Sindh Univ. Res. Jour. (Sci. Ser.) Vol. 48(3) 563-568 (2016)

SINDH UNIVERSITY RESEARCH JOURNAL (SCIENCE SERIES)

Policy-based Context-aware Architectural Adaptation in Pervasive Computing

G. LAGHARI++, L. D. DHOMEJA**, Y. A. MALKANI*, S. NIZAMANI***

Institute of Mathematics and Computer Science, University of Sindh, Jamshoro.

Received 17th March, 2016 and Revised 25th August 2016

Abstract: Context-aware adaptation is a rudimentary prerequisite for pervasive computing applications. It enables applications to adapt themselves in response to contextual changes in order to satisfy user needs and improve user experience with minimal or no user intervention. In existing adaptation approaches, various concerns involved in adaptation process (i.e. adaptation policies and adaptation mechanisms) are tightly coupled with the application being adapted, thus making the application difficult to build and modify at runtime. In this paper, we address this issue and propose our policy-based architecture-centric framework to context-aware adaptation.

Keywords: Runtime Adaptation, Context-awareness, Software architecture, ECA policy

1. INTRODUCTION

In 1991, Mark Weiser in his seminal paper (Weiser, 1991) introduced the notion of pervasive computing in which he predicted that computing will move beyond desktop and become ubiquitous and invisible to the user. Satyanarayanan (Satyanarayanan, 2001) defines invisibility as the "complete disappearance of pervasive computing technology from a user's consciousness" and approximates this to "minimal user distractions". To equip the capability of performing user tasks with minimal or no user distractions in pervasive computing applications requires them to adapt in changing contexts. Hence, context-aware adaptation becomes the quintessential requirement for numerous pervasive computing applications.

Context-aware adaptation is involves conferring the applications the ability to detect contextual changes, reason about the changes, and finally adapt themselves. To simplify development efforts, various approaches to context-aware adaptation exist including: (i) the approaches providing support for adaption in the programming language, (ii) the approaches providing support in the form of a middleware, and (iii) the approaches providing support via software architecture. A survey of these approaches can be found in (Aksit and Choukair, 2003; Mukhija, 2007). While adaptation approaches based on these categories have contributed towards simplifying the development efforts, our literature survey suggests that the approaches providing the support for adaptation via software architecture are more effective, for the support of adaptation being external and separate from the application being itself. Moreover, the approaches also operate at higher level of abstraction (i.e. software architecture).

The literature (Oreizy, *et al.,* 1998; Garlan, *et al.,* 2004; Georgas et al.,, 2009) suggests that existing architectural adaptation approaches have received more focus on the architectural reconfiguration specific methods, technologies, tool support, Whereas, a little attention has been paid on another important facet of context-aware adaptation—adaptation policies. The support for adaptation policies is finite in the sense that the policies have: (1) strong coupling with application code, and (2) no ability to dynamically modify them. Thus making applications difficult to build and modify at runtime. We propose our framework and develop mechanisms and supporting infrastructure to context-aware adaptation in pervasive computing environments that addresses these limitations of architecture-based adaptation approaches.

The rest of the paper is organized as follows. Section 2 presents background on context-aware adaptation and related work. In Section 3, we describe in detail our proposed approach and the prototype implementation of our approach. The results along with discussion are presented in Section 4, and finally Section 5 concludes the paper.

2. RELATED WORK

There exists a large body of research supporting compositional adaptation, also called as reconfiguration. These approaches can be classified as: (i) the approaches

++Corresponding Author: gulsher.laghari@usindh.edu.pk

* Institute of Mathematics and Computer Science, University of Sindh, Jamshoro, Pakistan

** Institute of Information and Communication Technology, University of Sindh, Jamshoro, Pakistan

*** Department of Computer Science, Sindh University Campus Mirpurkhas, Pakistan

Note: Part of this paper comes from M.Phil. thesis titled "Policy-based Context-aware Architectural Adaptation in Pervasive Computing" by the first author.

providing support for adaption in the programming language, (ii) the approaches providing support in the form of a middleware, and (iii) the approaches providing support via software architecture.

