Taming Software Adaptability with Architecture-Centric Framework

Bo Ding, Huaimin Wang, Dianxi Shi
School of Computer
National University of Defense Technology
Changsha, China
{dingbo, hmwang, dxshi}@nudt.edu.cn

Jiannong Cao
Department of Computing
Hong Kong Polytechnic University
Hong Kong, China
csjcao@comp.polyu.edu.hk

***Abstract*—In many cases, we would like to enhance the pre-defined adaptability of a running application, for example, to enable it to cope with a strange environment. To make such kind of runtime modifications is a challenging task. In existing engineering practices, the online policy upgrade approach just focuses on the modification of adaptation decision logic and lacks system-level means to assess the validity of an upgrade. This paper proposes a framework for adaptive software that supports the online reconfiguration of each concern in the "sensing-decision-execution" adaptation loop. To achieve this goal, our framework supports an architecture style which encapsulates adaptation concerns as software architecture elements. And then, it maintains a runtime architecture model to enable the dynamic reconfiguration of those elements as well as help to ensure the validity of a change. A third party can selectively add, remove or replace part of this model to enhance the running application's adaptability. We validated this framework by two cases extracted from real life.**

Keywords-adapitve software, software architecture, framework

I. INTRODUCTION

Consider the following scenario. A company provides an "anytime, anywhere" accessible service based on a cloud computing infrastructure [1] (e.g., Microsoft Live Mesh [2]). It adopts a server pool to process client requests. Since the operating cost is calculated by the cloud's pay-as-you-go strategy, the system has been built in the adaptability of dynamically resizing the pool according to the system load at design-time. However, two unexpected cases emerge at runtime: (*Case 1*) As there is a lag between applying for the computing resource and getting it ready in the cloud, we'd like to introduce some kind of load prediction mechanisms [3] into the adaptation process; (*Case 2*) some load peaks are found to be not from real clients but caused by a new type of network attack, and the system should be able to react discriminatively to them.

This scenario illustrates the concern of this paper, i.e., how to enhance software adaptability at runtime. Software adaptability is the extent to which software adapts to change in its environment [4]. By its enhancement, software can be improved to accommodate situations unexpected during the original development. There are many causes for this action, especially in pervasive computing settings, such as the tight coupling to the physical space, the openness of the computing world, and the cognitive limitations of the designers, etc.

Generally speaking, software adaptation can be achieved by detecting context changes (*sensing*), making appropriate decisions (*decision*), and acting accordingly (*execution*) [5, 6]. To enhance software adaptability, the original "sensing-decision-execution" adaptation loop should be modified. New code has to be introduced, for example, to detect the emerging context or upgrade the out-of-date adaptation logic (i.e., code to guide the adaptation decision [7]). For systems in which continuous availability is critical, this modification should be executed in an on-the-fly style, as in the aforementioned scenario.

An existing engineering approach to achieve this goal is the online policy upgrade [8]. In this method, the adaptation logic in software is separated as policies (or rules, strategies, etc.), and then a third party can upgrade them if necessary. However, in practice we may need to modify other parts of the adaptation loop instead of just the adaptation logic. For example, we need add a sensing means to detect the attack in Case 2. Besides, ad hoc policy upgrades without change management may break the application's integrity [9].

Software architecture can lay a foundation for systematic runtime software changes [9]. In this paper, we propose an architecture-centric framework for adaptive software named Auxo. With Auxo, all concerns in the adaptation loop (i.e., sensing, decision, and execution) can be reconfigured dynamically and independently. Besides, Auxo provides an architecture-level means to assess the validity of a change.

Our work centers on two concepts: software architecture style [10] and the runtime software architecture model [9]. *At the development time*, by introducing a dedicated architecture style, different adaptation concerns are separated into different architecture elements. *At runtime*, Auxo maintains an explicit architecture model that reflects the real running system. A third party can selectively modify the architecture elements in this model to enhance software adaptability, e.g., adding a component to detect new contexts or replacing a connector to upgrade a piece of adaptation logic. The real-time and global view provided by this model is also the foundation to evaluate architecture constraints, which can be used to guard against inappropriate architecture changes.

The rest of this paper is organized as follows. Section II presents the background and related work. Section III gives an overview of our methodology. Section IV describes some high-lighted implementation details. Section V presents two Auxo-based applications and illustrates how to enhance their adaptability at runtime.

