An Implementation of Case-based Memory of an Interface Agent by Lisp, Java, and C++

Seiji Koide, Ishikawajima-Harima Heavy Industries Co., Ltd.

Masanori Kawamura, Ishikawajima System Technology Co., Ltd.

An interface agent that mediates between an operator and a process plant was developed. The agent does not have expert knowledge but has cognitive mechanism for time series data and imitates elemental human cognitive process in plant operation. To realize the cognitive mechanism and the empirical learning ability of the agent, case-based reasoning (CBR) was adopted, and the Memory Organization Package (MOP) by Schank, *et al.* was utilized for the case memory. The agent can communicate with an operator in voice. A voice interactive system between an operator and the agent was also built with the Direct Memory Access Parser (DMAP) on top of MOP. In this paper, after introduction of MOP we describe the MOP implementation by the common lisp, Java, and C++, and report the comparison among them on usability and efficiency. This study was achieved as one of the Human Media Research and Development Projects under the Industrial Science and Technology Frontier program of Japan.

Key Words: Artificial Intelligence, Human Media, Interface Agent, Case-based Reasoning, and MOP

1. Introduction

These days the agent technology has been illuminated in many fields, including manufacturing [1]. In building an interface agent for process plant operation, we adopted not knowledge-based approach but human-centered approach [2, 3], where the agent does not have expert knowledge but have cognitive mechanism for time series data from process plants and imitates elemental human cognitive process in plant operation. The interface agent discriminates between experienced plant states and un-experienced states, and it detects abnormal states automatically if the agent has experienced them in the past and has been instructed that they were abnormal. This cognitive agent is able to learn from experience with teaching by an operator and grow its capability along with lifetime experience, starting without knowledge just as same as a human novice would become an expert. To realize the cognitive mechanism and the empirical learning ability, case-based reasoning (CBR) was adopted, and the Memory Organization Package (MOP) by Schank, *et al.* [4,5] was utilized. Hereafter, MOP means this case memory system instead of Meta-Object-Protocol.

The interface agent for plant operation mediates between a human operator and a process plant. The agent is able to interactively communicate with a human operator in voice [6,7]. An operator can command the agent itself in voice and other subsystems via the interface agent. The interface agent replies to the command of the operator by synthesized voice. The agent also explains what happens in the plant in voice.

Ishikawajima-Harima Heavy Industries Co., Ltd.
1-15 Toyosu 3-chome, Koto-ku, Tokyo 135-8732, Japan
TEL: +81-3-3534-3529, FAX: +81-3-3534-3510
email: seiji_koide@ihi.co.jp.

To realize voice interaction, off-the-shelf voice recognition system and text-to-speech voice synthesize system was equipped to the interface agent, and natural language processing system was built by utilizing the Direct Memory Access Parser (DMAP)[4] on top of MOP.

The micro MOP code had been described at the reference [4] by the subset of CLtL[8]. We rewrote it by the full set of CLtL2[9] and revised a part of code for the sake of efficiency. Java program of MOP was also coded for the use of our R&D project [10]. In other words, lisp program was for exploratory research of our own, and Java program is on account of Java base cooperative research project. Furthermore, C++ program using an object database ObjectStore was recently developed for the purpose of the practical use.

In this paper, we give the overview of MOP and the interface agent at first, then report our experience of MOP realization by the common lisp, Java, and C++.

This study has been done as a part of the Next Generation Plant Interface [10], that is one of the sub-program in the Human Media Project under the Industrial Science and Technology Frontier (ISTF) Program of Japan.

2. The Memory Organization Package

Memory Organization Package (MOP) is a case-based inference system developed by Schank, *et al.* An individual case on MOP has its abstractions, and slots or pairs of role and filler. You may deem a case realization (a MOP) is a sort of frame. MOP system is, however, more than simple frame system. In MOP system, a filler itself is also a MOP and a

bunch of slots is transformed a MOP and it is automatically located at appropriate position in the abstraction tree in case memory. A case or a bunch of pairs of attribute and value is memorized in case memory, depending on its slot fillers and memory contents. It is just like as intension decides extension in a concept tree.

