# Modification of Shared Resources in AL-1/D

Hideaki Okamura† okamura@mt.cs.keio.ac.jp
Yutaka Ishikawa†† yisikawa@etl.go.jp
Mario Tokoro† * mario@mt.cs.keio.ac.jp

† Department of Computer Science, Keio University
3-14-1 Hiyoshi, Kohoku-ku, Yokohama, Kanagawa, 223 Japan

†† Real World Computing Partnership, Tsukuba Research Center
1-6-1 Takezono, Tsukuba, Ibaraki, 305 Japan

**Abstract**

This paper describes the modification of shared resources in a reflective programming system AL-1/D. The systems where shared resources can be modified employ the concept of grouping. In these systems, member objects of a group share resources that belong to the group. AL-1/D adopts a difference approach. In AL-1/D, a meta object has information on shared resources without the grouping concept. Another topic in this paper is an issue about cooperations between default system resources and new user-defined resources.

## 1 Introduction

Shared resources in a programming system are computational resource that are shared by multiple objects such as a scheduler and a garbage collector (GC). A reflective programming system AL-1/D provides users with facilities for exchanging shared resources according to the characteristics of user application programs. This facilities enable users to execute application programs efficiently and to experiment with new algorithms on shared resources.

For example, let us consider a scheduler. Message priorities and time constraints are necessary for soft-realtime application programs. Some applications require efficient message handling, rather than support for priorities, when the programmer knows that multiple messages are not sent simultaneously in those applications. In another case, we need to modify an algorithm to enable it to handle messages after timeout, e.g. whether or not the execution of messages is retried; considering balance with communication costs. Next, let us consider a GC. There are some algorithms for the GC, such as mark and sweep, reference counting and so on. We want to exchange these algorithms and to evaluate new GC algorithms in a programming system not in a simulator.

In AL-1/D, the various abstraction levels of programming systems, from the language level to OS level, are defined as the metalevel. The metalevel can be decomposed into multiple models by a new reflection framework: Multi-Model Reflection Framework[2]. Six models are available in AL-1/D. For example, Operation model represents operational semantics of a base-level object such as message passing, Resource model represents OS functions including shared resources such as a scheduler and a GC.

This paper describes the modification of shared resources. Following aspects are mentioned:

- The mechanism of AL-1/D for modification of shared resources is shown. That is, a way for setting shared resources to an object is described. Conventional reflective systems, such as ABCL/R2[1] and Apertos[3], introduce concepts of grouping while AL-1/D does not employ these concepts.
- The mechanism for cooperation between default system shared resources and those that users define newly. In this paper, default system resources are called "default resources", and new user-defined resources are called "user-level resources".

* Also with Sony Computer Science Laboratory Inc. Takanawa Muse Building, 3-14-13 Higashi Gotanda, Shinagawa-ku, Tokyo, 141, Japan, mario@csl.sony.co.jp

In Section 2, a general mechanism in AL-1/D for exchanging shared resources is described. We show how user-level resources cooperate together with default resources. In Section 3, related works are described.

# 2 Meta-programming in AL-1/D

## 2.1 Resource Model in AL-1/D

The Resource model is used to modify share resources such as a scheduler and a GC in AL-1/D. Following programs indicates the defined of base-level object `One` and its meta object in the Resource model `ResOne`. `One` is defined from line 1 to line 3. Meta object `ResOne` is shown from line 5 to line 8. This meta object has internal variables such as `gc, scheduler, allocator, segment, context`. These represent a GC, a scheduler, a memory allocator, memory segments object of object `One`, an active context object in object `One` respectively.

```
1 [ base One
2   (delegation nil)
3 ]
4
```

```
5 [ meta Resource ResOne One
6   (delegation nil)
7   (vars gc scheduler allocator segment context)
8 ]
```

## 2.2 Modification

A way for modification of shared resources in AL-1/D is described. When new shared resource $R_{new}$ is set to base-level object $O_{base}$, following procedures are necessary:

- Add $R_{new}$ to the *resource manager object*.
- Assign $R_{new}$ to a variables corresponding to $R_{new}$ in the meta object in the Resource model of $O_{base}$.