There are many programming languages providing the support for compositional adaptation such as Context (Lincke, *et al.,* 2011). There are yet other programming languages that extend Java programming language to support compositional adaptation. These languages include Open Java, R-Java, Handi-Wrap, Adaptive Java, ContextJ, and ContextAJ. An exhaustive survey of these languages is provided by (Appeltauer, et al.,, 2009). The support for compositional adaptation is provided as features of the language per se. On one hand, these adaptation mechanisms are often highly specific to the applications themselves. On the other hand, they are tightly bound to the source code. Moreover, these approaches lack support for adaptation policies. The adaptation policies have the strong coupling with the application source code and no ability to dynamically modify them.

Middleware / runtime systems (Dhomeja, 2011; Mukhija, 2007) allow as another replacement where the support for adaptation is separate and external to the application. A detailed survey these approaches is provided by (Mukhija, 2007). These solutions provide APIs to address adaptation concerns. Indeed APIs contribute to reducing the complexity in development, yet the APIs are low-level abstractions, which entail a fair amount of system knowledge to code the applications.

Other approaches to adaptation are based on software architecture. The architecture of a software system is configuration of software components and connectors (Taylor, *et al.,* 2009). A software component encapsulates system's functionality and a software connector regulates interactions amongst components. The adaptation support at software architectural level offers great flexibility to reconfigure software systems as components are arranged in a loosely coupled manner. In addition, the approaches operate at very high level of abstraction—software architecture. Hence, the development focus pertains to the system structure (the whole) than program statements (the parts). Above discussion concludes that architecture-based adaptation is more flexible. We build our policy-based solution to context-aware adaptation on architectural adaptation.

Oreizy et al. (Oreizy, *et al.,* 1998) presented an architecture-based approach by exploiting the C2-style to adaptation. A prominent feature was a central model of software architecture for adaptation. Being a seminal work of architectural adaptation, the approach lacked the support for adaptation policies and runtime management their off.

PBAAM (Georgas and Taylor, 2009) uses rule-based adaptation policies to specify and manage the policies. These policies are both decoupled from the application and managed independently. Adaptation policy specifications are expressed in xADL. As xADL uses XML, it is verbose. Thus, involving more lines of code and making it difficult to read and debug.

Rainbow system (Garlan, *et al.,* 2004), is yet another reusable infrastructure supporting self-adaptation which is based on software architectures. The system uses an abstract model of software architecture for detecting constraint violation of the properties of the running system. The constraint violations cause the running system to adapt by invoking repair strategies. Since adaptation mechanisms are not part of the infrastructure, while writing application code the developers are also required to deal with adaptation acting by writing adaptation operators. Hence, Rainbow lacks the support for unplanned run-time adaptation.

In view of the above discussion, we advocate that a solution with (1) the capability of dynamic modification of adaptation policies expressed in a declarative language and (2) flexible in adding or modifying the application behavior without stopping and restarting the application is needed. In this paper, we are presenting such a solution.

3. MATERIALS AND METHODS

Our Proposed Approach

We present a policy-based approach to developing and executing adaptive context-aware applications at software architecture level. Our goal is to provide a modular and flexible framework and infrastructure for development and execution of adaptive context-aware applications. Towards this goal, our approach is based on software architectures and adaptation policies by following separation of concerns principle, in which all the concerns involved in adaptive context-aware application (adaptation policies, adaptation mechanisms) are separate and external to the application being adapted.

In our framework, the application is described in terms of initial software architecture of the application by writing configuration code. This code specifies the components and connectors used in the application, and their bindings. Reusable reconfiguration infrastructure (Section 5.1) is a component of our system that deals with application execution and carries out application adaptation. The application is initialized and loaded from the description of its initial software architecture. When the application is loaded, reusable reconfiguration management component also maintains an in-memory model of software architecture of the application. The in-memory model contains references to executing components and always represents current architecture of the application. It evolves over time whenever the application is adapted in response to changing contexts. The adaptation is achieved by reconfiguring this model. The changes in model are eventually enacted in the running application.

