LEAD: A Language for Adaptable Applications

Noriki Amano and Takuo Watanabe

Graduate School of Information Science,
Japan Advanced Institute of Science and Technology,
1-1 Asahidai, Tatsunokuchi, Ishikawa, 923-12, Japan,
Tel: +81–761–51–1256
Fax: +81–761–51–1149
Email: {n-amano, takuo}@jaist.ac.jp

Abstract: This paper reports a language-based approach to constructing *adaptable applications* — application level software systems which dynamically adapt themselves to their changing runtime environments. Adaptable applications are especially indispensable to mobile distributed computing. However, such flexible applications remain more expensive to develop, maintain and port compared to their 'fixed' counterparts. Our approach is based on a language mechanism called *generic procedure* which enhances the modular development of adaptable applications. Using this mechanism, an application programmer can properly separate the code of adaptation strategies from the description of his/her application's primary subject. The mechanism is realized in our language LEAD. Our current experimental implementation supports several sorts of adaptability in mobile computing environment.

1. Introduction

Nowadays, we'd better throw our old assumptions — computing takes place in a fixed/stable context — away. Typical examples lie in mobile computing. For example, 1) the physical location and network connectivity of a mobile host varies while its user moves around (*host mobility*), 2) within a single computer, the configuration of its hardware resources changes dynamically when *Plug & Play* devices such as PC cards are attached/detached, and 3) A software object called *mobile agent* can move around various kinds of computing sites (*code mobility*). Therefore, such systems have great potential for the changes of all elements (kind of computers, hardware & software configurations, physical environments, etc) in the moving of before and after. These changes have given a strong influence on executions of applications. Toward the changes of runtime environments, it is desirable that applications can make full use of the attributes in runtime environments, and handle various constraints in them. To realize the applications, we need the adaptation at operating system (OS) level and application level. The adaptation at OS-level is that *Plug & Play* of PC Cards, the dynamic alteration of *IP address*, etc, are supported by OS. The goal of this adaptation is to realize system transparency. However, that alone is insufficient. OS cannot realize the complete transparent environments about physical environments (bandwidth, network latency, etc). In contrast, the adaptation at application-level can handle the physical environments by changing applications' behaviors. For example, a network application changes its own behaviors according to the bandwidth and network latency. Therefore the application can realize a suitable behavior for the physical environments. We call this-type applications *adaptable applications*. The goal of them is not to realize system transparency but to make full use of the attributes in runtime environments, and to handle various constraints in them. Until now, the adaptation at OS-level has many researched and proposed. These OSs are called *adaptable OSs* [1, 2]. However, the merits of the adaptation at application-level are not a few. The language LEAD that we propose aims to provide the general & effective framework which realizes adaptable applications.

The procedures of applications in LEAD are changed dynamically by LEAD runtime system according to the changes of runtime environments. Applications in LEAD are realized by changing their procedures dynamically in runtime. This method has merits as follow. 1) The control mechanism of adaptation becomes simple. 2) This method can be applied to other languages which have *procedure call*. Besides, applications in LEAD form *meta-level architecture* [4]. This architecture, promotes the readability and extensibility of adaptable applications, makes their maintenance easier, and promotes the reusability of the code of its adaptation part in them.

The remainder of this paper is organized as follows. Section 2 mentions to overview of LEAD. We explain the architecture of applications in LEAD in Section 3. In Section 4, we discuss the effectiveness of LEAD and its applications. Finally, Section 5 concludes this paper.

2. Overview of LEAD

LEAD is a language to describe runtime environments where procedures are executed, suitable procedures for the environments and mechanism of adaptation. The primary design goals of LEAD are the following three points.

- LEAD makes the realization of adaptable applications easier.
- Applications in LEAD can be extend easily.
- LEAD can evolve from applications written in other languages into adaptable applications.

These goals are achieved by the following functionalities which LEAD has.

1) LEAD has many built-in functions which obtain the state information of environment elements (e-elements) which construct runtime environments (Tab.1). The information is indispensable to change the procedures of applications dynamically.

Table 1: A part of Built-in Functions

Built-in functions	Return value
`network`	network connectivity
`memory`	free memory size
`display`	display size
`power`	power source
`battery`	battery remainder

2) LEAD has a functionality to describe a pair of the state of runtime environments and a suitable procedure of applications for the state as a module. Using LEAD, we need not describe condition branch to control behaviors of applications according to the changes of runtime environments. Moreover, the module description is completely separated from applications. Namely, we can separate the code of its adaptation part from the code of its primary subject part of applications. This separation makes the readability and extensibility of applications high, and makes their maintenance easier. It also brings the following merits. a) The modules can be easily shared and/or reused among applications. b) We can easily append adaptable behaviors to applications written in other languages.

3) We can define the high abstract runtime environments (e-elements) and their states by combining the built-in functions. The high abstraction makes the readability of programs high, and makes their description brief.

