SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

September 2015, Vol. 58 092103:1–092103:15
doi: 10.1007/s11432-015-5307-9

Auxo: an architecture-centric framework supporting the online tuning of software adaptivity

WANG HuaiMin[1], DING Bo[1*], SHI DianXi[1], CAO JianNong[2] & Alvin T.S. Chan[2]

[1]*National Key Laboratory of Parallel and Distributed Processing, College of Computer, National University of Defense Technology, Changsha* 410073*, China;*
[2]*Department of Computing, Hong Kong Polytechnic University, Hong Kong, China*

Received September 30, 2014; accepted November 30, 2014; published online July 13, 2015

Abstract Adaptivity is the capacity of software to adjust itself to changes in its environment. A common approach to achieving adaptivity is to introduce dedicated code during software development stage. However, since those code fragments are designed a priori, self-adaptive software cannot handle situations adequately when the contextual changes go beyond those that are originally anticipated. In this case, the original built-in adaptivity should be tuned. For example, new code should be added to provide the capacity to sense the unexpected environment or to replace outdated adaptation decision logic. The technical challenges in this process, especially that of tuning software adaptivity at runtime, cannot be understated. In this paper, we propose an architecture-centric application framework for self-adaptive software named Auxo. Similar to existing work, our framework supports the development and running of self-adaptive software. Furthermore, our framework supports the tuning of software adaptivity without requiring the running self-adaptive software to be terminated. In short, the architecture style that we are introducing can encapsulate not only general functional logic but also the concerns in the self-adaptation loop (such as sensing, decision, and execution) as architecture elements. As a result, a third party, potentially the operator or an augmented software entity equipped with explicit domain knowledge, is able to dynamically and flexibly adjust the self-adaptation concerns through modifying the runtime software architecture. To truly exercise, validate, and evaluate our approach, we describe a self-adaptive application that was deployed on the framework, and conducted several experiments involving self-adaptation and the online tuning of software adaptivity.

Keywords software architecture, self-adaptive software, architecture style, application framework, software adaptation

Citation Wang H M, Ding B, Shi D X, et al. Auxo: an architecture-centric framework supporting the online tuning of software adaptivity. Sci China Inf Sci, 2015, 58: 092103(15), doi: 10.1007/s11432-015-5307-9

1 Introduction

In recent years, we have witnessed the increasing importance of software designed to operate in highly dynamic settings, such as pervasive computing [1] or cloud computing [2] environments. In contrast with traditional software, it shows the novel paradigm of growing construction and adaptive evolution [3]. This

* Corresponding author (email: dingbo@nudt.edu.cn)

trend has given rise to active research and development in the area of software adaptation [4–7], with the aim of coming up with software that will adjust its behavior in response to changes in the environment. A common approach in software engineering to achieve this goal is to introduce dedicated code during the software development stage, including code to detect environmental changes (*sensing*), code to guide adaptation decisions (*decision*), and code to adjust its business behavior accordingly (*execution*). By building such a "sensing-decision-execution" self-adaptation loop [7,8], the software is equipped with adaptivity, i.e., the capacity to adjust itself to changes in the environment.

However, since the "sensing-decision-execution" loop is designed a priori in the development stage, self-adaptive software may not be able to handle the situation adequately if the running environment goes beyond what was originally anticipated. This phenomenon is becoming increasingly probable in many "open-world" settings [9], such as long-life systems in which the designer's cognition is inevitably limited, or in the case of cyber-physical systems [10] that are tightly coupled with the real world. In order to cope with an unexpected environment, the original adaptivity in self-adaptive software requires tuning. For example, new code can be added to provide sensing of new contextual environments or to replace outdated adaptation logic (i.e., code to guide adaptation decisions [11]). For systems that require continuous availability, it is critically important that this kind of adjustment be executable on-the-fly. We name this kind of online adjustment of the self-adaptation loop as the online tuning of software adaptivity.

In this paper, we propose an application framework named Auxo, which supports both self-adaptation and the online tuning of software adaptivity. Similar to existing frameworks such as [11,12], our framework can support the construction and running of self-adaptive software. Moreover, our framework can support a third party to dynamically adjust each concern (i.e., sensing, decision, and execution) in the self-adaptation loop, to enable the running self-adaptive software to cope with unanticipated environments.

Our approach is based on the software architecture technology [13]. It mainly consists of two parts: the Auxo architecture style and the Auxo runtime infrastructure. Similar to other architecture-based approaches, our architecture style captures the structure and organization of the functional logic while providing intrinsic support for its adaptation. Importantly, the Auxo architecture style can also model the concerns in the self-adaptation loop. For example, a sensor and its associated data can be abstracted as a component of a special kind, while the adaptation logic can be encapsulated as connectors of a special kind. During runtime, the software can perform architecture-based self-adaptation according to its predefined adaptivity with the support of the runtime infrastructure. Meantime, a third party can selectively modify the architecture model to make changes to the self-adaptation loop, such as composing a specific component to detect an unanticipated contextual environment or replacing a specific connector to update a piece of adaptation logic. In addition, by abstracting software on an established architectural model, our infrastructure provides the foundational means to systematically evaluate architecture constraints, which can be used to guard against inappropriate adaptivity-tuning actions.

The main contributions of this paper include: (1) Describing and highlighting the challenges to the online tuning of software adaptivity. (2) Proposing an architecture-centric framework that concurrently supports self-adaptation and the online tuning of software adaptivity. (3) Validating our approach with a performance analysis based on two case studies and a real-life application. The rest of this paper is organized as follows. A further introduction to the research background is given in Section 2. Our framework is presented in Sections 3–5, with the architecture style presented in Section 3, the runtime infrastructure presented in Section 4, and those concepts reinforced in Section 5 by illustrating how self-adaptation and the online tuning of software adaptivity is realized. The case example and the evaluation results are presented in Section 6. In Section 7 our work is compared to related work.

2 Research background

This section begins with the presentation of a motivated scenario being extracted from a real-life cloud computing application. Subsequently, we give an overview of the technical foundation of our research, i.e., the runtime software architecture.

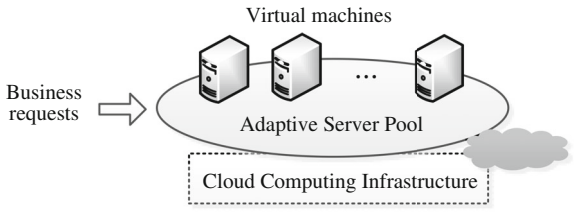

Figure 1 Motivating scenario-adaptive server pool.

