Object Location Control Using Meta-level Programming

Hideaki Okamura† Yutaka Ishikawa†† *

† Department of Computer Science, Keio University
3-14-1 Hiyoshi, Kohoku-ku, Kanagawa 223, Japan
†† Tsukuba Research Center, Real World Computing Partnership
1-6-1 Takezono, Tsukuba, Ibaraki 305, Japan

Abstract. In distributed environments, location control of objects among hosts is a crucial concern. This paper proposes a new mechanism of *object location control using meta-level programming* which provides the following advantages to programmers. First, the description of location control can be *separated* from the application program by exploiting the meta-level architecture. This separation makes it easy for programmers to understand application programs and change location control policies. Second, it is possible for programmers to control object location using *runtime information* provided at the meta-level such as the number of remote messages. This information enables programmers to control object location more flexibly than in traditional distributed languages. The mechanism proposed in this paper has been implemented on an *AL-1/D* distributed reflective programming system. We show that our mechanism of location control using meta-level programming provides reasonable performance for a distributed application.

1 Introduction

In distributed environments, location and distribution of objects among hosts (processors) which have no shared memories is a crucial concern. Object migration or object location control among hosts provides us benefits such as load sharing, efficient communication, fault tolerance, the utilization of special hardware and software capabilities, and data movement [8]. Several distributed systems with the object migration facilities have been developed [15, 1, 8, 2]. Also, a computational model, called the Computational Field Model [17], has been proposed for the optimized allocation of objects.

Most current distributed programming languages (DPLs), such as Emerald [8], Distributed Smalltalk [3], and COOL [7], provide the programmer with explicit object location control through programming languages. These languages have location control facilities such as explicit movement, object attachment, and parameter passing. However, these DPLs have two problems. First, the location

* This work was conducted under the partial support of the MITI project "New Models for Software Architecture" while the author was at Electrotechnical Laboratory.

control facility is incorporated into the application program together with computational algorithms for processing user requests, such as a search algorithm. Unnecessary of combination of the location control facility and the computational algorithm results in complicated programming flow and thus it is difficult for programmers who reuse existing application programs written by others to understand the program and change the location control policies used in the program. The second problem lies in the lack of information at runtime that can be used by the object location control facilities. The lack of runtime information makes it difficult for application programmers to use some useful location control facilities such as the object migration facility using the messages content for efficient communication.

In this paper, to overcome the problems of explicit user definition of location control through programming languages, we propose a new mechanism of *object location control using meta-level programming*. This mechanism is based on a *meta-object protocol* to achieve *separation of concerns*, that is, separating the user supplied computational algorithm and the mechanism needed to optimize and manage execution of the application program. This separation enables programmers to understand the program and change the location control policies because programmers can control object location through the programming language without complicating the application program. Our mechanism of object location control also provides runtime information at the meta-level where object behavior at runtime is defined. By using runtime information, the programmer can program object location control facilities more flexibly than that in existing DPLs which do not support capabilities for obtaining such information.

The location control mechanisms discussed in this paper have been implemented on our *AL-1/D* reflective programming system [13]. Based on the evaluation of application programs in AL-1/D, we discuss the trade-off between the overhead incurred by meta-level programming and performance improvement gained by object location control.

Section 2 discusses several useful object location control facilities which are necessary in DPLs and the problems in existing DPLs that support object migration. Next, in Section 3, we propose solutions to these problems by employing the mechanism of object location control using meta-level programming. In Section 4, we describe some examples of meta-level programming for object location control facilities in AL-1/D. In Section 5, we examine the trade-off mentioned above based on the evaluation of a distributed application program in AL-1/D. Related work is covered in Section 6. Finally, Section 7 summarizes this paper.

2 Object Migration in DPLs

In this section, we discuss several useful object location control facilities which are necessary in DPLs and the problems with existing DPLs.

2.1 Object Location Control Facilities

DPLs such as Emerald [8], Distributed Smalltalk [3] and COOL [7] feature a language primitive for explicit object movement. Programmers can move objects between hosts by using this primitive whenever necessary.

Programmers may also wish to explicitly specify which objects move together. If an object references other objects, these referenced objects may move together. For this purpose, Emerald allows the programmer to *attach* objects to other objects. When `object A` is attached to `object B`, `object A` moves together with `object B`. We call this facility, supported by Emerald, "*object attachment*".

An important object location control facility is the choice of *parameter passing* policies. When remote message passing or a remote procedure call is used, a programmer can select whether argument objects should be marshaled into the message. If a remote object invokes argument objects frequently after remote messages have been passed, argument objects should be moved by piggy-backing them on the sent message. This facility of object movement results in reducing remote communication cost [2]. Emerald and COOL provide this facility.

Fig. 1 is a sample program in Emerald, which illustrates *object attachment* and *parameter passing* facilities. This program is a part of the Emerald mail system. In line 1, the array of destination mailboxes is *attached* to the mail messages. When mail message is moved, the array pointed to at that time by `ToList` is moved with it. This may affect the performance of invocation on `ToList`. The operation starting at line 3 delivers the message to all the mailboxes on the `ToList`. If there is only one destination, *parameter passing* is used to move the mail message with the single destination mailbox which exists on the remote host (in Line 7).

```
 1 attached var ToList: Array.of[Mailbox]
 2
 3 operation Deliver
 4   var aMailbox: Mailbox
 5   if ToList.length = 1 then
 6     aMailbox ← ToList.getelement[ToList.lowerbound]
 7     aMailbox.Deliver[move self]
 8   else
 9     var i: Integer ← ToList.lowerbound
10     loop
11       exit when i > ToList.upperbound
12       aMailbox ← ToList.getelement[i]
13       aMailbox.Deliver[self]
14       i ← i + 1
15     end loop
16   end if
17 end Deliver
```

