Using autonomous components to improve runtime qualities of software

W. Jiao*

Institute of Software, School of EECS, Peking University
*Key Laboratory of High Confidence Software Technologies, Ministry of Education, People’s Republic of China
E-mail: jwp@sei.pku.edu.cn

Abstract: In the development of software systems, quality properties should be considered along with the development process so that the qualities 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. However, distributed autonomous software entities are developed and maintained independently by third parties and their executions and qualities are beyond the control of system developers. In this study, the notion of an autonomous component is used to model an independent autonomous software entity. An autonomous component encapsulates data types, associated operations and quality properties into a uniform syntactical unit, which provides a way to reason about the functional and non-functional properties of software systems and meanwhile offers a means of evaluating and assuring the qualities of software systems at runtime. This study also describes the implementation and running support of autonomous components and studies a case application system to demonstrate how autonomous components can be used to improve the qualities of the application system.

1 Introduction

Almost all of today’s software systems integrate autonomous software entities [1]. For example, complex distributed applications and systems of mobile devices can be modelled in terms of autonomous software entities [2, 3]; software entities situated on the Internet are decidedly autonomous and long-lived. Autonomous software entities may enter or quit environments autonomously and evolve their services continually without notifying others [4, 5]. Furthermore, they are independent. Their behaviour is out of control of third party or external software systems and their QoS relies on themselves as well as their deployment and execution environments, so neither their behaviour nor their QoS is deterministic or predictable. A software system has to be assembled at runtime by dynamically binding those autonomous software entities and requesting for services from them, and it has to approximate the current states of remote software entities [6] and rely on the experiences of trying those software entities to trace their services states and evaluate their QoS.

Methods to construct high-quality software systems via assembling autonomous software entities have become grand challenges.

To model and construct high-quality software systems, we should take into account both the functional requirements and the non-functional requirements (or quality properties). (In the literature, non-functional requirements, constraints, goals and qualities are often referred to the same properties of software systems [7].) Existing approaches usually model functionalities and quality properties separately. For example, in component-based software development [8], component developers are required to only focus on the business logic of applications whereas the non-functional properties are left to the component deployment phase, which will be specified in the deployment profile and finally guaranteed by the running support platform. However, the separation leads to many difficulties in assembling systems. It is especially difficult for analysts or maybe researchers to reason about the system-level quality properties based on the properties of components at the development phases because the qualities of software
systems are not only related to the involved components but also depend on the environmental resources, and the synthesis of quality properties cannot be achieved along with the integration of functionalities.

If we cannot reason about the properties (including both functional and non-functional properties) of software systems based on the properties of the involved components, it will be very difficult to construct high-quality software systems.

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

In our opinions, there is another abstraction crucial to the development of contemporary software systems, that is, quality abstraction. It is a necessity to integrate data abstraction, process abstraction and quality abstraction together in specifying and developing high-quality software systems.

In this paper, we aim at studying constituent elements for constructing high-quality software systems. However, neither traditional software components nor software agents are suitable just right for software systems involving autonomous software entities. Components are software entities that can be deployed independently and are subject to composition by third parties [8]. Traditionally, components are considered as passive software entities and they are called through their published interfaces. On the other side, although an agent is usually viewed as an active software entity situated in environments in pursuit of its design objectives [9], there is not a consensus of the definition of an agent. Unlike objects, there are still controversies and deviations on some aspects of agent, such as which structural and behavioural properties an agent must possess and radically what is or is not an agent. There are lots of tools or platforms for supporting the development of agents, but agents generated via the tools or platforms are quite dissimilar on structures as well as on behaviour modes (e.g. FIPA-OS[10], JADE[11], AGLETS[12]). There are also some methodologies for multi-agent systems (see [13–15] for surveys), but agents inside are coarse-grained and highly abstract. In the real world, especially on the Internet, not all of software entities can be considered as agents as specified in the methodologies. In this paper, we use autonomous component (abbreviated as AC below) to abstract and model autonomous software entities as a modelling unit and a foundation for constructing high-quality software systems.

Similarly as an object encapsulating its data types and associated operations in a single syntactical unit, an AC encapsulates data types for specifying its states, behaviour (or processes) associated with the data types for realising its functionalities, quality properties for representing its non-functional requirements and the quality evaluation and assurance mechanism for realising the quality requirements.

By encapsulating data, associated behaviour and quality properties into a unit, we can provide a rigorous means of inferring the properties (including functional and non-functional properties) of systems from the integrated ACs and can build a solid foundation for selecting ACs to integrate high-quality systems as well.

Briefly, the main contributions of this paper are listed as follows.

First, ACs are used to model autonomous software entities. In the model for AC, the approximate evaluations on the qualities of itself and its attributes (including internal data and actions and required external resources) and the quality assurance mechanism for monitoring and updating the evaluations are encapsulated together. Thus, an AC can dynamically evaluate the qualities based on its experiences on the use of resources and then autonomously make decisions on the selections of resources to satisfy its desired quality expectations.

Second, a prototype framework for supporting the run of ACs is built. In the framework, ACs use their behaviour rules to adjust the approximate evaluations about the qualities of external resources (i.e. required service provider candidates) and then automatically select the most appropriate candidates to request for services. Thus, ACs can obtain high qualities automatically and dynamically.

In the remainder of this context, Section 2 defines the concept and abstract structure of ACs. Section 3 describes the implementation structure and running support of ACs to offer a foundation for development of ACs. Section 4 studies a case application system to illustrate how ACs improve the runtime qualities (especially availability and performance) of the application automatically and dynamically. The last two sections detail some related work and make some concluding remarks on our current work.

2 Autonomous components

2.1 Definition

Components are software entities that can be deployed independently and are subject to composition by third parties [8]. Usually, people understand autonomy of components as the taking of independent action. Components are autonomous implies that they interact without any intervention from users or other controlling entities [16] and take actions based on their own decisions.

As an autonomous software entity, an AC should embody the capabilities of self-management and self-decision-making at least, for instance,
• Manage the information related to required external resources (i.e. other software entities it depends on to implement its functionalities), including capabilities and QoS of resources.

• Decide whether to contribute its capabilities to others (i.e. to offer its services to others).

• Decide on how to use the resources most appropriately.