II. BACKGROUND AND RELATED WORK

This section firstly gives a brief introduction to the foundation of our work, the runtime software architecture

978-1-4244-5328-3/09/$25.00 ©2009 IEEE

model. And then, we review the existing work in two relevant topics: architecture-centric frameworks for adaptive software, and enhancing software adaptability at runtime.

A. Runtime Software Architecture Model

Software architecture model represents software as a network of components bound together by connecters [11]. It abstracts away lines-of-codes and focuses on the overall system structure. Traditional software architecture models described by Architecture Description Languages (ADLs) are just design-time artifacts that help the designers profile and reason about the system in-the-large [12]. However, being aware that such a "big picture" can also contribute to software dynamic changes, researchers suggested the idea of maintaining an explicit and modifiable software architecture model at runtime [9]. It is causally connected with the real running system, i.e., any changes in this model will be mapped into the running system and vice versa.

To be specific, such a runtime model has the following immediate benefits: (1) providing a real-time and global perspective of the system, which can facilitate the planning and execution of software changes, (2) exposing important system-level properties and constraints, which helps to assess the validity of a change, and (3) making the change of a system external to its functional implementation details.

B. Architecture-centric Framework for Adaptive Software

Architecture-centric adaptation adapts software with the aid of a runtime software architecture model [13]. In order to maintain this model as well as its consistency with the running system, an application framework is usually necessary. A seminal work in this field is [14], which presents a conceptual model for architecture-centric adaptation and outlines a corresponding infrastructure. K-Component [7] maintains a graph-based architecture model and introduces ACDL (Adaptation Contract Description Language) to specify adaptation logic. Rainbow [15] emphasizes the reuse of infrastructure in the architecture-centric adaptation process. MADAM [16] uses a runtime architecture model to support the adaptation in mobile computing applications.

The above projects mainly focus on the construction and runtime support of software with predefined adaptability. For example, the adaptation strategy repertoire in Rainbow is assumed to be static and not changed after the deployment [17]. In contrast, Auxo uses the architecture model not only to realize the predefined adaptation actions but also to support a third party to modify the adaptation loop and enhance software adaptability at runtime. This is achieved by encapsulating sensing means and adaptation logic as general components and connectors.

C. Enhancing Software Adaptability at Runtime

This issue has been studied from two perspectives. The first one utilizes AI-based approaches such as reinforcement learning or genetic programming to enable software make non-programmed adaptation decisions [5, 14, 18]. However, limited by the state of art of AI techniques, those approaches still lack sufficient generality in engineering practices. The second view, which is more practicable and taken by this paper, assumes a third-party "oracle" (e.g., other software entities or an administrator) knows what should be modified in the adaptation loop. Therefore, the research focus is how to dynamically and efficiently realize the modification.

Several related projects based on the second view have paved the way for our work. ACT [19] 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 [8] specifies adaptation logic by policies and supports the online policy upgrade. The ACCORD autonomic element has an operational port which allows a third party to inject and mange rules that guides its runtime behavior [20]. Besides, many projects such as [7, 21] separate adaptation logic from functional implementations, which have the potential to be extended to support the online upgrade of adaptation logic.

In contrast with those works, Auxo enhances software adaptability by manipulating the runtime architecture model. The benefit of such a model has been enumerated earlier. Besides, we emphasize not only the upgrade of adaptation logic but also the reconfiguration of sensing means. This is because: (1) Adaptation is usually directly driven by the environment instead of software internal states in pervasive computing and (2) we cannot count on an "all-embracing" context service that can deal with all strange environments.

III. AUXO METHODOLOGY

This section gives an overview of the methodology to tame software adaptability in Auxo. It consists of two parts:

- According to the *Auxo architecture style*, the developer of Auxo applications encapsulates different concerns in the adaptation loop into different software architecture elements.
- When the application is running, the *Auxo runtime infrastructure* maintains an architecture model that is causally connected with the running application. By adding, removing or replacing the elements in this model, a third party can modify the adaptation loop to enhance the application's adaptability.

In the above process, we use the architecture constraint and its real-time evaluation to guard against inappropriate architecture changes.

A. Auxo Archtiecture Style

According to [10], 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.