It shroud be noted that the shared resource is not managed by the manager of an object group but by the meta object of a base-level object. A resource manager object is described in Section 2.3.

The following example program indicates method `replaceScheduler` in object `One`. This method adds a new scheduler object `UserSched` to object `One`.

```
1 [ method One replaceScheduler
2     ResourceManager addScheduler: UserSched;
3     (self meta: #scheduler) scheduler: UserSched
4 ]
```

In line 2, `UserSched` is added to resource manager object `ResourceManager`. In line 3, "**self** meta: #scheduler" returns a meta object that has variable `scheduler`, that is, a meta object in the Resource model of `One`. `UserSched` is assigned to variable `scheduler` in this meta object.

## 2.3 Invocation of Shared Resource

It is necessary to consider the problems of cooperations between shared resources in the system. For example, if multiple schedulers run in a system, following problems arise. (i) Can a scheduler run simultaneously when another scheduler is running? (ii) How a context created newly is allocated to the suitable scheduler? Let us consider GC. Which GC collects garbages that cannot be scavenged by any user-level GCs, such as objects cross-referred between groups?

In AL-1/D, default shared resources, which is used a entire virtual machine of AL-1/D, are called *default resources* while resources that users define newly are called *user-level resources*. A base-level object is handled by the default scheduler when users do not define new schedulers for the base-level object. Users can add user-level resources based on the way mentioned in Section 2.2.

To solve above-mentioned problems about cooperation between default resources and user-level resources, User-level resources are managed by the default resource. The *resource manager object* is introduced for the management of user-level resources. The resource manger object has the lists of user-level resources correspond to default resources. When the default resource controls user-level resources, the resource manager object is referred by default resources.

User-level schedulers are scheduled by the default scheduler. A context managed by a user-scheduler is forwarded when the context starts running. Controlling of default scheduler enable user-level schedulers to avoid conflicts each other. Similarly, the default GC controls user-level GCs. The default GC collect garbages that cannot be scavenged by any user-level GCs.

## 2.4 Example: User-level Scheduler

In this section, an example of the modification of scheduler is shown. The following program is the definition of user-level scheduler object `UserSched`. Method `start` initializes `UserSched`. In line 8, a table is create. This table is used to keep objects that use `UserSched`. In line 11, `UserSched` is added to the resource manager object `ResourceManager`.

Variable `evaluator` defined in line 3 is bound to object `Evaluator` that can access the default scheduler. `evaluator` is sent messages such as `add:` and `get`. By these message, a user-level schedulers and the default scheduler can cooperate with each other. By message `add:`, a context is forwarded to the default scheduler. By message `get`, a context is passed from the default scheduler to the user-level scheduler.

```
1 [ base UserSched
2   (delegation nil)
3   (vars objTable objnum  evaluator
4         firstcontext lastcontext)
5 ]
6
7 [ method UserSched start
8   objTable = SymTable create initialize: 32;
9   objnum = 0;
10  evaluator = Evaluator create;
11  ResourceManager addScheduler: self
12 ]
13
```

The following program implements the algorithm of round-robin scheduling. Method `run` is invoked by `Timer` object in the AL-1/D virtual machine and performs context switching.

```
1 [ method UserSched run
2   (vars ctxt nextctxt)
3   if (firstcontext != nil) then
4     ctxt = evaluator get: firstcontext;
5     nextctxt = ctxt next;
6     ctxt next: nil;
7     evaluator put: firstcontext;
8     if (nextctxt != nil) then
9       firstcontext = nextctxt;
10      lastcontext next: ctxt
11    else
12      firstcontext = ctxt;
13      lastcontext  = ctxt
14    end
15  end
16 ]
```

