Debugging Concurrent Systems Based on Object Groups

Yasuaki Honda[†]
Akinori Yonezawa

Department of Information Science,
Tokyo Institute of Technology,
Ookayama, Meguro-ku, Tokyo 152, Japan

Abstract

This paper presents a debugging method for Concurrent Object-Oriented Systems. Our method is based upon a new notion called Object Groups. An Object Group is a collection of objects which forms a natural unit for performing collective tasks. An Object Group's Task differs from C.Manning's *nested transaction* which is based on the nested request-reply bilateral message passing structures. Each Object Group's Task permits more general message passing structures. The language constructs which specify and use Object Groups have been introduced into an object-oriented concurrent language ABCL/1. The paper also describes ABCL/1's debugging tools based on Object Groups.

1 Introduction

In an object-oriented concurrent system, a number of objects are working concurrently and they communicate with each other by sending messages. Debugging programs on such a system is usually more difficult than debugging ones on a sequential system because bugs or errors may exist not only in individual behavior of each object, but also in communications and synchronization with each other.

As a basis for a debugging scheme for concurrent systems, it has been proposed to record all the events in which relevant objects are involved. The record of each object is often called a *task record* ([Manning 87]) or a *local history* ([Honda 86]). Since the data recorded in local histories are

[†]The author's current address is: Fuji Xerox, Co., Ltd. Akasaka, Tokyo, Japan.

low-level ones and not well-structured, it is often hard to make sense out of them directly. Based on the notion of *nested transactions*, C. Manning proposed a scheme to extract useful information for debugging from task records and implemented it as a part of his debugging system called, an Apiary Observatory[Manning 87]. A nested transaction comprises nested pairs of *request* and *reply* messages. His scheme is very powerful for debugging a system in which message passing structures are always bilateral with pairing of a request and a reply (the call/return pair).

In general, however, message passing structures in object-oriented concurrent systems[Yonezawa 87] do not necessarily conform to the request-reply bilateral protocol. To cope with such general message passing structures, the notion of *Object Groups* we propose provides a promising approach to debugging. (In this approach the data recorded in the local histories of individual objects are also the primary source of information.)

Generally, an Object Group reflects an abstraction of various problem and solution structures. When the programmer writes a large scale (object-oriented concurrent) system, s/he usually has in his/her mind a structure or an abstraction of the system. Currently proposed object-oriented concurrent languages do not provide the programmer with explicit structuring mechanisms: such structures or abstractions can only be expressed by using the implicit "knows-about" relationship among objects which is established by storing objects' names in state variables of objects.

An Object Group is a collection of objects which forms a natural unit for performing collective tasks. For example, consider to model a company in a concurrent object-oriented manner. A company basically consists of persons who are employed by the company. When the company is requested to perform some task, members of the company perform actions in cooperation and communicate with each other. Naturally, each person is modeled as an object. Actions performed by each person are simulated by procedures associated corresponding objects, and of course, communication among persons is simulated by message passing. But, should the company be modeled as an object? In fact, the company itself does not perform any task, but all the tasks are carried out by (a collection of) individual persons in the company. Thus it is often useful to model the company simply as a group of objects.

In this paper, we introduce the notion of Object Groups and its language constructs in our language ABCL/1[Yonezawa 86], which can be used to explicitly describe abstractions of structures of a large concurrent system. The rest of the paper mainly describes the applications of Object Groups to our debugging scheme and visual execution monitoring system. Also other debugging tools provided in ABCL/1 will be briefly presented. Note that the usefulness of the notion of Object Groups does not limit itself to debugging and monitoring: it is of course powerful in other activities such as designing systems and processor allocation as will be briefly discussed in Section 6.

2 The Computation Model and the Language ABCL/1

2.1 The Computation Model

The object oriented computing model assumed in this paper is a simplified version of our proposed model [Yonezawa 86]. In the model, each object has its own computing power and can perform its activity concurrently with the other objects. The computation model allows neither shared memory nor global clock. A group of objects forms a sparsely connected system.

An object has its own internal world, which consists of a local persistent memory and procedures inquiring/updating the local memory. They are called *state* and *script*, respectively. We assume that an object has its own local clock. The internal world of an object cannot be directly accessed from the other objects.

Objects interact with one another via *message passing*. In response to a message, an object executes one of the procedures in its script. Execution of a procedure (script) by an object consists of a sequence of actions of the following kinds:

- inquiring and updating the *state* of the object,
- creating new objects,
- sending and accepting messages, and
- returning a value as a reply to a received message.

Messages arriving at an object will be processed one at a time in a sequential manner.