**Definition 1:** An AC is a software entity independently deployed in the environment and reusable by third parties.

• The accomplishments of its functionality and quality properties depend on the internal data, associated operations and external resources, where external resources are loosely coupled with it and are capable of replying to its requests for services.

• It can evaluate the use of the external resources on the satisfactions of its functional and quality requirements.

• It can make decisions by itself on the use of its internal data, operations and external resources for achieving and improving its functional and quality objectives.

• It can serve others (maybe software systems or other ACs) via providing services; in other words, others can access its services from its exposed interface.

### 2.2 Model

As an encapsulation of data, operations and quality properties, the model of an AC should contain the specifications for the AC’s functional and quality objectives and the means for achieving the objectives (particularly, for quality objectives, the means may include evaluation, assurance and improvement of qualities).

On the other hand, as an abstraction of a software entity with autonomy, the model of an AC should embody decision-making capability for the AC. In the model, we use behaviour rules (w.r.t. *situation → action*) to represent the autonomous behaviour of an AC; and in the implementation, which will be described later, we embed a rule-based reasoning mechanism to offer the autonomous decision-making capability for an AC.

**Definition 2:** An AC can be modelled as a tuple as follows

\[
AC = \{\text{Dat, Srv, Act, Qua, Dcs, Inf}\} \quad (1)
\]

- **Dat** is the internal data of the AC. A data item is specified by its identity, value type and allowed range of values and \(\text{Dat} = \{\text{di} | \text{di} = (i\text{idat}, \text{type}, \text{range})\}\).

- **Srv** is the required external services (referred to as required services below) on which the AC depends to perform its objectives. We can assume that the world is formed purely by other ACs and the external services will be provided by ACs distributed in the environment. Suppose that \(\text{ACS}\) is the set of ACs, then

\[
\text{Srv} = \{(s, \text{pre-condi}, \text{post-condi}, \text{qos-exp-list}, \text{pc_list})\}
\]

\[
\text{pc_list} = \{(p, \text{gos}, \text{td}) | p \in \text{ACS} \land (\text{pre-condi} \land \text{run}(p) \Rightarrow \text{post-condi}); \text{td} : \text{gos} \rightarrow [0 \cdot \cdot 1]\}
\]

where \(s\) is the signature of a required service and it is specified via the pre-condition and post-condition, and \(\text{qos-exp-list}\) specifies the AC’s expectations for QoS of the service.

For each service, there may be multiple AC candidates providing it and \(\text{pc_list}\) manages the list of service provider candidates. All of these providers implement the service, that is, under the pre-condition and after the providers run, the post-condition of the service will be valid. Nevertheless, for each required service, the AC can select at most one provider candidate to request for the service at one time.

As external software entities, service providers evolve outside of the AC, join or quit the environment without notifying the AC and meanwhile their qualities often keep on changing at runtime. As a result, the qualities of the AC may change over time and, to some extent, is unpredictable. Nevertheless, the AC can select different service providers with varied qualities to pursue a high-quality realisation of its objectives.

For those remote service providers, the AC can only obtain approximate values of their qualities [6] and the values cannot be guaranteed to be the exact reflection of their real quality properties. In some sense, that the AC selects a service provider embodies a kind of trust of the AC on the service provider. We use \(\text{td} \) (abbr. for trust degree) to record the degree of trustworthiness of a provider candidate to indicate how much the AC trusts the candidate to reply to requests for the required service. Trust degree related to a provider is a function over the QoS of the provider obtained by the AC in use of the provider’s service, and the AC will select providers based on the trust degrees by some selection strategies. We will discuss in detail how the AC maintains the trust degrees and how it uses the trust information to select service providers below.

- **Act** is the set of actions that the AC performs. We use *action* here instead of *method* as that in the object-orientation is because we think that the AC is active and it has right to decide whether, when and how to take actions. An action can be implemented locally or via invoking required services.

\[
\text{Act} = \{a | a = (\text{sig}_a, \text{pre-condi}, \text{post-condi}, s\text{-list})\}
\]
where $s$-list $\subseteq$ Srv is the list of required services, $\text{sig}_a$ is the signature of action $a$ and it can be specified as $\text{sig}_a : \text{Dat} \times s$-list $\rightarrow$ Dat. The specification of an action is defined via pre-condition and post-condition, that is, $\text{pre-condi} \land \text{invoke}(\text{sig}_a) \Rightarrow \text{post-condi}$.

- $\text{Qua}$ specifies the expectations for qualities of the AC. The qualities of the AC depend on the AC's behaviour as well as the internal and external states. Suppose that $q$ is one of the required quality properties of the AC (e.g. reliability, availability, safety, security, performance etc.), then

$$\text{Qua} = \{ (q, \text{exp}, \text{sat}) | \text{exp} \in [0\ldots1]; \text{sat} : \text{Dat} \times \text{Srv} \times \text{Act} \rightarrow [0\ldots1] \}$$

where $q$ is the identity of a required quality property, $\text{exp}$ is the expectation value and $\text{sat}$ is the quality satisfaction degree evaluation function calculated based on the qualities of the AC's attributes. In principle, the base qualities of the attributes may be different from $q$. For instance, the AC's availability may depend on the security of its data items, the availability and safety of its external services, and the safety of its actions.

The quality dependencies between the AC and its attributes can be positive or negative. A positive (or negative) dependency means that the higher the quality of the attribute is, the higher (or lower) the AC's quality is. For instance, the AC's availability positively depends on the availability of its required services whereas negatively on the security of the latter.

- $\text{Dec}$ is the decisions (or behaviour rules) specifying how the AC reacts or adapts to the changes of its internal and external states. The behaviour rules specify not only how the AC performs actions to accomplish its functionalities but also how it evaluates the satisfaction of its quality expectations and adjusts its performance to maintain and improve the satisfaction.

A decision is defined by the action that the AC will initiate under the current state and quality expectations. It can be described as $d \in \text{Dec} : \text{Dat} \times \text{Srv} \times \text{Qua} \rightarrow \text{Act} \cup \{\varepsilon\}$, where $\varepsilon$ represents an inert action that does nothing.

- $\text{Inf}$ is the interfaces exposed by the AC for users to access its services. As a reusable software entity, the AC should expose its interface for the outside use, through which users can request the AC to take some actions. An interface is implemented by a process that involves a sequence of actions in a specific way.