As mentioned before, to adjust software adaptability, the "sensing-decision-execution" loop should be modified, and the change scope may be limited to only part of this loop. For example, in Case 1 of the motivated scenario, we can enable the load prediction by adding an extra load analysis process into the sensing stage; in Case 2, if the system has been built in the ability to block certain requests, we only need to add the means to detect the network threat and the corresponding adaptation logic. Based on those observations,

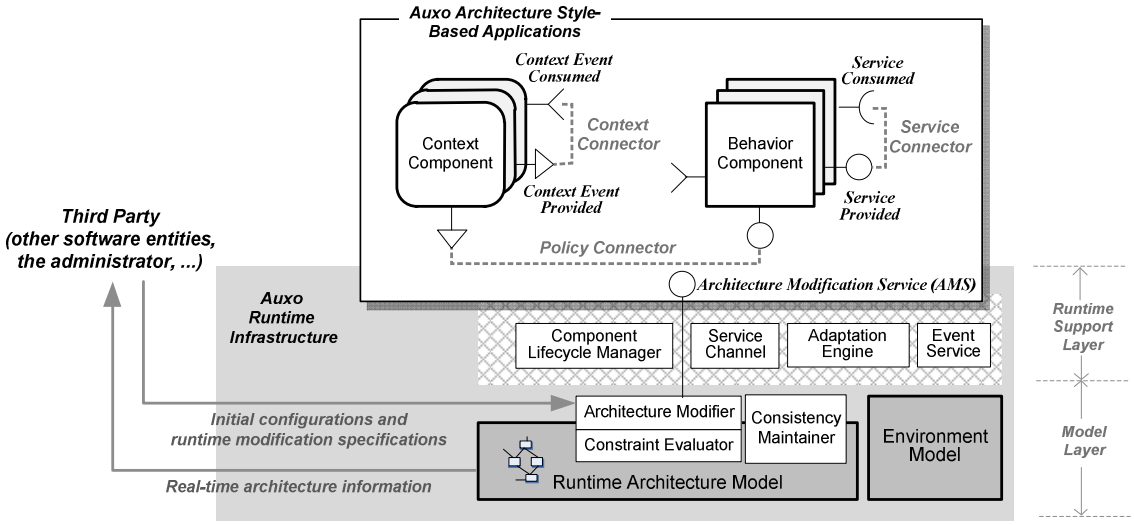


Figure 1. An Overview of Auxo framework

Auxo introduces an architecture style that supports the "sensing-decision-execution" loop and separates those three concerns as independent architecture elements.

1) Component and connector types. Auxo defines the following component and connector types, as depicted in the upper part of Figure 1.

- *Context components* encapsulate the sensing concern, which monitor the concerned environment changes and output them as context events. For example, a context component may be a software encapsulation of a physical sensor.
- *Behavior Components* encapsulate the execution concern and provide interface-based services.
- *Policy Connectors* encapsulate the decision concern, which bind context event providers and service providers together in the form of "when to do what". A service provider may be a behavior component or the Auxo framework itself (i.e., the Architecture Modification Service in Figure 1).
- *Service Connectors* bind services providers and consumers together.
- *Context Connectors* bind context event providers and consumers together, which make component-based context aggregation [22] possible.

2) Constraints. Auxo uses constraints to guard against inappropriate architecture changes and ensure the running application's integrity. Any change that violates a constraint will be rejected by the Auxo runtime infrastructure. There are two types of constraints: *default constraints* and *user-defined 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 motivated scenario at any time. More details can be found in Section IV.A

B. Auxo Runtime Infrastructure

The Auxo runtime infrastructure is designed to support the running as well as the online modification of Auxo architecture style-based applications. As shown in the lower part of Figure 1, the runtime infrastructure consists of two layers: the *runtime support layer* and the *model layer*.

The runtime support layer provides necessary supports for the running of Auxo components and connectors: the Service Channel reifies service connectors by dynamically locating the targets of service requests and transferring them; the Event Service reifies context connectors by providing the event publishing/subscription functions; the Adaptation Engine reifies policy connectors and drives adaptation actions; the Component Lifecycle Manager controls the component life-cycle, such as instantiation and deactivation.

The model layer reflects the environment and the application running on the framework. Its basic function is to provide the necessary information for the Adaptation Engine to "understand" the current context. The *environment model* is an up-to-date mirror image of the external environment, which is maintained by monitoring the context values carried by all context events; *the runtime architecture model* reflects the architecture of the running application, including components, connectors and their important states.

