An Approach for Constructing Dynamically Adaptable Component-based Software Systems using LEAD++

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
Email: {n-amano, takuo}@jaist.ac.jp

Abstract. A software system has *dynamic adaptability* if it can adapt itself to dynamically changing runtime environments. A component-based software system with dynamic adaptability (called *dynamically adaptable component-based software system*) can change its own behaviors depending on states of runtime environments in flexible way. To realize the well-suited behaviors for the states, it changes: *1*) behaviors of each component, *2*) the structure of components composition, in runtime. We have proposed a software model with dynamic adaptability called *DAS* and its description language *LEAD++*. The DAS model is a reflective software model, and LEAD++ is an object-oriented reflective language based on Java. In this paper, we mention how to construct dynamically adaptable component-based software systems using LEAD++.

1 Introduction

Nowadays, *component technology* is being watched with keen interest as an effective technology to develop software systems. It is a technology to construct a software system by combining software parts of Plug & Play type (called *components*). In general, software systems based on such components are called *component-based software systems*. There are ActiveX[17], JavaBeans[20], etc. as the typical component technologies. As the Internet has spread widely, such component technologies come to be used in open-end distributed systems. For such systems, there are fundamental architectures such as DCOM[18], JavaRMI[21], CORBA[19], etc. The component-based software systems based on such architectures are realized by communicating among components via the network.

It is extremely difficult for software engineers to develop component-based software systems in open-ended distributed systems. The reason why is that the engineers cannot obtain accurate information of each runtime environment where each component exists beforehand. Moreover, component-based software systems such as applets based on JavaBeans are downloaded & executed on several remote hosts in runtime. Thus, the runtime environments and their states cannot be specified beforehand. Such various runtime environments and their dynamic changes have given a strong influence on the execution of programs, which we cannot ignore.

Many component-based software systems are realized on the middleware systems without depending on specific hardware devices and/or operating systems. Java virtual machine is a typical instance of such middleware systems. However, even if such middleware systems can hide various runtime environments and their dynamic changes completely, the changes give a strong influence to upper-level applications if the changes are remarkable. Moreover, the hiding of runtime environments by middleware systems causes another problem. The upper-level applications cannot make full use of characteristic properties in each runtime environments such as special hardware devices, etc. because of the hiding.

To handle these problems, it is an effective way to change the behaviors of each software system depending on the states of runtime environments. However, such behaviors depend on properties of each software system. Thus, it is difficult to control such behaviors of each software system from its outside. From the reason, each software system should have the ability to be aware of the states' changes in runtime environments, and to change its own behaviors depending on the states. We call such ability *dynamic adaptability*. However, there is a limitation on runtime environments and their states that software engineers can predict beforehand. Thus, there is also a limitation on the dynamic adaptability that the engineers can give to software systems beforehand. Therefore, the mechanism of dynamic adaptability must be extensible. Such software systems with dynamic adaptability not only adapt themselves to dynamically changing runtime environments, but also change their own functionalities flexibly to make full use of the properties in the runtime environments. We have proposed a software model with dynamic adaptability called *DAS*[1, 2] and its description language LEAD++[3]. The DAS model is a reflective software model, and LEAD++ is an object-oriented reflective language based on Java.

Using LEAD++, we can construct component-based software systems with dynamic adaptability (called *dynamically adaptable software systems*) in flexible way. The basic ideas are that we change: *1*) behaviors of each component, *2*) the structure of components composition, depending on the states of runtime environments. These ideas are realized by using *adaptable procedure*, *event object* and *reflective adaptation mechanism* in LEAD++. An adaptable procedure is a variant of generic procedure (function) whose methods are dispatched depending on the states. An event object monitors & detects states' changes of runtime environments, and invokes the call-back methods for the changes. The Plug & Play of components can be realized by using the event object. The reflective adaptation mechanism realizes adaptation strategies of adaptable procedures.

The remainder of this paper is organized as follows. Section 2 introduces the basic concepts of our dynamically adaptable component-based software systems. Section 3 gives an overview of the DAS model and its description language LEAD++. Section 4 shows how to construct dynamically adaptable component-based software systems using LEAD++ in detail. Section 5 mentions related works. Finally, Section 6 mentions our future works and concludes this paper.