Fig. 1. Sample Program in Emerald

Other useful location control facilities which have not been incorporated into existing DPLs must be considered. One such facility is object movement when an object sends remote messages. If a programmer knows that the sender ob-

ject will send many messages to the receiver object, the sender object should migrate to the receiver object's host when communication is initiated to reduce communication cost. We call this migration facility *sender migration.*

Another useful facility is object movement after a certain number of messages is sent to an object. For example, we can consider the case when an object starts to move if the number of remote messages from a sender object to a remote receiver object exceeds some threshold set by the programmer. We call this migration facility *dynamic migration.*

Finally, object migration according to the receiver's name is also useful. For example, if an application programmer knows that `Object A` will send many messages to `Object B`, `Object A` migrates to `Object B`'s host when the first message is sent from `Object A` to `Object B`. We call this migration facility *receiver-name migration.*

2.2 Problems with Object Migration in DPLs

In existing DPLs, the unnecessary combination of computational algorithms and location control facilities, and the lack of runtime information are problems for object migration. These problems are discussed in this section.

Combining Computational Algorithm and Location Control

Several facilities mentioned above are suitable for writing distributed applications, such as mail systems, that are concerned mainly with data movement. There are many applications, however, in which data movement is not essential and some other function should be emphasized. Examples include search systems for distributed data, simulation by distributing tasks to hosts, etc. In these applications, location control is an important part of performance optimization and should be performed at a lower operating level than the computational algorithm which simply calculates the result needed by the programmer. Also, the programmer may not wish to be concerned about object location, but may instead want to concentrate on describing an algorithm to produce desired results.

For instance, when a programmer designs a data search system in a distributed environment, the programmer should concentrate on the search algorithm to be used. In some cases, the programmer may want to apply an existing search algorithm used in centralized systems. Object location and movement are of no importance, unless data is located on distributed resources or when efficiency must be emphasized. Combining location control and the computational algorithm into a single program makes the program complicated and, hence, difficult for programmers who reuse existing application programs written by others to understand or change.

Fig. 2 shows an example in which a location control mechanism and the computational algorithm are combined into a single program. In many traditional DPLs, object mobility is incorporated as in this example. The language used in this example has message passing syntax like Smalltalk [6] and a language

```
 1 [ object Searcher
 2   (delegation nil)
 3 ]
 4
 5 [ method Searcher search: nm with: sec with: age
 6   (vars data strm)
 7   while (data != nil) do
 8     if (fnum <= 30) then
 9       data = dmgr next;
10     else
11       data = dmgr next[move];        /* move if (number of messages > 30) */
12       fnum = 0;
13     end
14     if (dmgr isRemote) then          /* count remote messages */
15       fnum = fnum + 1;
16     end;
17     /* pattern matching, stream creation */
18     if (((data name) isMatch: nm) && ((data section) isMatch: sec) &&
19         ((data age) <= age)) then
20         if (strm == nil) then
21           strm = data dataToStream
22         else
23           strm add: (data dataToStream)
24         end;
25         dnum = dnum + 1;
26         if (dnum >= bufnum) then
27           return strm
28         end
29     end
30   end;
31   return "done"
32 ]
```

Fig. 2. Location Control Mechanism and Computational Algorithm Combined into a Single Program

primitive that supports object mobility so that it can easily be compared to AL-1/D programs described later. In this program, an object `Searcher` obtains data from the object `dmgr` located on the remote host and selects data items which match the keys specified by the user. Lines 1 to 3 define of the object `Searcher`. This language is not class-based but prototype-based like AL-1/D. Delegation is employed via a message forwarding mechanism. The `delegation` entry at line 2 describes that the delegation object of `Searcher` is `nil`, that is, there is no delegation object. Lines 5 to 32 define the method `search:with:with` which defines the behavior of the object `Searcher` while searching requested keys. The `vars` entry in the method describes temporary variables. The message expression "`data = dmgr next;`" denotes that a message `next` is sent to the object `dmgr`, and a return value is bound to the temporary variable `data`. Lines 18 to 25 describe a pattern matching algorithm. According to lines 8 to 16, if the number of remote messages exceeds the maximum 30, a base-level object is moved by the primitive `move`. Thus, the program contains an extra algorithm for location control. This makes it difficult to understand the search and pattern-matching algorithms in the program. Such combination of computational algorithms and

location control also makes it difficult to change the optimization policy. For instance, if the programmer needs information on the kinds of receiver objects instead of the number of remote messages, the whole program in Fig. 2 has to be recompiled and loaded in the system. But in order to do so, the application program must be terminated.

Lack of Runtime Information

Some object location control facilities such as "dynamic migration" and "receiver-name migration", described in Section 2.1 need runtime information, such as the number of messages and the receiver names. For example, to count the number of messages between `Objects A` and `B` regardless of message content, it is necessary to access to the message sending mechanism to trap messages. If the message sending mechanism can be accessed, we can count the messages each time they are sent. If information on the receiver objects is important, receiver names are needed. In this case, a message sending mechanism must also be accessible to retrieve receiver name from the message contents.

Traditional DPLs lack the facility which make it possible to access such runtime information from the message handling mechanism. This mechanism would be built into the programming language system during system design if the facility is necessary, however, it is difficult to predict all runtime information that is needed by the application programmer.