The adaptation decision logic is defined as specification of high-level declarative Event-Condition-Action (ECA) adaptation policies. These policies subscribe to a specific context event or a set of context events and encapsulate specifications for adaptation actions expressed as architectural changes. When the context event occurs and the condition is true, an appropriate policy will be triggered and executed. This will cause reusable reconfiguration management component to make an architectural change in in-memory architectural model, which is then enacted in the running system. The ECA policies are external to the application, specified separately and independently (independently of configuration code of the application), and are dynamically managed (added to and removed from the system dynamically at any time throughout the life cycle of application). This provides a clean separation of concerns between adaptation policies and other aspects of architecture-centric adaptation (i.e. adaptation mechanisms, adaptation policies, and application being adapted are all separate and external to each other). Dynamic modifiability of adaptation policies is an essential requirement of applications running in pervasive computing environments. As user needs and preferences may change any time during the execution cycle of the application, thus, require run-time modification to the user policies and application functionality.

Prototype Implementation

We have implemented a prototype of our framework. The overall system that realizes our proposed approach is composed of three components: reusable reconfiguration infrastructure, policy infrastructure, and context monitoring service **(Fig. 1).** These components are described in subsequent sub-sections.

Reusable Reconfiguration Infrastructure

This component encapsulates adaptation mechanisms that provide the support for application adaptation at architectural level. It supports the loading, initialization of the application (from an initial description of its software architecture), and adaptation of the application. The infrastructure maintains an in-memory model of the software architecture of the application. The in-memory model is initialized from the initial specification of the application architecture. It contains reference to executing components and always represents current architecture of the application. It evolves over time whenever application is adapted in response to changing contexts. The application is adapted by reconfiguring this model. The changes in model are enacted in the running application.

The reconfiguration infrastructure has several sub-components: a parser, a configurator, a cache manager, and a remote listener. The parser reads and parses the file containing initial architecture description and generates a list of configuration commands. The parser also generates a list of configuration commands from the adaptation message sent by the remote listener.

The configurator performs two main tasks. First, when the parser component gives it a list of configuration commands, it executes those commands. The commands are, in fact, software architectural actions such as addition of components and their bindings to initialize the application. Second, the configurator carries out application adaptation by changing the in-memory architectural model, and thereby adapting the application.

The remote listener is implemented as an RMI service. This is exported by reconfiguration infrastructure. The policy system interacts with reconfiguration infrastructure to receive the adaptation message. The adaptation message comprises architectural adaptation commands expressed in action part of the policy. After receiving the adaptation message, the remote listener sends the message to the parser to generate a list of commands involved. The configurator to carry out application adaptation, then, executes the list of commands generated by the parser.

When cache support is enabled, cache manager maintains references to the components removed from the application. This helps avoid reloading the components when they are required later on, thereby significantly improving performance.

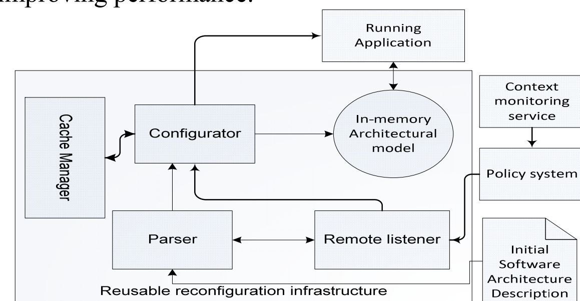

Fig. 1. System architecture of the proposed approach

Reusable policy system

We use third party system called Ponder2 (Twidle, *et al.,* 2009) which provides support for specification, enforcement, and dynamic management of adaptation policies. We choose Ponder2, as it is a lightweight, self-contained, and extensible policy system that can be used across all devices from small resource-constrained devices to complex environments. In addition, Ponder2 uses a declarative language called PonderTalk to specify polices, which provides better transparency to the developers (Dhomeja, 2011). Ponder2 ECA policies are separate units of execution that execute independently of the application and can be dynamically modified, added to, or removed from the policy system.

Context monitoring service

Currently, we have implemented simulated context widgets, which are implemented as Ponder2 managed objects that provide contextual information to policies. Contextual information is in the form of Ponder2 events

to which adaptation policies subscribe. When these events occur, the policies subscribing to them get triggered and executed.

4. RESULTS AND DISCUSSION

We have implemented and tested some hypothetical applications including smart notice board, context-aware compression server, and location-based message delivery. We demonstrate effectiveness our approach through location based message delivery system. The location based message delivery system presents the messages for a user, received from a remote source, to the user on her nearest device. For example, the user may prefer receiving the messages on smart TV whenever she is watching TV in the TV hall. Similarly, she may prefer to receive messages on the smart phone when she is in the bedroom.