2 Dynamically Adaptable Component-based Software Systems

2.1 Basic Concepts of Component-based Software Systems

We assume basic concepts of component-based software systems as follows.

- A component is an object based on a class.
- The components can be instantiated in runtime, and can interact with other ones (Plug & Play).
- The connection among components is based on message passing (in other words, method invocation).

Such component-based software system is represented as a direct graph. We call it *component-graph*. In a component-graph, the nodes are components, and the edges are paths of message passing. The actions of a component-based software system are also represented as the routes of message passing.

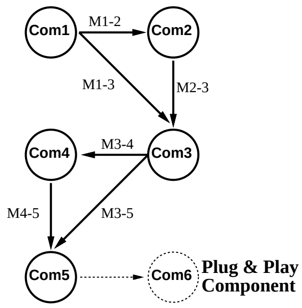

Fig. 1. A component-graph.

In Fig.1, Com$n$ represents a component, and Mm-n represents a message from Com$m$ to Com$n$. The message corresponds to a method invocation. Namely, Mm-n means that Com$m$ invokes a method in Com$n$. The potential component such as Com6 that has not been instantiated yet is drawn using a dotted line.

2.2 Basic Approaches

We adopt the following approaches to introduce dynamic adaptability into basic component-based software systems that we mentioned in Sect.2.1.

- We change the structure of components composition dynamically (namely, the alteration of component-graph).
- We also change behaviors of each component dynamically.

We aim to change the structure of components composition (component-graph) depending on the states of runtime environments in flexible way. Such alteration of component-graph is classified as follows.

1. Alteration of control flow:
   we change the routes of message passing among components dynamically.
2. Alteration of graph structure:
   we instantiate components dynamically, and change the graph structure by adding such Plug & Play components to the graph.

The alteration of control flow aims to change behaviors of component-based software systems by changing the routes of message passing without modifying the structure of component-graph. The alteration is based on the selection of components depending on the states of runtime environments. Namely, a dynamically adaptable component-based software system consists of several components that implement well-suited behaviors for the states. Then, it selects well-suited components depending on the states by using its own selection mechanism of components (called *adaptation mechanism*). Here, we assume that software engineers develop several components depending on the predictable states of runtime environments beforehand. However, it is not desirable to instantiate all components beforehand for effective utilization of computer resources. Thus, it is an effective way to instantiate & utilize components in runtime depending on the states. The alteration of graph structure is based on such dynamic activation of components. It also includes the alteration of control flow. We assume that the adjustment of interfaces among components (namely, program interfaces based on method invocation) will be assured after these alterations.

We also aim to realize the adaptation mechanism to select well-suited components depending on the states of runtime environments in flexible way. Since there is a limitation on runtime environments and their states that software engineers can anticipate beforehand, there is also a limitation on components for the states. By changing the selection policy (called *adaptation strategy*) of components depending on the states in flexible way, we can make full use of limited components. This approach may not provide the best behaviors of component-based software systems depending on the states, however we can expect the better behaviors by changing the adaptation strategy in flexible way.

It is an effective way to change the structure of components composition depending on the states of runtime environments. However, there may be a requirement that we want to change fine-grained behaviors in a component. Although we can handle such requirement to realize the nesting of components and/or hierarchy of ones, such solution is complicated because we must define new components for fine-grained behaviors. If there is the mechanism that can change behaviors in a component depending on the states of runtime environments, we can make the description & structure simpler.

Fig.2 shows such component-based software system and its adaptable behaviors. The two component-graphs in Fig.2 represent the different behaviors of the same system in the state (A) and (B) of a runtime environment. When the

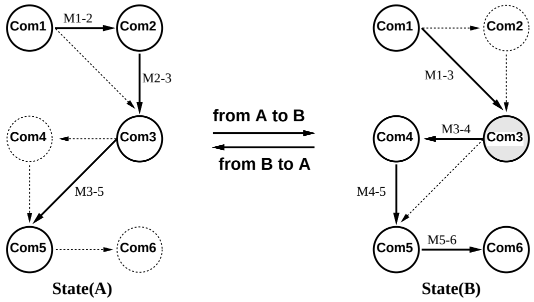

Fig. 2. A dynamically adaptable component-based software system.

state changes from (A) to (B), the component-based software system realizes the following adaptable behaviors.