There are two types of message passing, *asynchronous* and *synchronous*, which are called *past-type* and *now-type*, respectively. Immediately after sending a message asynchronously, an object can continue its computation. The sender object of an asynchronous message transmission expects no reply. In contrast, the sender object of a *synchronous* message transmission cannot resume its computation until the reply to this message arrives.

When a message is sent to an object to request the object to do some task, it is often useful to be able to specify where the result or reply of the requested task should be sent. This information is called *reply-destination* in our computation model (See [Yonezawa 86] for more details). Each synchronous message transmission implicitly specifies the reply-destination. The receiver object of a synchronous message transmission can forward the reply-destination to another object which is responsible for returning a reply. Note that the reply-destination specified by a synchronous message transmission can be forwarded to more than two objects. Some of them may return replies to the reply destination. In this case, the first message returned is considered as the reply for the synchronous message transmission.[1]

We assume that messages satisfy the *transmission ordering law*: Suppose that two messages M and M' have the same sender and the same receiver. If M and M' are sent in this order according

[1] This feature is used in the example of tree structured dictionary given in Section 3.1.

to the local clock of the sender, they are received by the receiver in the same order. In general, if M and M' are sent from the different objects and received by the same one, their arrival order cannot be determined.

2.2 The Language Constructs

In the language ABCL/1, an object is defined by the following form:

```
[object <object-name>
  (state [<state-variable> := <initial-value>] ...)
  (script
    (=> <message-pattern> @ <variable>  <behavior-description>)
                          ⋮
    (=> <message-pattern> @ <variable>  <behavior-description>))]
```

The <state-variable>s of an object are the variables which represent the internal *state* of the object. An object defined by the above form accepts a message which matches against some <message-pattern>. When a message matches against the pattern, the reply destination of the message (if it exists) is bound to the corresponding <variable>. After accepting a message, the object will perform a sequence of actions described in the corresponding <behavior-description>. The state variable declaration part and <variable> are optional.

In the <behavior-description>, sequential computation is written using Lisp-like forms and message passing and returning a reply are written in the following forms:

```
[<object> <== <message>]        ;; synchronous message passing (now-type)
[<object> <= <message>]         ;; asynchronous message passing (past-type)
[<object> <= <message> @ <reply-destination>]
                                ;; asynchronous message passing with reply destination
!<reply>                        ;; returning a reply
```

3 Object Groups in ABCL/1

3.1 Characteristics of Object Groups

In this subsection, we will discuss the characteristics of Object Groups. Possible operations on Object Groups and the tasks of Object Groups are illustrated by using a tree structured dictionary.

A tree structured dictionary is a dictionary represented by a binary tree. In a tree structured dictionary, each dictionary entry is represented as an object. This object corresponds to an entered word and remembers both its spelling and meaning. This object is called *a word object* and it is created by an object *CreateWord*.

The nodes of the tree are word objects. Each node may have at most two sub-trees. A word object, which is also a node of the tree, can accept messages `[:search <word>]` and `[:update [<word> <meaning>]]`. Upon receiving one of these messages, a word object compares `<word>` with its remembered spelling. If the spelling equals to `<word>`, then the meaning it holds is returned if the request is *:search*, or the word object updates its meaning if the request is *:update*. Otherwise, the word object sends the same message to one of its sub-trees according to the result of the comparison between the spell and `<word>`.

To ask the meaning of a word, a request message `[:search <word>]` is sent to the root of the tree. A tree search is performed inside the dictionary. If the word has already been entered, the meaning of the word is returned from the corresponding word object of the tree dictionary. When more than one *:search* request arrives, the search tasks are performed in parallel.

There is a tree-manager object which keeps the root word objeot of the tree structure. This dictionary also has a cache memory which remembers the last ten referred words. This cache memory is represented as an object called Cache. When the tree-manager object is requested to search a word, it sends `[:search <word>]` messages to both the root object of the tree structured dictionary and the cache object in parallel. (Figure 1)

The objects appeared in the tree structured dictionary example form a natural unit for performing collective tasks, such as searching or updating. In order to describe this structure explicitly, we can define an Object Group that consists of the tree-manager object, the cache object and word objects.