3 Location Control Using Meta-level Programming

In this section, we propose a mechanism for object location control using meta-level programming to overcome the problems described in the previous section: unnecessary combination of computational algorithms and location control facilities, and lack of runtime information. This location control mechanism is not supported by the basic language specification for writing computational algorithm, but by meta-level programming facilities which are based on the concept of *computational reflection* and *meta-level architecture*. The meta-level architecture is employed to achieve separation of concerns, that is, separating the user supplied computational algorithm and mechanism needed to optimize and manage it. With this architecture it is also possible to provide runtime information to the application program because this information can be gained by accessing object behavior defined at the meta-level.

3.1 Reflection and Meta-level Architecture

Computational reflection is a useful concept for modifying a system according to the programmer's requirements [16, 11, 18, 19, 12]. A system supporting the reflection concept is called a *reflective system*. A reflective system has a *meta-level architecture*, which consists of two levels; the base-level and the meta-level. To expose or reify its internals, a reflective system embodies reifiable data

that represents or implements the structural and computational aspects of the system at the meta-level. Such data must be dynamically self-accessible and self-modifiable by the user program. Furthermore, modification by the user must be 'reflected' to the actual computational state of the user program. This property is termed *causal-connection* [12]. We call programming at the meta-level *meta-level programming.* Computational reflection is employed by functional languages such as 3-Lisp [16] and BROWN [5], object-oriented languages such as 3-KRS [11] and CLOS [9], concurrent object-oriented languages such as ABCL/R [18], and distributed systems such as the Apertos OS [19] and ABCL/R2 [12].

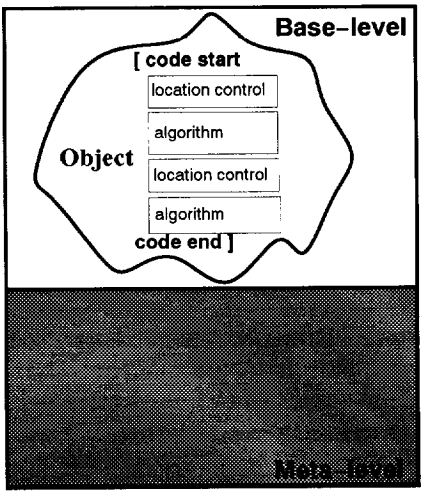

Fig. 3. Traditional DPL (Closed Implementation)

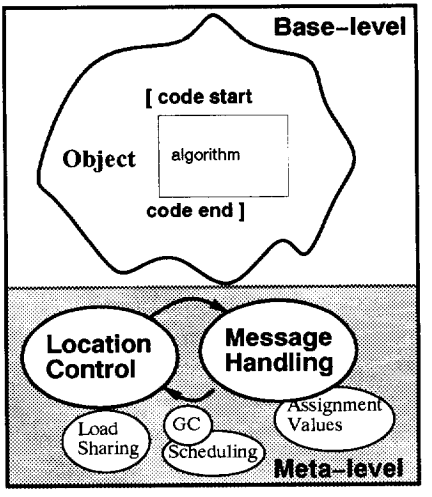

Fig. 4. Meta-level Architecture (Open Implementation)

3.2 Separation of Concerns

We describe the difference between programming in a traditional DPL and programming in a system with a meta-level architecture. Fig. 3 shows programming in a traditional DPL, where system implementation, that is, the meta-level, is closed to the application programmer. This black-box style of system design is called *closed implementation.* If location control facilities are needed in such a system, location control primitives must be introduced in the application program flow. This makes it difficult to understand the application program. A system design that does not use this style, is called *open implementation* [10]. This implementation is supported by a meta-level architecture. In open implementation systems, system components and operations, such as a location control mechanism and a message sending mechanism, are accessible by a programmer as in Fig. 4. These components and operation are programmed with a well-defined protocol called the *meta-object protocol.* In the open implementation system, meta-level programs and base-level programs are separated thus

a *separation of concerns* can be introduced. This enables programmers to easily read and construct programs, and to customize optimization code, such as location control, without changing the base-level program. In the the open implementation system with meta-level architecture, location control mechanisms are encapsulated and are not provided to users who do not need optimization while it can be accessed by users who require optimization.

3.3 Flexible Location Control Facilities

In addition, meta-level programming solves the problem incurred by the lack of runtime information. In meta-level programming it is possible to use runtime information provided at the meta-level by the meta-level architecture, and so programmers can use flexible location control facilities. In the open implementation system, the application programmer can access a message passing mechanism defined at the meta-level. By programming the message passing mechanism and adding a message monitoring facility based on this mechanism, runtime information such as the number of message and the receiver's name can be made available to the programmer. Using this information, the programmer can modify the object location control facility accordingly.

If a object location control policy can be programmed at the meta-level according to runtime information, we can modify it to suit the characteristics of the application program. We can also flexibly control object location because more complex requirements of applications such as those requiring information pertaining to both communication cost and object name leading to migration can be satisfied through meta-level programming with runtime information.

4 Meta-Level Programming in AL-1/D

In this section, we cover the object location control facilities by meta-level programming provided by the *AL-1/D* distributed reflective system. AL-1/D is a concurrent object-based reflective programming system that may be used over distributed environments [13]. An AL-1/D system consists of a compiler and a virtual machine, that is, a bytecode interpreter, constructed on a UNIX-based operating system. Basically, programs are constructed using concurrent objects and messages. The message sending syntax is similar to that of ConcurrentSmalltalk [20]. Multiple virtual machines may be connected to each other by networks. Distributed programming facilities, remote message passing, and object migration are provided. Since AL-1/D has a meta-level architecture, we can program the location control facility specified to the meta-level thus taking into consideration the separation of concerns described previously. We can also achieve flexible location control using runtime information. Some examples of meta-level programming are presented.

