify the selection probabilities based on the current trust values.

4.3 Verification of Quality Properties

To verify the prototype that it can gain the theoretically high quality, we implement a simple application based on the prototype.

The application is a travel agency, which is designed to assist the customer to reserve flights. To accomplish the flight reservation, the agency may depend on different friends who may assist the agency in varied ways. For instance, the agency may directly reserve the flight on the airline company that offers the flight course, so it will directly depend on the company's service. The agency may also have to depend on other agencies when it tries to reserve a long-distance flight that includes transfers among different airline companies. The dependency relationships between the agency and other service providers are shown in Fig.2.

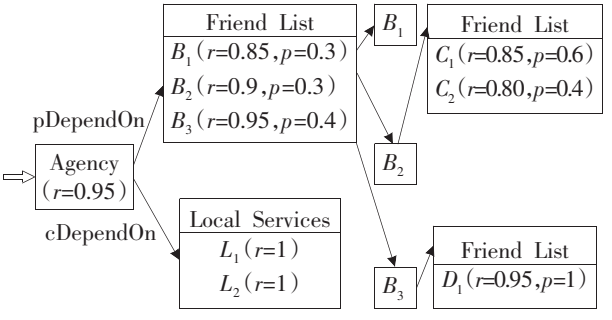

Fig.2 Relationships among service providers

图2 服务供应商之间的关系

As Fig.2 shows, the agency is concatenatedly dependent on its local services and parallelly dependent on its friends. For the simplicity, we assume that the local services are always reliable. In the friend list, r represents the reliability of a service and p denotes the probability of selecting the service provider.

Theoretically, the highest reliability of the system is about 85.74% if the invocation path ($Agency \rightarrow B_3 \rightarrow D_1$) is selected, the worst reliability is about 68.4% for the path($Agency \rightarrow B_2 \rightarrow C_2$), and the average reliability is about 80% which can be calculated as following.

$$R=R_A \times (R_{B_1} \times P_{B_1} + R_{B_2} \times (R_{C_1} \times P_{C_1} + R_{C_2} \times P_{C_2}) \times P_{B_2} + R_{B_3} \times R_{D_1} \times P_{D_1} \times P_{B_3}) = 79.81\%$$

Experimentally, we run the application system for 20 000 times, and the reliability of the system is recorded in Fig.3.

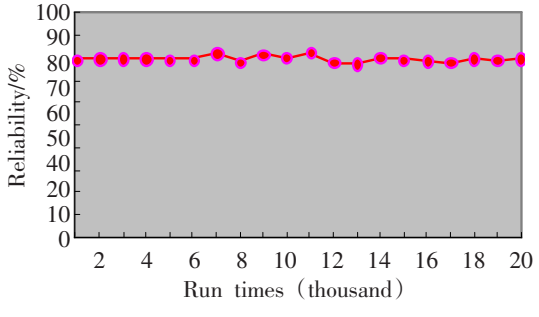

Fig.3 Average reliability of the application

图 3 应用软件的平均可靠性

As Fig.3 shows, the reliability of the system vibrates around 80%.

4.4 Improvement of Quality Properties

From the above theoretical analysis, we can conclude that a higher quality (e.g., the reliability) of the depended service will result in a higher reliability of the whole system and a higher probability of selecting the more reliable depended service will also lead to a more reliable system.

To make the application gain the highest reliability, we implement the AQTs with the ability to adjust the trust values and the selection probabilities autonomously via designating new adaptive behavior rules to AQTs.

- First, raise the selection probability of depended services with higher trust values to make sure that more reliable services are selected more frequently.
- Second, modify the trust values of depended services so that successful services would be trusted more whilst failed services would be trusted less.
- Thirdly, when another service provider enters the environment, the AQT will add the newcomer into its friend list automatically so that the application could gain a higher reliability than before once if the newcomer has a very high reliability.

In the upgrade application, we allow the maximal selection probability to be adjusted to 0.90 and the minimal probability to 0.09 so that all depended services will have chances to be selected and always-unreliable services will be discarded from the friend list.

After 20 000 times of run, the change of the reliability of the application is shown as Fig.4.

As Fig.4 shows, with the increasing of the run times, the reliability of the application becomes higher and gradually (when the application runs about 11 thousand of times) tends to the theoretically maximal value, 85.74%.

Finally, we designedly deploy a new service provider B_4 in the environment, whose reliability is

as high as 0.95 and on which the agency will directly depend.

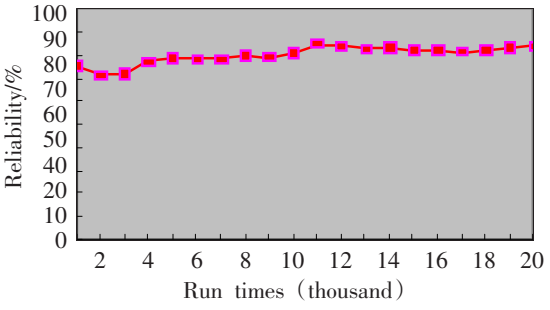

Fig.4 Gaining the highest reliability

图 4 获得最高可靠性

The reliability of the application is shown as Fig.5 in another 20 000 times of run. When the application runs about 10 thousand of times, the new deployed service provider enters the agency's vision. Then, the application will more frequently select the new path including the new highly reliable provider and gain a higher reliability.