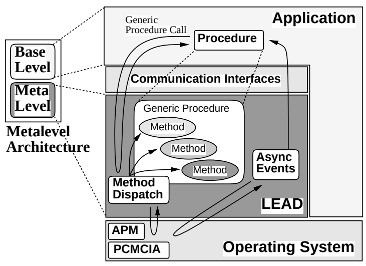
4) LEAD has sufficient descriptive power to implement the mechanism of adaptation called *adaptation strategy*. The adaptation strategy is an algorithm to change dynamically procedures of applications according to the states of runtime environments. LEAD runtime system provides the adaptation strategy as a procedure which can be customized. We can modify it, and can add new strategies which we implement to it.

The other characteristic functionalities of LEAD are *asynchronous events*, *remote execution* of procedures, etc. Asynchronous events are the functionality which informs applications the occurrence of the changes asynchronously when runtime environments have changed. Remote execution of procedures is the functionality which executes the procedures on remote hosts according to the state of runtime environments. LEAD also has the functionality of inter process communication. When we evolve from applications written in other languages into adaptable applications by using LEAD, LEAD runtime system and the applications use this functionality and communicate with each other effectively.

3. The Architecture of Applications in LEAD

Adaptable applications are constructed from applications and LEAD runtime system (LRS). LRS is a server which provides various functionalities which support dynamically adaptable behaviors for applications. An application is client which takes the services which LRS provides. Besides, LRS is *meta-level* which controls applications' procedures in runtime, and an application is *base-level* controlled by LRS. Namely, applications in LEAD form *meta-level architecture* [4] (Fig.1). This architecture completely separates the code of its adaptation part from the code of its primary subject part of applications. Since this separation promotes the readability and extensibility of applications, their maintenance becomes easier.

Figure 1: The Architecture of Applications in LEAD



The procedures of adaptable applications are separated from applications themselves, and are described in LRS (*meta-level*). We call these procedures *generic procedures*. A generic procedure is a set of *method* (Fig.2). A method is constructed from a pair of the state of runtime environments and a suitable procedure of the application for the state. Namely, a generic procedure is an interface for some methods. Each method for a generic procedure is an actual procedure of the application according to the state of runtime environments. LRS selects a suitable method for the state of runtime environments from some methods. Thus applications can easily realize adaptable behaviors. The state of runtime environments in a method is the state of environment elements (e-elements) which construct the runtime environments. The e-elements in LEAD are bound to the built-in functions (Tab.1). There are two ways to implement a method as follows. We describe, 1) the whole code of procedure, 2) only procedure's name, in a method. The above 2 is available to evolve from applications written in compiler-based language into adaptable applications. Moreover, since a method is described with the state of runtime environments where the method is selected, we need not describe condition branch for adaptation explicitly. Adaptable applications by using generic procedures and their methods have the following merits. 1) We can easily extend them by adding methods without modifying original source codes in applications. 2) We can also easily share and/or reuse generic procedures and methods among them.

Figure 2: Generic Procedures & Methods

Generic Procedure	=	(Name, Lang, App, E-list)
Name	:	Generic Procedure's name
Lang	:	Application's description language name
App	:	Application's name
E-list	:	List of e-elements
Method	**=**	**(Name, Lang, App, S-list, Loc, Proc)**
Name	:	Generic Procedure's name
Lang	:	Application's description language name
App	:	Application's name
S-list	:	List of the state of e-elements
Loc	:	Execution location
Proc	:	Application's procedure {code | name}

LRS has interfaces to communicate with applications (Tab.2). These interfaces for each language which describes applications are provided. Applications can call generic procedures in the meta-level through the interfaces.

Table 2: Primary Interface Functions

Interface functions	Use
`GenericProc`	generic procedure call
`RegistEvent`	async events entry
`EventProc`	async events' procedure entry
`GenericStrategy`	alteration of adaptation strategy

When an application does *generic procedure call* via them, First, LRS obtains the state information of runtime environments from operating system. Secondly, LRS selects a suitable method for the state. Then the code in the method (actual procedure code) is sent to the application via them, and the code is executed on the application. These consecutive processings are combined into *generic procedure call* using the interfaces. We have only to do *generic procedure call* via them to realize adaptable applications.

Adaptation strategy is the strategy of method selection which selects a suitable method for the state of runtime environments from some methods for a generic procedure. LRS has default strategies. These strategies are methods in LEAD. Adaptation strategy is implemented in LEAD as a generic procedure. Thus, we can alter the strategies dynamically according to the state of runtime environments, can modify them, and can easily append new strategies to them by adding methods. Besides, using interfaces, applications can change the strategies in runtime. The result of alteration of them reflects the method-selection necessarily. Namely, applications in LEAD can realize *reflective* [4, 7] behaviors. Behaving reflectively, applications can realize to limit their functionalities temporarily according to the state of runtime environments, and can release the limitation.