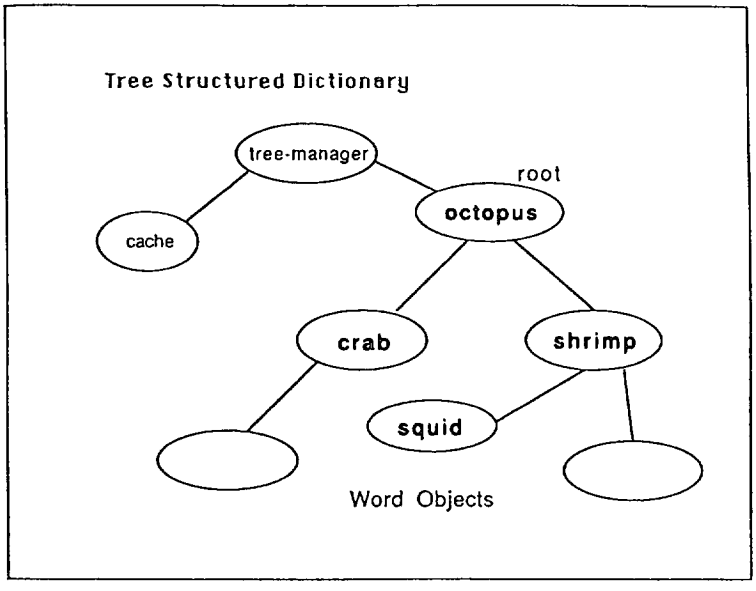

Figure 1: Objects in the tree structured dictionary

From this example, the following characteristics of the group of objects are observed.

- *Some members of the group are fixed, while some are not fixed and dynamically added to or deleted from the group.*

The tree-manager object and the cache object are the fixed members of the tree structured dictionary group. That is, these objects originally belong to the tree structured dictionary group since the group was created. In contrast, word objects are dynamically added to the tree structure when they are created. Some word objects may be removed from the tree structured dictionary group when words are removed from the dictionary.

- *Causally connected individual actions of objects in the group form a task of the group.*
- *Some of the group's fixed members represent the group as a whole and request messages sent to them often trigger the group's collective tasks.*

When a request `[:search <word>]` comes to the tree-manager object which represents the whole dictionary, it sends the same requests to the root node of the tree and the cache object in parallel. The meaning of the word is searched in both the cache and the tree in parallel and, if found, then it is returned. The individual actions of the objects form the searching task of the tree structured dictionary group. Note that if the cache hits, two replies are sent back, one from the cache and the other from the tree. (In this case the reply that arrived first is picked and the second one is ignored. See Section 2.1 for more details.) This kind of message passing structure cannot be conformed to the request-reply bilateral protocol.

Since these characteristics may be observed widely in various problem domains, the notion of Object Groups is useful in concurrent object-oriented systems. We incorporate the language constructs for Object Groups into our language ABCL/1.

3.2 Language Constructs for Object Groups

Object groups may be created dynamically during the execution of a program. We use the following notation to specify an *object-group* in ABCL/1:

```
[object-group <group-name>
  (fixed-members
    [<member-name> = <object-creation-form>]
                          ⋮
    [<member-name> = <object-creation-form>])
  (initialize
    <abcl-form> ...)]
```

Every object created by evaluating <object-creation-form> is a fixed member of the Object Group. Fixed members can be accessed by specifying associated <member-name>s and the name of the group they belong to. After all the fixed members are created, the <abcl-form>s are evaluated as the initialization of the group.

As an object-group form is evaluated, an object is created which manages and maintains the members of the group. The object is associated with <group-name>. Any request to an Object Group should be sent to this object.

The following is an example of a definition of an Object Group.

```
[object-group tree-dictionary
  (fixed-members
    [Cache = [Object
                (state [cache-entry := nil]
                       Manager)
                (script
                  (=> [:cooperate-with obj]
                      [Manager := obj])
                  (=> [:search word]
                      if word exists in cache-entry
                        then return its meaning in cache-entry
                        else do nothing.)
                  (=> [:update [word meaning]]
                      update the cache-entry.))]]
    [tree-manager =
        [Object
          (state [RootNode := [CreateWord <== :new]]
                 Cache)
          (script
            (=> [:cooperate-with obj]
                [Cache := obj])
            (=> [:search word] @ RD
                [[Cache RootNode] <= [:search word] @ RD])
                  ;; A message [:search word] and the reply destination RD
                  ;; are sent to both Cache and RootNode in parallel.
            (=> [:update data]
                [RootNode <= [:update data]]))]])
  (initialize
    [Cache <= [:cooperate-with Manager]]
    [Manager <= [:cooperate-with Cache]])]
```

This object-group form defines the tree structured dictionary described in Section 3.1. When this form is evaluated, the tree-manager object and the cache object are created as the fixed members of the group. Then the initialization forms are evaluated.