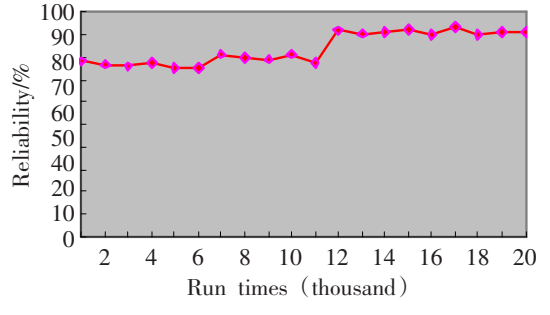

Fig.5 Adapting to a higher reliability

图 5 适应较高可靠性

5 Related Work

By now, many component models, besides those popular ones based on commercial standards such as COM, CORBA and J2EE, have been proposed to model software entities. For example, [5] proposes a unifying component description language for integrated descriptions of structure and behavior of software components and component-based software systems. [6] describes how components are specified at the interface level, design level and how they are composed.

There is also some work on modeling components considering the environments. For example, [7] presents a component model for encapsulating services, which allows for the adjustment of structure and behavior of autonomous components to change or previously unknown contexts in which they need to operate. [8] introduces service-oriented concepts into a component model and execution environment and defines a model for dynamically available components.

However, those component models do not include quality properties. Quality properties are independent of components at the design phase and they have to be coped with after components are deployed.

There is other work on relating non-functional properties (or quality) with functionalities of software entities. For example, [9] introduces a framework dealing with interface signature, interface constraints, interface packaging and configurations, and non-functional properties of software components. [10] presents a framework for integrating NFRs into the ER and OO models, where NFRs are modeled as special objects. [11] and [12] propose two formal description languages for non-functional attributes, NoFun and Process-NFL, respectively, which combine non-functional specifications and architectural descriptions. However, in that work, quality is not specified as inseparable properties of software components but as data and operations.

In addition, much work focuses on modeling software quality[13]. For example, [14] surveys a significant number of quality models and meanwhile describes the design of a quality model that assesses a software product's efficiency and effectiveness. However, those proposed quality models are usually merely concerned with the quality properties of software entities after software entities have been developed, which cannot be used to reason about the quality properties of software entities at the modeling and design phases. [15] and [16] propose some rigorous approaches to specifying non-functional attributes, which can provide a good expressiveness on the specifications. Nevertheless, their approaches are too mathematical to be mastered and used by the developers of software systems.

In the study of the relationships and compositions of quality properties, some work also takes the software architecture or the process into consideration. [17] defines key requirements for an architecture-based approach to composing components and provides an architecture definition language RADL with a focus on compositionality and extra-functional properties. However, the extra-functional properties with which the work deals are different from those quality with which high quality software systems are concerned. [18] proposes a comprehensive framework for representing and using non-functional requirements during the development process, which is unlike our work where the process is to specify the interactions among components.

6 Conclusions and Future Work

In the development of high quality software systems, multiple quality properties should be considered along with the development process so that the quality of software systems can be inferred and predicted at the specification and design stages and be evaluated and verified at the deployment and execution stages.

In this paper, we introduc a new kind of abstraction into the development of Internetware, i.e., quality abstraction, and put forward a new abstraction notion to encapsulate data, operations and quality properties in an uniform unit, i.e., the abstract quality type. So far as we know, it is the first time to model data, operations and quality in a single syntactical unit and relate a software entity's quality properties directly to the entity's attributes. Like ADT, AQT provides a rigorous way to reason about the functional and non-functional properties.

We also explored the composition of AQTs by considering the software architecture and the process of interactions. The study on the composition of AQTs will offer a solid foundation for reasoning about the whole quality properties of Internetware since the whole quality properties rely on both the systems' internal states and the external environments.

At last, this paper describes a prototype to verify and improve the quality of Internetware. Although the prototype is so preliminary and only deals with the reliability of the application, it can be easily extended for other quality properties.

In the next stage, we will investigate the methodology of constructing high quality Internetware

based on AQTs and upgrade the prototype for more quality properties.

References:

[1] Papazoglou M, Georgakopoulos D. Service-oriented computing[J]. Communications of the ACM, 2003,46(10):25-28.

[2] Pamas D L. On a 'Buzzword', hierarchical structure[C]// the Proceedings of IFIF Congress 74. [S.l.]: North Holland Publishing Company, 1974:336-339.

[3] Shaw M, Garlan D. Software architecture: perspectives on an emerging discipline[M]. NJ, USA: Prentice Hall PTR, 1996.

[4] Talib M A, Yang Z K, Ilyas Q M. Modeling the flow in dynamic web services composition[J]. Information Technology Journal, 2004,3(2)184-187.