2.1 Motivating scenario and challenges

Consider the following scenario. A server pool is adopted to process the client requests of a cloud service. Since the operating cost is calculated by the cloud's "pay-as-you-go" strategy, the size of the server pool is not fixed and the system is equipped to dynamically adapt by enlarging the pool when the system load exceeds the pool's current capacity (Figure 1). The prebuilt self-adaptation loop consists of (1) the code for *sensing* the system load, (2) the code to guide the *decision* to resize the server pool, and (3) the code responsible for the *execution* of applying resources from the cloud infrastructure and adding them to the server pool.

However, two unexpected cases emerge at runtime: (Case 1) the administrator finds that there is a lag between applying for the computing resource and getting it ready in the cloud, and he or she would like to introduce some kind of load prediction mechanism [14]; (Case 2) some load peaks are found to be not from the real clients but caused by a new type of network attack, and the system should be able to react discriminatively towards them.

Both of these cases were not anticipated during the development stage. Therefore, the original built-in adaptivity cannot meet the new requirements. Since this system has to provide "anytime accessible" services, we have to tune its adaptivity in an on-the-fly style. In other words, the prebuilt "sensing-decision-execution" loop has to be adjusted online. In Case 1 of the motivating scenario, we can enable the load prediction by simply adding an extra load analysis process into the sensing stage; and in Case 2, if the system has a built-in ability to block certain requests, we only need to add the means to detect the network threat as well as the corresponding adaptation logic.

The above scenario illustrates the concept of the online tuning of software adaptivity and its importance in real life. It also implies the challenges in its realization. (1) We have to adjust different parts of the self-adaptation loop in those two cases. (2) Those above-mentioned actions are performed at runtime. (3) Since self-adaptation and the tuning of adaptivity may concur at runtime, it is necessary to provide an approach that integrates the support for those two kinds of actions.

2.2 Runtime software architecture

Our approach is architecture-centric; in particular, it is based on the runtime software architecture technology. Software architecture represents the software as a network of components bound together by connecters [15]. It abstracts away lines-of-codes and focuses on the overall structure and system-level properties. Traditional software architecture models described by Architecture Description Languages (ADLs) are just design-time artifacts that help designers to profile and reason about the system in-the-large [16]. However, researchers have found that such a "big picture" can also contribute to changes in runtime software. By maintaining the knowledge of the runtime architecture of the system, concerns involving the high-level structure can be separated from concerns involving the concrete functional implementation. The high-level knowledge of the target system can provide the basis and foundational environment for systematic runtime change management. For example, it can facilitate the planning and execution of dynamic changes since a "bird's eye view" of the runtime system is maintained. In addition, it can help to assess the validity of a change by exposing important system-level constraints.

There are two key concepts in software architecture that are related to our work.

(1) Software architecture style. In order to capture the knowledge about the runtime architecture, the software usually has to be developed according to a specific paradigm. According to [17], a software

architecture style "defines a vocabulary of components and connector types, and a set of constraints on how they can be combined". In other words, it defines what the internal coarse-grained structure of an application looks like. Typical architecture styles include simple ones such as client/server and peer to peer, and more complex styles such as C2 and CORBA [13].

(2) Runtime software architecture model. Maintaining an explicit runtime software architecture model (RSA model in short) is a common approach to capturing the real-time architecture knowledge in most recent work (cf., Subsection 7.1). This model is causally connected with the real running system, such that any changes in this model will be mapped into the running system and vice versa. It can act as a source of information for software to introspect itself as well as the entry to manipulating the software architecture. In order to maintain the RSA model, an application framework is usually necessary.

In this paper, we present the Auxo framework that underpins the concept of realizing self-adaptation and the online tuning of software adaptivity. More specifically, applications are developed according to the Auxo architecture style, and those above two kinds of actions can be made with the support of the Auxo runtime infrastructure, which maintains an explicit and modifiable RSA model. The architecture style and the runtime infrastructure will be introduced respectively in the following two sections.

3 Auxo architecture style

As mentioned in Section 1, the prominent feature of the Auxo architecture style is that it can encapsulate the concerns in the self-adaptation loop (i.e., sensing, decision and executing) as architecture elements, laying the foundation for possible future tuning of software adaptivity at runtime. In this section, we introduce this architecture style by presenting the types of architecture elements in this style and an ADL named AuxoADL to reify this style, along with the development process of Auxo-based applications.

3.1 Types of architecture elements

There are three kinds of atomic architecture elements in the Auxo architecture style: components, connectors, and constraints. These elements serve the important role of architecturally identifying the core components of the system and how they are related and linked to one another through different connector and interface types.

3.1.1 *Components and connectors*

The Auxo components and connectors can essentially be classified according to the interfaces that they provide or bind to. Therefore, it is fitting that we introduce the interface types first. There are three types of interfaces in Auxo (Figure 2): *service interfaces* that provide the software service in the form of a set of functions, *context interfaces* that output the environmental changes in the form of events, and *reflection interfaces* that contain the introspection and intercession functions to adjust the behavior of a single component.

Based on those interfaces, Auxo defines the following component and connector types (Figure 2). Note that in contrast with other component models, we introduce context components and policy connectors specifically to encapsulate concerns in the self-adaptation process.

(1) *Context Components* encapsulate the sensing concern in the self-adaptation process. It monitors the environmental changes that are concerned and outputs them as context events. For example, a context component may be the software encapsulation of a physical sensor or a piece of code to detect the availability of computing resources.

(2) *Behavior Components* encapsulate the execution concern in the self-adaptation process as well as the general functional logic of the software. They provide interface-based services to the outside world and consume services that other components provide. A behavior component can also have a reflection interface to enable the outside world to access its states and adjust its behavior, if necessary.

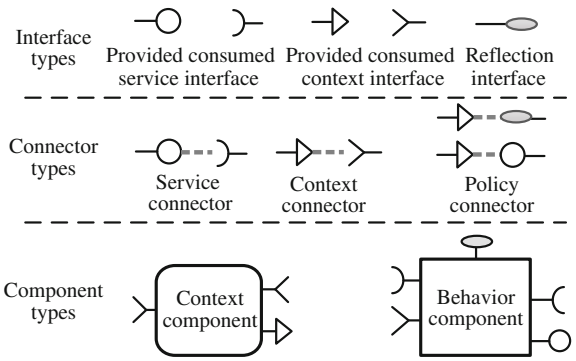

Figure 2 Interfaces, connectors and components in Auxo architecture style.

(3) *Policy Connectors* encapsulate the decision concern in the self-adaptation process, which bind context event providers and service providers together in the form of "when to do what". A service provider may be a service interface or a reflection interface of a behavior component.

(4) *Service Connectors* bind service providers and consumers together.