In method `add:` a context created newly is added to `UserSched`. This method are invoked by the Resource model of a base-level object that uses `UserSched`, and appends the context to the end of a ready queue. There is the program of the Resource model from line 11 to line 15. Method `schedule` invokes method `add:` of `UserSched`. The invocation of the Resource model is mentioned in Section 2.5.

```
1 [ method UserSched add: ctxt
2   if (firstcontext == nil) 'pthen
3     firstcontext = ctxt;
4     lastcontext  = ctxt
5   else
6     lastcontext next: ctxt;
7     lastcontext = ctxt
8   end
9 ]
10
11 [ method ResOne schedule
12   if (context != nil) then
13     scheduler add: context
14   end
15 ]
```

## 2.5 Example: Invocation of Scheduler

In this section the way how a user-level scheduler is invoked is described.

A user-level scheduler is invoked from a method in the Resource model of a base-level object as mentioned in the previous section. The Resource model is invoked by events shown as the following table. In this table, a "context sequence" means a sequence of contexts created in an application program.

| Event | Occurrence |
|---|---|
| **add-context** | a context sequence is newly created. |
| **delete-context** | the last context in a context sequence finishes |
| **suspend-context** | a running context is suspended by message sending from the context and a new context is created. |
| **resume-context** | a suspended context is resumed. |

Methods corresponding to these events are defined in a meta object in the Resource model. For example, when event `add-context` occurs, the AL-1/D virtual machine invokes method `add:`, shown in Section 2.4

### 2.6 Example: Modification of Scheduler

The following program defines method `add:` of another scheduler object `UserSched2`. In this method, a currently executing context is suspended and a new context is executed earlier if the new context has an `interrupt` flag. When the new context finishes, the suspended context is resumed. The interrupt flag stands for the higher priority than the ordinary execution of a context. Lines 3 to 6 are different from `UserSched` defined Section 2.4. In these lines, a current context is suspended and a new context start being executed.

```
1 /* another version */
2 [ method UserSched2 add: ctxt
3   if (ctxt flag == #interrupt) then
4     evaluator get: firstcontext;
5     ctxt waiting: firstcontext;
6     evaluator put: ctxt
7   else
8     if (firstcontext == nil) then
9       firstcontext = ctxt;
10      lastcontext  = ctxt
11    else
12      lastcontext next: ctxt;
13      lastcontext = ctxt
14    end
15  end
16 ]
```

## 3 Discussion

In conventional reflective systems, concepts for managing multiple objects are employed; such as a *group* in ABCL/R2 and a *meta space* in Apertos. In AL-1/D, shared resources belong to a meta object in the Resource model of a base-level object that uses these resources without grouping.

The benefits of managing share resources without grouping are following: (i) The number of groups does not gain though every resource would be exchanged. For example, if `object A` uses `scheduler 1` and `GC 1`, `object B` uses `scheduler 2` and `GC 1`, then each object needs two groups: one is for the scheduler and the other is for the GC. (ii) The ownership of shared resource can be defined. For example, let us consider that we want to count the number of times of scheduling for an object. Exchanging the method of a meta object that invokes the scheduler enable us to count the number of times without harmful influence to other objects.

The faults of our approach is following: (i) Conceptually, the addition of shared resources have no semantics in the framework of reflection in our system while, in ABCL/R2, it is defined as hybrid group reflection. In AL-1/D, the addition of shared resources is defined as "addition to the system". (ii) Physically, the mechanisms cannot be realized by using the concept of grouping such as multi-cast communication to members of a group.

## References


[1] H. Masuhara, S. Matsuoka, T. Watanabe, and A. Yonezawa. Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently. In *Proceedings of OOPSLA'92*, October 1992.

[2] Hideaki Okamura, Yutaka Ishikawa, and Mario Tokoro. AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework. In *Proceedings of the International Workshop on New Models for Software Architecture'92 Reflection and Meta-level Architecture*, November 1992.

[3] Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proceedings of OOPSLA'92*, October 1992.