This application is composed of three software components (**Fig. 2**). The MessageReceiver component receives messages from the remote source. The MessageForwarder component reads messages from MessageReceiver component and sends them to the device nearest to the user to be displayed. The third component displays the messages. In the demonstration application, we implement two components SmartTV and SmartPhone, which simulate the devices for displaying messages.

Fig. 2. Initial architecture of the application.

The first step in developing adaptive context-aware application in our framework is expression of core functionality (functional concerns) of the application at architectural level. This requires expressing an initial architecture of the application (**Fig. 3**), which is then mapped into running application. Note that in Figure 3, "mf" and "mr" are ports where service is required or provided.

Once the application is running, the adaptation policies, which are non-functional concerns, can then be associated with the running application. The demonstration application includes two adaptation policies: one for the location TV hall **(Fig. 4)** and the other for bedroom **(Fig. 5).** Both of these policies subscribe to user location context event. The TV hall policy (**Fig. 4**) subscribes to user location context event (line 2). When user location context event occurs and the location is "TV Hall" (line 3), action part of the policy is performed. The action part includes sending adaptation message (lines 4, 5 and 6) to reusable reconfiguration infrastructure, which eventually adapts the application.

In the bedroom policy, when the user is in bedroom then the display device component is replaced with SmartPhone component **(Fig. 5).** When the location context event occurs, both policies are evaluated. If the user location is bedroom, the bedroom policy is enforced and the action part of this policy (lines 4, 5 and 6) sends the adaptation message to reconfiguration infrastructure. The reconfiguration infrastructure loads the SmartPhone component, removes the SmartTV component, replaces it with SmartPhone, and binds it with MessageForwarder component **(Fig. 6).** Thereafter, all the messages received from the remote source are forwarded and displayed on SmartPhone component. The overall working mechanism of the system is shown in (**Fig. 7**).

```
1. policy := root/factory/ecapolicy create.
2. policy event: root/event/userlocationevent;
3. condition: [ :type :value | value == "TV Hall" ];
4. action: [ :type :value |
5.  config reconfig: "add component SmartTV as tv".
6.  replace component sp with tv".
7. active: true.
```

Fig. 3. Description of initial architecture of the application

```
1. add component SmartTV as tv
2. add component MessageForwarder as msgF
3. add component MessageReceiver as msgR
4. add connector Conn as con_r
5. add connector Conn as con_dvc
6. bind tv at mf to msgF at mf using con_dvc
7. bind msgR at mr to msgF at mr using con_r
8. start msgF
```

Fig. 4. TV hall policy

```
1. policy := root/factory/ecapolicy create.
2. policy event: root/event/ userlocationevent;
3. condition: [ :type :value | value == "Bedroom" ];
4. action: [ :type :value |
5. config reconfig: "add component SmartPhone as sp;
6. replace component tv with sp".
5. active: true.
```

Fig. 5. Bedroom policy

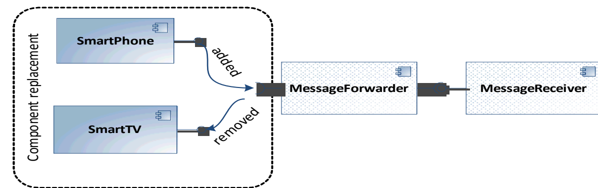

Fig. 6. Application adaptation.

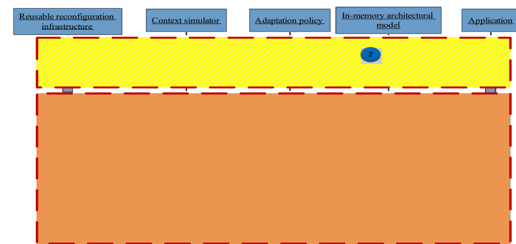

Fig. 7. Working mechanism of the proposed approach

We have demonstrated through the example scenario that our proposed approach allows development of adaptive context-aware applications from software architecture specifications and policy specification. Furthermore, the proposed approach allows separate treatment of various concerns involved in the development of adaptive context-aware applications; adaptation policies, adaptation mechanisms and application being adapted. In particular, adaptation policies are dynamically modifiable. These features allow dynamic programmability of applications and make development of adaptive context-aware applications easier.