The runtime architecture model is the real focus of this paper. In addition to representing the application at the meta-level, it plays key roles in the online architecture modification: (1) It is causally connected with the running system and acts as an entry to manipulate the architecture; (2) it lays a foundation for the constraint evaluation; (3) it helps to locate the indirectly affected architecture elements. Details can be found in Section IV.C and IV.D.

IV. AUXO IMPLEMENTATION

The Auxo framework currently supports C++ language and has been tested on x86 PCs with Windows and Linux, Windows Mobile-based devices and iMote2 [23] sensors with ARM-Linux. It is realized based on our previous work, a distributed computing middleware named StarBus [24].

This section highlights certain implementation details of Auxo, including a brief introduction to a style-specific ADL,

the runtime realization of the "sensing-decision-execution" adaptation loop, the architecture modification protocol which can reconfigure the adaptation loop, and the consistency maintenance in the modification process.

A. Reifying Auxo Architecture Style with AuxoADL

In Auxo implementation, a dedicated ADL, AuxoADL, is introduced to define Auxo components, connectors as well as the overall structure of an application. It plays key roles in the development of Auxo applications.

1) Defining Components An Auxo component is strictly defined by the services/context events it provides and consumes. The syntax to define components in AuxoADL is an enhancement of OMG's IDL2 [25]. The IDL2 keyword *interface* is used to define services. Other major extensions include the keywords *component*, *context*, *uses*, *supports, state and multiple,* etc.

Figure 2.a is an example of the component definition. Its primary purpose is to make the meta-information of a component explicit, thus laying the foundation of future architecture manipulations. Besides, the definition is also the basis of the component development: An AuxoADL to C++ compiler can map the definition into a component skeleton, and then the developers can fill in the business logic.

2) Defining Configurations An AuxoADL configuration is an application's "blueprint". It is composed of component instances, connectors and constraints. An example is shown in Figure 2.b, whose background is the case in Section V.A. Similar to the work in ABC/ADL [26] and OpenCom [27], such a configuration can be directly instantiated into a running application by the runtime infrastructure.

Connectors are defined directly in the configuration. Service and context connectors are defined by the roles in the binding relationship. In contrast, the policy connector is more complex. It is defined in the "Event-Condition-Action" (ECA) [28] form. Those three parts separately specify the context event that triggers the adaptation, the conditions on the environment and the application internal states that should be satisfied before the adaptation, and the concrete actions to realize the adaptation. The syntax to define Condition and Action is partly based on CORBAScript [29]. We hide unnecessary features and enhance it to access Auxo components, the environment model and the runtime architecture model.

Developers can also specify invariants on the architecture model, namely, the user-defined constraints mentioned earlier. We provide a set of functions to facilitate the model check. Several examples are shown in Table I.

TABLE I. SAMPLE FUNCTIONS TO CHECK ARCHITECTURE MODEL

Function	Usage
connected (ci1, ci2)	True if component instance *ci1* and *ci2* is directly connected by a connector.
instance_count(c)	Counting the instance of component *c*.
connector_count (ci.i)	Counting the connectors attached to a specified service/context event port.

B. Runtime Adaptation Process

Although the implementation of the predefined adaptation is not the focus of this paper, we have to explain it before we dig into its online reconfiguration. We illustrate it with the policy connector *PoolEnlarger* in Figure 2.b.

The sketched runtime interaction is shown in Figure 3. When the application is instantiated according to its AuxoADL descriptions, the Adaptation Engine reads in the policy connector. At runtime, the context component *LM* can initiate an adaptation process by publishing the context event *Overload* with the value *true* (#1). The Environment Model gets this value by monitoring all context events (#2). The Adaptation Engine is activated by this context event (#3), and then judges the Condition of the *PoolEnlarger* by querying the context values from the environment model (#4) and the component states (#5). If the condition is satisfied, it executes the script in the Action part, i.e., activating an idle

a)AdptiveLoadBalancer.cdl

```
interface SizeAdjustment{...};//Service defined in IDL2
interface RequestProcessor {...};
interface ServerActivator {...};
...
component PoolController {  //Defining a behavior component
      provides SizeAdjustment sa;  //Service provided
      uses multiple ServerActivator sat;  //Service consumed
      state long ActivatedServerCount, TotalServerCount; // States
      ...
};
component ServerProxy {  //Defining a behavior component
      provides RequestProcessor pr;
      provides ProxyManagement pm;
      uses multiple RequestProcessor pr_to_server;
      ...
};
component LoadMonitor{ //Defining a context component
      provides context bool Overload; //Context event  provided
      ...;
};
...
```