4.1 Revised Base-level Program

Unlike programs in traditional DPL (Fig. 2), AL-1/D programs are separated into a base-level program (Fig. 5) and a meta-level program (Fig. 6). The "**base**" expression as in the line 1 of Fig. 5 is used to depict a base-level object, which in this case is the `Searcher` object. In this program, location control is separated from the search algorithm. That is, the program becomes location-independent and can be understood easily, because there is no extra algorithm for location control included.

```
 1 [ base Searcher
 2   (delegation nil)
 3 ]
 4
 5 [ method Searcher search: nm with: sec with: age
 6   (vars data strm)
 7   while (data != nil) do
 8     data = dmgr next
 9     if (((data name) isMatch: nm) && ((data section) isMatch: sec) &&
10         ((data age) <= age)) then
11         if (strm == nil) then
12           strm = data dataToStream
13         else
14           strm add: (data dataToStream)
15         end;
16         dnum = dnum + 1;
17         if (dnum >= bufnum) then
18           return strm
19         end
20     end
21   end;
22   return "done"
23 ]
```

Fig. 5. Revised Base-level Program

4.2 Encapsulated Location Control Facilities at Meta-level

Next, we discuss meta-level programming in AL-1/D. A meta-level program shown in Fig. 6 encapsulates location control of the program in Fig. 2. This meta-level program uses the *DE (Distributed Environment) model*, a meta object provided by AL-1/D. The DE model is a meta-object that represents the distributed environment and consists of object location information, a name server object, remote host information and a network. The DE model is mainly responsible for describing remote message sending.

Lines 1 to 7 show the definition of the meta-object `DESearcher` that is the DE model of the base-level object `Searcher`. The `vars` entry in lines 3 to 6 shows that `DESearcher` has `localHost`, `nameServer`, `remoteInfo`, and `network` as its internal variables, called the *meta-level variables*. These variables are bound

```
 1 [ meta DE DESearcher Searcher
 2   (delegation nil)
 3   (vars localHost          /* object location */
 4         nameServer         /* name server  */
 5         remoteInfo         /* remote information */
 6         network            /* network */  )
 7 ]
 8
 9 [ method DESearcher ssend: rcvr message: msg
10   (vars rcvrLocation info)
11   rcvrLocation = nameServer location: rcvr;
12   if (rcvrLocation != localHost) then
13     info = remoteInfo at: #remotesend; /* Count remote messages */
14     remoteInfo at: #remotesend put: (info++);
15     /* if (number < 30), migration starts */
16     if (info > 30) then
17       state migrating: rcvrLocation;      /* migration */
18       (rcvr meta: #request:) request: msg
19     else      /* else, remote sending */
20       network send: rcvr  message: msg  host: rcvrLocation
21     end
22   end
23 ]
```

Fig. 6. Migration based on Number of Messages

to the components of the base-level object `Searcher` when reification occurs. The method "`ssend:message:`" in the DE model defines its behavior when sending a remote message. When a message send is initiated, this method is invoked by a message which contains a receiver object and a message object as arguments. The location control facilities can be combined with the monitoring facility for remote messages. This combination allows runtime information obtained from the location control mechanism to be used. Fig. 6 defines the method `ssend:message:` of the object `DESearcher`, the DE model of the object `Searcher`. This method counts the number of remote messages. If the number exceeds the maximum 30, the base-level object migrates. The number of remote messages is stored in the `#remotesend` entry in lines 13 to 14. This number is used in line 16. The "`state migrating:`" expression moves a base-level object to the remote host specified in the argument.

Although there are many lines of code in this program, the essential code is located in lines 12 to 18. This program was constructed by modifying the program of the default object behavior (see Fig. 7 which is described later).

We now show some examples of changing the optimization policy by modifying the meta-level program. In systems with the meta-level architecture, the code needed for optimization part can be modified without changing the base-level computational algorithm. In Fig. 7, the method `ssend:message:` is modified so that the base-level object `Searcher` does not migrate to remote hosts. This is the default definition for the method `ssend:message:`. In Fig. 8, object location is controlled by the object names rather than the number of remote messages. If the receiver's name is "Special", a sender base-level object `Searcher` is moved

to the receiver object's host according to lines 7 and 8. Notice that the base-level program depicted in Fig. 6 does not have to be modified at all. In AL-1/D, since users can replace the methods of the DE model dynamically by compiling and loading source codes, the location control policy can be changed by the application program according to runtime information. This allows easy prototyping of the location control policy, which is a difficult task in traditional DPLs.

```
1 [ method DESearcher ssend: rcvr message: msg
2   (vars rcvrLocation)
3   /* Get location of receiver object */
4   rcvrLocation = nameServer location: rcvr;
5   if (rcvrLocation == localHost) then
6     (rcvr meta: #request:) request: msg    /* If local, send local message */
7   else /* If remote, send remote message */
8     network send: rcvr  message: msg host: rcvrLocation
9   end
10 ]
```

Fig. 7. Without Migration (default)

```
1 [ method DESearcher ssend: rcvr message: msg
2   (vars rcvrLocation)
3   rcvrLocation = nameServer location: rcvr;
4   if (rcvrLocation == localHost) then
5     (rcvr meta: #request:) request: msg
6   else
7     if ((rcvr name) == #Special) then /* Move, if receiver name is "Special" */
8       state migrating: rcvrLocation;
9       (rcvr meta: #request:) request: msg
10    else
11      network send: rcvr  message: msg host: rcvrLocation
12    end
13  end
14 ]
```