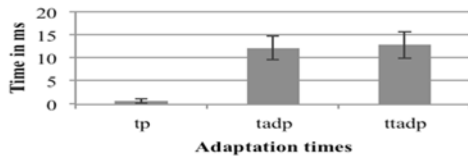

Fig. 8. Total adaptation time ($t_{tadp} = t_p + t_{adp}$) along with standard deviation when cache disabled

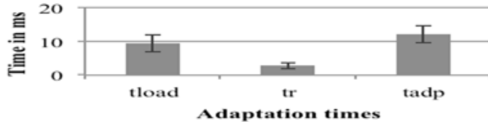

Fig. 9. Application adaptation time ($t_{adp} = t_{load} + t_r$) along with standard deviation when cache disabled

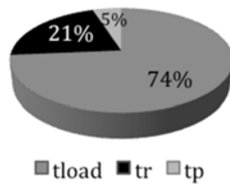

Fig. 10. Total Adaptation time ($t_{tadp} = t_p + t_{load} + t_r$) when cache disabled

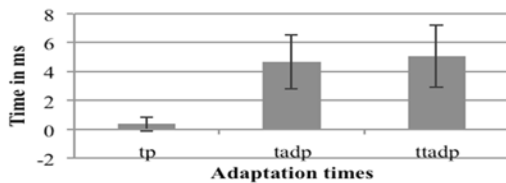

Fig. 11. Total Adaptation time ($t_{tadp} = t_p + t_{adp}$) along with standard deviation when cache enabled

Evaluation

We, now, substantiate the effectiveness of our proposed approach through performance evaluation and support for dynamic modifiability of adaptation concerns.

Performance Analysis

To study the performance of our system, we conduct tests by executing the hypothetical application (described in Section 6) and measuring total adaptation time taken by our system. Total adaptation time is measured from a point when a context is sent by the context monitor to the policy system, until the application is adapted in response to policy evaluation. This time t_{tadp} is a sum of time t_p taken by policy system (for policy enforcement) and the time t_{adp} taken by reconfiguration infrastructure to adapt the application. It can be expressed as:

$$t_{tadp} = t_p + t_{adp}$$

The application adaptation time t_{adp} taken by reconfiguration infrastructure is a sum of time t_{load} to load the new component in memory and the time t_r to reconfigure the architecture.

$$t_{adp} = t_{load} + t_r$$

The Performance tests are conducted on a Windows platform (CPU Intel Core i3-3120M 2.50 GHz, RAM 2GB, OS Windows 7 Ultimate 64-bit Service Pack 1, Java version JDK1.7.0). All the infrastructural elements of the system run on a single machine. To achieve a better measurement, the adaptation policy is triggered thirty (30) times, which responds to user location context event. The tests are conducted when the cache support is not enabled and also when cache support is enabled.

The average timings, when cache support is not enabled, are presented in (**Fig. 8 to Fig.11**). The graph in (**Fig. 8**) shows the total adaptation time t_{tadp}, the time t_p for policy enforcement, and the application adaptation time t_{adp}. The standard deviation is also shown. The average application adaptation time t_{adp}, the component load time t_{load}, and the architectural reconfiguration time t_r are shown in (**Fig. 9**). The pie chart (**Fig. 10**) shows the total adaptation time t_{tadp} split into the percentage of times taken by each: the policy enforcement t_p, the component loading t_{load}, and the architectural reconfiguration time t_r. It is also indicated in (**Fig. 10**) that the largest contribution in overall adaptation time is provided by component loading time, while the performance overhead of policies and architectural reconfiguration is minimal.

The times are shown in (**Fig. 11**) when cache support is enabled. The figure indicates that caching support improves the performance by avoiding component reloading t_{load}, thereby minimizing the total adaptation time t_{tadp}.

Dynamic modifiability of adaptation policies

In this section, we evaluate main feature of our research that our proposed approach supports dynamic modifiability of adaptation policies. Through demonstration application, we show how policies are dynamically modified without stopping the application. The demonstration application has two adaptation policies: TV hall policy and bedroom policy. We use bedroom policy and modify it to demonstrate dynamic modifiability of this policy.