(5) *Context Connectors* bind context event providers and consumers together. With this connector, a context/behavior component can consume context events directly, which makes component-based context aggregation [18] and flexible context handling possible.

3.1.2 *Architecture constraints*

Auxo uses predefined architecture invariants (called architecture constraints) to guard against inappropriate changes to the software architecture. Any runtime change that violates a predefined constraint will be rejected by the Auxo runtime infrastructure. There are two types of constraints: style built-in constraints and customized constraints. The former are built in the Auxo architecture style. For example, the service connector can only bind service providers and consumers together. The latter are explicitly defined by application developers. For example, we can specify that there should be one and only one server pool controller in the motivating scenario at any time. Architecture constraints are defined by AuxoADL which will be introduced in the next subsection.

3.2 AuxoADL: a style-specific ADL

In the following description, we illustrate how to use AuxoADL, an architecture description language to capture and model the Auxo architecture elements that we introduced earlier. We make use of some sample code that will form the background for our motivating scenario.

(1) Defining components. The syntax to define components in AuxoADL is based on an enhancement of OMG's IDL [19]. The IDL2 keyword interface is used to define service interfaces. The reflection states/functions are directly defined in the component definition using the keyword *reflection*. Other major extensions to IDL2 include the keywords *component*, *context_event*, *uses*, *supports*, and *multiple*, and so forth. Figure 3(a) is an example of the component definition in our motivating scenario. In this example, after defining a set of services, we define two behavior components and one context component named *LoadMonitor*. This context component outputs an *OverLoad* context event with a Boolean value to indicate whether or not there are too many client requests.

(2) Defining Connectors. Service and context connectors are defined by their roles (i.e., the provider and the consumer) in the binding relationship. In contrast, the definition of policy connector is richer but more complicated. A policy connector encapsulates the adaptation logic in the form of "Event-Condition-Action" (ECA) [20] (Figure 3(b)). Those three parts respectively specify the context event that triggers the adaptation, the conditions on the environment or the internal states of the application that should be checked before the adaptation, and the concrete actions that should be executed when the event occurs and the conditions are satisfied.

a) AdaptiveLoadBalancer.cdl

```
interface SizeAdjustment{…};//Services defined in IDL2
interface RequestProcessor {…};
interface ServerActivator {…};
…
component PoolController {  //Defining a behavior component
   provides SizeAdjustment sa;  //Service provided
   uses multiple ServerActivator sat; //Service consumed
…
};
component ServerProxy {  //Defining a behavior component
   provides RequestProcessor pr;
   uses multiple RequestProcessor pr_to_server;
   reflection{ //Defining reflection states and methods
      readonly long threadPoolSize;
      voidaddToBlockList(string address);
      …
   }
   …
};
component LoadMonitor{ //Defining a context component
   provides context_event bool Overload; //Context provided
   …;
};
…
```

b) AdaptiveLoadBalancer.cf

```
configuration AdaptiveLoadBalancer { //Defining a configuration
…
   component_instance PoolController PC;
   component_instance LoadMontior LM;
   component_instance ServerProxy SP;
   …
   policy_connector PoolEnlarger //Defining a policy connector
      event LM.Overload;
      condition ((LM.Overload==true)&&
        PC.sa.ActivatedServerCount<PC.sa.TotalServerCount));
      action {
         server=PC.sa.AddServer();
         Auxo.AAS.AddServiceConnector(“”, “SP”, “pr_to_server”,
         sever.getname(), “pr”);
       };
   …
   constraint UniquePoolController //ensuring the uniqueness of PC
      Auxo.AAS.InstanceCount(“PoolController”)==1;
   constraint AtLeastTwoServers //ensuring at least two activated servers
      Auxo.AAS.AttachedConnectorCount(“SP”, “pr_to_server”)>=2;
   …
};
```

Figure 3 AuxoADL code fragments in the self-adaptive server pool.

Table 1 Sample functions in Auxo.AAS

Function	Usage
IsConnected (c1, i1, c2, i2)	True if the interfaces c1.i1 and c2.i2 are connected by a connector.
InstanceCount (c)	Counts the instances of component c.
AttachedConnectorCount (c, i)	Counts the number of connectors that are attached to the interface c.i.
AddServiceConnector(name, c1, i1, c2, i2)	Adds a service connector that binds c1.i1 and c2.i2 together.
DeleteConnectorByPort(c1, i1, c2, i2)	Deletes the connector that binds c1.i1 and c2.i2 together.

The syntax to define the *Condition* and *Action* is partially based on the syntax of CORBAScript[1)]. We hide unnecessary features and enhance it to access Auxo components and the RSA model. The latter is realized by invoking a service interface named AAS (short for Architecture Access Service), which is attached to a virtualized component whose name is “Auxo”. Auxo.AAS mainly provides two kinds of functions (Table 1): the functions to get and evaluate architecture information and the functions to modify the RSA model. For example, the *AddServiceConnector* function in the Policy Connector *PoolEnlarger* can dynamically add a service connector.

(3) Defining Constraints. As shown in Figure 3(b), developers can also specify invariants on the software architecture, namely, the customized architecture constraints. Basically, a constraint is a script-based expression that returns a Boolean value, which can invoke the functions of getting and evaluating architecture information in the Auxo.AAS service interface.

3.3 Developing a self-adaptive application

The process of developing an Auxo application consists of the following two parts. Part 1 is optional, i.e., if all desirable components can be found in the component repository, the developers can skip this part.

Part 1: Component development. First, the developers define the outward feature of a context or behavior component using AuxoADL, as shown in Figure 3(a). The purpose of such a component definition is two-fold. On one hand, it explicitly exposes the meta-information of the component, laying the foundation for the possible manipulation of the architecture. On the other hand, it is also the basis for constructing components: A dedicated compiler can map the definition into a programming language-specific component skeleton, from which the developers can fill in the details of the implementation.

1) CORBA Scripting Language Specification. http://www.omg. org/spec/SCRPT/, v1.1, 2002.

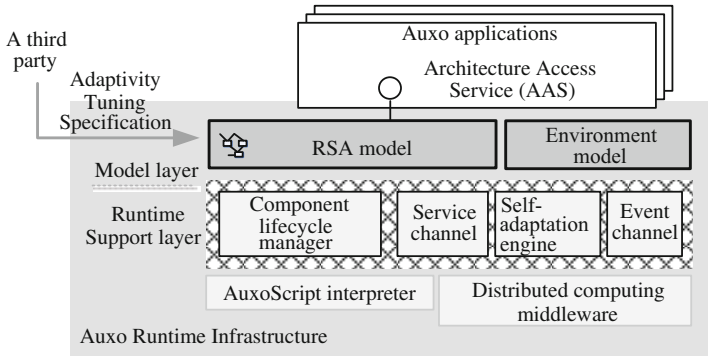

Figure 4 Overview of Auxo runtime infrastructure.