b) AdptiveLoadBalancer.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.ActivatedServerCount<PC.TotalServerCount));
      action {
         server=PC.sa.AddServer();
         Auxo.AMS.AddServiceConnector(SP.pr_to_server, server.pr);
       };
  ...
  constraint UniquePoolController //ensuring the uniqueness of PC
     instance_count("PoolController")==1;
  constraint AtLeastTwoServers //ensuring at least two activated servers
     connector_count(SP.pr_to_server)>=2;
     ...
};
```

Figure 2. AuxoADL Code Fragments in Adaptive Server Pool

server (#6) and then invoking the Auxo.AMS service to connect this server to the server proxy (#7).

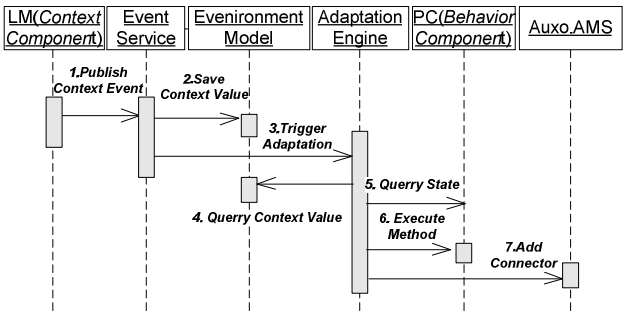


Figure 3. Sketched Interaction in Adaptation Process

C. Architecture Online Modification Protocol

In Auxo, an online architecture modification action may be triggered by a policy connector (e.g., #7 in Figure 3) or a third-party who would like to adjust software adaptability.

To allow a third party to specify the desired architecture modification at runtime, the Action clause of the policy connector can be independently used as an Architecture Modification Language [30]. For example, the following code replaces the context component *LM* with a newer one which has the ability of load prediction.

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

The Architecture Modifier in the runtime infrastructure can accept such online modification specifications. Then, the architecture will be modified by a three-phase protocol:

- *Phase 1: Modifying the runtime architecture model*. The Architecture Modifier builds a checkpoint of the runtime architecture model firstly. Then it modifies this model according to the specification it received. In this process, the indirectly affected elements are also located. In our example, the connectors attached to LM are modified according to *PortMapping*.
- *Phase 2: Evaluating architecture constraints.* The Constraint Evaluator evaluates all default and user-defined constraints. If any constraint is violated (i.e., not true), the modifications in phase 1 will be rollback and an error message will be generated.
- *Phase 3: Enacting changes to the real system*. The changes in the architecture model are mapped into the running system by the Consistency Maintainer. More details can be found in the next subsection.

D. Consistency Maintenance in Online Modification

At runtime, Auxo maintains two kinds of architecture consistency [14]: *1) Model-to-Implementation Consistency*, i.e, the consistency between the architecture model and the running system, is realized by reflection [7, 31] – the runtime support layer acts as a bidirectional bridge between the model and the system. *2) State Consistency,* i.e., the consistency of system states before and after a modification action, is partly ensured by state serialization/deserialization and the atomicity of each modification action.

Take the component replacement for an example. Phase 3 of the protocol in Section IV.C is unfolded as follows:

1) The runtime support layer is notified with the changes in the architecture model;

2) The Service Channel blocks all requests and the Event Service caches all events to the old component;

3) The Component Lifecycle Manager gets the package of the new component from the repository, instantiates it and waits for the old component to finish its current requests (if any). After that, the Component Lifecycle Manager invokes the state serializing method of the old component and the state deserializing method of the new one. Those methods are assumed to be realized by the component developers.

4) The Service Channel updates the affected service binding relationships, the Event Service updates the affected publishing/subscribing relationships, and the Adaptation Engine reloads the affected policy connectors.