Let us assume the user preference has changed. She is now interested to receive messages into her email inbox instead of smart phone when she is in bedroom. This requires modifying lines 5 and 6 of the bedroom policy (**Fig. 5**). The modified bedroom policy is shown in (**Fig. 12**).

Next step is to load the modified policy through Ponder2 shell without shutting down and restarting the application. After the policy is loaded, now in response to location context event (location = "Bedroom") the messages are forwarded to email inbox.

This demonstrates that our proposed approach supports dynamic modification of adaptation concerns without shutting down and restarting the application and thereby ensuring continuous availability of the application.

```
1. policy := root/factory/ecapolicy create.
2. policy event: root/event/ userlocationevent;
3. condition: [ :type :value | value == "Bedroom" ];
4. action: [ :type :value|
5. config reconfig: "add component EmailIbox as ei;
6 replace component tv with ei".
7. active: true.
```

Fig. 12. Modified bedroom policy

5. CONCLUSION

In this paper, we have presented an approach to context-aware adaptation in pervasive computing, which is based on software architectures and declarative ECA policies. Our approach handles all the concerns involved in adaptation process (adaptation policies, adaptation mechanisms, and an application being adapted) separately. In particular, handling of adaptation concerns as separate and dynamically modifiable ECA policies is a key feature of our approach, which is lacking in existing software-architecture based adaptation approaches. The support for dynamic modifiability of adaptation policies and separation of concerns in the proposed approach provides the ease of development and dynamic programmability of applications with minimal performance overhead of policies and architectural reconfiguration.

REFERENCES:

Aksit, M., Z. Choukair, (2003). Dynamic, adaptive and reconfigurable systems overview and prospective vision. Paper presented at the Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on.

Appeltauer, M., R Hirschfeld,. M. Haupt, J. Lincke, and M. Perscheid, (2009). A comparison of context-oriented programming languages. Paper presented at the International Workshop on Context-Oriented Programming.

Dashofy, E. M., Hoek, A. V.D.. and R. N. Taylor, (2005). A comprehensive approach for the development of modular software architecture description languages. ACM Transactions on Software Engineering and Methodology (TOSEM), 14(2), 199-245.

Dhomeja, L. D. (2011). Supporting policy-based contextual reconfiguration and adaptation in ubiquitous computing, PhD Thesis. (PhD Thesis), University of Sussex.

Garlan, D., S.W. Cheng, A. C Huang, B.. Schmerl, and P. Steenkiste, (2004). Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. Computer, 37(10), 46-54. doi:10.1109/mc.2004.175

Georgas, J. C., and R. N. Taylor, (2009). Policy-based architectural adaptation management: Robotics domain case studies Software Engineering for Self-Adaptive Systems 89-108: Springer.

Lincke, J., M. Appeltauer, B. Steinert, and R. Hirschfeld, (2011). An open implementation for context-oriented layer composition in ContextJS. Sci. Comput. Program., 76(12), 1194-1209. doi:10.1016/j.scico.2010.11.013

Mukhija, A. (2007). CASA A Framework for Dynamic Adaptive Applications. PhD thesis, University of Zurich, Switzerland.

Oreizy, P., N. Medvidovic, and R. N. Taylor, (1998). Architecture-based runtime software evolution. Paper presented at the Proceedings of the 20th international conference on Software engineering, Kyoto, Japan.

Satyanarayanan, M. (2001). Pervasive computing: vision and challenges. Personal Communications, IEEE, 8(4), 10-17. doi:citeulike-article-id:203956
doi: 10.1109/98.943998

Taylor, R. N., N. Medvidovic, and E.M. Dashofy, (2009). Software Architecture: Foundations, Theory, and Practice: Wiley Publishing.

Twidle, K., N. Dulay, E. Lupu, and M. Sloman,(2009). Ponder2: A policy system for autonomous pervasive environments. Paper presented at the Autonomic and Autonomous Systems, 2009. ICAS'09. Fifth International Conference on.

Weiser, M. (1991). The computer for the 21st century. Scientific American., 265, 94-104.
doi:10.1145/329124.32912