Fig. 8. Migration based on Receiver Name

4.3 Programming Location Control Facilities of DPLs in AL-1/D

Examples in this subsection illustrate that AL-1/D encapsulates the location control of traditional DPLs at the meta-level. We described parameter passing and "*object attachment*" policy in DPLs using the DE model of AL-1/D. These examples show that several location control facilities of DPLs are encapsulated at the meta-level of AL-1/D. Behavior equivalent to that of the program described in traditional languages can be encapsulated into the AL-1/D meta-level. This makes both programming ordinal computational algorithms and optimization easy. By sharing meta-objects (e.g. through a delegation mechanism), we can also reuse the optimization policy with other application programs.

```
1 [ method DESearcher ssend: rcvr message: msg
2   (vars rcvrLocation args i obj)
3   rcvrLocation = nameServer location: rcvr;
4   if (rcvrLocation == localHost) then
5     (rcvr meta: #request:) request: msg
6   else
7     network send: rcvr  message: msg
8                         host: rcvrLocation;
9     /* Get and Move argument objects */
10    if ((msg selector) == #next:) then
11      for (i = 0) to (i < args size)
12            step (i++) do
13        obj = args at: i;
14        if (obj size > 1) then
15          obj move: rcvrLocation
16        end
17      end
18    end
19  end
20 ]
```

Fig. 9. Parameter Passing Using DE model

```
1 [ method DESearcher ssend: rcvr message: msg
2   (vars rcvrLocation)
3   rcvrLocation = nameServer location: rcvr;
4   if (rcvrLocation == localHost) then
5     (rcvr meta: #request:) request: msg
6   else
7     /* if remote, migration starts */
8     state migrating: rcvrLocation
9     /* after migration, local message send */
10    (rcvr meta: #request:) request: msg
11  end
12 ]
```

Fig. 10. Sender Movement Using DE model

```
1 [ method DESearcher ssend: rcvr message: msg
2   (vars rcvrLocation objList i)
3   rcvrLocation = nameServer location: rcvr;
4   if (rcvrLocation == localHost) then
5     (rcvr meta: #request:) request: msg
6   else
7     objList = base readMVal: #ivars;
8     state migrating: rcvrLocation;
9     (rcvr meta: #request:) request: msg;
10    /* Get and Move internal objects */
11    for (i = 0) to (i < objList size)
12                  step (i++) do
13      (obj at: i) move: rcvrLocation
14    end
15  end
16 ]
```

Fig. 11. Object Attachment Using DE model

First, we describe the parameter passing facility. We restrict the description granularity using *call-by-move* semantics similar to that in Emerald [8][2], which allows programmers to specify a move primitive at any place in the base-level program. In our parameter passing facility, however, the programmer chooses the objects which can migrate. That is, based on the kind of the message selector and the argument object, it is possible to decide whether an argument object is to be piggy-backed on the remote message which is sent by the sender object. Our purpose for the parameter passing description is to reduce the number of remote references after migration rather than to control data movement from an application program. We believe parameter our passing facility is sufficient for object location control, because we can control the migration cost of argument objects and the number of remote references according to migrating object size and the kinds of receiver objects. Though parameter our passing facility restricts object location control, we emphasize that meta-level programming separates location control from the base-level program. Fig. 9 defines parameter passing by meta-level programming for the method `ssend:message:`. Lines 10 to 18 are different from the default definition. The DE model retrieves argument objects from the message object `msg` as an array object. If the message selector is "`next:`" and the size of an element object included in the argument array is greater than 1, the element object initiates the move by sending the "**move:**" message[3](lines 12 to 17).

In the Fig. 10, if the receiver object is at a remote host, the sender object migrates to the receiver's host. We call this movement "*sender movement*". In line 8, "`state` statement" is used to initiate the migration of the base-level object `Searcher`. Using the state statement, we can change the state of the base-level object. When the base-level object is moved by a meta-object such as the DE model, its state changes [4]. In Fig. 10, the object state changes into "`migrating`", and the base-level object starts migration to the destination host.

Fig. 11 defines *object attachment*. The variable `objList` is an array object listing attachment objects. In existing DPLs, an expression to define `obj1` and `obj2` as objects attached to `base-obj` would be represented by adding `obj1` and `obj2` to `objList` in the DE model of `base-obj`. In Fig. 11, all internal variables of the base-level object are moved along with the sender object, which means all internal variable objects are attached to the sender object. In line 7 of Fig. 11, an

[2] Emerald provides two parameter passing modes: *call-by-move* and *call-by-visit*. The argument object may return to the source host of a call in call-by-move mode, and remain at the destination host in call-by-visit mode.

[3] To move an object except for the base-level object, we use the explicit description using the "**move:**" message. When object `obj` receives a "**move:**" message, it migrates to a remote host specified by the parameter. In AL-1/D, object movement is represented by another model called the *Migration model*. This model converts an object into a message packet and sends the packet to the network. Object migration cannot be defined at the meta-level but at the base-level in AL-1/D

[4] Object migration of the base-level object is defined as a state transition to unify behavior at the meta-level. A model uses state transitions to it check the object behavior is reflected consistently [14].

internal variable slot of the base-level object is reified as an array object. Here, we use the model, called the *Operation model*, to reference an internal variable slot defined in a base-level object. A programmer can use the special message `readMVal:model:` to reference the *meta-level variables* of each model. We can program methods for attaching several other internal objects to the base-level object to be moved.

5 Performance Evaluation

In this section we present our evaluation of the object location control facilities in AL-1/D. To control object location, our AL-1/D code invokes meta-objects. As convenient as it may be, this is not very efficient because of the overheads incurred by *reification* and *reflection*. *Reification* is the behavior exhibited when a meta-object is invoked and the actual computational state contents at the meta-level are retrieved. *Reflection* is the behavior exhibited when a meta-object finishes execution and its modifications are reflected to the actual computational state. Although description with meta-level programs is useful for location control, we must also consider the trade-off between the benefits of object location control and the drawbacks of meta-level programming which present an overhead. To do so, we evaluate an application program in AL-1/D, using a *distributed search system* as an example.