5) The Component Lifecycle Manager destructs the old component and activates the new one.

6) The Service Channel and the Event Service dispatch the cached requests and context events to the new component.

The above actions are coordinated by the Consistency Maintainer, and their implementations are partly based on the existing mechanisms in StarBus, the bottom middleware of Auxo. For example, all service invocation requests are transparently serialized by StarBus, which enables the request blocking and caching in step 2.

V. CASE STUDIES

We realized two experimental applications extracted from real life based on the Auxo framework: the Adaptive Server Pool and the Smart Fire Alarm. This Section presents their designs and illustrates how to enhance their adaptability at runtime.

A. Adaptive Server Pool

This application is similar to the system presented at the beginning of this paper, except that we do not realize it on a real cloud computing infrastructure. As shown in Figure 4, this application is made up of three parts: a set of clients sending requests, a server pool processing the requests, and a load balancer which not only distributes requests but also controls the pool size by activating/deactivating servers.

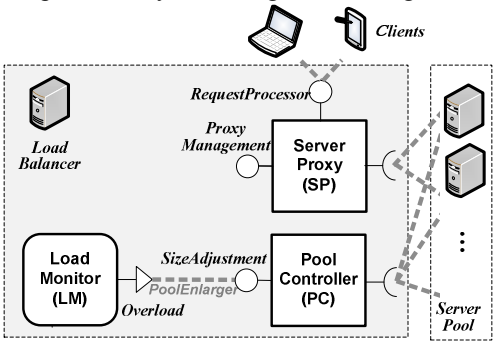


Figure 4. An Overview of Adaptive Server Pool

Take the process of adaptively enlarging the server pool as an example. The initial configuration to achieve this goal is shown in the gray box in Figure 4. According to the Auxo architecture style, the adaptation concerns are encapsulated as follows in the development time:

- *Sensing*. The context component *LM* monitors the request response time to determine whether or not the system is overloaded. In our implementation, the response time is collected by the ORB interceptor [19] registered in the bottom middleware.
- *Decision*. The connector *PoolEnlarger* encapsulates the adaptation logic, i.e., activating an idle server in the pool when the system is overloaded.
- *Execution*. This concern is encapsulated as follows: the behavior component *SP* distributes the requests to the backend servers; the behavior component *PC* activates and deactivates servers on demand; the *Auxo.AMS* service provided by the Auxo runtime infrastructure can access and manipulate the runtime architecture model (cf. Figure 2).

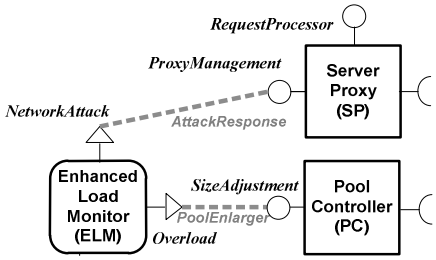
The AuxoADL-based descriptions of the above elements have been shown in Figure 2. At runtime, the adaptability defined by those elements can be enhanced as follows:

(Case 1: Adding Load Prediction) We replace the component LM with a newer one which has the ability to predict the system load based on a simple windowed-mean model [3]. The online modification specification to achieve this goal has been presented in Section IV.C.

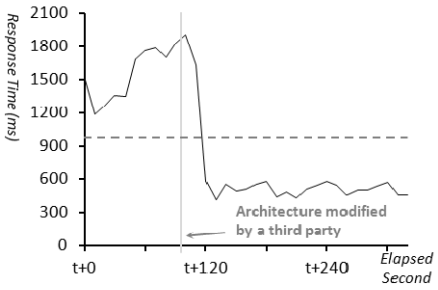
(Case 2: Reacting attacks discriminatively) We simulate a simple Denial-of-Service (DoS) attack scenario, i.e., a client tries to saturate the service with continuous requests. As shown in Figure 5.a, the context component *LM* is replaced with *ELM* that can detect DoS attacks by counting the request frequency and output a context event whose value is the attacker's address. At the same time, we add a policy connector, *AttackResponse*, which encapsulates the adaptation logic whose content is "invoking *SP* to block the attacker when a *NetworkAttack* context event is published".

Figure 5.b 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 has get the ability to detect and block the attacker, and the response time dropped to the normal level shortly.

Note that the component *SP* is not affected since it already has the functionality of blocking certain users, and the service of the whole system is not interrupted.