- Com*1* sends a message to Com*3* directly without sending a message to Com*2* (skip the process of Com*2*).
- Com*4* is instantiated. Com*3* sends a message to Com*4*, and it also sends a message to Com*5* via Com*4* (add to the process of Com*4*).
- Com*6* is instantiated. Com*5* sends a message to Com*6*.
- The inside behaviors of Com*3* are changed.

If the state changes from (B) to (A) again, the opposite behaviors will be occurred. Although the selection mechanism of components is not shown in Fig.2, for simple, each component-based software system has it inside. Then, by changing the selection policy of components, each system can realize its own adaptable behaviors in flexible way. For example, we can realize adaptable behaviors such as the temporary restriction of software systems' functionalities and its release without modifying the graph structure (the alteration of control flow only).

We conclude that the following mechanisms are required to realize dynamically adaptable component-based software systems based on our approaches.

- The mechanism to change the structure of components composition depending on the states of runtime environments.
- The mechanism to instantiate components in runtime (Plug & Play).
- The mechanism to change inside behaviors of a component in runtime.

3 Overview of LEAD++

We have proposed a software model with dynamic adaptability called *DAS*[1, 2] and its description language LEAD++[3]. The DAS model is a reflective software model, and LEAD++ is an object-oriented reflective language that is an extension of Java. The syntax of LEAD++ is based on Java with some additional extensions. The prototype of LEAD++ is a pre-processor of Java. LEAD++ provides effective mechanisms to construct dynamically adaptable component-based software systems in flexible way.

3.1 The DAS Model

The following are basic concepts of the DAS model.

- It provides *environmental objects* that represent the local & virtual runtime environments depending on the abstractions of procedures in software systems. They reflect the states of actual runtime environments, and can be defined & instantiated arbitrarily in a software system.
- It provides *event objects* that monitor the states' changes of environmental objects. When they detect the changes (called *state events*), they invoke procedures (called *call-back methods*) for the changes in the software systems that use the environmental objects.
- It provides *adaptable procedures*. An adaptable procedure is a variant of generic procedure (function). It consists of several *adaptable methods* that implement well-suited behaviors depending on the states of environmental objects. An adaptable method consists of conditions (dispatch conditions) for an environmental object and actual runtime codes. When an adaptable procedure is invoked, a well-suited adaptable method is selected & executed depending on the states. We call the selection mechanism *adaptation mechanism*, and call the selection strategies *adaptation strategies*. An adaptation strategy is specified every environmental object.
- It provides a *reflective adaptation mechanism*. The main mechanism consists of a *strategy object* that implements adaptation strategies and a *dispatch object* that controls adaptable procedures using the strategy object. The adaptation strategy is realized using an adaptable procedure in reflective way. Thus, the mechanism (in other words, adaptation strategy) can be changed depending on the states of environmental objects.
- A software system based on the DAS model forms a meta-level architecture (Fig.3). The meta-level consists of the descriptions for dynamic adaptability. The base-level consists of primary subject domain codes (including adaptable procedures and call-back methods) in the software system.

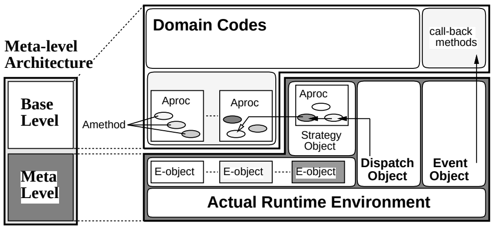

Fig. 3. The architecture of dynamically adaptable software systems based on the DAS model.

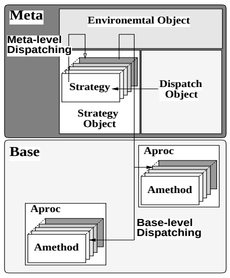

Fig. 4. The reflective adaptation mechanism.