In MOP the conception of distance is obtained by counting the number of nodes in abstraction hierarchy tree from one node to another as the degree of kinship. The notion of similarity is also obtained according to the distance in concept tree, as same as an ape is similar to Homo sapience in taxonomy. In concept tree an intermediate node represents a concept. Therefore it is possible to acquire new concepts from a number of cases, even though it is yet open how an abstraction tree of MOPs is re-organized in accepting new input cases.

In language learning, it is argued that children have some intrinsic bias to learn new words. It is also discussed that conceptual abstraction or constraints play important role in language learning, *i.e.*, the notion of constraint explains immediate mapping of a new word to a reference when a child encounters a new object in the world with a new word. We think that MOP provides such mechanism as human empirical learning.

As a consequence, it is possible that an agent provided with case memory is capable of learning new concepts from experience as children learn their language from experience. The mechanism will work well in mapping from events in plants to literal symbols.

3. Demonstration of the Interface Agent

For the purpose of demonstration of the interface agent performance, the agent was connected to a simulator of a crude distillation column (topper) and the peripherals in an oil refinery plant instead of a real plant. In this demonstration the labeling mechanism for plant states, *i.e.*, NORMAL/ABNORMAL was implemented into the interface agent. The function of foreseeing from experience is also implemented. The details of implementation are described at [2].

In the beginning of the demonstration, the agent has no experience or no case in memory. Then the agent asks to an operator whether it is NORMAL or ABNORMAL when it encounters new unknown states of the plant. The operator responds the question from the agent on account that he or she thinks.

The scenario of the demonstration is that the degradation of the drawing pump of the Whole Straight Run gasoline (WSR) at the outlet of an overhead drum in the topper causes the elevation of the distillated liquid level in the overhead drum. The agent always monitors some of process data including the distillate liquid level, and detects the anomaly caused by the WSR pump degradation. The log of first trial in the demonstration is as follows.

1. Agent: Is this abnormal?
   (Because the agent has no experience.)
2. Operator: Normal.
   (The agent labels NORMAL to this plant state.)
3. The degradation of WSR pump has occurred.
4. Agent: The level of overhead drum has elevated. Is this abnormal?
   (The agent sensed the change of the level. So, the agent generated the appropriate explanation and question.)
5. Operator: Normal.
   (Because the amount of elevation is still little although the change is detected.)
6. Agent: The level of overhead drum is increasing. The level of overhead drum has elevated. Is this abnormal?
   (The agent sensed the level was different now from the previous value. Then, the agent makes the messages and speak it.)
7. Operator: Abnormal
   (The operator confirmed at the trend graph that the level of overhead was fairy elevated and still increasing.)
8. Operator: Show the counteraction
   (The agent asks the semantic representation interface sub-system [13] to derive the counteraction by means-ends analysis.)
9. The agent told the derived counteraction, *i.e.*, reducing the inlet flow of the main column by regulating the flow control valve at the inlet of the oil heater.
10. The operator took the counteraction.

After this first trial, the second trial was performed that is exactly same except that the agent knows plant situation at the first trial and all of states are labeled NORMAL or ABNORMAL in memory.

1. Agent: The plant state has become normal.
   (because the agent knows this state was normal)
2. The degradation of WSR pump has occurred.
3. Agent: It may become abnormal.
   (because the agent predicted an abnormal state would come after this state by empirical foreseeing)
4. Agent: The level of overhead drum is increasing. The level of overhead drum has elevated. It has become abnormal. The same abnormality has been experienced in the past. At that time the following counteraction was taken. Blab, blab, blab, …
   (because the agent monitored also operator's action with set-points of control variables)

Note that in this second trial the agent did not ask anything, and spoke the messages based on its experience.

4. MOP by the Common Lisp

The micro MOP system in the reference [4] was written by the subset of CLtL on condition of the use of personal computers in the second half of 1980s. It was reasonable because the aim of the book was education instead of practical use. However, the program codes included the textbook should be revised if the full set of the common lisp was available in today's personal computers.

At the beginning of the project, the following issues were pointed on condition of the use of the common lisp.