5.1 Application Program: Distributed Search System

The distributed search system searches for data in databases distributed on several hosts, according to keywords input by the user. The configuration of this application is shown in Fig. 12.

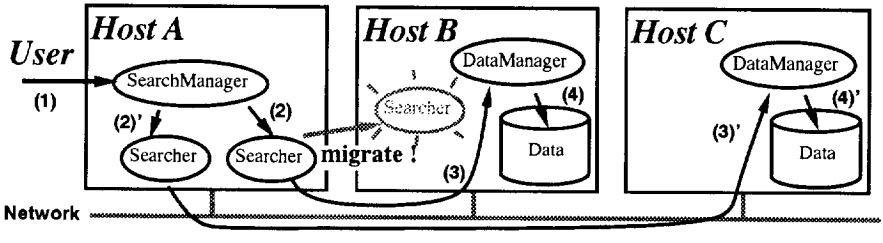

Fig. 12. Distributed Search System

This system consists of three kinds of objects: `SearchManager`, `Searcher`, and `DataManager`. These objects are responsible for the following tasks:

(1) The `SearchManager` object provides a user interface. This object receives keywords from the user and exists on the user's host, *Host A*.

(2) When `SearchManager` receives a request to search, it simultaneously invokes several `Searcher` objects. The number of `Searcher` objects corresponds to the number of hosts that contain databases. In Fig. 12, since there are two such hosts in our example: *Host B* and *Host C*, two `Searcher` objects are invoked. Synchronization between `Searcher` objects is managed by the `SearchManager` object.
(3) The `Searcher` object accesses the `DataManager` object on the host with databases.
(4) The `DataManager` object provides an interface between the `Searcher` object and databases. The `DataManager` object does not migrate to any other host and retrieves data from the database.

The description of these objects' behavior is location-independent at the base-level. In addition, we control object location by meta-level programming. There are several ways to control location. In this evaluation, we control object location by considering remote message sending between the `Searcher` and `DataManager` objects. To evaluate this, we employ programs described in Fig. 7, Fig. 10 and Fig. 6 as meta-level programs of the DE model of the `Searcher` object. The `Searcher` objects migrate to other hosts at appropriate times.

5.2 Basic Performance

Communication and migration costs were measured on Sun SPARCStation 2 workstations connected via 10 Mbps Ethernet. Each workstation has a 40 MHz SPARC CPU with 32 MB of physical memory. A virtual machine with the same specification was executed on each workstation, using SunOS as the platform. TCP/IP was used for the network protocol.

Communication Cost

Communication costs between sender and receiver objects were measured. We made comparisons of: (1) local and remote message passing and (2) whether the DE model is defined, that is, whether meta-level programming is used or not. A receiver object only returns an integer value when it receives a message. The result is the average value of total time when 10,000 messages are sent and received. The message packet size is fixed, regardless of whether the DE model is defined. To analyze remote message sending results, we also measured communication cost using a packet with the UNIX socket interface, whose size is the same as that of a message packet in AL-1/D. The results are shown in Table 1. The "ratio" depicted in this table is the ratio of the communication cost with the DE model to that without the DE model.

The difference between remote message sending and UNIX socket communication is 1.32 milliseconds. This is the time required for creating a message packet at the sender host and regenerating it at the receiver host.

When users do not write a meta-level programs, the AL-1/D virtual machine executes internal codes by default (without invoking meta objects) considering

efficiency. This enables AL-1/D to send local messages as quickly as ConcurrentSmalltalk [20], a concurrent object-oriented programming system without the meta-level programming facility. When the DE model is defined, local message sending becomes 20 times more expensive. There are three reasons for this: generation of internal objects of the DE model, context switching required to invoke the DE model, and execution of code defined in the DE model. Our present implementation of AL-1/D has not been completely optimized. We expect that optimizing object creation will reduce the overhead behalf. On the other hand, for remote message send, the overhead incurred by meta-level programming is low, 30% of the cost without the DE model, because much more cost is necessary for remote message sending.

Table 1. Communication Cost

	A: time (ms) without meta	B: time (ms) with meta	ratio (B/A)
Local	0.04	0.87	21.2
Remote	2.60	3.40	1.3
UNIX socket	1.28	-	-

Table 2. Migration Cost

	size (byte)	time (ms)
(i) Only `Searcher`	364	7.10
(ii) `Searcher` + DE (dormant)	608	8.64
(iii) `Searcher` + DE (running)	4494	15.20

Migration Cost

The migration cost of a `Searcher` object is shown in Table 2. The cost includes the time taken to receive an acknowledgement from the destination host. The first column in this table indicates three cases: (i) migration of the dormant `Searcher` object without the DE model, (ii) migration of the `Searcher` object with the DE model, but the `Searcher` object and DE model are both dormant, and (iii) migration when DE model is running. Execution contexts are packed into the migration stream[5]. (iii) is for object when migration occurs while executing the "*sender movement*" program (Fig. 10). The migration stream in (iii) is bigger than (i) or (ii) because it includes several messages and execution contexts. Analyzing (iii), we found that it takes the same time needed to send

[5] Only the current context migrates. Contexts which invoke the current context do not migrate.

messages on the network for migration stream generation and object regeneration.