When the state of runtime environments changes, if there is a mechanism which informs applications of occurrence of the changes, some applications can be realized efficiently. LRS has this functionality as asynchronous events. This functionality is used with generic procedures. An event means the occurrence of the state change of e-element which corresponds to the event name. We define generic procedures as actions for an event. When the change occurs, LRS notifies the applications of the occurrence of the change, then the *generic procedure call* is done automatically from the applications. To use asynchronous events, applications must request events to LRS, and must define generic procedures for events in LRS via communication interfaces.

4. Discussion

We have implemented the prototype of LEAD and two examples of adaptable applications in LEAD. The examples are full-screen based text editor and resource monitor. They are described in Tcl/Tk only at first. Then we evolve from them into adaptable applications. We have make sure the effectiveness of LEAD and its applications from these experiments. If we implement the two examples in Tcl/Tk only, the problems occur as follows. 1) The applications programs become complicated, because we must describe the deep nesting of condition branch to classify various states of runtime environments, and the code of its adaptation part (adaptation code) and the code of its primary subject part (primary code) of applications are mixed. 2)

From the above 1, the readability and extensibility of the programs become low, and their maintenance becomes difficult. 3) Since Tcl/Tk has no asynchronous events such as LEAD's, the applications must poll to obtain the change of runtime environments. 4) Adaptable applications in Tcl/Tk only are not reusable. Moreover, Tcl/Tk has no built-in functions which obtain the state of runtime environments.

Using LEAD, we can solve these problems. 1, 2) Using LEAD, the adaptation code is completely separated from the primary code of the application. Thus, the readability of programs becomes high, and their maintenance becomes easier. Applications in LEAD can be also extend easily by adding methods. 3) Applications in LEAD need not poll, since LEAD has asynchronous events. 4) The generic procedures in LEAD can be shared and/or reused among applications. Actually, our examples share a generic procedure. Besides, LEAD has many built-in functions to obtain the state of runtime environments.

There are some existing works which try to support adaptable applications by using language [5, 6, 8]. Toward these linguistic approaches, LEAD has the following merits. 1) LEAD provides many built-in functions which obtain the state of runtime environments. 2) LEAD can handle the change of environment of various abstractions. 3) LEAD has asynchronous events' mechanism which informs applications of the state change of runtime environments. 4) LEAD can evolve from applications which describe in other languages into adaptable applications.

Java [3], Telescript [9], etc, support *mobile code*. The programmers of applications based on them do not consider the environment which the applications are executed. Thus, there is potentiality that their applications do not work well on the other hosts. For example, we implement the application which needs large memory, then sent it to the other hosts which have small memory, and execute it on there. In this case, it is obvious that the execution will fail. We can avoid this situation by wrapping them with LEAD.

5. Conclusion

A application in LEAD forms *meta-level architecture*. The code of its adaptation part is completely separated from the code of its primary subject part. This separation promotes the readability of application programs, and makes their maintenance easier. Besides, generic procedures can be shared and/or reused among applications, and we can easily extend applications by adding methods.

In LEAD, a method and the method-selection description form a program module. Thus, we need not describe condition branch for adaptation explicitly. Besides, Our approach is well-suited to other languages which have *procedure call*.

Acknowledgments

We are grateful to Prof. Tatsuo Nakajima and Dr. Kazuhiro Ogata at JAIST for discussions and comments. This work is supported in part by the IPA (Information technology Promotion Agency, Japan) Advanced Information Technology Program.

References

[1] Brain N Bershad, Stefan Savage, Przemyslaw Pardyak, and Emin Gun Sirer. Extensibility, safety and performance in the spin operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, pages 267–284, 1995.

[2] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, 1995.

[3] James Gosling, Bill Joy, and Guy Steele. The Java Language Specification. The Java Series. Addison-wesley publishing company, 1996.

[4] Pattie Maes. Computational reflection. Ph. D. Thesis 87-2, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1987.

[5] Hideyuki Nakashima. Organic programming for cooperative computation. In *Proceedings of the World-Wide Computing and its Applications (WWCA'97)*, pages B–1–3–1–8, March 1997.

[6] Hideaki Okamura and Yutaka Ishikawa. Object location control using meta-level programming. In *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP'94)*, pages 299–319, July 1994.

[7] Brian Cantwell Smith. Reflection and Semantics in Lisp. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'84)*, pages 23–35, 1984.

[8] Shigeru Watari and Yasuaki Honda and Mario Tokoro. Morphe: A Constraint-Based Object-Oriented Language Supporting Situated Knowledge. Technical Report SCSL-TR-92-005, Sony Computer Science Laboratory Inc, 1992.

[9] James E White. Telescript Technology – Mobile Agents, 1991. General Magic Inc, February, 1996.