1. Slot table by property list to the structure
   In the micro MOP, a table implemented by the property list were used in order to make association of role and filler. The structure should be utilized in order to realize a frame as same as CLOS[11] or PCL.
2. Mapping from name to frame by property table to the hash array
   As same as slot table, a table by the property list was used for the mapping from MOP name to MOP frame. The hash array should be utilized for the mapping.
3. Dynamic programming by CLOS or Meta-Object-Protocol
   The program would be more flexible and adaptive by use of CLOS or Meta-Object-Protocol (yet another MOP)[12].
4. Garbage free programming
   No consing programming is desirable for the efficiency.
5. Robust programming
   To make the system more robust, the introductions of transaction, rapping by with-xxxxx macro, etc. are desirable.
6. GUI for MOP editor

Up to the present, the frame realization by the structure and mapping by the hash array were achieved. In fact, the efficiency of execution was most important concern in realization of *situation awareness* of the cognitive agent. The agent must sense the change of plant situation in a few seconds from sensor data readings. The use of CLOS was not initiated by same reason.

In addition to the use of the structure and hash array, one more revision was carried out for the sake of efficiency. In the micro MOP, a MOP was represented and handled by its name. It implies that the system must traverse from name to frame and frame to name, *i.e.*, retrieving a frame from name, getting a slot from the frame, accessing from role to filler (MOP name), then MOP name to MOP frame, again and again. This overhead of traversing was not negligible for our application. Therefore, every reference to MOP was changed by name to frame object itself.

5. MOP by Java

In general, the most elemental question in translation of program from lisp to another is how much in details the code in hand should be translated. We must set translation level appropriately. At one extreme, only algorithm level is translated, and at the other extreme, lisp emulator is developed in another language. In fact, actual translation falls into some point between the extremes. The reality depends on features of a particular application. In this case, we stood a very close point to developing lisp emulator. Java has garbage collection as same as lisp. We were anxious about the semantics in details, *i.e.*, inheritance logic, set or queue or stack by list, etc. Therefore, it seemed that developing of lisp emulator by Java was rather easy and suitable. However, our goal is not lisp emulator but MOP system. Therefore, some account was taken into for the sake of efficiency.

5.1 List or Consing

List or consing is realized newly developed Java class Cell at first and ListHolder later. **List1** shows a part of class Cell code.

List1:

```
public class Cell {

        private Object car_;
        private Object cdr_;

        public Cell() {
                this.car_ = null;
                this.cdr_ = null;
        }

        public Object car() {
                return car_;
        }
        public Object cdr() {
                return cdr_;
        }
```

After some experiences of translation and execution of this typical cell realization, this straightforward implementation of list was changed to Java class ListHolder that includes entry definition internally for the sake of efficiency. **List2** shows a part of class ListHolder code. Apart from legitimate programming manner in Java, iteration is done by not the Iterator interface but updating current entry.

List2:

```
public class ListHolder implements Serializable {

        private Entry header ;
        private int size ;
        private Entry tail ;
        private transient Entry current ;
        private transient int nextIndex ;

        public ListHolder () {
                header = null ;
                size = 0 ;
                nextIndex = 0 ;
                tail = null ;
                current = null ;
        }

        public Object getFirst () {
                if (size == 0)
                throw new NoSuchElementException () ;
                return header.element ;
        }

    private static class Entry
                    implements Serializable {
        Object element;
        Entry next;

        Entry(Object element, Entry next) {
            this.element = element;
            this.next = next;
        }
        }

        public void listIterator () {
                current = header ;
                nextIndex = 0 ;
        }

        public Object next () {
            if (nextIndex == size)
                throw new NoSuchElementException() ;
                Object ret = current.element ;
                current = current.next ;
                nextIndex++ ;
                return ret ;
        }
```

}

5.2 Null List and Boolean

In lisp NIL represents null list and Boolean false. Opposing to lisp, Java is strongly typed language and NIL in lisp must be translated exact null list or Boolean false according to the context. The word NULLP is preferred as predicate name, because the word `null` is reserved in Java. In List1 `nullp` checks if the `car` and `cdr` variables hold null, and in List2 `nullp` checks if the length of list is zero.

5.3 Package, Symbol, and String

The functionality of lisp package was discarded. A puzzling question was symbol. Sometime symbol was implemented as same as lisp and sometime symbol was virtually represented by Java string. In the former, lisp string was translated to Java string, in the latter lisp, string was translated to double quoted Java string.