3.2 Reflective Adaptation Mechanism

The adaptable method dispatching is realized from two levels dispatching (Fig.4). When an adaptable procedure is invoked in the base-level, the invocation is caught by a dispatch object in the meta-level. The dispatch object instantiates a strategy object. Then, the dispatch object picks up an adaptable method (adaptation strategy) in the strategy object and executes it. The adaptable method (adaptation strategy) dispatches the adaptable methods (including itself) in the strategy object depending on the states of an environmental object (*meta-level dispatching*). Moreover, using the adaptation strategy selected by the meta-level dispatching, an adaptable method in the invoked adaptable procedure is dispatched depending on the states (*base-level dispatching*). Here, we assume that the default meta-level dispatching is realized depending on the value of a variable called *strategy variable* in an environmental object (a string of strategy's name). However, such default meta-level dispatching is not fixed. We can customize it flexibly by using simple protocol that LEAD++ provides.

The meta-level dispatching has given a strong influence on the adaptable behaviors in the base-level. Furthermore, the software systems based on the DAS model can change the behaviors of the meta-level from the base-level by modifying the values of strategy variables in environmental objects. Such reflective adaptation mechanism has the following advantages.

- We can customize the adaptation mechanism in dynamic & flexible way depending on the states of runtime environments (environmental objects).
- We can uniformly extend & change the adaptation mechanism in the meta-level using the same mechanism as an adaptable procedure in the base-level.

By changing adaptation strategies depending on the states of runtime environments, we can make full use of the limited adaptable methods. Moreover, we can realize adaptable behaviors in flexible way such as the temporary restriction of software systems' functionalities and its release without modifying the descriptions of adaptable procedures.

3.3 Adaptable Methods

An adaptable procedure is a set of adaptable methods. An adaptable method consists of conditions for the state of an environmental object and actual runtime codes. LEAD++ provides the following syntax to describe adaptable methods.

```
public adaptable type-name method-name (arg, ... ) {
  StateCond {
    doc: "...";        // document string of this adaptable method
    type: type name; // type of body:
    body: { ... }      // condition codes for states of an environmental object
  } RunCode { ... }  // actual runtime codes
}
```

An adaptable method is defined by using the modifier `adaptable`. It consists of `StateCond` clause and `RunCode` clause. The former is the part of condition codes for the state of an environmental object, and the latter is the part of actual runtime codes for the state. The both of them are translated into ordinary methods by LEAD++ translator. They are called *condition method* and *code method*, respectively. The body of condition method is the field `body:`, and its return type is the field `type:`. The body of code method is `RunCode` clause, and the return type of code method is the same type as the adaptable method definition. The arguments of the both methods are also the same arguments as the adaptable method definition. Here, `adaptable`, `StateCond`, `doc:`, `type:` `body:` and `RunCode` are reserved words in LEAD++.

LEAD++ provides the class of standard environmental objects. To describe conditions for the state of an environmental object, we use methods in the class. The following are part of such methods (Table.1).

Table 1. Part of methods in environmental objects.

Methods	Contents
`getDisplayWidth()`	display size (width)
`getDisplayHeight()`	display size (height)
`getPowerSource()`	power source
`getBatteryRemainder()`	battery remainder
`getFreeMemory()`	free memory size

The conditions for the state of an environmental object are represented as the set of conditions using these methods. Adaptation strategies decide how to evaluate the set. LEAD++ provides several basic strategies (see [1]). For instance, as the simplest strategies, there are AND strategy and OR one. The former evaluates dispatch conditions in adaptable methods by using AND ($\wedge$), and the latter evaluates them by using OR ($\vee$).

Strategy objects that provide adaptation strategies are dynamically instantiated by dispatch objects. Thus, we need to specify dispatch objects that control adaptable methods. LEAD++ provides the following syntax to specify a dispatch object.

`DispatchObj:`*obj-class-name* `{`adaptable-method-name`,...}`

In above code, *obj-class-name* is the class name of dispatch object. The code represents that a dispatch object based on the class is instantiated in runtime, and the adaptable methods in the brace are controlled by the dispatch object. We can describe the names of several adaptable methods in the brace by punctuating them. `DispatchObj:` is also a reserved word in LEAD++. We invoke an adaptable method as follows.