Thus, an AC is modelled in a hierarchy (Fig. 1). In the hierarchy, the specification layer specifies the outside view of the AC, the implementation layer lists the elements used in the achievements of the functional and quality objectives of the AC and the reference layer specifies the references to other ACs (i.e. service providers) which the AC depends on to request for the required services.

For an AC, whenever it perceives another AC appearing in the system, it will register itself about its services to the latter. The latter may append it to the service provider candidates list based on the registration information. When the AC is
committed to providing services, it will invoke other ACs that provide the required services.

After an AC is activated, it will run for serving the outside world or adapting for its own good. When the AC receives a request for service from outside, it will make decisions on whether or not to reply to the request; and if it decides to reply to the request, it will take actions to perform the service. When the AC infers that it cannot satisfy its quality expectations, it can decide to keep inert simply. Ideally, the AC could keep itself always in a good service condition by making some changes (i.e. adaptations) to improve its survival state, such as taking actions to modify its internal data to ensure itself always in a good condition and re-selecting service providers to make the service states of required services better. For instance, suppose there is a data item to record the amount of a special expendable source, taking some actions to increase the amount can create more possibility for the AC to achieve its objectives.

In process of providing services, the AC may select different service providers to request for services dynamically and even re-plan the implementation of its services on-the-fly when the states of its own or the service providers for required services change.

2.3 Actions

The actions include operations for performing the functional objectives as well as the quality objectives of the AC. Concretely, the actions can be divided into the following categories:

- Functionality-relevant, to realise the functionality of the AC in the usual sense, such as manipulating the data and invoking required services.

- Quality-relevant, including those (i) to evaluate the qualities of the AC and (ii) to improve the qualities of the AC. When the qualities of the AC depend on those of other services, the AC can adopt a strategy of selecting the most appropriate service providers to improve its qualities.

- Management-relevant, including those (i) to perceive and collect the information about existence of other ACs; (ii) to register as a service provider candidate to other ACs that depend on its services to realise their objectives; (iii) to manage the information about required services, including maintaining the list of service provider candidates, evaluating the qualities of service providers, adjusting the degrees of trustworthiness of providers, and (iv) to record and maintain the history that the AC has gone through so that the AC could make its decisions based on its experiences.

In [17], we describe a method using the simulated annealing algorithm to adjust the trust degree related to a service provider, which is primarily based on the history of trying the service provider. Concretely, trust degree can be defined as a function as follows

\[ td = \frac{\text{syn}(|\text{dif}(\text{qos, qos-exp}))|)}{\text{temperature}} \]  

(2)

where \( \text{dif}(\cdot) \) is the function of computing the deviation between the actual obtained qos and expected qos of the provider, \( \text{syn}(\cdot) \) is the function of synthesising multiple deviation qos values specified in the qos-exp-list and temperature acts as an indicator to represent the degree of freedom to select the service provider.

Initially, the temperature of the service provider is set with a relatively high value. It will decrease gradually (i.e. annealing) if the provider is in a good service condition and subsequently is selected to provide service frequently; and on the other side, it will increase (i.e. heating) if the quality of the provider turns poor. When the temperature of the service provider is too high, the provider will have little opportunity to be selected and it may be eliminated from the candidates list of the AC. On the contrary, when the temperature is too low it will impede the AC to request services from other service providers. Therefore there are an upper limit and a lower limit for the temperature so that the AC need not waste its time on selecting those poor quality service providers and meanwhile all service providers still kept in the candidates list have chances to be selected.

Finally, the AC can adopt special strategies to select appropriate service providers based on the quality expectations of its own, the dependency relationships (i.e. positive or negative dependencies) between its quality properties and the required services, and the degrees of trust on service providers.

2.4 Quality evaluations

As mentioned above, an AC depends on its attributes (i.e. its internal data, actions and required services) to achieve its design objectives, including its computation and quality expectations.

When the qualities of an AC depend on those of others, the measurements of the qualities can be calculated from those dependent qualities. Nevertheless, for different application systems, they may be concerned about different quality properties; and meanwhile, for different dependency relationships and varied application requirements, the computations of qualities may be distinct.

2.4.1 Reliability: For example, reliability implies the ability of a system to operate correctly according to its specification for a given time interval [18]. Traditionally, system reliability is measured in terms of mean time between failures or the failure probability (FP) of a system function. The basic concept of the methods for predicting the reliability of component-based systems is to identify
failure probabilities for each service of a component and determine the reliability of system functions based on the invocation sequences and dependencies between the low-level services and the specific system function [19, 20].

For an AC, its reliability can be defined as a function over the failure probabilities of the actions involved in the implementation of its interfaces. When an action invokes services, the FP of the action depends on the invocations of the service providers. For a service provider of an AC, it is considered that the provider fails to provide the required service if the run of the provider violates the specification of the required service, that is, failed(p) iff (pre-cond ∧ run(p) ⇒ post-cond).

Furthermore, the satisfaction of the AC at reliability can be computed as deviation between the calculated reliability value and the expected value.

For instance, suppose that the reliabilities of the actions of the AC are r1, r2, . . . , rn, respectively (when an action is implemented by invoking a required service, the reliability of the action is actually the reliability of the required service), the reliability of the AC is defined as a function over the reliabilities of the actions, that is, Relac = ∏i ri; on the contrary, if these actions are never be performed at one time, the reliability of the AC can be defined as the average of the reliabilities of the actions, that is, Relac = ∑i ri/N. Finally, the satisfaction of the AC at reliability can be expressed as sat = 1 − (| min(Rebc − exp, 0)|/exp).

2.4.2 Availability: Availability is an attribute similar to reliability [21] and both of them are about the ability of the system to operate correctly though the former’s concern is the ability to provide correct services and recover from failures at a certain point in time when the services are requested. Many evaluation methods for reliability have been adapted and extended to perform availability evaluations [22].

Nevertheless, to differentiate availability from reliability, we need only to care about whether service providers reply to the service requests in time since the corrections of the services have been handled by reliability. For a service provider, its availability can be calculated by the probability of replying to requests for service, that is, P(r) = Num of Replies/Total Num of Requests.

Then the total availability of an AC can be computed based on the availability of required services since the local actions of the AC can be considered to be always available.