Part 2: Architecture definition. In this stage, the application developers select desirable components and specify connectors to bind those components together. The product is an architecture configuration that serves as a "blueprint" of the application. As shown in Figure 3(b) an architecture configuration is composed of three parts: component instance declaration, connector definitions and architecture constraint definitions. Along with the component definition (Figure 3(a)) and the component package, such an architecture configuration can be directly instantiated into a running application with the help of the Auxo infrastructure.

Note that in the architecture configuration, not only the function logic but also the sensing, decision, and execution concerns in the self-adaptation loop are encapsulated as components and connectors. For example, the context component *LoadMonitor* encapsulates the sensing means, and the policy connector *PoolEnlarger* encapsulates the adaptation logic in Figure 3.

4 Auxo runtime infrastructure

We have presented the process of developing self-adaptive applications as well as how to encapsulate all concerns in the self-adaptation loop as architecture elements. The next challenge is how to provide runtime support for those applications.

4.1 Infrastructure overview

As shown in Figure 4, the Auxo runtime infrastructure is built on top of a CORBA-based distributed computing middleware named StarBus, our previous work presented in [21,22]. Another substrate software entity is the AuxoScript interpreter being reponsible for interpreting scripts embedded in the architecture configuration. The script interpreter is mainly adapted from a CORBAScript interpreter in the underlying middleware, since the script syntax in AuxoADL is mainly based on that of CORBAScript as mentioned in Subsection 3.2.

Inside the Auxo runtime infrastructure, there is a layered architecture which is composed of *the runtime support layer* and *the model layer*. There are four major functional units in the runtime support layer: (1) the Service Channel reifies service connectors by dynamically locating the targets of service requests and transferring them. It is realized by virtue of the object dynamic locating and remote invocation mechanisms of the underlying distributed computing middleware; (2) the Event Service reifies context connectors by providing the event publishing/subscription functions; (3) the Self-Adaptation Engine reifies policy connectors and drives adaptation actions. This engine subscribes to context events from the Event service, so that whenever a matched event is received it will check the Condition part and execute the Action part (if the check passes) of the corresponding policy connectors. The check and execution is mainly realized by invoking the underlying AuxoScript interpreter; (4) the Component Lifecycle Manager provides support for the component's life-cycle control such as component instantiation, suspension, and deactivation. It also provides basic execution support for the components, for example, to get the handle of the Service Channel.

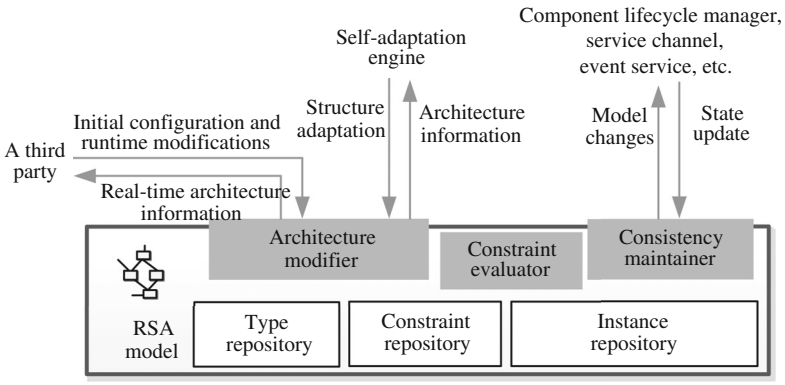

Figure 5 Internal structure of Auxo RSA model.

The model layer reflects both the environment and the running application. Its basic function is to provide the necessary information for the Self-Adaptation Engine to "understand" the current environment as well as the application. The environment model captures the contexts of the external environment and the RSA model is a high-level view of the running applications. More details about those two models can be found in the next subsection.

4.2 Internal structure of runtime models

Figure 5 shows the internal structure of the RSA model. Its major interaction relationships with the outside world are also presented. This model consists of three repositories and a set of assistant functional units: (1) The type repository stores the component definitions and the descriptions of related data types, which are necessary for performing type checks in the architecture manipulation process. (2) The constraint repository stores the customized constraints, which will be evaluated by the Constraint Evaluator, a functional unit is the RSA model, in the architecture modification process. (3) The instance repository stores descriptions of all running component instances and connectors, including their static information (such as its unique id) and dynamic states (such as its current lifecycle state). The initial contents of those repositories come from the AuxoADL-based description of the application as exemplified in Figure 3. At runtime, the instance repository is maintained by the Consistency Maintainer, which invokes the entities in the runtime support layer (such as the Service Channel, the Event Channel, and the Component Lifecycle manager) to map the model modifications to the real system and vice versa.

In contrast with that of the RSA model, the structure of the environment model is relatively simple. In brief, it can be regarded as a table in which each line stores the up-to-date state of a dimension of the environment, such as the temperature captured by a sensor or the availability of an external computing resource detected by a specific API. Each line is paired with a context event port of a context component and dynamically maintained by the value carried by events published through this port.

4.3 Architecture modification protocol

In Auxo, the adaptation actions can be classified into two types: the invocation of the reflection interface (or service interface) of a general behavior component, and the modification of the software architecture through the invocation of Auxo.AAS. The former can be easily realized with the aid of the underlying distributed middleware. The latter is realized by the Architecture Modifier, as shown in Figure 5. When receiving a runtime modification request, it will modify the running application according to a three-phase protocol:

Phase 1: modifying the RSA model. The Architecture Modifier first builds a backup of the RSA model. Then, it modifies the RSA model according to the modification request. For example, the *AddServiceConnector* request in the policy connector shown in Figure 3 adds a connector to a running application.

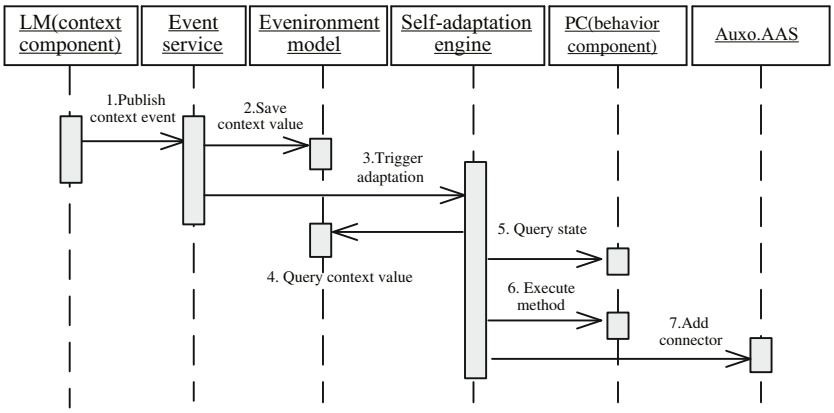

Figure 6 Sketch of interaction in runtime self-adaptation process.