`adaptable:`*obj.adapt-method*(environmental-object,...);

We use the keyword `adaptable:` to invoke adaptable methods. In above code, *obj* is the object whose class includes the definition of adaptable method *adapt-method*. The first argument of adaptable method must be an environmental object. The other arguments are specified after the second.

3.4 State Events

Event objects receive environmental objects and monitor states' changes (state events) of the environmental objects. When they detect the changes, they invoke call-back methods for the changes in the software systems that use the environmental objects.

Table 2. Part of state events & call-back methods.

State events	Contents	Call-back methods
power event	power source	`powerSourceChanged()`
battery event	battery remainder	`batteryRemainderChanged()`
memory event	free memory size	`memoryFreeSizeChanged()`
timer event	time interval	`timerInterrupt()`

Table.2 shows part of state events that standard event objects in LEAD++ provide. To use an event object, we specify the interface that defines call-back methods (`StandardCallBack`), and implement them. Then, we activate the event object as follows.

`event:enable`(current-obj, environmental-obj);

In above code, `event:` is a reserved word in LEAD++, and `enable()` is a method in the event object. We specify the current object that receives the state event and the monitored environmental object by the event object as its arguments. An event object is executed as another thread. We deactivate an event object as follows.

```
event:disable();
```

3.5 Metaobjects & Protocol

In LEAD++, environmental objects, event objects, strategy objects and dispatch objects are called *metaobjects*. They are opened to the users as the extensible meta system of LEAD++. Table.3 shows such metaobjects classes in LEAD++. We can customize the metaobjects using the simple protocol (*metaobject protocol*) in LEAD++. It is summarized as follows.

Table 3. Metaobjects classes in LEAD++.

Class name	Contents
`StandardEnv`	standard environmental objects
`StandardEventCtrl`	standard event objects
`StandardStrategy`	standard strategy objects
`StandardCtrl`	standard dispatch objects

- Inheriting the metaobjects classes.
- Overriding and/or overloading their methods of the classes.
- Adding adaptable methods to the classes.

4 Dynamically Adaptable Component-based Software Systems in LEAD++

We discussed about the required mechanisms to realize dynamically adaptable component-based software systems in Sect.2.2. They are summarized as follows.

- The mechanism to change inside behaviors of a component.
- The mechanism to instantiate components in runtime (Plug & Play).
- The mechanism to change the structure of components composition.

Using adaptable methods in LEAD++, we can realize the first mechanism easily. Namely, we describe the behaviors of a component depending on the states of runtime environments as adaptable methods by using LEAD++. Then, we can instantiate components in runtime by using event objects in LEAD++. Moreover, we can change the structure of components composition (component-graph) by using adaptable methods, event objects and the reflective adaptation mechanism (namely, the alteration of adaptation strategies) in LEAD++.

4.1 Construction of Component-Graph

We adopt the following approaches to construct a component-graph.

- We define a base object of a component-based software system Such object is called *template object*.
- A template object defines slots (variables) to store components.
- A template object instantiates components, and constructs the component-graph by invoking methods in components.
- A template object instantiates components in runtime by using event objects in LEAD++, and add nodes (components) to the component-graph.

We assume that we can set template objects in slots of the other template objects. The nesting of template objects promotes the grouping and hierarchy of components.

Then, to utilize computer resources effectively, a template object does not instantiate all components at a time. It instantiates them according to the necessity, and binds them to slots. In a component-graph representation, we distinguish the slots with components and empty slots. The former is called *active nodes*, and the latter is called *non-active nodes*. In the left-hand side component-graph in Fig.5, Com*1*, Com*3* and Com*5* are active nodes, and Com*2* and Com*4* are non-active nodes.

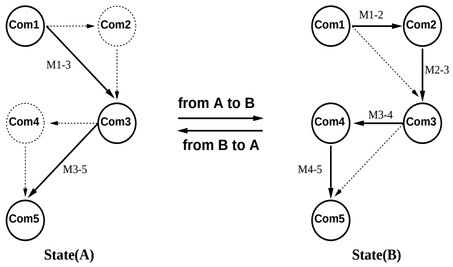

Fig. 5. The construction & alteration of component-graph.