2.4.3 Performance: In [23], many performance prediction techniques are surveyed and described. For service providers located at remote sites, we cannot adopt those model-based techniques to evaluate their performance attribute since we may not own those service providers and say nothing of knowing their implementation models. Nevertheless, when an AC tries to request for services from them in the system, they execute in a real world and we can obtain good estimations of their performance.

Therefore the performance of a service provider is the actual reply time of the provider and the performance of an AC is the real execution time in process of providing services (via invoking required services and scheduling its actions).

2.4.4 Timeliness (real-time): When the execution time of an AC to provide a service cannot meet hard deadlines [24], it will be considered that the AC does not satisfy its timeliness requirement. Timeliness is also concerned with the performance of the system. Differently, there are usually some properties like worst-case execution times, execution periods or deadlines for specifying timeliness requirements. For a required service, qos-exp-list can be used to express the AC’s expectation for timeliness of the service.

Timeliness will put negative impact on reliability and availability of the system. When there are timeliness requirements, a service provider is considered to be failed if it cannot reply to the requests for service in time, even if it replies with a correct result.

2.4.5 Safety: Safety is freedom from unacceptable risks. Literally, safety is the situation where no hazard is present. A safety requirement is a description of a hazard incorporating the tolerable probability of the hazard [20]. There have been many techniques proposed to identify the probability of hazards in the early design phase based on some generic evaluation models (e.g. Component Fault Trees [25], Failure Propagation and Transformation Notation-Modules [26]).

Different from reliability that is concerned about the failures of functionality of the system, safety is about the value failures of data [27]. Therefore a more appropriate safety evaluation approach to analysing the probability of failures is based on the information about the impacts of actions on data of the system at runtime instead of the architecture of components and the system at design time.

For an action of an AC, it is considered as a failure if the returns of the action are out of the allowed ranges of values of the influenced data items. Suppose that there is an action act : Dat × ActV → di0 × di1 × · · · × din, dii (1 ≤ i ≤ n) ∈ Dat, then it is a failure to execute the action if invoke(sig.) ⇒ ∃i: 1 ≤ i ≤ n(valueOf(di) /∈ dii.range). Furthermore, when the action is realised via performing a required service, the failure of the action also implies the failure of the required service. Thus, safety of a
service provider can be quantified by the value \( FP \) based on the returns of the provider in time when it is requested for services.

### 2.5 Decisions (behaviour rules)

The decisions of an AC originate from the AC’s reactions to the changes of its internal and external states. Correspondingly to actions of the AC, the decisions can also be classified into several categories as below.

- Functionality actions relevant, such as (i) to do nothing, that is, not to serve the outside temporarily or permanently, when the AC finds its quality cannot satisfy its expectations. When the AC decides not to provide services again, it is equivalent that the AC quits the system. (ii) To start to execute functional actions, that is, to be committed to providing services. (iii) To plan the realisations of its exported services, that is, to implement its exposed interfaces.

- Quality actions relevant, such as to select appropriate service provider candidates based on the quality evaluations and the trust degrees when the qualities are dissatisfied.

For the rules related to the AC’s qualities, they can further be divided into three categories.

1. Rules for evaluating the qualities of service provider candidates and the qualities of the AC itself.

After the AC selects and invokes a service provider (identified by \( x \) below), it will collect the information about the run of the service provider, such as returns and reply time, and then re-evaluate the qos of the service provider based on the obtained information.

- Selected(\( x \)) and Invoked(\( x \)) \( \rightarrow \) Re-evaluate the qualities of \( x \).

Every time when the AC takes an action, the AC will call its quality satisfaction degree evaluation functions to evaluate its overall qualities and further adjust its behaviour based on the evaluations.

- An action is performed \( \rightarrow \) Re-evaluate the qualities of the AC.

2. Rules for maintaining the trust degrees related to service provider candidates.

Because the quality of a service provider may change over time, the trust degree related to that service provider should be adjusted correspondingly.

- QualityRe-evaluated(\( x \)) \( \rightarrow \) Adjust the trust degree related to \( x \).

When a new AC enters the environment, it should register to another AC as one of service provider candidates of a required service of the latter, and then it will be assigned with an initial trust degree by the latter so that it could be requested for services.

- isNewcome(\( x \)) \( \rightarrow \) Assign an initial trust degree value related to \( x \). Generally, the initial trust degree is low.

3. Rules for maintaining the service provider candidates lists.

When the qos of a service provider is poorer than tolerable, the AC will discard it from the service provider candidates list.

- \( \exists (s \in AC..Srv) \land \exists (p \in s.pc_list \land x \in p) \land qualityIsTooPoor(x) \rightarrow Remove x from the candidates list. qualityIsTooPoor(x) \) will be calculated based on the re-evaluated value of the quality of \( x \) and the quality expectation on the required service that \( x \) will provide, that is, \( qualityIsTooPoor(x) \) is valid if the current quality of \( x \) is worse by far than the expectation of the AC on the required service.

On the other hand, when a new service provider candidate enters the environment, the AC will add the service provider into its service provider candidates list.

- isNewcome(\( x \)) \( \rightarrow \) Insert \( x \) into the candidates list.

This kind of rules will maintain the relationships between the AC and its service providers to be up-to-date and make the AC possible to accomplish its objectives under the dynamic environment.

### 3 Implementation and running support of ACs

#### 3.1 Implementation of AC

As described above, the interface of an AC will be implemented by a process which is recursively specified as the compositions of sequential, branched and iterative actions of the AC. In [28], we described an algorithm for translating a BPEL-specific process into equivalent behaviour rules. Thus, an AC is implemented with the following structure (Fig. 2):

- The ‘rule engine’ is the kernel of the AC, which will reason about the AC’s behaviour based on the states of the AC’s own and the required services.

- The ‘service provider candidates lists’ maintain the lists of provider candidates (i.e. other remote ACs) for the required services and the trust degrees related to service providers.

- The implementations of the ‘actions’ may reference to external service providers. Through the running support platform of ACs to be mentioned below, the AC can select appropriate service providers to implement its actions.

In the implementation, we integrate a rule engine, Drools [29], which uses the Rete [30] algorithm to process rules and
provides object-oriented interfaces for integrating the rule engine into systems developed in Java. Using Drools, the AC can reason about its behaviour based on declarative rules.