Every fixed member is accessed by specifying their name and the name of the object group that it belongs to. The following functions are provided for this purpose.

```
(group-member <group-name> <fixed-member-name>)
(fixed-member-name <group-name>)
```

On evaluating the first form, a fixed member of the group <group-name> which is associated with <fixed-member-name> is returned. On evaluating the second form, a list of the names of all

the fixed members are returned. Using the first form, for example, it is possible to get the meaning of a word by evaluating the following form:

```
[(group-member tree-dictionary 'tree-manager) <== [:search word]]
```

It is possible to add an object to an object group, or to delete an object from an object group by sending a request to the object group (in fact an object which manages the members of the group). Ordinary message passing forms are used with the following special message patterns:

```
[<object-group> <= [:add an-object]]
[<object-group> <= [:delete an-object]]
```

In the tree structured dictionary example, immediately after the creation of a new word object the following form should be evaluated to add it to the tree structured dictionary group. This form is evaluated in the CreateWord object, which creates the `new-word` object.

```
[tree-dictionary <= [:add new-word]]
```

4 Debugging Scheme based on Object Groups

4.1 Local History

As a basis for our debugging scheme, the ABCL/1 system records the local history for each object during the debug mode execution of a program. The local history of an object is a sequence of events that occur in the object. Sending or receiving a message and object creation are such events, and they are recorded as the local history as they occur. Information related to each event, such as the sender name and the message content is also recorded as the local history.

Each object keeps its local time. The local time of an event is also recorded in the local history together with the event. The local time is incremented by one unit time when the object sends a message or creates an object. When an object S sends a message M to an object R at local time Ts of S, Ts is also sent along with the message M as a time-stamp. When the object R receives this message, the sender S, the message M, and Ts are recorded in the local history of R. Suppose Tr is the local time of R immediately before it receives M. The local time of this receiving event is the maximum of $1 + Ts$ and $1 + Tr$.

By using the local time mechanism, the global history of the entire system can be re-constructed from the local event history recorded in each object, for the local history keeps a partial order of events. Note that the local history mechanism is similar to the *task record* of [Manning 87]. He used it for constructing nested transactions. But we will use the information obtained from the local history to define and construct the *Object Group's Task* described in the next section.

The ABCL/1 debugging system provides a tool called the ABCL/1 Inspector, which can be used to examine the local history of an object directly. More details of the tool will be given in Section 5.

4.2 Object Group's Task

An *Object Group's Task* is a collection of causally connected actions of individual objects in the group. When one of the object group's fixed members receives a message from outside the group, an *Object Group's Task* starts. The first message sent from outside the group to trigger the Object Group's Task is called an *entry message*. The receiver of the message causes message transmissions to other members of the group. Such other members, in turn, perform their actions, and so on. All the actions performed by the members of the group are included in the Object Group's Task. Sending messages to more than two members of the group may cause a branch of the Group's Task. Sending a message to an object outside the group or sending no messages terminates a branch of the task. An Object Group's Task is considered to be terminated when all the branches are terminated. Note that such an Object Group's Task cannot be captured by Manning's nested transactions.

Since Object Groups reflect the problem domain structure, Object Groups' Tasks reflect meaningful tasks in the domain. For example, when the tree-manager of the tree structured dictionary group receives a [:search word] message from outside, a task of the tree structured dictionary group begins. The tree-manager object sends a [:search word] message to the root node and the cache simultaneously. This causes a branch in the Object Group's Task. They start their actions upon receiving the [:search word] message. All the actions performed by the tree-manager, the root node, and the cache are parts of the group's task. If the cache does not know the meaning of the word, it does not send any message to other objects. So a branch of this group's task is considered to terminate at the cache. The root node, after receiving [:search word], sends the same message to its RightNode or LeftNode, then a tree search begins. If a word object which remembers the specified spelling exists, it returns the meaning of the word and the branch of the Group's Task terminates at the word object.

All information needed to trace an Object Group's Task is included in the local history of the members of that group.

4.3 Object Group Examiner

Object Group Examiner (OGE) is a tool to examine each task of an object group. Evaluating the following form activates the OGE:

```
(oge-start <object-group>)
```

OGE displays the following information on its sub-windows (Figure 2):

- *the fixed members* of <object-group>,
- *entry messages* of one of the fixed members,
- An *Object Group's Task* triggered by one of the entry messages, and
- *entered commands* for OGE