Phase 2: evaluating architecture constraints. Taking the modified RSA model as an input, the Constraint Evaluator in Figure 5 evaluates all style built-in and customized constraints. The ability to evaluate style built-in constraints has been hardcoded in the Constraint Evaluator. Since the customized constraints are basically script-based expressions, they are processed by evaluating those expressions. If any constraint is violated, the instance repository in the RSA model will be rebuilt by the Consistency Maintainer and other parts will be restored from the backup of the RSA model. The modification process will be terminated and an error log will be generated.

Phase 3: enacting changes to the real system. The changes in the RSA model are mapped into the running system by the Consistency Maintainer in Figure 5. It invokes the interfaces of the entities in the runtime support layer to achieve this goal. For example, if the description of a new component is added to the RSA model, the Component Lifecycle Management will instantiate it according to this description.

5 Putting things together

In this section, we illustrate how to realize software self-adaptation and the online tuning of software adaptivity that is made possible through the synergistic support of the Auxo architecture style and the Auxo runtime infrastructure.

5.1 Achieving adaptivity

We illustrate the realization of runtime self-adaptation with the policy connector *PoolEnlarger* presented in Figure 3(b). To realize the self-adaptation action specified by this policy connector, the functional units in the Auxo runtime infrastructure interact as shown in Figure 6. When the application is instantiated according to its AuxoADL descriptions, the Adaptation Engine in the infrastructure reads in the policy connector, and the environment model adds a line in its table to cache the value carried by the context event Overload. Subsequently, at runtime, the context component *LM* publishes the context event Overload with the value true when there are too many client requests (#1). The environment model gets this value by monitoring all context events and saves it (#2). The Self-Adaptation Engine is activated by this context event (#3), and then evaluates the Condition part of the *PoolEnlarger* by querying the context values from the environment model (#4) and the component states (#5). If the specified condition is satisfied, it executes the script in the action part, i.e., activating an idle server (#6) and then invoking the Auxo.AAS service to connect this server to the server proxy (#7).

5.2 Online tuning of adaptivity

When a third party such as an operator finds that the above self-adaptation process cannot adequately handle an unexpected environment, he or she can tune the adaptivity by adjusting the original self-adaptation process dynamically. The Auxo framework primarily focuses on support for the specification

```
target_configuration AdaptiveLoadBalancer
component_instance LoadMontiorWithPrediction LP;
action {
    PortMapping=StringSeq("LM.Overload", "LP.Overload");
    Auxo.AAS.ReplaceComponent("LM", "LP", PortMapping);
};
```

Figure 7 Example of adaptivity tuning specification.

and enaction of the online tuning of software adaptivity. The framework is oblivious of what should be tuned and when, because this kind of action is triggered by the third party instead of by the software itself.

Since all concerns in the self-adaptation loop have been encapsulated as architecture elements, we can easily realize the tuning by modifying the architecture through the Auxo.AAS interface. To allow a third party to specify the desirable tuning actions at runtime, the Action clause of the policy connector can be independently used as an Architecture Modification Language (AML) [23]. For example, the adaptivity tuning specification in Figure 7 replaces the context component *LM* with a newer one, which has the ability to predict loads. Such a specification can be injected by a GUI tool named AuxoME that we designed based on GME (General Modeling Environment) [24] or by directly invoking the management interface of the Auxo runtime infrastructure. After receiving an adaptivity tuning specification that contains architecture modification actions, the runtime infrastructure will execute the steps presented in Subsection 4.3.

5.3 Distributed extension of Auxo

A hidden assumption in the previous description is that the self-adaptation and online adaptivity tuning actions only take place on a single instance of the Auxo runtime infrastructure. However, in real large-scale systems, those actions usually cross physical nodes. For example, the self-adaptive server pool in our motivating scenario may involve a server pool controller and a set of servers. Since the Auxo framework is built on top of a distributed computing middleware, our approach can be easily extended to a distributed computing environment by leveraging this underlying facility. In Auxo, the cross-node adaptation logic is also specified by policy connectors. In contrast with the "local" policy connectors, those policy connectors have access to remote components, refer to remote context events, or use the Auxo.AAS service to manipulate a remote RSA model. To support this kind of policy connector, the Auxo runtime infrastructure has been extended to support event publishing/subscription across nodes, and the remote access of architecture models through the underlying middleware. For architecture elements that involve two or more nodes (e.g., a cross-node connector), only one "host" node maintains a comprehensive description in its RSA model. Other nodes access it by the remote access mechanism.

6 Case studies and evaluation

Auxo supports C++ language and can run on Windows, Linux, WinCE and embedded Linux. In this section, we describe an application that we have developed and implemented on the Auxo framework. We aim to illustrate (1) how to develop self-adaptive applications according to the Auxo architecture style and (2) how to tune software adaptivity when an unanticipated environment emerges.

This application is very similar to the motivating scenario presented at the beginning of this paper. Its prototype is a realistic email service in a large organization, which has been deployed for several years. The original version of this application has a fixed-size server pool to process newly arrived emails. However, we found that most of the time this pool was operating far below its capacity. Thus, we decided to port it to the Auxo framework and introduce a server pool, the size of which varies according to the system load. As shown in Figure 8, this application is made up of two parts: a set of servers processing the emails and a load balancer that not only distributes the emails but also controls the size of the pool by activating and deactivating the servers.

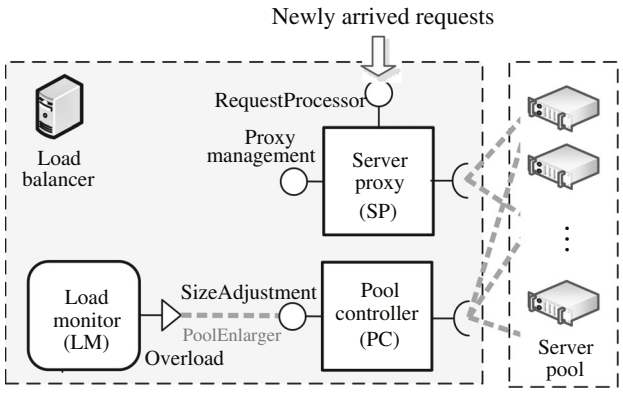

Figure 8 Initial configuration of self-adaptive server pool.