### 3.2 Running support of AC

To support the run of ACs, there are three commonly used strategies.

- The first is to extend the existing runtime support for components, for instance, implementing a new type of component container for providing space for the executions of ACs and managing the lifecycles of ACs.

- The second is to pack an AC including its function modules into an indivisible whole and then deploy the AC onto some existing runtime support platform. In this case, developers should implement an assembling module for packing the modules.

- The third is to deploy an AC’s modules as separate components on an existing runtime support platform and meanwhile compose a suitable deployment profile. When the deployment profile is uploaded onto the platform, the platform will use the profile to weave the separately deployed modules into a logically united component.

We adopt the third strategy to support the run of ACs so that it is not necessary to implement a new platform to support the executions of ACs or program the rule engine into all ACs, and further the actions of ACs could be implemented as independent modules.

We select the Spring Framework [31], which is a lightweight J2EE Application Framework and one of the most remarkable open source frameworks, as the support platform of ACs. The modules of ACs can be deployed as plain EJB components on the Spring Framework.

Concretely, the modules of an AC include the information manager, the rule engine, the required services manager and the action manager, and the AC will be synthesised from the interactions among these modules when they are deployed. The interactions among the modules are shown in Fig. 3, in which the dotted-line rectangle denotes the boundary of the AC.

- Initially, when the AC joins the system, it first ‘registers’ itself as a potential service provider candidate to those ACs it knows. On the other side, the ‘required service manager’ of the AC stores the registration information in the ‘service provider candidates lists’. (Currently, we just implemented the method of matching required services based on the literal notations of interfaces.) The implementation of the mechanism of discovery and registration of required service provider candidates borrows the idea of peer discovery in hierarchical P2P systems [32], such as

![Figure 2 Implementation structure of AC](image)

![Figure 3 Runtime structure of AC](image)
KaZaA [33] and eDonkey [34]. For an AC, its required service manager is also responsible for registering service provider candidates and publishing (or broadcasting) the registration information.

- When the AC receives a ‘request’ for service from another AC, it will insert the request as a ‘fact’ into the information base of the ‘rule engine’. After this, the rule engine will start to reason about the behaviour of the AC based on its information base and the states of the AC’s ‘internal data’ and the ‘required services’.

- When a behaviour rule is triggered, the rule engine will perform the action specified by the behaviour rule, to reply to the request (or implement the interface), to evaluate the qualities of service providers, to adjust the trust degrees related to service providers or to choose service providers from the candidates.

- When an ‘action’ is implemented via invoking external services and when it is performed, the AC will send requests for services to the chosen service providers.

- After that, the required service manager will capture the returned information from service providers and update the information into the ‘service provider candidates lists’. The captured information is also a part of the information base of the rule engine, so the rule engine may trigger behaviour rules relevant to the management of the required services once the ‘service provider candidates lists’ are modified.

As mentioned above, to assemble an AC from its constituent modules (e.g. rule engine, actions and required service manager), there should be a deployment profile to specify the interconnections of the separate modules. We use a tool called JET (Java Emitter Templates) released by the EMF (Eclipse Modelling Framework) project [35] to implement the automated generation of the deployment profile of an AC [36]. The JET is a universal template engine and it includes the JETEmitter class which implements a method called generate() to translate a template into a text file. Thus, the JET can generate different format texts (e.g. SQL, XML and Java code) based on user-customised templates.

Before using the JET to generate an AC’s deployment profile, we design a template specific to the Spring Framework (Fig. 4).

In the template, line 2 is used to generate the Spring deployment profile head, line 9 defines the unique identity of an AC; lines 11–23 are used to generate the attributes of the AC, such as interface (currently, we presume that an AC just implements one interface), behaviour rules and required services; lines 23–35 are used to generate the interconnections between the AC and its modules, such as

```xml
1  ...<bean id="AComponent" class="edu.pku.ac.AComponent">  
2  <property name="name">  
3  <value>"AComponent"</value>  
4  </property>  
5  <property name="interface">  
6  <value>"edu.pku.ac.AComponent"</value>  
7  </property>  
8  <property name="requiredServices">  
9  <value>"edu.pku.ac.AComponent"</value>  
10  </property>  
11  </bean>  
12  ...<dependency>  
13  <bean id="ruleEngine">  
14  <property name="ruleBase">  
15  <value>"edu.pku.ac.AComponent"</value>  
16  </property>  
17  </bean>  
18  <bean id="infoManager">  
19  <property name="infoBase">  
20  <value>"edu.pku.ac.AComponent"</value>  
21  </property>  
22  </bean>  
23  <bean id="servicesManager">  
24  <property name="serviceBase">  
25  <value>"edu.pku.ac.AComponent"</value>  
26  </property>  
27  </bean>  
28  ...</dependency>
```

**Figure 4** Deployment profile template specific to the spring framework for ACs
information manager, required services manager, actions manager and rule engine. The deployment profiles of the modules are defined by using other templates and we do not present them because of the space limit.

4 Case study

4.1 Example application system

To verify that ACs can be used to implement high-quality software applications, we implement a simple application system by using ACs.

The application simulates a travel agency, which is designed to assist customers to reserve flights and hotels and suggest travel plans for customers. The agency is implemented as an AC with the following features:

1. Functionality-relevant actions
In process of providing travel assistance, the agency will take actions such as ‘capturing travel requirements’ of customers, ‘reserving flights’, ‘reserving hotels’ and ‘suggesting travel plans’ for customers.

2. Required services
While reserving flights, reserving hotels and planning travel plans for customers, the agency will invoke some external services specific to ‘flight reservation’, ‘hotel reservation’ and ‘travel plan generation’. Every external service will be provided by some service provider candidates and candidates may be implemented differently, for example, a candidate can provide an independent reservation service and another candidate may further depend on other flight reservation services to reserve flights when it tries to reserve a long-distance flight that includes transfers among different airline companies (Fig. 5).

3. Quality expectations
In the current implementation, we mainly focus on the ‘availability’ and ‘timeliness’ of the agency. To express the expectations on timeliness of the required services, the agency defines an upper limit of response time for each required service.

4. Quality-relevant actions
Correspondingly, the agency will call a group of methods, which will be described in the next sub-section, to evaluate the qualities of service provider candidates and improve its own.