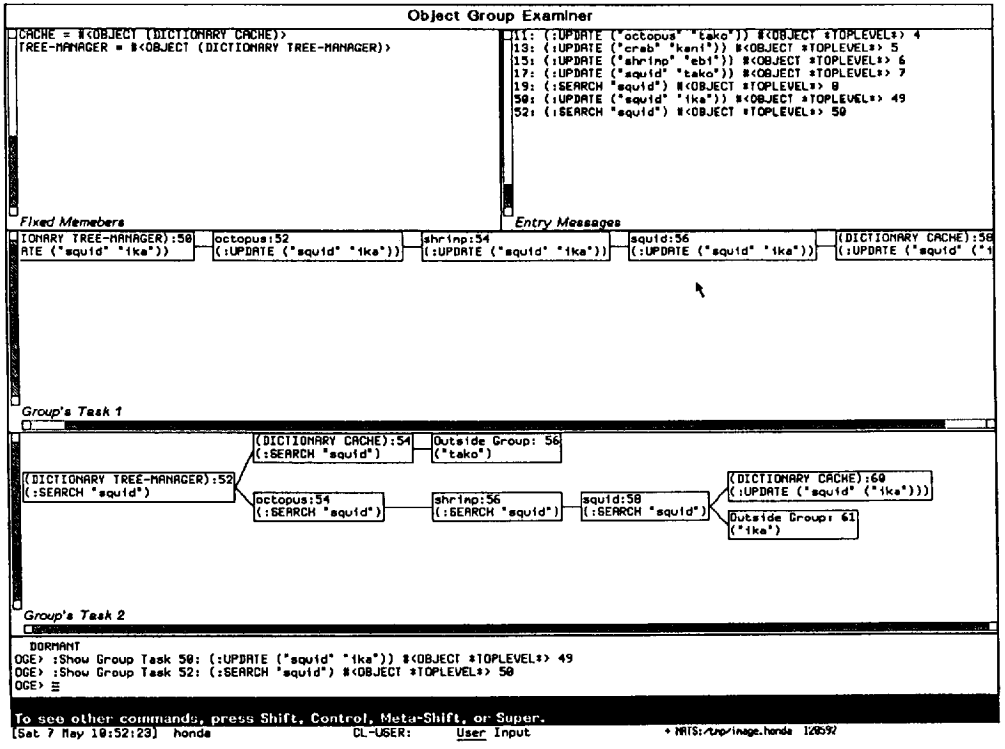

Figure 2: Object Group Examiner Window: the tree structured dictionary is examined.

On the upper left sub-window, all the fixed members of the tree dictionary group (*tree-manager* and *cache*) are displayed. These objects are mouse-sensitive and, by pointing one of them by mouse, entry messages of the selected object are displayed on the upper right sub-window which is called an *entry messages sub-window.*

Each line of the entry messages sub-window corresponds to each entry message and it includes *the received time*, *the message content*, *the sender*, and *the send time.* In Figure 2, entry messages of the tree-manager are displayed. Note that at 17, the tree-manager receives [:update ["squid" "tako"]] message. This message is wrong because "tako" does not mean a squid in Japanese.[2] To correct the wrong registration, at 50, [:update ["squid" "ika"]] is received. Each entry message is

[2]The word tako and ika mean octopus and squid in Japanese respectively.

also mouse-sensitive and, by pointing one of them, an object group's task invoked by the message is displayed in one of two windows which are labelled *Object Group's Task 1 and 2* in Figure 2. Two of Group's Tasks can be examined simultaneously on these windows.

On the *Object Group's Task 1* sub-window, the Object Group's Task triggered by [:update ["squid" "ika"]] received at 50 by the tree-manager is displayed. Each box appeared in the sub-window represents an individual action of an object invoked by receiving a message. For example, the left-most box represents the tree-manager's action invoked by receiving [:update ["squid" "ika"]] at 50. The tree-manager sends the same message to the root node of the dictionary. The second box represents the action of the root node (which is, in fact, a word object). The root node sends the same message to its appropriate sub-tree. The third box represents the action of the intermediate node between the root node and the node which remembers the meaning of "squid" (The action of the node is the same as the action of the root node). The fourth box represents the action of the node which remembers the meaning of "squid". It updates the meaning, then sends an [:update ["squid" "ika"]] message to the cache object. The last (right-most) box represents the action of the cache. It updates its local memory and sends no message to other objects. The *:update* task terminates at this point.

A more complicated Group's Task can be observed in the *Group's Task 2* sub-window. The Group's Task displayed on this sub-window is invoked by receiving the message [:search "squid"] by the tree-manager at 52. It is observed that after the tree-manager receives the *:search* request, it sends the same messages to both the cache and the root node in parallel. The cache sends a reply message "tako" (which is the wrong answer. See footnote.) to outside the group. Concurrently, a tree search occurs and the right answer "ika" is sent to outside the group. These two replies are sent back to the same reply destination and the first arrival is used as the reply. In this example, the wrong answer "tako" is used instead of "ika." This seems to be a bug in this program because the out-of-date meaning "tako" is returned for the *:search* request (at 52) which is sent after the *:update* request (at 50). The reason for the bug can be found by looking at the *Group's Task* sub-windows. In the *Group's Task 1* sub-window, the cache object receives the [:update ["squid" "ika"]] at 58, while the cache object receives the [:search "squid"] at 54. This *:update* action should have been completed before this *:search* action.