At the present, MOP name is represented by Java string, and associated to its frame object through Java hash table.

5.4 Funcall or Apply

Daemon functionality is very important in MOP programming. When a new case is added into memory, if an abstraction MOP of newly created MOP instance has a daemon, it is invoked. The invoked daemon may create new MOPs. It can be deemed that the influence of a newly added case propagates in memory through daemons. This daemon functionality is implemented by the funcall or apply in lisp and it was translated into Java reflection Method.invoke().

List3 shows a top-level lisp function in micro CHEF program [4]. In this program, the function `chef` is called with a bunch of slots such as `((:meat BEEF) (:vege GREEN-PEPPERS) (:style STIR-FRY))` and all of functions such as memorizing, adapting, repairing, etc., or calculating the steps of recipe in short, are executed via daemons defined in memory.

List3:

```
(defun chef (slots)
  "SHEF <slot-list>
   finds or creates a case under M-RECIPE with
   the given slots and returns it."
  (let ((instance (slots->mop slots '(M-RECIPE) T)))
    (and (get-filler instance ':steps)
         instance)))
```

5.5 S-form Reader and Writer

Preceding the development of MOP system in Java, S-form reader and writer is programmed in Java. Exactly, the reader is realized in class mop.Reader, and the writer is realized by overriding the method toString for every defined class. This allows us to program Java code in same feeling as lisp by looking at printed matters, and allows to transfer the ontology that is developed by lisp MOP to Java MOP. This Java program is also available to communicate with other agents using agent communication language like KQML.

5.6 Class Hierarchy and Dependency

List4 illustrates class hierarchy in mop package in Java. **List5** shows the dependency relationship among them. Note that in List5 every class uses ListHolder, though it was not listed. FrameHashtable class and Index class including IndexEntry are independent from others except ListHolder. The instance of FrameHashtable class provides a memory of frames, and the instance of Index provides a memory of indices. The Mop class has two variables, a frame memory and an index memory, where the instance of FrameHashtable and Index class are stored when the Mop class is instantiated. A number of methods in Mop class delegate their task to FrameHashtable class and Index class. A slot is an object that has two variables, a role and filler. A frame consists of name, slots, abstracts, and properties. An index entry is an object that has two variables, labels and items. An instance of Slot and IndexEntry is not a list, but it is externally printed as a non-dotted list of two elements.

List4:

```
class java.lang.Object
    class java.util.Dictionary
        class java.util.Hashtable
             (implements java.lang.Cloneable,
              java.util.Map,
              java.io.Serializable)
            class mop.FrameHashtable
    class mop.Frame
          (implements java.io.Serializable)
    class mop.Index
          (implements java.io.Serializable)
    class mop.IndexEntry
          (implements java.io.Serializable)
    class mop.ListHolder
          (implements java.io.Serializable)
    class mop.Mop
    class java.io.Reader
        class java.io.FilterReader
            class java.io.PushbackReader
                class mop.MopReader
    class mop.Slot (implements
java.io.Serializable)
```

List5:

```
   mop.Mop
      mop.Slot
      mop.FrameHashtable
      mop.Frame
            mop.Slot
      mop.Index
            mop.IndexEntry
   mop.MopReader
```

5.7 Programming in Java

In typical usage of Mop in Java, an application package should import mop package, and public application class should extend Mop class. The instantiation of application that extends Mop class implicitly initiates the Mop constructor. The main method typically executes the following steps.

(1) create a new instance of application, that causes creation of a frame memory and an index memory.
(2) clear both memories, that implies creation of `M_ROOT` node and intrinsic MOPs in a frame memory.
(3) load the application ontology, that implies that an abstraction tree on the application is made starting at `M_ROOT` node.
(4) execute the top level method of application.

For instance, the translated code of micro Judge program [4] is enveloped as shown in **List6**.

List6:
```
import mop.* ;
import java.util.* ;
import java.io.* ;
public class Judge extends Mop {
    public static void main (String args []) {
        Judge nj = new Judge () ;
        nj.clearMemory () ;
        nj.loadMem () ;
        nj.judgeDemo () ;
    }
    <Judge program in Java>
}
```

6. MOP by C++

So far, MOPs in lisp and Java are not persistent. In addition that they must be persistent, a huge number of MOPs must be stored and retrieved from memory. In other words, very large-scale database functionality was needed. We thought that object database is much more appropriate than relational database in order to build case-based memory. MOP code in Java has been translated to C++ using ObjectStore.