[5] Teschke T, Ritter J. Towards a foundation of component-oriented software reference models[C]//Butler G, Jarzabek S. Generative and component-based software engineering. LNCS 2177: the Second International Symposium, GCSE 2000, Erfurt, Germany, October 9-12, 2000, Revised Papers. [S.l.]: Springer, 2001:70-84.

[6] Liu Z M, He J F, Li X S. Contract-oriented development of component software[C]//the Proceedings of IFIP WCC-TCS2004, Toulouse, France, 2004:355-372.

[7] Ben-Shaul I, Holder O, Lavva B. Dynamic adaptation and deployment of distributed components in Hadas[J]. IEEE Transactions on Software Engineering, 2001,27(9): 769-787.

[8] Humberto C, Richard H. Autonomous adaptation to dynamic availability using a service-oriented component model[C]// the 26th International Conference on Software Engineering (ICSE'04), Edinburgh, Scotland, United Kingdom, 2004: 614-623.

[9] Han J. A comprehensive interface definition framework for software components[C]//the 1998 Asia-Pacific Software Engineering Conference, Taipei, Taiwan, 1998:110-117.

[10] Cysneiros L M, do Prado Leite J C S, de Melo Sabat Neto J. A framework for integrating non-functional requirements into conceptual models[J]. Requirements Engineering, 2001,6:97-115.

[11] Franch X, Botella P. Putting non-functional requirements into software architecture[C]//Proceedings of the Ninth International Workshop Software Specification and Design. [S.l.]: IEEE Press, 1998,:60-67.

[12] Rosa N S, Justo G R R, Cunha P R F. A framework for building non-functional software architectures[C]//Proceedings of the 2001 ACM Symposium on Applied Computing. [S.l.]: ACM Press, 2001:141-147.

[13] Chung L, Nixon B, Yu E, et al. Non-functional requirements in software engineering[M]. [S.l.]: Kluwer Publishing, 2000.

[14] Ortega M, Perez M, Rojas T. Construction of a systemic quality model for evaluating a software product[J]. Software Quality Journal, 2003,11:219-242.

[15] Issarny V, Bidan C, Saridakis T. Achieving middleware customization in a configuration-based development environment: experience with the Aster prototype[C]//Proceedings of the 4th International Conference on Configurable Distributed Systems, Maryland, 1998:207-214.

[16] Zarras A, Issarny V. A framework for systematic synthesis of transactional middleware[C]//Proceedings of Middleware'98, England, 1998:257-272.

[17] Schmidt H. Trustworthy components compositionality and prediction[J]. The Journal of Systems and Software, 2003 (65):215-225.

[18] Mylopoulos J, Chung L, Nixon B. Representing and using non-functional requirements: a process-oriented approach[J]. IEEE Trans on Software Eng, Special Issue on Knowledge Representation and Reasoning in Software Development, 1992,18(6):483-497.

QU Youtian was born in 1962. He received the B.S. degree in Architecture of Computer Science from Nanjing University in 1986. He is a professor and master supervisor at Zhejiang Normal University. His research interests include component techniques, Agent-oriented software engineering and intelligent database techniques, etc.

瞿有甜(1962-),男,浙江东阳人,1986 年于南京大学获理学学士学位,浙江师范大学教授,主要研究领域为面向 Agent 软件工程、智能数据库技术等。

WU Zejun was born in 1972. He received the B.S. degree in Computer Science from Yangtze University in 1994. He is an economic engineer at Hubei Branch of China Construction Bank.

吴泽君(1972-),男,湖北天门人,1994 年于江汉石油学院(现长江大学)获工学学士学位,中国建设银行湖北分行工程师、经济师。

JIAO Wenpin was born in 1969. He received the B.S. and M.S. degrees in Computer Science from East China University of Science and Technology in 1991 and 1997, respectively, and the Ph.D. degree from Chinese Academy of Sciences in 2000. He is an associate professor at Peking University. His research interests include software engineering, intelligent software, internetware and formal methods, etc.

焦文品(1969-),男,湖北天门人,2000 年于中国科学院软件研究所获工学博士学位,北京大学副教授,主要研究领域为软件工程、智能软件等。

CHEN Tianzhou was born in 1970. He received the Ph.D. degree in Computer Science from Zhejiang University in 1998. He is a professor and doctoral supervisor at Zhejiang University. His research interests include embedded system and architecture.

陈天洲(1970-),男,浙江丽水人,1998 年于浙江大学获计算机应用专业博士学位,浙江大学教授、博导,主要研究领域为嵌入式系统、体系结构,发表论文 140 余篇,负责 20 余项国家基金、国防预研、863、省科技计划项目。

HE Guolong was born in 1966. He received the Ph.D. degree in Mathematics from Zhejiang University in 2004. He is an associate professor at Zhejiang Normal University. His research interests include functional analysis and factal geometry.

何国龙(1966-),男,浙江新昌人,2004 年于浙江大学获理学博士学位,浙江师范大学副教授,主要研究领域为计算数学、泛函分析,已发表论文 20 余篇。