At this level, each action performed by each member object is not displayed in detail. To see the detail, a box which represents an individual action of an object should be pointed by mouse. The ABCL/1 Inspector is invoked to see the detail of that action which is a part of the object's local history. (See Section 5 for ABCL/1 Inspector.)

4.4 Visual Execution Monitor

We have been developing a tool called Visual Execution Monitor, which displays the changes of general status for each object in real time. The status of an object includes its name and current mode (*dormant*, *wait* or *active*). A programmer can designate additional information for each object to be displayed on Visual Execution Monitor. In the word object example, the spelling remembered by an object and its current mode can be seen on the display (Figure 3). The state variable *entered-word* of the word object, which holds the spelling of the entered word, is designated by the programmer.

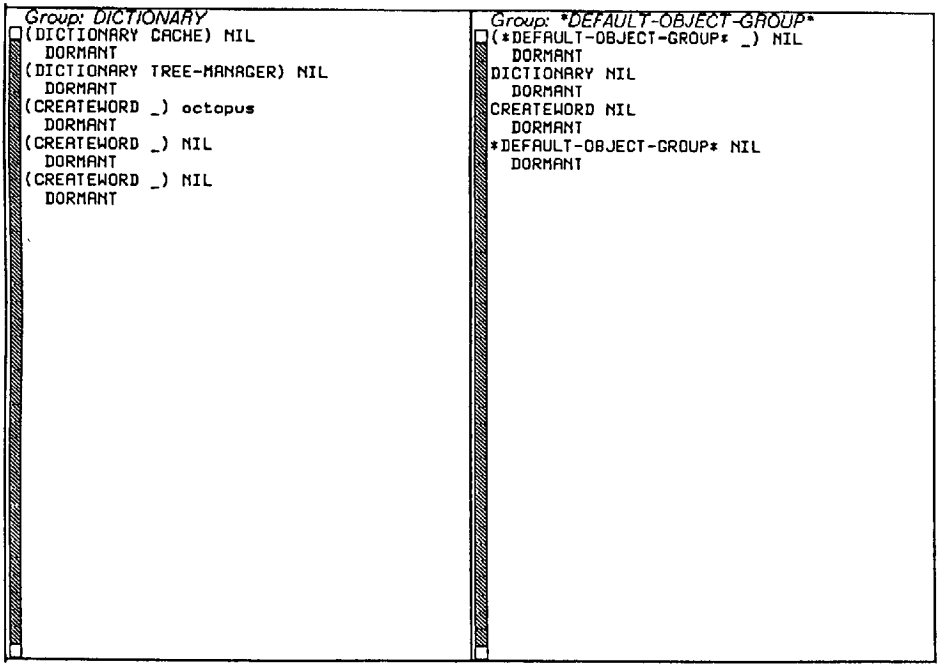

Figure 3: Visual Execution Monitor 1

Visual Execution Monitor uses Object Groups as the strategy of placing objects on the window. All the members of an object group is placed on a particular region of the window, so the dynamic changes of members during the execution such as addition and deletion can also be observed in real time. The window of the Visual Execution Monitor can be divided into more than two sub-windows to display many Object Groups simultaneously.

The left sub-window of Figure 3 shows the status of each object in the tree structured dictionary group after a word "octopus" is entered. The label displayed on top of the sub-window