a) Modified Software Architecture Model



b) Request Response Time

Figure 5. Adding the ability to detect and block new attack

B. Smart Fire Alarm

In this application, we simulate the fire alarm process in a smart room. The system aggregates the smoke density, the temperature and the humidity in the room to determine the fire probability. If it is beyond a specific threshold, an alarm is triggered and a warning message is popped on the host's PDA. The initial architecture configuration as well as a key policy connector is shown in Figure 6.

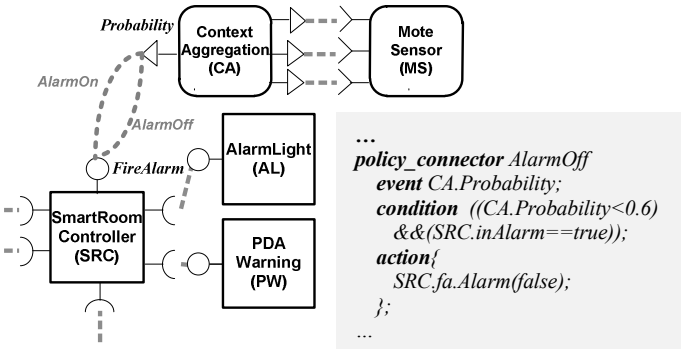


Figure 6. Initial Architecture Configuration of Smart Fire Alarm

Consider the following scenario in which we have to adjust its adaptability. The room is rented to be used as a kitchen. To avoid the frequent false alarms in the cooking process, we would like to decrease the sensitivity of this system to a more appropriate level. Since the adaptation logic has been encapsulated as policy connecters, we can achieve this goal by simply upgrading them. In particular, the Condition part of the connectors should be modified.

The participant devices in this case include desktop computers, PDAs and iMote2 sensors. Part of the fire alarm devices (e.g, the alarm light) is realized by simulation. Figure 7 shows some running snapshots of this application.

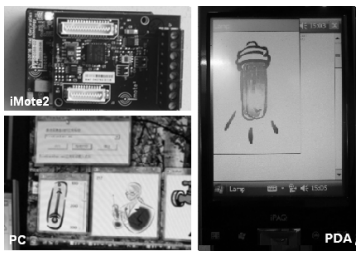


Figure 7. Running Snapshots of Smart Fire Alarm

VI. CONCLUSSION

Adaptive software may be required to accommodate situations not anticipated during the original development. At this moment, we have to enhance its predefined adaptability by introducing new code into the "sensing-decision-execution" adaptation loop. In this paper, we proposed Auxo, an architecture-centric framework for adaptive software, which supports the online reconfiguration of each concern in the adaptation loop. And we use architecture constraints to guard against inappropriate architecture changes in this process. Our future work includes the decentralized coordination of adaptation actions and the formal foundation of AuxoADL.

ACKNOWLEDGMENT

The authors would like to thank all those who contributed to the implementation of the Auxo framework and the reviewers for their valuable comments. This work is partially supported by National Science Foundation of China (no. 90818028), National 973 program of China (no. 2005CB321800), the Outstanding Youth Grant of NSF of China (no. 60625203) and the "Core Electronic Devices, High-End General Chip and Fundamental Software" project (no. 2009ZX01042-001).

REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph and R. H. Katz, et al., "Above the Clouds: A Berkeley View of Cloud Computing," EECS Department, University of California, EECS-2009-28, 2009

[2] Microsoft, "Live Mesh Beta," http://www.mesh.com

[3] P. A. Dinda and D. R. O'Hallaron, "Host Load Prediction Using Linear models," Cluster Computing, vol. 3, 2000, pp. 265-280.

[4] N. Subramanian and L. Chung, "Metrics for software adaptability," in Proceedings of Software Quality Management Conference, 2001, pp. 95-108.

[5] M. Salehie and L. Tahvildari, "Self-Adaptive Software: Landscape and Research Challenges," ACM Transactions on Autonomous and Adaptive Systems, vol. 4, 2009, pp. 1-42.

[6] J. Andersson, R. de Lemos and S. Malek, and D. Weyns, "Modeling Dimensions of Self-Adaptive Software Systems," , 2009, pp. 27-47.

[7] J. Dowling and V. Cahill, "The K-Component Architecture Meta-model for Self-Adaptive Software," in Proceedings of International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, 2001, pp. 81-88.

[8] J. Keeney, "Completely Unanticipated Dynamic Adaptation of Software," Phd. Thesis: Department of Computer Science, Trinity College, 2004.