6.1 Achieving adaptivity in server pool

The architecture configuration to realize a self-adaptive server pool is shown in the gray box in Figure 8. The behavior component *SP* is responsible for encapsulating the function logic of receiving and distributing the requests to the backend servers. The "Sensing-Decision-Execution" self-adaptation loop is encapsulated as follows: (1) Sensing. The context component *LM* determines whether or not the system is overloaded by monitoring the average request response time. In our implementation, the response time is collected by an ORB interceptor [25] registered with the StarBus middleware. (2) Decision. The connector *PoolEnlarger* encapsulates the adaptation logic, namely, activating an idle server when the system is overloaded. (3) Execution. The behavior component *PC* encapsulates the concerns of the adaptation execution. It controls the size of the server pool by dynamically adding and deleting servers on demand. In addition, the general architecture adaptation execution code is encapsulated in the Auxo.AAS interface.

To show the benefits of introducing such a self-adaptive server pool, we compared its response time and operating cost with those of a fixed-size implementation. The testing workload is designed on the basis of a real-life sample and characterized by a massive increase in the traffic profile (Figure 9(a)). The experiment is conducted on a cluster composed of 11 Xeon servers (one as the load balancer and ten as the servers in the server pool) with a gigabyte Ethernet infrastructure.

In this experiment, the request response time is collected and outputted by the *LM* component. Assuming that a server has an operating cost e per minute, and the running time of the server s_i is t_i, the total operating cost of the server pool can be calculated by the following formula:

$$C_{\text{Pool}} = \sum_{s_i \in \text{Pool}} t_i e.$$

The size of the non-adaptive server pool is set to 9, and the size of the self-adaptive one ranges between 7 and 10 according to the system load and driven by the policy connector *PoolEnlarger*. As shown in Figure 9 (b) and (c), although the non-adaptive server pool has the higher operating cost ($360e$ vs. $334e$), it has a lower performance (180.51 ms vs. 144.07 ms).

6.2 Online tuning of software adaptivity

The design of the self-adaptive server pool presented earlier implies some assumptions about the running environment. For example, in the setup, we assume that no client is malicious. However, the real running environment may invalidate those assumptions. In this subsection, we illustrate how to accommodate the self-adaptive server pool to take into consideration the two unexpected cases presented at the beginning of this paper. With the support of the Auxo framework, the initial architecture configuration in Figure 8 can be dynamically modified by the administrator as follows.

Case 1: adding load prediction. We replace the context component *LM* with a newer one, which has the ability to predict the system load based on a simple windowed-mean model [14]. The adaptivity

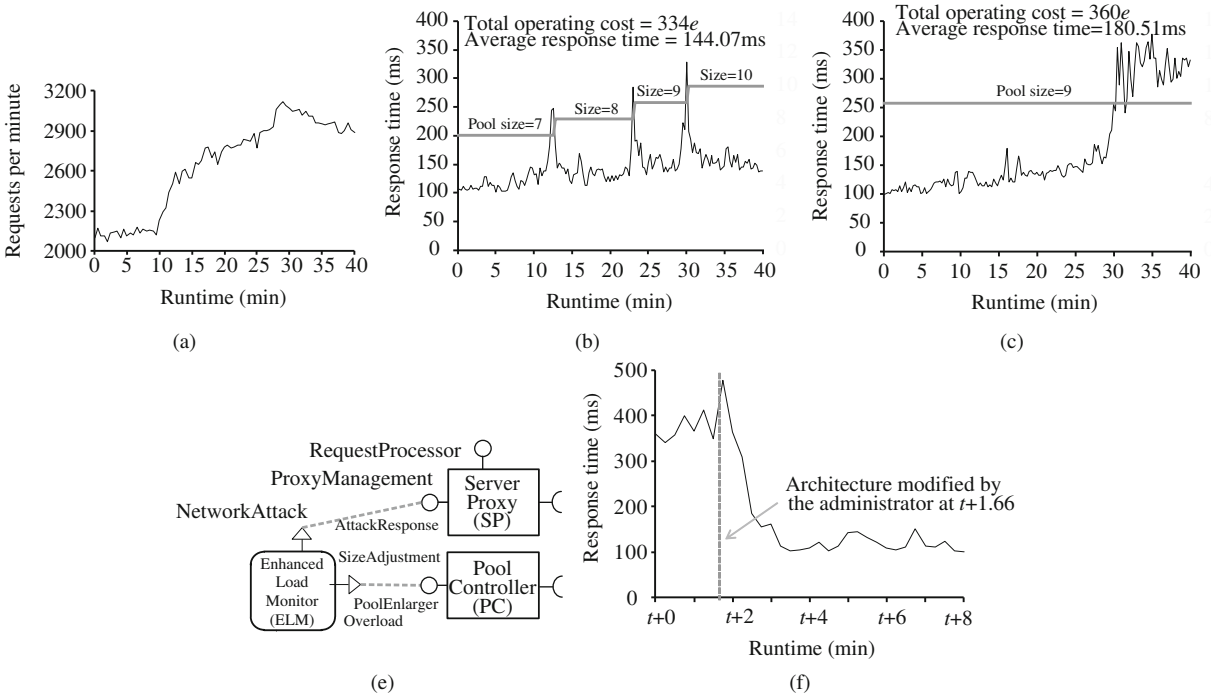

Figure 9 Experiment results in self-adaptive server pool. (a) Testing workload; (b) response time of the self-adaptive server pool; (c) response time of the non-adaptive server pool; (d) modified software architecture in Case 2; (e) response time before and after the modification.

tuning specification to achieve this goal has been presented in Figure 7.

Case 2: reacting to attacks discriminatively. We simulate a Denial-of-Service (DoS) attack scenario, i.e., a client tries to saturate the service with continuous requests. As shown in Figure 9(d), in order to deal with this unexpected attack, the context component *LM* is replaced with *ELM*, which can detect the threat by counting each client's request frequency and output a context event whose value is the attacker's address. Additionally, we add a policy connector, *AttackResponse*, which encapsulates the adaptation logic that invokes *SP.ProxyManagement* to block the attacker when a *NetworkAttack* event is published.

Figure 9(e) shows the collected performance data in this experiment: Before the architecture modification, although all servers have been activated, the request response time is still increasing because of the attack; after the modification, the system is equipped with the ability to detect and block the attacker, and the response time quickly drops to the normal level. Note that the overall service of the whole system is not interrupted.

7 Related work

In recent years, self-adaptive software has been studied extensively in the context of software engineering. For a comprehensive survey on the state-of-art, we refer the reader to [6]. In this section, we aim to focus on two topics that are of utmost relevance to Auxo: the framework for architecture-based self-adaptation and the approaches to enhance runtime software adaptivity.