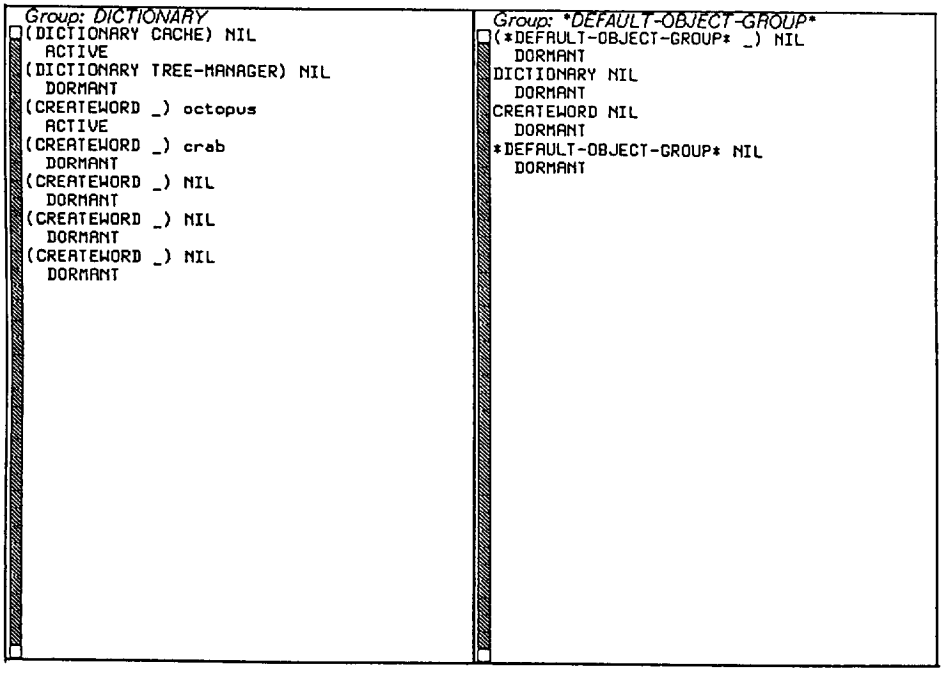

Figure 4: Visual Execution Monitor 2

shows the name of the group. Each object's information consists of two lines. The first two lines indicate the information about the cache object: the user designated information is nil (in fact, nothing is designated) and the current mode is *dormant*. The next two lines are the tree-manager's information. The information about a word object which remembers the word "octopus" is shown in the next two lines. There are two word objects which no words are entered. On Figure 4 is displayed the status of each object at a moment of carrying out the *:search* task. The cache and the root node (which corresponds to the word "octopus") are active at this moment.

J.Joyce et.al employed a process grouping technique to manage their graphical state display system called Mona [Joyce et.al 87], which is a part of their distributed system monitor. Processes are graphically displayed on a screen and, when a process changes its running state, the display is updated. A process group is created by pointing processes displayed on the screen using mouse. After a group is created, it can be repositioned, or can be incorporated into other groups as if it were an indivisible unit.

In Mona, processes are grouped by the instruction from the user monitoring system execution. In our scheme, Object Groups are specified by the program text. J. Joyce et.al use their process

grouping only to improve the display, while we use Object Groups not only to improve the display but also to extract debugging information (See Section 4.3).

5 Other ABCL/1 Debugging Tools

Our simplified computation model of ABCL/1 is based on both sequentiality within an object and parallelism among objects. Thus debugging a program written in ABCL/1 requires both a scheme suitable for parallelism and that for sequentiality. We have already discussed the former one in the preceding sections. To deal with sequentiality, the current ABCL/1 system has three major tools. They are based on the ideas of traditional debugging tools.

The ABCL/1 Inspector can be used to examine the general information about an object such as the value of state variables and the local history. Objects relevant to each event are recorded in the local history. In Figure 5, the local history of the cache object is displayed in the upper sub-window. In the lower left sub-window, variable values are displayed. All the message patterns acceptable to the cache object is displayed in the lower right sub-window.

The traditional trace and break-point facilities are also provided. The trace facility simply prints the event information when an event takes place. The trace-switch can be set on selected objects or every object involved in the system so that the programmer can selectively trace objects.

The break-point facility stops the program execution when a [break-point] form is executed in an object. The object which causes the break can be examined by using the break-point top-level commands or by invoking the ABCL/1 Inspector.

6 Discussions

We have introduced the notion of Object Groups and incorporated its language constructs into ABCL/1. Its application to the debugging scheme for concurrent systems have been presented. The Object Group Examiner was presented as a tool for observing Object Group's Tasks. Also the Visual Execution Monitor was presented which can be used to observe the dynamic changes of object groups and the status of component objects in real time.

The Object Group Examiner and the Visual Execution Monitor have been incorporated into our ABCL/1 environment and we have used these tools to debug ABCL/1 programs. One of interesting example programs is a context free grammar parallel parsing algorithm proposed by Yonezawa and Ohsawa [Yonezawa 88]. Our debugging tools are very useful to debug and monitor the program itself, and the grammar rules written for the parallel parser. Also in demonstrating the parallel parser to the people who have no idea of the algorithm, these tools are very helpful because they clearly display the cooperative behavior of objects in a very structured way.