ObjectStore has no dedicated access command such as SQL. Instead it looks like a sort of virtual memory from the viewpoint of programmer. The virtual memory space in client machine is mapped to the database space, and when page fault occurred by accessing a persistent object, ObjectStore module in the client machine reads the page including the target object into cache in client machine. You do not need to take care of the database handling in programming except opening and closing of database at the beginning and end of execution. If you want to keep the coherency between the cache and the database in the middle of execution, you may commit it at anytime. In our application of the interface agent, the committing was executed at each time of the iteration in which the current process data was read and processed.

In addition of setting of the transaction, one more key issuer in usage of ObjectStore is a database root. At first you need to explicitly get a database root from the database. **List7** shows a portion for setting a database root in Judge program, and **List8** shows a portion for getting a database root. Note that a symbol dictionary is used as FrameHashtable.

List7:
```
num = new(db ,Number::get_os_typespec()) Number (0) ;
ListHolder::setSegment (os_segment::of (num)) ;
Newjudge* nj = new(os_segment::of (num),
                   Newjudge::get_os_typespec())
                Newjudge () ;
Symbol::setDict (os_segment::of (num),
                 nj->getSymbolDict ()) ;
root = db->create_root ("judge");
root->set_value (nj);
nj->clearMemory () ;
nj->loadMem () ;
nj->judgeDemo () ;
```

List8:
```
db = os_database::open (dbName) ;
root = db->find_root ("judge");
nj = (Newjudge*)root->get_value ();
num = new(db, Number::get_os_typespec()) Number (0) ;
ListHolder::setSegment (os_segment::of (num)) ;
Symbol::setDict (os_segment::of (num),
                 nj->getSymbolDict ()) ;
```

ObjectStore has a list class named `os_list` in class library. Therefore, ListHolder is implemented with `os_list` as shown in **List9**, which shows a part of ListHolder codes. **Table 1** shows the difference of implementation between Java and ObjectStore C++.

List9:
```
os_segment* ListHolder::_sg ;
ListHolder::ListHolder()
{
  _list = &os_list::create (os_segment::of (this)) ;
}

ListHolder::~ListHolder()
{
  if (del == 0) {
    os_cursor c = os_cursor (*_list) ;
    HMObject* elem ;
    for (elem = (HMObject*)c.first () ; c.more () ;
         elem = (HMObject*)c.next ()) {
      if (elem == 0) break ;
      delete elem ;
    }
    _list->destroy (*_list) ;
  }

}

os_cursor ListHolder::getCursor () {
  return os_cursor::os_cursor (*_list) ;
}

HMObject* ListHolder::getFirst () {
  return (HMObject*)_list->retrieve_first () ;
}
```

Table 1 The Difference of Implementation

	Java	ObjectStore
FrameHashtable	extends java.util.Hashtable	composed of os_Dictionary
Index	composed of mop.ListHolder	composed of ListHolder
ListHolder	original	composed of os_list

7. Comparison among Lisp, Java, and C++

7.1 Code in MOP

In lisp, arithmetic calculation functions can accept any type of number. In Java, the operation is very strictly typed for number class object so that coding is required for each type of given parameters. This is very tedious. This trouble is resolved by overriding of arithmetic operators in C++. List10 shows the same portion in Judge program that includes arithmetic operation.

List10:
```
                    -- Lisp --
(let ((result (- this-severity prev-severity)))
  (setf prev-severity this-severity)
  result)

                    -- Java --
if (thisSeverity instanceof Integer)
  if (prevSeverity instanceof Integer)
    result =
     (Object)new Integer (
       ((Integer)thisSeverity).intValue () –
       ((Integer)prevSeverity).intValue ()) ;
  else
```

```
    result =
     (Object)new Double (
       ((Integer)thisSeverity).doubleValue () –
       ((Double)prevSeverity).doubleValue ()) ;
else
  if (prevSeverity instanceof Integer)
    result =
      (Object)new Double (
        ((Double)thisSeverity).doubleValue () –
        ((Integer)prevSeverity).doubleValue ()) ;
  else
    result =
      (Object)new Double (
        ((Double)thisSeverity).doubleValue () –
        ((Double)prevSeverity).doubleValue ()) ;
```

-- Object Store C++ --

```
result =
  new(os_segment::of (this),
      Number::get_os_typespec ())
  Number (*thisSeverity - *prevSeverity) ;
prevSeverity = thisSeverity ;
```

In application level, it is quite easy to imagine coding form among lisp, Java, and C++. Two examples of original lisp code, translated Java code, and ObjectStore C++ code are shown in **List11** and **12**.