The following descriptions are part of the class definition of template object that forms the component-graph in Fig.5.

```
public class TempObj {
  public static Comp1 com1;  // Slot for Com1
  public static Comp2 com2;  // Slot for Com2
  public static Comp3 com3;  // Slot for Com3
  public static TempObj currentObj;
  ...
  public TempObj() { // Components are set in slots
    com1 = new Comp1();
    com3 = new Comp3();
  }
  public static void main(String[] args) {
    currentObj = new TempObj();
    com1.sendMessage(com3);
    ...
  }
  ...
}
```

For simple, we omit the class definition of each component. In above descriptions, all components are not bound to slots at a time. Here, we assume that Com*3* is passed into Com*1*, and the methods in Com*3* are invoked by Com*1*. We also omit the detail definition of method `sendMessage()` in Com*1*, for simple. Moreover, we assume that Com*3* is also a template object that define slots for Com*4* and Com*5*, and binds them to the slots depending on the states of runtime environments.

4.2 Alteration of Component-Graph

The alteration of component-graph is classified as the alterations of graph structure and control flow (see Sect.2.2). They are realized as follows.

1. Alteration of graph structure:
   realization based on dynamic instantiations of components by event objects.
2. Alteration of control flow:
   - realization based on adaptable methods.
   - realization based on the reflective adaptation.

On the alteration of control flow, we only mention the realization based on adaptable methods in this paper due to the limitation of space.

Alteration of Graph Structure
The alteration of graph structure points to the operation that we make non-active nodes active ones depending on the states of runtime environments. The operation can be realized by using event objects in LEAD++. For example, in the left-hand side component-graph in Fig.5, Com*2* is a non-active node. Here, we assume the following specification. When the power source is changed from AC power to battery, Com*2* becomes an active node. We can realize such specification by adding the following codes to the definition of the template object that we mentioned in Sect.4.1.

```
public class TempObj implements StandardCallBack {
  // Specifying the event object
  EventObj:StandardEventCtrl;
  ...
  public static void main(String[] args) {
    // Instantiation of the environmental object
    StandardEnv envObj = new StandardEnv();
    // Activation of the event object
    event:enable(currentObj,envObj);
    ...
  }
  public void powerSourceChanged(StateEvent evt, StandardEnv env) {
    com2 = new Com2();
  }
  ...
}
```

We specify an event object, and activate the event object. In above codes, the argument `currentObj` means this template object itself that receives the power event from the event object. Then, we specify the interface `StandardCallBack` for the call-back methods, and define the call-back methods. The call-back method for the power event is `powerSourceChanged()` (see Table.2). When the state of runtime environment (here, the environmental object based on the class `StandardEnv`) changes, namely the power source changes from AC power to battery, vice versa, the call-back method is invoked by the event object. In the call-back method, Com*2* is instantiated and bound to the slot `com2`. On the contrary, there may be the requirement that we make Com*2* (an active node) a non-active node, when the power source changes from battery to AC power. We satisfy such requirement by defining the following adaptable methods, and by invoking them in the call-back method.

```
public adaptable void slotBind(StandardEnv env) {
  StateCond { // Power source is AC-power
    doc: "AC-power. So, com2 is not active.";
    type: boolean;
    body: {return(env.power() == "AC-power");}}
  RunCode {com2 = null;}
}

public adaptable void slotBind(StandardEnv env) {
  StateCond { // Power source is Battery
    doc: "Battery. So, com2 is active.";
    type: boolean;
    body: {return(env.power() == "Battery");}}
  RunCode {com2 = new Com2();}
}
```

Here, we assume that the method `power()` in the environmental object obtains the kind of power source, and returns the string `AC-power` (in the case of AC power) or `Battery` (in the case of battery), respectively. Invoking the adaptable method `slotBind()` in the call-back method, we can change the graph structure depending on the power source (namely, we can make Com*2* an active/non-active node). The following are the descriptions.

```
public void powerSourceChanged(StateEvent evt, StandardEnv env) {
  adaptable:currentObj.slotBind(env);
  ...
}
```

Here, we assume that `currentObj` is the template object, and the adaptable method `slotBind()` is defined in the template object. Then, `StateEvent` is the class that defines the information of state events in detail.