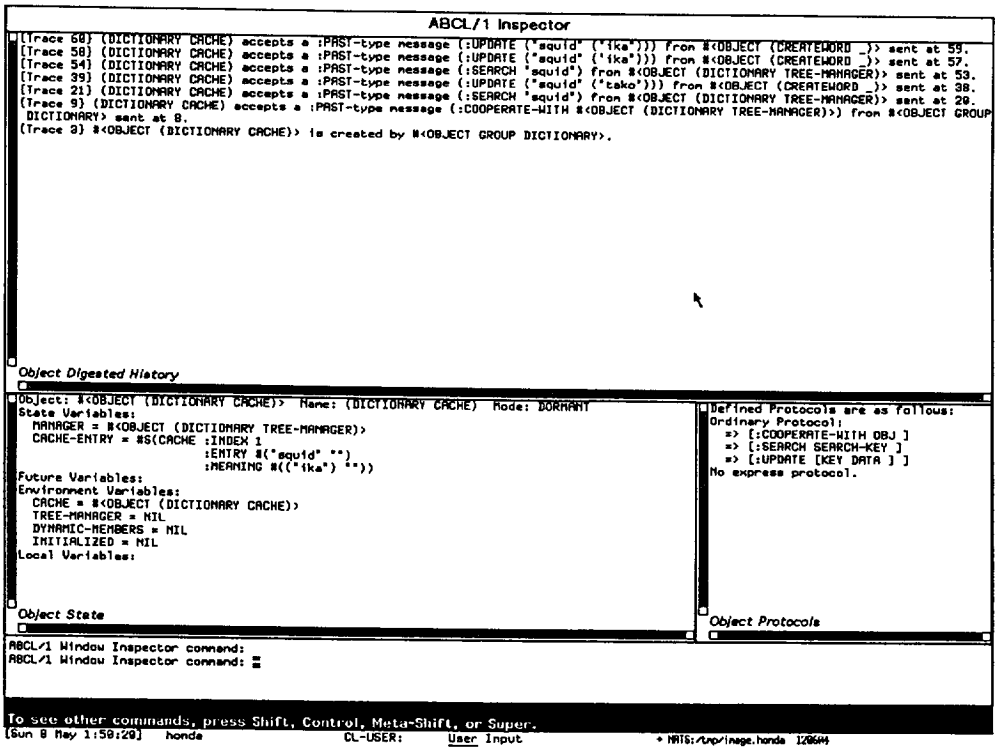

Figure 5: ABCL/1 Inspector: The cache is inspected.

Complementarily to these tools, we feel that tools are needed to present the overall behavior of a system (which might consist of multiple Object Groups). Such tools can be used to determine which Object Group's Task contains bugs. Note that the objective of these tools is the same as that of the behavioral abstraction approach given in [Bates and Wileden 83]. Bates and Wilden define an event description language. Using this language, they describe abstract events in a bottom up manner. In contrast, Object Groups' Tasks, which correspond to abstract events, can be defined more naturally by Object Groups.

The notion of Object Groups can be applied to areas other than debugging. For instance, it can be expected that objects included in an object group are so tightly connected that they communicate with each other more frequently than with objects outside the object group. This means that, in a distributed environment, to put all the members in an object group on a single processor may reduce the inter-processor communication which often degrades system performance.

We do not claim that the current design of Object Group is complete. It imposes several restrictions. For example, the current design does not allow the hierarchy of object groups. Every object must belong to only one object group. Furthermore, a member that ought to be accessed by *group-member* function must be a fixed member. The membership of an object and its access method is tightly connected. In the future design, these points need to be improved to provide more flexible and useful features.

7 Acknowledgment

We would like to appreciate various comments made by other members of the ABCL Project, E.Shibayama, T.Takada, and K.Sano.

References

[Bates and Wileden 83] P.C.Bates and J.C.Wileden.: High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach, *The Journal of Systems and Software 3*, 1983.

[Honda 86] Y.Honda and A.Yonezawa.: A Debugging Scheme for an Object-Oriented Concurrent Language, (in Japanese) *Proceedings of 3rd Annual JSSST Conference*, 1986.

[Harter et.al 85] P.K.Harter,Jr., D.M.Heimbigner and R.King.: IDD: An Interactive Distributed Debugger, *Proceedings of International Conf. on Distributed Computing Systems*, 1985.

[Joyce et.al 87] J.Joyce, G.Lomow, K.Slind, and B.Unger: Monitoring Distributed Systems, *ACM Trans. on Computer Systems*, Vol.5, No.2, 1987

[Manning 87] C.R.Manning: Traveler: An Apiary Observatory, *Proceedings of European Conf. on Object Oriented Programming*, 1987.

[Yonezawa 86] A.Yonezawa, J-P. Briot, and E. Shibayama: Object-Oriented Concurrent Programming in ABCL/1, *Proceedings of Object-Oriented Programming System, Languages and Applications*, 1986.

[Yonezawa 87] A.Yonezawa and M.Tokoro (Eds): *Object-Oriented Concurrent Programming*, The MIT Press, 1987.

[Yonezawa 88] A.Yonezawa and I.Ohsawa: Object-Oriented Parallel Parsing for Context-Free Grammars, *Proceedings of International Conf. on Computational Linguistics*, Budapest, August, 1988.