5.3 Application Program Evaluation

Search time of the distributed search system was measured by modifying the number of data items stored in remote databases. We used several location control strategies. Data items in the databases consists of three character strings. Users can query by using three keys. Each key specified by one character is the first character of the value. If there is any matched data, the `Searcher` object returns the string that is generated from matched data to `SearchManager`.

The `Searcher` object scans the database as long as matched data exists. When all the data is scanned, the search completes. To simplify the comparison, queries were made so that the `Searcher` object returns only the first two matched data items matched. The total size of the sent packet is fixed when the number of returned data items is fixed.

Table 3. Measurement Factors

	meta-object	sender migration	# of message
(a)	–	–	–
(b)	**DE**	–	–
(c)	**DE**	**YES**	–
(d)	**DE**	**YES**	**COUNT**

We measured four situations of execution as shown in Table 3. In (a) and (b), no objects migrate. (a) is the result obtained when `Searcher` executes without meta-level programming in the DE model. (b) is the result obtained when the program in Fig. 7 is used. That is, the `Searcher` and `DataManager` send remote messages to each other. Comparison of (a) and (b) shows the overhead incurred by meta-level programming. (c) is the result of executing the "sender movement" program in Fig. 10. As expected, (c) shows improvement over (b). (d) is the result of a more flexible location control facility than (c). In (d), the number of remote messages is counted and the sender object moves, if the number of messages exceeds a programmed value. We expect (d) to result in a further performance improvement than (c). Results are shown in Fig. 13.

First, we compare (a) and (b). Since (b) includes overhead from using the DE model, (b) takes more time than (a). Next, let us concentrate on the results of (b) and (c). Both search times increase with the number of data items, although (b) increases at a greater rate. This is because, for (b), if `Searcher` does not move, as the number of data items increases, the number of remote messages between `Searcher` and `DataManager` increases. For (c), however, the number of remote messages does not increase once `Searcher` migrates to the `DataManager`'s host.

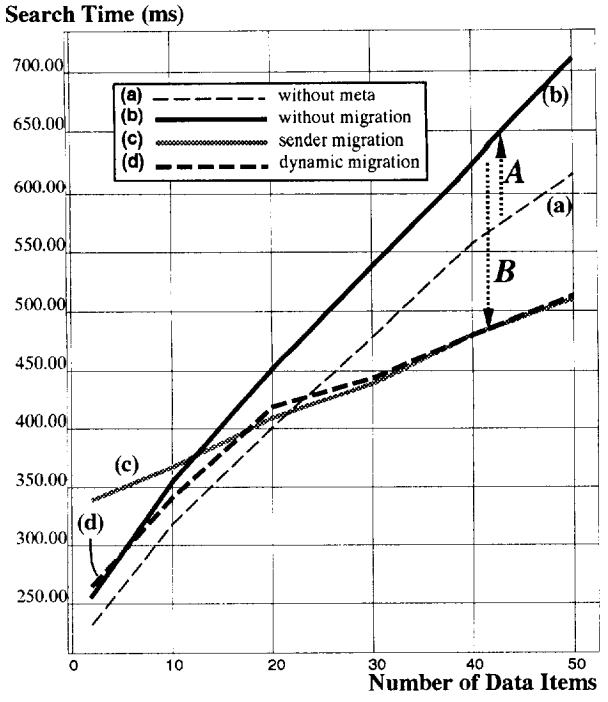

Fig. 13. Search Time and Number of Data Items

There is a certain threshold for numbers of data items where the search time of (c) is greater than that of (b). This threshold can be determined for numbers of data items less than 10. This happens because the results are greatly influenced by migration cost. Using this characteristic in our distributed search system, we can create a flexible meta-level program. Let us consider the case where a user randomly queries multiple databases. Each database can have up to 50 data items. To minimize the average search time, we can use the program of the DE model described Fig. 6. This program lets the `Searcher` object move if the number of remote messages is greater than the user-defined maximum value. When the number of data items in a database is 10, we found this value to be 30. The result with a threshold of 30 is indicated by (d) in Fig. 13. We call this migration mechanism *dynamic migration*. This method result in better average search times than those in (b) or (c).

Performance of (b) relative to (a) drops because of the overhead incurred by meta-object invocation. However, by meta-level programming in (c) and (d), the performance improvement gained by location control (shown as $\downarrow B$ in Fig. 13) is greater than the overhead incurred by meta-object invocation (shown as $\uparrow A$). The overhead incurred by using meta-objects is generally large. In our ap-

plication, however, this overhead has small influence because the cost of remote message send and object migration is as great as the overhead shown in Tables 1 and 2. Although the network performance will improve, this result will not change because of the costs incurred by migration stream creation and object regeneration. On the contrary, we expect performance improvements within a host and efficient implementation techniques of meta-level architecture will reduce the overhead incurred by meta-level programming. Our results in this section show object control by meta-level programming efficient as well as increasing programmability. Location control combined with the message handling mechanism is also useful for efficiency.

With these results, we can define the *characteristics* of our distributed search system relating to the DE model. These *characteristics* consist of information such as communication costs, the number of data items in databases, the number of matched data items, etc. If such *characteristics* are available, it becomes easier to make programming decisions such as whether to write meta-level programs and how to define meta-objects.

6 Related Work

In this section, we discuss related work, comparing our work to other distributed reflective systems. ABCL/R2 [12] is a reflective programming system based on the *Hybrid Group Architecture*. The grouping concept is used to represent distributed environments. In ABCL/R2, modification of object location is defined as migration among groups. A group can be the unit for resource sharing and can represent a host. ABCL/R2 employs object migration to modify the execution environment of an object. The system does not exhibit concrete components and strategies for object mobility. Thus, to our knowledge, ABCL/R2 does not support object location control as described in this paper.