5. Management-relevant actions
The agency adjusts the trust degrees related to service providers according to the evaluation on the qualities of the providers.

6. Interface
The agency exposes an interface to ‘assist customers to make travel plans’ and the process of providing travel assistance for customers is depicted as follows (Fig. 6).

7. Decisions
The agency makes decisions on choosing appropriate service provider candidates by using the specific selection strategy to implement its actions. On the other hand, the process of providing travel assistance of the agency will be transformed into behaviour rules and the agency will decide the executions of those behaviour rules to provide services for customers.

4.2 Actions

In the description of the behaviour of an AC, we have mentioned that the behaviour rules will trigger actions to perform the functional and quality goals of the AC. Among the actions relevant to the evaluation and improvement of the qualities of the AC, there are two categories of significant actions, that is, (i) re-evaluating the qualities of service providers for the required services of the AC according to the use (i.e. invocations) of the providers and (ii) adjusting the trust degrees related to the providers based on the evaluation results.

4.2.1 Re-evaluation of availabilities: The availability of a software entity is the measurement of the accessibility of the software entity in a timely manner [37]. For a service provider candidate, its availability can be quantified as an approximate value of the successful invocation ratio during a given period (e.g. 100 invocations). The agency uses the following algorithm to (re-)evaluate the availability of a service provider, where SuccessRatio denotes the successful ratio of invoking the provider’s service.

- Initially, SuccessRatio = 0.
- If the provider successfully responds to the current request from the agency, the agency will raise the success ratio of invocations related to the provider, that is, let SuccessRatio = (SuccessRatio × 100 + 1)/(100 + 1).

Figure 5 Dependency relationships between the agency and required services

Note: The figure shows the dependency relationships between the agency and required services, including Flight Reservation, Hotel Reservation, and Travel Plan Generation.

Figure 6 Process of the travel agency

In the process, customers list their travel requirements first, the agency reserves the flight and hotel for customers then, and the agency provides the travel suggestion for customers at last.
• When the value of HotServiceIndicator reaches the top limit \( \beta \times \beta \), that is, HotServiceIndicator \( \geq \beta \times \beta \), let HotServiceIndicator = \( \beta \times \beta / 2 \).

4. The current trust degree related to a service provider candidate is calculated as

\[
\text{td} = e^{\left[ \alpha \times (\text{dif}_{\text{avail}} + \text{dif}_{\text{timeliness}}) / \beta - (\text{HotServiceIndicator} / \beta) \right]}
\]

The computation for trust degrees is defined as an exponential function so that the trust degrees could be converged quickly and consequently the agency could locate and select the best quality provider candidates to request for services as quickly as possible.

\[
\text{dif}_{\text{timeliness}} = \begin{cases} 0, & \text{if } \text{ActualRT} > \text{WorstRT} \\ -\ln \left( 1 - \frac{\text{WorstRT} - \text{ActualRT}}{\text{WorstRT}} \right), & \text{otherwise} \end{cases}
\]
to service provider candidates with high QoS. Thus, the more the agency trusts a service provider candidate, the more probably it will select the candidate for services.

Suppose that there are $N$ candidates for a special required service, the agency will use a distribution probability for selecting candidates based on the following calculation

$$P_i = \frac{t_d}{\sum_{j=1}^{N} t_d_j}$$  \hspace{1cm} (5)

in which $t_d$ is the current trust degree related to the $i$th service provider candidate and $P_i$ is the corresponding probability of selecting the candidate.

Furthermore, when a selected candidate fails to reply to the service request of the agency, the agency can retry at most $N$ times to select another candidate if the QoS of the agency is tolerable (for instance, the subsequently selected candidate is possible to reply to the request in time, that is, meet the agency’s timeliness requirement).

4.4 Experiments

4.4.1 Settings: For each required service, there are five service provider candidates deployed on a cluster of servers. These servers are with the same configuration, that is, with an Intel® Core™2 Duo CPU E7300 @ 2.66 GHz, 2.66 GHz and 1.99 GB memory.

The availabilities of the service provider candidates are compliant with the normal distribution and the range of the availabilities is from 60 to 95%. Meanwhile, the response times of the provider candidates ranged from 100 to 1000 ms and the expected response times corresponding to the required services of the agency are all set to 500 ms.

As mentioned before, $\alpha$ and $\beta$ in formula (4) are two empirical constant values in the system and they together will impact on the changes of trust degrees. Currently, $\alpha$ is designated to 4.6 and $\beta$ is set to be 40. Nevertheless, we still compare the experimental results when $\alpha$ and $\beta$ are assigned with different values to evaluate the impacts of the values of $\alpha$ and $\beta$.

In addition, to evaluate the effectiveness of the strategy of selecting service provider candidates, we compare the currently adopted strategy with two generic selection strategies. The first strategy (referred to as MTFS, for example, ‘most trusted first selected’) is always to select the most trustworthy service provider candidates first. In this strategy, the system will first try the service provider candidates for a specified times (e.g. 100 times) and obtain the initial trust degrees related to the candidates. The system will dynamically adjust the trust degrees as well in the later tries. The second strategy (referred to as RSAS e.g. ‘recently successful always selected’) prefers to select the service provider that succeeds to reply to the request recently. In this strategy, the system does not accumulate its experiences on trying the service provider candidates and it will randomly select another candidate when the recently selected candidate fails to reply to the request of the system.

4.4.2 Experimental results: We test the qualities of the system (i.e. the agency) from several aspects as follows:

1. First, the dynamic changes of the availability of the system under the assumption that the system is not concerned about its timeliness requirement (i.e. $\omega_1 = 1$ and $\omega_2 = 0$). In this case, the experiments are divided into two sub-categories.

   • (Case 1.1). The system is considered to be failed whenever it finds that a selected service provider candidate is unavailable to reply to the request for the required service. This group of experiments is designed for showing that the system can find the most available service providers efficiently and meanwhile gain a high availability as quickly as possible autonomously.

   • (Case 1.2). While requesting for a service, the system will continue to choose another service provider candidate when the currently selected candidate is unavailable until the newly selected candidate is available or no more candidates are selectable. These experiments are designed for verifying that the system can improve its overall availability effectively when there is more than one service provider candidate for a required service.