List11:

-- Lisp --

```
(defun judge (slots)
  "JUDGE <slot-list>
   finds or creates a case under M-CRIME with the
given
   slots and returns it."
  (let ((instance
         (slots->mop slots
                     (list (frame-of 'M-CRIME))
T)))
    (and (get-filler instance :sentence)
         instance)))
```

-- Java --

```
public Object judge (ListHolder slots) {
    ListHolder list = new ListHolder();
    list.addLast( frameOf("m_crime") );
    Object instance = slotsToMop(slots, list, true);
    Object sent = getFiller(instance, ":sentence");
    if (sent == null ||
        (sent instanceof ListHolder &&
         ((ListHolder)sent).nullp() )) {
       return (Object)new ListHolder();
    }
    else {
       return (Object)instance;
    }
}
```

-- Object Store C++ --

```
HMObject* Newjudge::judge (ListHolder* slots) {
   ListHolder* list =
      new( os_segment::of(this),
           ListHolder::get_os_typespec() )
      ListHolder();
   list->addLast( frameOf("m_crime") );
   HMObject* instance =
      slotsToMop(slots, list, true);
   HMObject* sent =
      getFiller(instance,
                Symbol::create(":sentence") );
   if (sent == 0 ||
       (dynamic_cast<ListHolder*> (sent) != 0 &&
        ((ListHolder*)sent)->nullp ())) {
      return new( os_segment::of(this),
                  ListHolder::get_os_typespec() )
             ListHolder();
   }
   else {
      return instance;
   }
}
```

List12:

-- Lisp --

```
(defun mop-calc (slots)
  "MOP-CALC <slot-list>
   finds the specialization of m-calc with the
   given slots and returns its :value filler."
  (let ((instance
         (slots->mop
           slots (list (frame-of 'm-calc)) NIL)))
    (and instance (get-filler instance :value ))))
```

-- Java --

```
public Object mopCalc (ListHolder slots) {
    ListHolder list = new ListHolder();
    list.addLast(frameOf("m_calc"));
    Object instance =
        slotsToMop(slots, list, false);
    if (instance == null) {
        return null;
    }
    Object ret = getFiller(instance, ":value");
    return ret;
}
```

-- ObjectStore C++ --

```
HMObject* Newjudge::mopCalc (ListHolder* slots) {
   ListHolder* list =
      new( os_segment::of(this),
           ListHolder::get_os_typespec() )
      ListHolder();
   list->addLast(frameOf ("m_calc"));
   HMObject* instance =
      slotsToMop(slots,list,false);
   if (instance == 0) {
      return 0;
   }
   HMObject* ret =
     getFiller(instance,Symbol::create(":value"));
   return ret;
}
```

The comparison of the amount of codes in MOP program is shown in **Table 2**. Where all of comments and descriptions are stripped off, and the braces, parenthesis, commas, and semicolons are also stripped off on number of token. In Java , the code of ListHolder class and mop.Reader class was not taken into account of in fairness to comparison. In ObjectStore, the code of Boolean, Number, etc., those object classes are programmed only for the purpose of storing them into slots, are also ignored in fairness. The amount of code is approximately three to four times larger in Java and four to five times larger in C++ than lisp. We think this is not so strange.