Alteration of Control Flow using Adaptable Methods

As an example, we consider the adaptable behaviors of Com*1* in the component-graph in Fig.5. From the assumption, Com*1* directly invokes the method in Com*3* when the power source is AC power. If the power source is battery, Com*1* invokes the method in Com*2*, and invokes the method in Com*3* via Com*2*. Using adaptable methods, we can easily realize such behaviors of Com*1*. Here, we try to describe the part of invocation of the method `sendMessage()` as adaptable methods.

```
public adaptable void powerAction(StandardEnv env) {
  StateCond { // Power source is AC-power
    doc: "AC-power. So, com2 is not active.";
    type: boolean;
    body: {return(env.power() == "AC-power");}}
  RunCode {com1.sendMessage(com3);}
}
```

```
public adaptable void powerAction(StandardEnv env) {
  StateCond { // Power source is Battery
    doc: "Battery. So, com2 is active.";
    type: boolean;
    body: {return(env.power() == "Battery");}}
  RunCode {com2 = new Com2(com3);
           com1.sendMessage(com2);}
}
```

These codes satisfy above specification of Com*1*. Namely, Com*3* is passed into Com*1*, and Com*1* invokes the method in Com*3*, when the power source is AC power (the 1st definition of `powerAction()`). If the power source is battery, Com*2* is instantiated and bound to the slot `com2`, and then Com*2* is passed into Com*1* (the 2nd definition of `powerAction()`). In this case, we assume that Com*1* invokes the method in Com*2*, and invokes the method in Com*3* via Com*2*. To invoke the method in Com*3* through Com*2*, Com*3* is passed into Com*2* at the instantiation of Com*2*.

By invoking such adaptable methods in the template object, we can realize component-based software systems that satisfies above specification. The following are part of the descriptions.

```
public static void main(String[] args) {
  ...
  adaptable:currentObj.powerAction(new StandardEnv());
}
```

In above codes, `currentObj` is also the template object itself.

5 Related Works

There are ActiveX/DCOM[17, 18], JavaBeans/JavaRMI[20, 21] and CORBA[19] as the typical component technologies and architectures. However, they have not the adaptation mechanism and/or framework for dynamically changing runtime environments. Although CORBA has the mechanism of dynamic binding to combine among components in flexible way, the architecture of CORBA is rather complicated. Thus, it is not easy to construct component-based software systems with dynamic adaptability by using CORBA only.

As the works for constructing dynamically adaptable component-based software systems, there are Demeter Method[11], Adaptive Plug & Play Components[12], Architectural Reflection[4], etc.

As the other related works, there are Compositional Interface[14], Compositional Adaptation[8], Object-Graph Customization[7], AOP[10], etc. These works do not aim to construct component-based software systems directly. However, they provide effective mechanisms to realize component-based software systems and their adaptable behaviors.

There are also ABCL/R[16], CLOS[9], AL-1/D[13], OpenC++[6], etc. as related works of the other object-oriented reflective languages. Although their design is mainly aimed for language (or runtime extension), they have potentiality for constructing dynamically adaptable software systems.

Moreover, there are the language Cecil[5] and the middleware DART[15] that provide the mechanism such as adaptable methods in LEAD++.

Toward these related works, the following are features of the DAS model.

- It is a generic model that is independent of specific paradigms and/or description languages.
- The adaptation mechanism can adapt itself to dynamically changing runtime environments.
- The adaptation mechanism can be modified/extended by adding adaptable methods incrementally without modifying the structure.
- It provides asynchronous events mechanism (event object) depending on the abstractions of the local & virtual runtime environments.

However, there are also weak points of the DAS model. For example, when we develop dynamically adaptable software systems based on the DAS model in practice, the following problems occur.

- When we describe adaptable procedures, we must consider the adaptation strategies for them.
- We must pay attention to the alteration of adaptation strategies sufficiently.