2. Second (Case 2), the performance of the system when the timeliness of service provider candidates are taken into consideration (i.e. $\omega_1 = 0$ and $\omega_2 = 1$). In this case, we are just concerned about how fast the system can run when the system succeeds to provide services for customers. Whenever a service provider fails to reply to the request of the system, the system will be activated into another round of runs and the failures of the system will not be counted.

3. Third (Case 3), the synthesized quality of the system under the situation that both availability and timeliness of the system are taken into consideration (i.e. $\omega_1 = 0.5$ and $\omega_2 = 0.5$). In this case, when a service provider candidate for a required service is unavailable and there is still enough time (i.e. the upper limit of response time is not run out of yet) to select other provider candidates, the system will choose another candidate to request for the required service. These experiments are designed for verifying that the system can improve its overall quality (i.e. both availability and performance) simultaneously.

4. Finally (Case 4), the adaptation of the system under the situations that existing service providers quit the system and new service providers enter the system. In this case, we purposely remove some service providers with good qualities from the servers and then deploy some new service providers with better qualities on the servers. These experiments are designed for verifying that the system can
adapt to the changes of the environment and can always provide higher quality services for customers.

In case 1.1, as shown in Fig. 7, since the ACs (including the agency and the service providers) adopt the same adjustment strategy to improve their availabilities, their availabilities are not stable at the early stage so that the availability of the system is not stable. After a number of rounds of runs, those ACs connect to their most available service providers and then enter into relatively stable states. Consequently, the system finds out its most available service providers soon and then relies on them to perform its requests. With the run times increasing, the availability of the system becomes higher.

Comparatively, when the system adopts the MTFS strategy, the system can reach the statistically maximum value more quickly than our approach and the overall quality is ever better than that of our approach (Nevertheless, we will show its shortage in case 4 below). When the system adopts the RSAS strategy, the quality of the system always keeps at a rather low level since the system cannot find the best quality service providers all along.

In addition, as shown in Fig. 8, when $\alpha$ is assigned with a bigger value and $\beta$ is smaller than the current settings mentioned above, the quality of the system can converge to the maximum value quickly. Nevertheless, because the system may switch among different service provider candidates more frequently, the quality of the system will fluctuate and it seems worse (since the statistical values of the quality are average values). On the contrary, when $\alpha$ is assigned with a smaller value and $\beta$ is with a bigger value, the system has to take a long time to obtain its best quality though the maximum value of the quality is not better than that of the current settings.

In case 1.2, the selection strategy realises a redundancy-based fault-tolerant mechanism. Theoretically, suppose there are $N$ service provider candidates for a required service, the availability of the required service can be calculated as

$$\text{Rel}_{\text{AC}} = \sum_{i=1}^{n} r_i - \sum_{i_1=1; i_2 \neq i_1}^{n} r_{i_1} \times r_{i_2} + \cdots + (\alpha-1)^{n-1}$$

$$\times \sum_{i_1, \ldots, i_n=1; i_1 \neq i_2 \neq \cdots \neq i_n}^{n} \prod_{j=1}^{n} r_{i_j}$$

Figure 7 Finding high availability service providers and gaining higher availability through requesting for services from the found service providers

Figure 8 Impacts of the setting of parameters $\alpha$ and $\beta$
As shown in Fig. 9, the system can almost always be executed successfully when the number of service provider candidates for each required service is more than 1. The results also show that the system can enter a stable state very quickly.

In case 2, we assume that the execute time of the local computations of the agency can be ignored. As shown in Fig. 10, the performance of the system turns better very quickly and the system will always request for services from those better performance service providers after the system tries them for a while.

In case 3, because a high availability service provider candidate may not always be with a high performance, the overall availability of the system will decrease when the timeliness requirements of the system are taken into account (Fig. 11a). On the other side, a high performance service provider candidate may have a low availability and thus the overall performance of the system will turn worse (Fig. 11b). In calculation of the average performance, only those successful runs of the system are counted. Nevertheless, the system can quickly find those service provider candidates with higher availability and better performance as well and then stably provide high-quality services for customers.

In case 4, as shown in Fig. 12, at stage 2, the agency's availability and performance turns worse immediately after the best quality service providers quit the system, but the agency can find those better quality service providers to request for services and gain a relatively better overall quality soon. At stage 3, the agency changes to depend on those newly joining service providers with qualities better than those existing service providers and finally gains a higher quality than before.

Comparatively, when the system adopts the MTFS strategy, although the system can quickly enter a good service state when the environment is stable and its quality does not drop down dramatically when those best quality service providers quit the environment, it cannot adapt to
the changes of the environment since it does not leave any opportunities for the newly joining service providers. When the system adopts the RSAS strategy, the quality of the system has very limited improvement though there are better service providers joining in the system.

5 Related work

5.1 Component models

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, Teschke and Ritter [38] propose a unifying component description language for integrated descriptions of structure and behaviour of software components and component-based software systems. Ben-Shaul et al. [39] present a component model for encapsulating services, which allows for the adjustment of structure and behaviour of components to changing or previously unknown contexts. Humberto and Richard [40] introduce service-oriented concepts into a component model and execution environment and define a model for dynamically available components. Maamar et al. [41] discuss the seamless combination of context and policy to manage behaviour that Web services expose during composition and in response to changes in the environment. The Fractal model [42] is endowed with arbitrary reflective capabilities and supports developing components with adaptation policies. Arbab [43] introduces the notion of Abstract Behaviour Type as a higher-level alternative to abstract data type (ADT) to model for both components and their composition.

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.
5.2 Autonomous component

Recently, there is much work on modelling autonomous software entities. Kral and Zemlicka [44] describe the needs for ACs and gives some comparisons between ACs and autonomous agents. In [4], an autonomy-oriented computing paradigm is described, in which systems are formed of ACs. In [45], the state/event LTL properties of software systems assembled from ACs are formally studied.

Our component model is similar to the autonomous entity model described in [4]. However, our model integrates quality properties. In addition, our model does not mention the goals of an autonomous entity since goal is an intentional concept and it is too virtual to be implemented in a concrete software entity.

5.3 Agents and multi-agent systems

Many of the properties of ACs are similar to those of agents and there is already a great deal of research work on agents (see the roadmap for agent research published by the AgentLink [46]). Like agents [47], ACs can take the control over their internal states and their behaviour modes and dynamically adapt and respond to the changes of their environments. Nevertheless, ACs are different from agents on some aspects. For example, agents are developed for the achievements of specific goals whereas ACs are mainly for reuse and assembly like traditional components. Agents are active software systems taking actions on behalf of their owners whereas ACs do not need to be standalone executable software entities, although they are independent of ones who integrate and deploy them.