Table 2 Comparison of the Amount of MOP Code

	# of Line (ratio)	# of Token (ratio)
Lisp	928 (1.0)	3401 (1.0)
Java	3522 (3.8)	10230 (3.0)
ObjectStore	4432 (4.8)	13060 (3.8)

7.2 Calculation Speed

The interface agent can detect un-experienced situation with sensory data. **Figure 1** shows the result of off-line calculation with the record file of sensory data for a period of one hour from an incinerator plant. At the bottom in the figure, the messages from the agent that tells it is unknown are plotted at the time spot. At the beginning, the agent often detected unknown situation because it had no cases in memory, then recognized that the situation was similar to what the agent experienced. When the momentary variation

occurred, the agent detected it but it was temporal. When the situation was dramatically changed, the agent continued to detect during the period of changing.

This calculation was carried out in lisp, Java, and ObjectStore C++. The result on calculation speed is shown in **Table 3.** In this execution, the condition of data was very severe for ObjectStore. As shown in Figure 1, one of sensory data was very unstable and scattered in full range at each reading. As a result, for most of 900 sets of logging data in one hour the disk access for paging was occurred at each set reading. We think that sensory data condition is much more stable in ordinary plants such as oil refinery plants, chemical plants, power plants, etc., and we can improve the efficiency of calculation by consulting an expert of ObjectStore programming.

Table3 The Comparison of Calculation Speed

Lisp (ratio)	Java (ratio)	ObjectStore (ratio)
1min54s (1.0)	2min33s (1.34)	32min58s (16.6)

Pentium II, 400MHz, Windows-NT
AllegroCL 5.0.1 under IDE, JDK1.1.7 with JIT, ObjectStore5.0

8. Conclusion

An interface agent that mediates between an operator and a process plant was developed. The memory of the agent was realized as case-based memory using the Memory Organization Package (MOP).

The original micro MOP program by Schank, *et al*. was revised for the sake of efficiency using the full set of common lisp, then the lisp program revised was translated into Java program. Furthermore, the Java program was translated into C++ code using ObjectStore API. This incremental stepwise translation enabled us to focus the attention to keeping the semantic equality in details and object-oriented programming in the translation from lisp to Java, and to focus to object database usage in the translation from Java to ObjectStore C++. As a result, we achieved the development of the interface agent smoothly and successfully, and case-based inference engine, MOP program, was left in language lisp, Java, and ObjectStore C++.

Lisp MOP and Java MOP ensured that we port the application program developed for the interface agent from lisp in exploratory programming into Java for the Java based cooperative project. We believe that ObjectStore MOP will ensure that the interface agent handles very large-scale case memory in practical use.

9. Acknowledgements

This paper was prepared under an Entrustment Contract with the Laboratories of Image Information Science and Technology (LIST) from the New Energy and Industrial Technology Development Organization (NEDO) in concern with the Human Media Research and Development Project under the Industrial Science and Technology Frontier (ISTF) program of the Ministry of International Trade and Industry (MITI) of Japan. We appreciate the leader of project, Prof. Mizoguchi, and the sub-leader, Prof. Gofuku.

10. References

[1] Agent'98, the workshop of agents in manufacturing.
[2] Koide, S. and S. Yamauchi, Interface Agent for Process Plant Operation Towards the Next Generation Interface, Proceedings of the 1999 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM'99), pp.197-202, 1999, IEEE.
[3] Endo, M., *et al*., Development of Human-Machine Interface Composed of Virtual Reality and Interface Agent on Process Plant Operation, International Conference on Systems, Man, and Cybernetics (SMC'99), pp.V-636-641, 1999, IEEE.
[4] Riesbeck, C.K., and R.C. Schank, *Inside Case-Based Reasoning*, 1989, LEA.
[5] Schank, R.C., A. Kass, and C.K. Riesbeck, *Inside Case-Based Explanation*, 1994, LEA.
[6] Koide, S. and S. Suzuki, Voice Interaction of an Interface Agent for Process Plant (Japanese), Proceedings of 1999 Annual Conference of Japan Society of Artificial Intelligence, pp.229-230, 1999, IEEE.
[7] Suzuki, S., *et al*., Voice Interaction Model of an Interface Agent for Process Plants (Japanese), SIG-SLUD-9902-2 (10/14), pp.7-10, 1999.
[8] Steel Jr., G.L., Common Lisp the Language, 1984, DEC.
[9] Steel Jr., G.L., Common Lisp the Language Second Edition, 1990, DEC.
[10] Mizoguchi, R., *et al.* Human Media Interface System for the Next Generation Plant Operation, International Conference on Systems, Man, and Cybernetics (SMC'99), pp.V-630-635, 1999, IEEE.
[11] Paepcke, A. (ed.), Object-Oriented Programming the CLOS Perspective, 1993, MIT.
[12] Kiczales, et al., The Art of the Metaobject Protocol, 1991, MIT.
[13] Gofuku, A. and Y. Tanaka, Display of Diagnostic Information from Multiple Viewpoints in an Anomalous Situation of Complex Plants(SMC'99), pp.V-642-647, 1999, IEEE.
[14] Schank, R.C. *Dynamic Memory*, 1982, Cambridge Univ. Press.