7.1 Framework for architecture-based self-adaptation

Architecture-based software self-adaptation makes use of runtime software architecture techniques to facilitate self-adaptation. On the one hand, the runtime architecture information in the RSA model can be used as a source for software to introspect itself in the sensing stage of the self-adaptation loop, such as the work in [12]. On the other hand, by maintaining a modifiable RSA model that is causally connected to the real system, self-adaptive software can manipulate its own architecture in the execution

stage of the self-adaptation loop. A typical example is the seminal work in [5], in which a general-purpose process for the architecture-based self-adaptation is proposed. In order to maintain a RSA model as well as its consistency with the real system, an application framework is usually necessary. K-Component [11] enables software self-adaptation based on asynchronous architectural reflection. The component container maintains a graph-based run-time RSA model and performs self-adaptation actions according to the predefined adaptation contracts specified by ACDL (Adaptation Contract Description Language). Rainbow [12] emphasizes the reusing of facilities and knowledge in the software self-adaptation process. It introduces an architecture layer in the infrastructure that maintains the RSA model and triggers the adaptation action according to the predefined strategies. MADAM [26] and its follow-up project MUSIC [27] is designed for self-adaptive software in the mobile computing environment. It maintains two kinds of RSA model, the framework model and the instance model, and dynamically maps the former to the later through selecting component variants according to changes in the environment. DiVA [28] focuses on adaptation at the architectural level and introduces a set of models (such as the requirements model and the adaptation model) at design time. At runtime, it uses a model to represent the running system at a higher level of abstraction, which is causally connected to the real system. In the OpenCOM's reflective extensions, an architecture meta-model that is casually connected with the real system is introduced [29]. Its purpose is to support the coarse-grained topological inspection, adaptation, and extension of the structure of a dynamic target system. Internetware [30] employs the RSA model and reflection mechanisms to support online self-organization and self-adaptation.

Existing projects in this field mainly focus on achieving adaptivity by realizing the self-adaptation loop predefined by the software developers. They are not concerned with the online tuning of software adaptivity. For example, the adaptation strategy repository in Rainbow is assumed to be static and unchanged after the deployment [31]. In contrast, our framework integrates support of self-adaptation and online tuning of software adaptivity together. As a result, the self-adaptive software gains the ability to cope with an environment that was not anticipated at the time of its development.

7.2 Runtime enhancement of software adaptivity

Recently, there has been a surge of interest in uncertainty in the software self-adaptation process. A set of research has been conducted at the requirements, design, and run-time levels [32,33]. In the following, we focus on existing work related to the runtime adjustment/enhancement of software adaptivity to cope with unanticipated environments. This issue has been investigated from the following two perspectives.

A common approach adopted to deal with the uncertainty in environment is enhancing the reasoning ability of self-adaptive software. Researchers are trying to enable self-adaptive software to make non-programmed adaption decisions by utility computing, fuzzy logic, reinforcement learning, and other AI-related techniques. Several projects such as Rainbow, MADAM, CAMPUS [34], and task-based adaptation [35] support the selection of adaptation actions based on utility computing, which enables them to cope with uncertainty to some extent. FUSION [36] proposes a learning-based approach to allow the automatic online fine tuning of the adaptation logic to unanticipated conditions. K-Component supports coordination among software entities by reinforcement learning. POISED [37] builds on possibility theory to assess both the positive and negative consequences of uncertainty. It makes self-adaptation decisions that result in the best range of potential behavior. In [38], an approach based on evolutionary computing to deal with uncertainty in self-adaptation is discussed. We believe that the aforementioned AI-based techniques will be the foundation for running self-adaptive software in unpredictable environments. However, limited by current AI technology, existing work based on this approach still lacks sufficient generality with regard to real-life software engineering practices. Besides, this approach only focuses on the enhancement of the decision ability in the self-adaptation loop.

Another, more practicable, approach assumes that there exists a third-party "oracle" (e.g., an operator) who knows what should be modified in the newly emerged environment and when. Thus, the major research focus is how to dynamically and efficiently enact the just-in-time modification with flexibility. This view is taken by our approach. Several related projects have paved the way for our work. ACT [25] enables the runtime improvement to CORBA-based applications by dynamically registering Object

Request Broker (ORB) interceptors. For example, a newly registered rule interceptor can enable the application to make new adaptation decisions. Chisel [39] specifies the adaptation logic by policies and supports the online upgrading of those policies. The ACCORD autonomic element has an operational port that allows a third party to inject and manage rules that guide its runtime behavior [40]. In addition, the importance of separating adaptation logic from the functional implementation of software has been widely acknowledged and many projects on self-adaptive software such as [11,28,41] encapsulate the adaptation logic as independent entities, which are called policies, rules, strategies, or something similar. They have the potential to be extended to support the dynamic upgrading of the adaptation logic.

However, the above projects only focus on separating concerns relating to the self-adaptation decision. In contrast, our approach emphasizes the clear separation of sensing, decision and execution. As a result, we can support the independent dynamic adjustment of those concerns. In addition, we note that many existing works on dynamic policy updating do not provide a systematic approach to managing change. In contrast, we provide such a means by adopting existing techniques in the field of software architecture, namely, the runtime evaluation of architecture constraints.

8 Conclusion

Developing self-adaptive software is a challenging task. This condition becomes even more challenging when the real running environment presents contextual situations that are unanticipated when the software was originally developed. In this paper, we presented an architecture-centric application framework for self-adaptive software named Auxo. Similar to existing work, it can support the development and running of self-adaptive software. Importantly, this framework can also support the online tuning of software adaptivity through the dynamic adjustment of the existing self-adaptation loop, which can be used by a third party to accommodate the running self-adaptive software to unexpected environments. We validated and evaluated our approach by designing and deploying several applications based on the Auxo framework.

Acknowledgements

This work was supported by National Natural Science Foundation of China (Grant Nos. 91118008, 61202117). The authors would like to thank all of those who contributed to the implementation of the Auxo framework, as well as those who gave valuable suggestions and comments on this paper.