There is not an agreed formal model for agents that can specify both the functional and the non-functional requirements at the abstract level. The constructions of agents are basically ad hoc and non-standardised even though they may be declared conformed to some standards (e.g. the FIPA [48]). In addition, not all of the software entities on the Internet are agents and it is impractical to model or wrap all Internet-based software entities as agents. Therefore, in this paper, we adopted the notion of AC as a compromise between agent and traditional passive component, which uses some features of agents to support the autonomy of software entities on the one hand and adopt the technology of traditional components to realise the functional and non-functional properties of autonomous software entities on the other hand.

In multi-agent systems (MAS), much work focuses on the web services composition based on agents (e.g. [40, 49–52]) and Huhns and Singh [53] discuss the research directions in that area. However, current work is still focusing on the composition of functionalities of software systems. Meanwhile, there is also some work discussing about trust. Ramchurn et al. [54] provide a survey on the state-of-the-art of the areas of trust and reputation in MAS. In [55], a formal language for expressing the relevant conditions of belief and trust is described. Similarly as our work, Maximilien and Singh [56] propose a multi-agent framework, in which agents are automatically selected according to the associated trust levels. Differently, ACs in our work use a learning algorithm based on the annealing algorithm to adjust the trust degrees related to service providers.

5.4 Non-functional properties

There is much work on relating non-functional properties (or qualities) with functionalities of software entities. For example, Han [57] introduces a framework dealing with interface signature, interface constraints, interface packaging and configurations, and non-functional properties of software components. Cysneiros et al. [58] present a framework for integrating non-functional requirements (NFRs) into the ER and object-orientation (OO) models, where NFRs are modelled as special objects. However, in that work, qualities are not separable properties of software components like data and operations.

In addition, a lot of work focuses on modelling software qualities [59]. For example, Mylopoulos et al. [7] and Musa et al. [18] survey a significant number of quality models and 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 modelling and design phases. Our work is different in that it is concerned with those properties that are emergent from the executions of software systems, that is, runtime properties such as reliability, availability, performance and so on.

Some related work on QoS has been done in the area of service selection. Zeng et al. [60] present a platform that addresses the issue of selecting web services for the purpose of their composition in a way that maximises user satisfaction expressed as utility functions over QoS attributes. Ran [61] presents an approach to storing quality information (e.g. availability) of web services into the UDDI registry. The UDDI service is responsible for monitoring and updating service states and returning service candidates with satisfied quality to the client. However, service availability is a property that depends on not only the service implementation but also the network states between service requestors and providers. Another typical approach is to delegate customer requests to a service broker [16], which invokes appropriate service(s) and then sends the result back to the original client. However, this approach needs the central broker to process a large number of client requests, where the broker often becomes the performance bottleneck when the number of clients and services scales up.
In the study of the relationships and compositions of quality properties, some work also takes into consideration the software architecture or the process. Rosa et al. [62] propose the Parmenides, which is an architecture-central development framework. The framework supports the specifications of non-functional requirements, the integrations of functional and non-functional parts, and the mappings from non-functional requirements to the implementations on a specific platform. However, the framework does not touch upon the analysis and verification of non-functional properties, and further it cannot cope with the non-functional properties related to the global behaviours of software systems. Franch and Botella [63] describe a formal language, NoFun, to combine non-functional properties into software architectural description languages. However, NoFun cannot express the relationships among non-functional properties. In addition, Mylopoulos et al. [7] propose a comprehensive framework for representing and using non-functional requirements during the development process.

There is also much effort focusing on improving system availability at runtime. When components are developed and deployed in a single organisation, the lifecycles of components are controlled by a centralised management server and then the availability of systems composed of that type of components can be improved via system redeployment [64]. However, on the Internet, redeployment techniques are unsuitable because ACs are deployed by third parties and cannot be migrated. Moreover, an AC’s availability may change at runtime, whereas existing redeployment algorithms often consider it as a static value. Yu and Vahdat [65] discuss the means of improving availability of Web services. Birman and van Renesse [66] describe an approach to improving the availability of Web services via modelling autonomic behaviours in Web services. However, that work is mainly focusing on the availability of a single service. As we have mentioned before, improving the availability of a service locally cannot guarantee that the service is subsequently highly available in the dynamic and unstable network environment. In [67], the process flow for designing and implementing high-availability technologies is discussed. Sutcliffe et al. [68] develops an evolutionary computing tool for requirements analysis and optimisation of component-based systems, in which designs can be optimised on aspects of reliability, performance and costs.

6 Conclusion 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 properties 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 used the concept of an AC to encapsulate data, behaviours and quality properties in a uniform unit. We also described the implementation and running support of ACs and studied a case application system. Although the application system is preliminary and only deals with availability and timeliness, we believe it can be extended for other quality properties. How to approximately evaluate other related qualities of remote service providers and how to deal with multiple quality properties will be our future work.

In the current work, the quality values of service provider candidates are scattered in different ACs and they are not inconsistent, and ACs have to take more time to obtain the real quality values of service providers since an AC newly participating in the system may be unable to obtain the up-to-date estimated quality values of existing service providers. At the next stage, we will offer some knowledge sharing mechanism so that ACs could quickly obtain the most recent quality properties of existing service providers.

In the next stage, we will study the formal properties of ACs and AC-based software systems in a more rigorous way to make it possible to formally infer, predict and assure of the qualities of software systems along with the development of software systems. Further, we will investigate the methodology of constructing high-quality software systems based on ACs and upgrade the prototype for more quality properties.

7 Acknowledgment

First of all, we thank the anonymous referees, whose useful comments have greatly improved the presentation of this paper, and Dr. Xi Sun and Mr. Lei Zhuang, who provided some experiment results when they were graduate students with the Institute of Software, EECS at Peking University. This work was supported in part by the National Basic Research Program of China (973) (2009CB320700), the National Key Technology R&D Program of China (2008AA010139), the Science Fund for Creative Groups of China (60821003) and the National Natural Science Foundation of China (60773151).

8 References