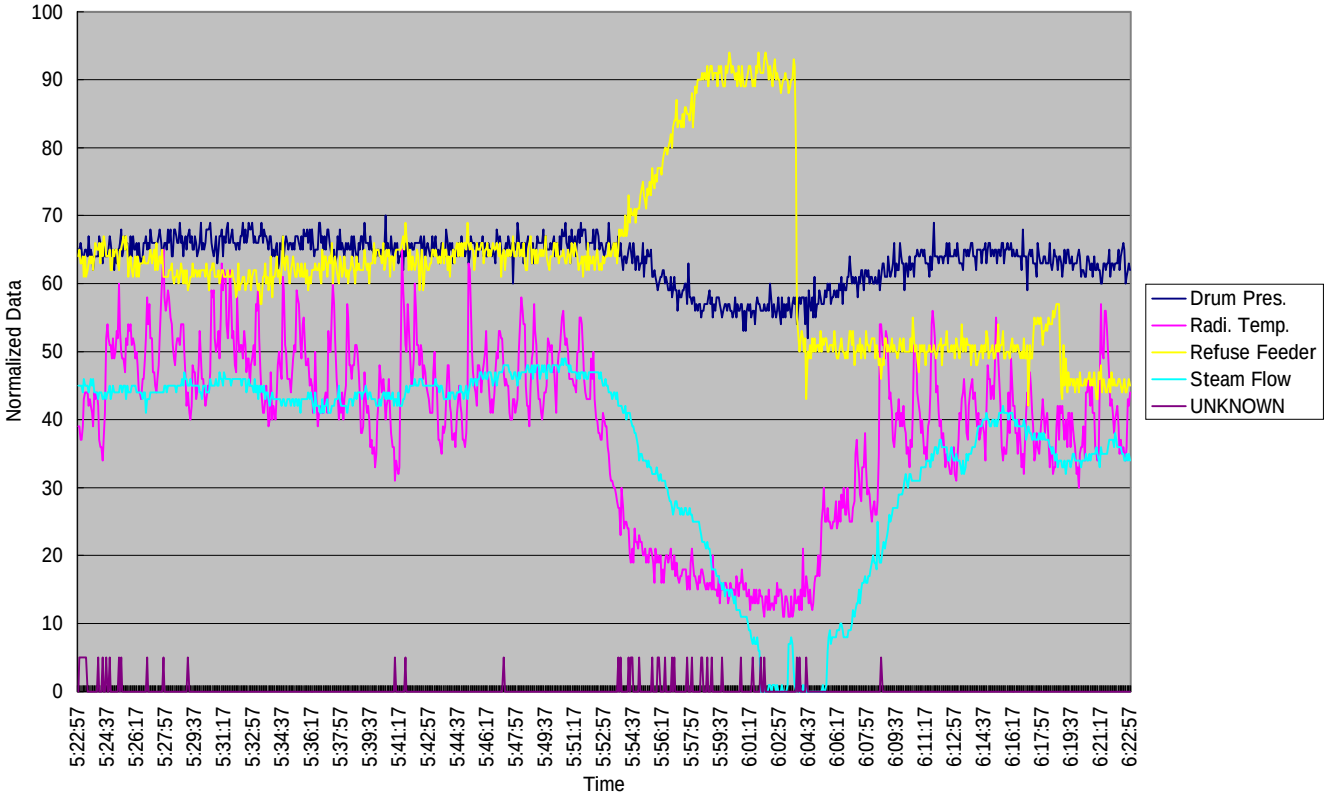

Figure 1 Unknown Situation Detection by Interface Agent