References

1 Weiser M. The computer for the 21st century. Sci Amer, 1991, 265: 94–104
2 Armbrust M, Fox A, Griffith R, et al. A view of cloud computing. Commun ACM, 2010, 53: 50–58
3 Wang H M, Wu W J, Mao X J, et al. Growing construction and adaptive evolution of complex software system (in Chinese). Sci Sin Inform, 2014, 44: 743–761
4 Cheng B H C, de Lemos R, Giese H, et al. Software engineering for self-adaptive systems: a research roadmap. In: Cheng B H C, de Lemos R, Inverardi P, et al., eds. Software Engineering for Self-adaptive Systems. Berlin/Heidelberg: Springer, 2009. 1–26
5 Oreizy P, Gorlick M M, Taylor R N, et al. An architecture-based approach to self-adaptive software. IEEE Intell syst, 1999, 14: 54–62
6 Salehie M, Tahvildari L. Self-adaptive software: Landscape and research challenges. ACM Trans Auton Adapt Syst, 2009, 4: 1–42
7 Chan A T S, Chuang S N. MobiPADS: a reflective middleware for context-aware mobile computing. IEEE Trans Softw Eng, 2003, 29: 1072–1085
8 Andersson J, de Lemos R, Malek S, et al. Modeling dimensions of self-adaptive software systems. In: Cheng B H C, de Lemos R, Inverardi P, et al., eds. Software Engineering for Self-adaptive Systems. Berlin/Heidelberg: Springer, 2009. 27–47
9 Baresi L, Di Nitto E, Ghezzi C. Toward open-world software: issue and challenges. Computer, 2006, 39: 36–43
10 Lee E A. Cyber-physical systems-are computing foundations adequate? In: Proceedings of NSF Workshop on Cyber-Physical Systems: Research Motivation, Techniques and Roadmap, Austin, 2006. 1–10

11 Dowling J, Cahill V. The k-component architecture meta-model for self-adaptive software. In: Proceedings of International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Kyoto, 2001. 81–88
12 Garlan D, Cheng S W, Huang A C, et al. Rainbow: architecture-based self-adaptation with reusable infrastructure. Computer, 2004, 37: 46–54
13 Taylor R N, Medvidovic N, Dashofy E M. Software Architecture: Foundations, Theory, and Practice. New York: John Wiley & Sons, 2009
14 Dinda P A, O'Hallaron D R. Host load prediction using linear models. Cluster Comput, 2000, 3: 265–280
15 Perry D E, Wolf A L. Foundations for the study of software architecture. ACM SIGSOFT Softw Eng Notes, 1992, 17: 40–52
16 Medvidovic N, Taylor R N. A classification and comparison framework for software architecture description languages. IEEE Trans Softw Eng, 2000, 26: 70–93
17 Shaw M, Garlan D. Software architecture: perspectives on an emerging discipline. Englewood Cliffs: Prentice Hall, 1996
18 Baldauf M, Dustdar S, Rosenberg F. A survey on context-aware systems. Int J Ad Hoc Ubiquit Comput, 2007, 2: 263–277
19 Yang Z, Duddy K. CORBA: a platform for distributed object computing. SIGOPS Oper Syst Rev, 1996, 30: 4–31
20 McCarthy D, Dayal U. The architecture of an active database management system. ACM SIGMOD Record, 1989, 18: 215–224
21 Wang H M, Wang Y F, Tang Y B. StarBus+: distributed object middleware practice for Internet computing. J Comput Sci Technol, 2005, 20: 542–551
22 Ding B, Shi D X, Wang H M. Towards pervasive middleware: a CORBA3-compliant infrastructure. In: Proceedings of International Symposium on Pervasive Computing and Applications, Urumuqi, 2006. 139–144
23 Wermelinger M A. Specification of software architecture reconfiguration. Dissertation for the Doctoral Degree. Lisboa: Universidade Nova de Lisboa, 1999
24 Davis J. GME: the generic modeling environment. In: Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Anaheim, 2003. 82–83
25 Sadjadi S M, McKinley P K. ACT: an adaptive CORBA template to support unanticipated adaptation. In: Proceedings of International Conference on Distributed Computing Systems, Tokyo, 2004. 74–83
26 Floch J, Hallsteinsen S, Stav E, et al. Using architecture models for runtime adaptability. IEEE Softw, 2006, 23: 62–70
27 Rouvoy R, Barone P, Ding Y, et al. Music: middleware support for self-adaptation in ubiquitous and service-oriented environments. In: Cheng B H C, de Lemos R, Inverardi P, et al., eds. Software Engineering for Self-adaptive Systems. Berlin/Heidelberg: Springer, 2009. 164–182
28 Morin B, Barais O, Jezequel J, et al. Models@run.time to support dynamic adaptation. IEEE Comput, 2009, 42: 44–51
29 Coulson G, Blair G, Grace P, et al. A generic component model for building systems software. ACM Trans Comput Syst, 2008, 26: 1–42
30 Mei H, Huang G, Xie T. Internetware: a software paradigm for Internet computing. Computer, 2012, 45: 26–31
31 Cheng S W. Rainbow: cost-effective software architecture-based self-adaptation. Dissertation for the Doctoral Degree. Pittsburgh: Carnegie Mellon University, 2008
32 Whittle J, Sawyer P, Bencomo N, et al. Relax: incorporating uncertainty into the specification of self-adaptive systems. In: Proceedings of International Requirements Engineering Conference, Atlanta, 2009. 79–88
33 Ramirez A J, Jensen A C, Cheng B H C. A taxonomy of uncertainty for dynamically adaptive systems. In: Proceedings of International Symposium on Software Engineering for Adaptive and Self-managing Systems, Zurich, 2012. 99–108
34 Wei E J Y, Chan A T S. CAMPUS: a middleware for automated context-aware adaptation decision making at run time. Pervasive Mob Comput, 2013, 9: 35–56
35 Sousa J P, Poladian V, Garlan D, et al. Task-based adaptation for ubiquitous computing. IEEE Trans Syst Man Cybern Part C-Appl Rev, 2006, 36: 328–340
36 Elkhodary A, Esfahani N, Malek S. FUSION: a framework for engineering self-tuning self-adaptive software systems. In: Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering, Santa Fe, 2010. 7–16
37 Esfahani N, Kouroshfar E, Malek S. Taming uncertainty in self-adaptive software. In: Proceedings of ACM SIGSOFT symposium and European conference on Foundations of software engineering, New York, 2011. 234–244
38 McKinley P K, Cheng B H C, Ramirez A J, et al. Applying evolutionary computation to mitigate uncertainty in dynamically-adaptive, high-assurance middleware. J Internet Serv Appl, 2012, 3: 51–58
39 Keeney J. Completely unanticipated dynamic adaptation of software. Dissertation for the Doctoral Degree. Dublin: Trinity College, 2004
40 Liu H, Parashar M. Accord: a programming framework for autonomic applications. IEEE Trans Syst Man Cybern Part C-Appl Rev, 2006, 36: 341–352
41 Wang Q. Towards a rule model for self-adaptive software. ACM SIGSOFT Softw Eng Notes, 2005, 30: 8