The Apertos OS [19] incorporates the concept of *meta-space* corresponding to a group of ABCL/R2 and supports the object migration facility. However, since object location control facilities are shared by the members in a meta-space, it is difficult to describe the facilities which should be defined in every object such as *object attachment*. Also, Apertos does not yet supported a high-level programming language like AL-1/D for writing application programs. In both these systems, the trade-off between the improvement of efficiency gained by using location control and the drop in efficiency incurred by employing meta-level programming based on the evaluation as described in Section 5 has not been studied.

OpenC++ [4] is a reflective programming language, which has no interpreter, based on C++. In OpenC++, we can marshal argument objects into the remote message by changing the message sending mechanism. The message sending mechanism, however, is modified in every class rather than in every object. Also, it is difficult for OpenC++ to change the migration policy while the application program is running.

7 Summary

This paper presented the advantages of incorporating an object location control facility using meta-level programming into distributed systems. Object location control is a practical application of meta-level programming on a reflective system.

Meta-level programming provides the following benefits to improve programmability. It separates code for efficiency from the application program and clarifies the programming flow. Programs can be tuned based on object location without changing the basic computational algorithm through the capability of separation of concerns between the base-level and the meta-level. It also provides flexible location control facilities using runtime information such as the number of remote messages. These facilities are more flexible than those provided by traditional DPLs.

Our object location control facilities have been implemented in the AL-1/D distributed reflective programming system. We discussed the meta-level components needed to control object location and showed several useful examples of meta-level programming. Meta-level programming in AL-1/D enables description of facilities similar to those provided by traditional DPLs, such as "parameter passing" and "object attachment". With AL-1/D, we can also support the location control facilities combined with the message handling mechanisms based on application program characteristics. For example, we can describe object location control based on the number of remote messages and the receiver object's names.

In AL-1/D, the location control facilities can be combined with message handling mechanisms based on application program characteristics. For example, we can describe object location control based on the number of remote messages and the name of receiver objects.

By evaluating an application system created on AL-1/D, we discussed the trade-off between performance optimization from using location control and the overhead incurred by meta-level programming. As a result, in a distributed environment where we can use remote message sending and object migration, location control by meta-level programming is efficient while improving programmability. If programmers take this trade-off into account, meta-level programming can be effective.

Acknowledgements

We would like to thank Professor Mario Tokoro and Atsushi Shionozaki at Keio University, and Gregor Kiczales and members of his group at Xerox Parc for their helpful comments on earlier drafts of this paper.

References

1. Y. Artsy and R. Finkel. Designing a Process Migration Facility - The Charlotte Experience. *IEEE COMPUTER*, 22(9):47–56, October 1989.

2. J. Bacon and K.G. Hamilton. Distibuted computing with RPC: the Cambridge approach. In *Proceeding of IFIP Conference on Distibuted Computing*, North-Holland, October 1987.
3. John. K. Bennett. The Design and Implementation of Distributed Smalltalk. In *Proceedings of ACM OOPSLA'87*, pages 318 330, October 1987.
4. Shigeru Chiba and Takashi Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. In *Proceedings of ECOOP'93*, July 1993.
5. D. Friedman and M. Wand. Reification: Reflection without meta-physics. In *Proceedings of ACM Symposium on Lisp and Functional Programming*, pages 348–355, August 1984.
6. A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison Wesley, 1983.
7. Sabine Habert and Laurence Mosseri. COOL: Kernel Support for Object-Oriented Environments. In *Proceedings of ECOOP/OOPSLA'90*, pages 269–277, October 1990.
8. E. Jul, H. Levy, N. Hutchinson and A. Black. Fine-grained mobility in the Emerald system. *ACM Transaction of Computer Systems*, 6(1):109–133, February 1988.
9. G. Kiczales, J. Des Rivieres and D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.
10. Gregor Kiczales. Towards a new model of abstraction in software engineering. In *Proceedings of the International Workshop on New Models for Software Architecture'92 Reflection and Meta-level Architecture*, November 1992.
11. Pattie Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of ACM OOPSLA'87*, pages 147–155, 1987.
12. H. Masuhara, S. Matsuoka, T. Watanabe and A. Yonezawa. Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently. In *Proceedings of ACM OOPSLA'92*, October 1992.
13. 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.
14. Hideaki Okamura, Yutaka Ishikawa and Mario Tokoro. Metalevel Decomposition in AL-1/D. In Nishio and Yonezawa, editors, *Proceedings of the International Symposium on Object Technologies for Advanced Software*, pages 110–127, Springer-Verlag, November 1993. Lecture Note in Computer Science, No. 742.
15. Michael L. Powell and Barton P. Miller. Process Migration in DEMOS/MP. *ACM Operating System Review*, 17(5):110–119, October 1983.
16. B. C. Smith. *Reflection and Semantics in LISP.* Technical Report CSLI-84-8, Stanford University Center for the Study of Language and Information, 1984.
17. Mario Tokoro. Computational Field Model: Toward a New Computing Model/Methodology for Open Distributed Environment. In *Proceedings of the 2nd IEEE Workshop on Future Trends in Distributed Computing Systems*, September 1990.
18. Takuo Watanabe and Akinori Yonezawa. Reflection in an Object-Oriented Concurrent Language. In *Proceedings of ACM OOPSLA'88*, pages 306–315, 1988.
19. Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proceedings of ACM OOPSLA'92*, pages 414–434, October 1992.
20. Yasuhiko Yokote and Mario Tokoro. The Design and Implementation of ConcurrentSmalltalk. In *Proceedings of ACM OOPSLA'86*, pages 331–340, 1986.