6 Concluding Remarks & Future Works

We have mentioned how to construct component-based software systems with dynamic adaptability using LEAD++. A dynamically adaptable component-based software system is represented as the graph that connects with several components (component-graph). The adaptable behaviors for dynamically changing runtime environments are realized as follows: *1*) the alteration of behaviors in each component, *2*) the alteration of the structure & control flow of component-graph. Using adaptable methods, event objects and the reflective adaptation mechanism in LEAD++, we can realize adaptable behaviors of component-based software systems in flexible way.

As the future works, we plan to introduce *delegation object* into the DAS model and to extend LEAD++. It resolves the adjustment problem of program interfaces among components after the alteration of the structure & control flow of component-graph. Then, we also plan to develop practical component-based software systems with dynamic adaptability by using LEAD++ and JavaBeans. As the example, we consider to develop dynamically adaptable applets based on JavaBeans using animations & sounds. Since the prototype of LEAD++ is a pre-processor of Java, it is easy to combine our approaches with JavaBeans.

Acknowledgments

We are grateful to Prof. Tatsuo Nakajima, Dr. Kazuhiro Ogata and the members of the language design laboratory at JAIST for discussions and comments.

References

1. N. Amano and T. Watanabe. A Procedural Model of Dynamic Adaptability and its Description Language. *Proc. IWPSE'98*, pp. 103–107, April 1998.
2. N. Amano and T. Watanabe. LEAD: Linguistic Approach to Dynamic Adaptability for Practical Applications. *Proc. IFIP Conf. System Implementation 2000*, Chapman & Hall, pp. 277–290, February 1998.
3. N. Amano and T. Watanabe. LEAD++: An Object-Oriented Reflective Language for Dynamically Adaptable Software Model. *IEICE Trans*, E82-A(6):1009–1016, June 1999.
4. W. Cazzola, A. Savigni, A. Sosio, and Francesco. Architectural Reflection: Concepts, Design and Evaluation. *Technical Report RI-DSI 234-99, DSI, University degli Studi di Milano*, May 1999.
5. C. Chambers. Predicate Classes. *Proc. ECOOP'93, LNCS, Vol. 707*, pp. 268–296, July 1993.
6. S. Chiba and T. Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. *Proc. ECOOP'93, LNCS, Vol. 707*, pp. 482–501, July 1993.
7. A. Hokimoto, K. Kurihara, and T. Nakajima. An Approach for Constructing Mobile Applications using Service Proxies. *Proc. ICDCS'96*, May 1996.
8. 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.
9. G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.
10. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. *Proc. ECOOP'97, LNCS, Vol. 1241*, pp. 220–242, 1997.
11. K. J. Lieberherr. *Adaptive Object-Oriented Software – The Demeter Method – with Propagation Pattern*. The PWS Series in Computer Science. PWS Publishing Company, 1996.
12. M. Mezini and K. Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development. *Proc. ACM Conf. OOPSLA'98*, pp. 97–116, 1998.
13. H. Okamura and Y. Ishikawa. Object Location Control Using Meta-level Programming. *Proc. ECOOP'94, LNCS, Vol. 821*, pp. 299–319, July 1994.
14. R. K. Raj and H. M. Levy. A Compositional Model for Software Reuse. S. Cook ed., *Proc. ECOOP'89*, pp. 3–24. Cambridge University Press, July 1989.
15. P.-G. Raverdy, H. L. V. Gong, and R. Lea. DART: A Reflective Middleware for Adaptive Applications. *Technical Report of Center for Computational Physics, University of Tsukuba 98-4*, pp. 37–40, October 1998.
16. T. Watanabe and A. Yonezawa. Reflection in an Object-Oriented Concurrent Language. *Proc. ACM Conf. OOPSLA'88*, pp. 306–315, 1988.
17. A. Denning. *ActiveX Controls Inside Out*. Microsoft Press, 1997.
18. D. Rogerson. *Inside COM*. Microsoft Press, 1997.
19. OMG: *CORBA 2.0*, http://www.omg.org/library/specindx.htm.
20. JavaSoft: *JavaBeans*, http://www.javasoft.com/beans/.
21. JavaSoft: *JavaRMI*, http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi.