# LEAD++: An Object-Oriented Reflective Language for Dynamically Adaptable Software

**Noriki Amano and Takuo Watanabe**

Graduate School of Information Science,
Japan Advanced Institute of Science and Technology
1-1 Asahidai, Tatsunokuchi, Ishikawa, 923-1292, Japan
Tel: +81–761–51–1256 Fax: +81–761–51–1149
Email: {n-amano, takuo}@jaist.ac.jp

## Abstract

A system has *dynamic adaptability* if it can adapt itself to dynamically changing runtime environments. As open-ended distributed systems and mobile computing systems have spread widely, the need for software with dynamic adaptability increases. We propose a model of software with dynamic adaptability and the object-oriented reflective language LEAD++ based on the model. The basic idea is that we introduce the mechanism which changes *procedure invocations* dynamically depending on the states of runtime environments. The mechanism forms a meta-level architecture, and realizes dynamic adaptability in reflective way. LEAD++ provides the mechanism as a language functionality. We are currently implementing a prototype of LEAD++ as a preprocessor of Java. Using LEAD++, we can systematically describe dynamically adaptable applets, mobile objects, etc.

## 1 Introduction

Nowadays open-ended distributed systems and mobile computing systems have come into wide use. When we design software for such systems, we cannot obtain accurate information of runtime environments and their dynamic changes beforehand; e.g., hardware configurations (esp. including PC card devices), software configurations, logical/physical location, etc. The changes of runtime environments have given a strong influence on the execution of the programs, which we cannot ignore.

For example, applets in Java are downloaded to any remote hosts, and then they are executed on the remote hosts. Moreover, mobile agents in Java [6], Telescript[11], etc. can move around various kinds of computing sites via the network in runtime. However, they are designed and implemented without considering the runtime environments of remote hosts/sites. Thus, they may not work well in some runtime environments of remote hosts/sites.

Then, Java virtual machine (VM) hides the states of runtime environments and their dynamic changes from the applets. Namely, Java VM provides *system transparency* for applets. However, because of the system transparency, applets cannot make full use of each runtime environment's properties; e.g., hardware & software resources, special devices, etc.

To handle these situations, it is an effective way to change those applications' behaviors depending on each runtime environment and its states. However it is nearly impossible to develop such specific applications individually. Moreover we cannot predict the runtime environments and their states beforehand. Thus the software must have the ability which can adapt itself to them in runtime. We call the ability *dynamic adaptabil-*

*ity*. Dynamically adaptable software can change its own functionalities flexibly to make full use of the runtime environments' properties.

Since we cannot obtain accurate information of runtime environments and their states beforehand, there is limitation on dynamic adaptability that we can give to the software beforehand. Thus, the mechanism which realizes dynamic adaptability of software must be constructed flexibly, and must be able to adapt itself to the states of runtime environments. We can satisfy the requirement by using reflection [7].

We propose a model of dynamically adaptable software and its description language. The model forms a meta-level architecture [7], and the mechanism for dynamic adaptability is based on procedures. It changes procedure invocations dynamically depending on the states of runtime environments, and realizes the dynamic adaptability in reflective way. To construct the model, we had designed & implemented the language LEAD [1], and we are currently working on the language LEAD++ which is a successor of LEAD. By using them, we can systematically describe the dynamically adaptable software.

The remainder of this paper is organized as follows. Section 2 introduces the basic concepts of our software model more detail. Section 3 gives an overview of the language LEAD++. Section 4 mentions related works. Finally, Section 5 concludes this paper.

# 2 The Model

## 2.1 Basic Concepts

We show the key ideas for our dynamically adaptable software (DAS) model (Fig.1) as follows.

*1*) It forms a meta-level architecture, thus the codes of adaptation mechanism are separated from the codes of primary subject domain.

*2*) It has functions to obtain the states of runtime environments (low-level library functions).

*3*) It has *environmental objects*. They express local & virtual runtime environments depending on the abstraction of procedures in DAS.

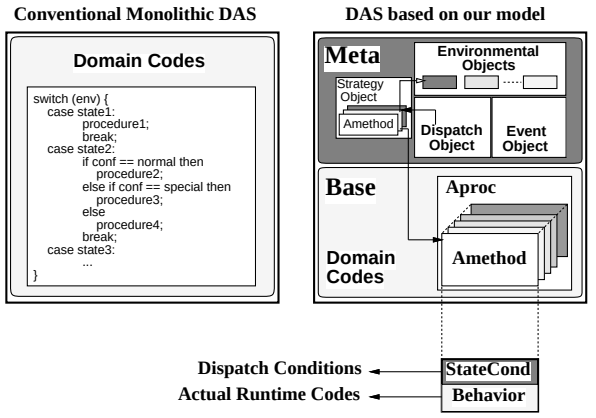


Figure 1: Comparing Dynamically Adaptable Software models.

*4*) The mechanism for dynamic adaptability is based on *adaptable procedures*. An adaptable procedure has several *adaptable methods* in it. An adaptable method consists of the conditions for states of environmental object and the actual runtime codes for the states. When an adaptable procedure is invoked, the adaptable method which satisfies its conditions is selected. Then, its actual runtime codes are executed. We call the policy which selects the adaptable method depending on the states *adaptation strategy*.

*5*) It has *dispatch objects* and *strategy objects*. The dispatch objects control adaptable procedures by using the strategy objects. A strategy object has an adaptable procedure which implements adaptation strategies as adaptable methods. Namely, it has the data which represent itself, and it can manipulate the data within itself (reflection) [7].

*6*) It has *event objects* which obtain the occurrences of states' changes in environmental objects, and then invoke procedures in DAS corresponding the states' changes. We call the occurrences *state events*, and call the invoked procedures *call-back methods*.

## 2.2 Dynamic Adaptability

The dynamic adaptability of our model is based on *states of runtime environments* (states of en-

vironmental objects), *adaptable procedures* and *adaptation strategies*. We define them here.

**Definition 1** (**states of runtime environments**)
We assume that the number of the elements (*environmental elements*) in runtime environments is finite. Let $N$ be the number of environmental elements, $Value_i$ $(i = 1, \cdots, N)$ be the set of values which express the states of environmental elements. *Rstate*, the set of states of runtime environments, is defined as:

$$Rstate = Value_1 \times \cdots \times Value_N.$$

■

*Rstate* is the set of the transitional states of runtime environments, then its element expresses a state of runtime environments at a certain moment.

**Definition 2** (**adaptable methods**)
*StateCond* is the set of conditions that select an adaptable method. Let *Behavior* be the set of actual behaviors. *Amethod*, the set of adaptable methods, is defined as:

$$Amethod = StateCond \times Behavior.$$

An element of *StateCond* is the *dispatch condition* in an adaptable method. It is defined as:

$$StateCond = Cond_1 \times \cdots \times Cond_N.$$

An element of $Cond_i$ $(i = 1, \cdots, N)$ is a condition for the $i$-th environmental element. $Cond_i$ is expressed as the set of functions which take $Value_i$, and return *true*, *false* or *undefined*.

$$Cond_i = Value_i \rightarrow \{true,\ false,\ undefined\}.$$

■

**Definition 3** (**adaptable procedures**)
*Aproc*, the set of adaptable procedures, is the set of finite sequences of adaptable methods.

$$Aproc = Amehtod^+.$$

Note that $X^+$ denotes the set of finite sequences of elements of $X$. ■

**Definition 4** (**adaptation strategies**)
*Strategy*, the set of adaptation strategies, is defined as:

$$Strategy = Rstate \times Aproc \rightarrow Amethod.$$

■

An adaptation strategy is a function which takes the state of runtime environment and an adaptable procedure, and returns the adaptable method which satisfies its dispatch condition for the state.

# 3 Overview of LEAD++

LEAD++ is an object-oriented reflective language based on our model. It provides dispatch objects which realize adaptable procedures, and we can customize the objects by using reflection flexibly & dynamically. LEAD++ also provides environmental objects, strategy objects, event objects, etc., which we mentioned in Sect.2. We call these objects *metaobjects* in LEAD++. We can also customize them using *metaobject protocol* (MOP) in LEAD++. DAS in LEAD++ forms a meta-level architecture as in Fig.2.

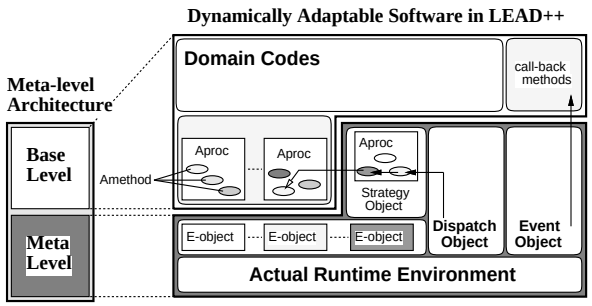


Figure 2: DAS in LEAD++

## 3.1 Environmental Objects

Environmental objects (E-objects) express virtual runtime environments. They reflect states of actual runtime environments, and their abstraction depends on the abstraction of procedures in DAS. Namely, they are highly abstract runtime

environments. Environmental objects can be created arbitrarily in DAS, and can be assigned to adaptable procedures in it. Namely, they are also local runtime environments which adaptable procedures use. An environmental object can be shared by several adaptable procedures. Environmental objects have several instance valuables and methods to obtain their states. The methods consist of the low-level library functions in LEAD++.

### 3.2 Reflective Dispatching

The internal actions of an adaptable procedure invocation are followings. The dispatch object generates a strategy object, and executes the runtime codes of default strategy. The codes select an adaptation strategy depending on the states of an environmental object (*meta-level dispatching*). Then, the codes of selected strategy are executed. They select an adaptable method in the invoked adaptable procedure depending on the states (*base-level dispatching*). Finally, the codes of the selected adaptable method are executed. The dispatching of adaptable methods is based on reflection like this.

Although the default meta-level dispatching is realized depending on the value of an instance valuable (string of a strategy name) in each environmental object, we can customize it. For example, to realize the dispatching, we can also use any other states in an environmental object; e.g., power source, free memory size, battery remainder, etc. By changing adaptation strategy depending on the states of environmental objects, we can realize the dynamic adaptability in flexible way such as the temporary restriction of DAS's functionalities and its release depending on the states. Moreover, since the states of runtime environments which we can predict beforehand are limited, the adaptable methods which we can describe beforehand are also limited. We aim to utilize the limited adaptable methods effectively by changing adaptation strategies.

### 3.3 State Events

The event object takes an environmental object, and monitors the states' changes of it. When the event object detects the states' changes (state events) in the environmental object, it invokes the call-back method in DAS corresponding the event. The abstraction of state events depends on the abstraction of environmental objects. Using the event object, we can implement DAS based on *states event driven* style efficiently & briefly.

### 3.4 Metaobject Protocol

To customize metaobjects (environmental objects, dispatch objects, strategy objects and event objects), LEAD++ provides a simple MOP. These metaobjects are defined as well as ordinary objects in LEAD++. We can customize them by defining the subclasses of their classes (overriding/overloading their methods, and/or adding new methods, etc.). Using the MOP, we can realize the following customizations.

*1*) We can realize local & virtual runtime environments and their states depending on the abstraction of each adaptable procedure by customizing environmental objects. *2*) We can exchange the strategy object for more suitable one for each adaptable procedure, and can also change the default adaptation strategy into any other by customizing dispatch objects. *3*) By adding and/or modifying the adaptable methods in the strategy objects, we can improve the dynamic adaptability. *4*) Customizing the event objects, we can define the highly abstract state events depending on the abstraction of environmental objects, and can add and/or modify the invoked call-back methods corresponding the events.

### 3.5 Current Implementation

The prototype of LEAD++ is implemented as a pre-processor of Java. It consists of a translator, class libraries and low-level library functions. The syntax of the language is based on Java (with some additional extensions). The transla-

tor translates LEAD++ source codes into pure Java source codes. The class libraries realize adaptable methods and metaobjects. Using the prototype, we can realize dynamically adaptable applets, mobile objects, etc. in Java by rewriting the existing ones a little. The current developing platform is WindowsNT/95. We had already developed the class libraries and several low-level library functions.

## 4 Related Works

There are several works on the linguistic approach for DAS: Rome [2], Clovers [10], Compositional Adaptation [4], Gaea [8], AL-1/D [9], etc. Then, as object-oriented reflective languages, there are also CLOS [5], Open C++ [3], etc. Our model and the language LEAD++ have the following features toward those works. *1)* We can realize dynamic adaptability based on procedures in application programs. Thus, we can apply our model to software in existing languages which have *procedure invocations* mechanism. *2)* We can improve the dynamic adaptability of DAS by adding and/or modifying their adaptable procedures and methods. *3)* The adaptation mechanism is based on reflection and we can customize it flexibly and dynamically.

## 5 Concluding Remarks

We have proposed a dynamically adaptable software model and its description language. Our model is simple, however it realizes the dynamic adaptability in reflective & flexible way. We plan to implement dynamically adaptable applets and mobile objects in LEAD++.

## References


[1] N. Amano and T. Watanabe. LEAD: Linguistic Approach to Dynamic Adaptability for Practical Applications. In *Proc. of IFIP Conf. on System Implementation 2000*, pp. 277–290, Chapman & Hall, 1998.

[2] B. Carre, L. Dekker, and J. Geib. Multiple and Evolutive Representation in the Rome Language. In *Proc. of TOOLS'90*, June 1990.

[3] S. Chiba and T. Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. In *Proc. of ECOOP'93, LNCS, Vol. 707, Springer-Verlag*, pp. 482–501, 1993.

[4] Y. Honda, S. Watari, E. Osawa, and M. Tokoro. Compositional Adaptation: A New Method for Constructing Software in Open-Ended Systems. Technical Report SCSL-TR-90-012, Sony CSL Inc, 1991.

[5] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, 1991.

[6] D. B. Lange, M. Oshima, and K. Kosaka. Aglets: Programming Mobile Agents in Java. In *Proc. of WWCA'97, LNCS, Vol. 1274, Springer-Verlag*, pp. 253–266, 1997.

[7] P. Maes and D. Nardi, editors. *Meta-level Architecture and Reflection.* ELSEVIER SCIENCE PUBLISHERS B.V., 1988.

[8] H. Nakashima. Organic Programming for Cooperative Computation. In *Proc. of WWCA'97, LNCS, Vol. 1274, Springer-Verlag*, pp. 133–141, 1997.

[9] H. Okamura, Y. Ishikawa, and M. Tokoro. AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework. In *Proc. of IMSA'92*, 1992.

[10] L. A. Stein and S. B. Zdonik. Clovers: The Dynamic Behavior of Types and Instances. Technical Report CS-89-42, Department of Computer Science, Brown University, 1989.

[11] J. E. White. Telescript Technology – Mobile Agents – White Paper. February 1996.