[9] P. Oreizy, N. Medvidovic and R. N. Taylor, "Architecture-based Runtime Software Evolution," in Proceedings of International Conference on Software Engineering, 1998, pp. 177-186.

[10] M. Shaw and D. Garlan, Software Architecture: Perspectives on an Emerging Discipline. New Jersey: Prentice Hall, 1996.

[11] D. E. Perry and A. L. Wolf, "Foundations for the Study of Software Architecture," ACM SIGSOFT Software Engineering Notes, vol. 17, 1992, pp. 40-52.

[12] N. Medvidovic and R. N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," IEEE Transactions on Software Engineering, vol. 26, 2000, pp. 70-93.

[13] R. N.Taylor, N. Medvidovic and E. M. Dashofy, "Software Architecture: Foundations, Theory, and Practice," New York: John Wiley & Sons, 2008.

[14] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner and G. Johnson, et al., "An Architecture-Based Approach to Self-Adaptive Software," IEEE Intelligent Systems, vol. 14, 1999, pp. 54-62.

[15] D. Garlan, S. W. Cheng, A. C. Huang, B. Schmerl and P. Steenkiste, "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure," IEEE Computer, vol. 37, 2004, pp. 46-54.

[16] J. Floch, S. Hallsteinsen and E. Stav, F. Eliassen, K. Lund, and E. Gjorven, "Using Architecture Models for Runtime Adaptability," IEEE software, vol. 23, 2006, pp. 62-70.

[17] S. W. Cheng, "Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation," Phd. Thesis: School of Computer Science, Carnegie Mellon University, 2008.

[18] J. Dowling, "The Decentralised Coordination of Self-adaptive Components for Autonomic Distributed Systems," Phd. Thesis: University of Dublin, Trinity College, 2004.

[19] S. M. Sadjadi and P. K. McKinley, "ACT: An Adaptive CORBA Template to Support Unanticipated Adaptation," in Proceedings of the 24th International Conference on Distributed Computing Systems, 2004, pp. 74-83.

[20] H. Liu and M. Parashar, "Accord: a programming framework for autonomic applications," IEEE Transactions on Systems, Man, and Cybernetics, vol. 36, 2006, pp. 341-352.

[21] Q. Wang, "Towards a Rule Model for Self-Adaptive Software ," SIGSOFT Software Engineering Notes , vol. 30, 2005 , pp. 1-5.

[22] M. Baldauf, S. Dustdar and F. Rosenberg, "A Survey on Context-aware Systems," International Journal of Ad Hoc and Ubiquitous Computing, vol. 2, 2007, pp. 263-277.

[23] L. Nachman, J. Huang and J. Shahabdeen, R. Adler, and R. Kling, "Imote2: Serious Computation at the Edge," in Proceedings of International Wireless Communications and Mobile Computing Conference, 2008, pp. 1118-1123.

[24] B. Ding, D. Shi and H. Wang, "Towards Pervasive Middleware: A CORBA3-Compliant Infrastructure," in Proceedings of International Symposium on Pervasive Computing and Applications, 2006, pp. 139-144.

[25] Z. Yang and K. Duddy, "CORBA: A Platform for Distributed Object Computing," ACM SIGOPS Operating Systems Review, vol. 30, 1996, pp. 4-31.

[26] H. Mei, J. C. Chang and F. Q. Yang, "Software Component Composition Based on ADL and Middleware," Science in China Series F: Information Sciences, vol. 44, 2001, pp. 136-151.

[27] A. Joolia, T. Batista and G. Coulson, and A. Gomes, "Mapping ADL specifications to an efficient and reconfigurable runtime component platform," in Proceedings IEEE/IFIP Conference on Software Architecture, 2005, pp. 131-140.

[28] D. McCarthy and U. Dayal, "The Architecture of an Active Database Management System," ACM SIGMOD Record, vol. 18, 1989, pp. 215-224.

[29] OMG, "CORBA scripting language specification," , 1.1 ed, 2002.

[30] M. A. Wermelinger, "Specification of Software Architecture Reconfiguration,". Phd. Thesis: Universidade Nova de Lisboa, 1999.

[31] F. Tisato, A. Savigni and W. Cazzola, and A. Sosio, "Architectural Reflection: Realising Software Architectures via Reflective Activities," in Proceedings of International Engineering Distributed Objects Workshop, 2000, pp. 102-115.