ソフトウェア基礎論　27－3
プログラミング言語　19－3
（1988. 12. 9）

Basic Functions of a Parallel Coordinated Problem Solving System: *Harmonia*

(並列協調型問題解決システム *Harmonia* の基本機能)

Rikio Onai
尾内理紀夫

NTT Software Laboratories
NTT ソフトウェア研究所

3-9-11 Midori-cho, Musashino-shi, Tokyo 180 Japan

あらまし 疎結合ELISより構成される並列協調型問題解決システム*Harmonia*の基本機能、特に、通信機能、動的オブジェクト指向プログラミング機能、実時計合わせ機能について述べる。これらの機能はマルチパラダイム言語TAOにより実現されている。

Abstract
A parallel coordinated computing system *Harmonia* is composed of loosely-coupled computers which solve problems coherently and harmoniously in parallel. This paper describes the basic functions of *Harmonia*. It permits various parallel coordinated problem solving techniques using communication functions which can be used independently of hardware configuration, dynamic object-oriented programing functions for knowledge modification, and clock synchronizing function. The experimental hardware of *Harmonia* consists of six AI work-station ELISs connected by Ethernet and basic functions of *Harmonia* are implemented using the multiple-paradigm language, TAO.

1 Introduction

The importance of Distributed Artificial Intelligence (DAI) increases as computers become more powerful, smaller and cheaper, network development broadens and expert systems grow. DAI consists of distributed, coordinated AIs and is related to a wide range of computer science research areas, such as distributed processing, parallel processing and AI.

Future AI systems must possess and maintain remarkably complex and vast knowledge to be 'intelligent'. This will be possible if AIs are distributed and operated in parallel. There are also AI problems with characteristics which is natural spatial or functional distribution.

Parallel coordinated problem-solving [Filman 84] is part of DAI. Functional division of a problem into subproblems with comparatively large granularity is characteristic of parallel coordinated problem-solving. There have already been several research efforts on parallel coordinated problem-solving; for example, Contract Net [Smith 85], Scientific Community Metaphor [Kornfeld 81], and Hearsay-II [Erman 80].

Our definition of parallel coordinated problem-solving is that agents in a distributed system, which consists of many loosely-coupled computers, coordinate one another to solve problems coherently.

'Loosely-coupled' means that agents spend most of their time computing rather than communicating. Both control and data are distributed.

'Coordination' is different from 'cooperation'. 'Cooperation' occurs when agents with identical or nonconflicting goals help one another to solve problems. On the other hand, 'coordination' occurs when conflicting agents compromise with one another to solve problems. In other words, 'coordination' = 'cooperation' + 'compromise'.

'Coherent problem-solving' means that all the agents finally reach the goal, but some of the time, some agents may move in the opposite direction to the goal.

Our aim in developing *Harmonia* is to establish the fundamental technology for parallel coordinated problem-solving. This paper describes the basic functions of *Harmonia*, especially the communication functions which can be used independently of the hardware configuration, dynamic object-oriented programing functions for knowledge modification, and the synchronizing function of real clocks. The experimental hardware of *Harmonia* consists of six AI work-station ELISs [Watanabe 87]connected by Ethernet. *Harmonia's* basic functions are implemented using the multiple-paradigm language, TAO [Takeuchi 86].

2 Organization of *Harmonia*

This section outlines the organization of the parallel coordinated problem solving system *Harmonia*.

Harmonia is composed of loosely-coupled computers which contain Intelligent Agents (IAs). An IA is an active entity which can communicate with other IAs to solve problems coherently and harmoniously in parallel. *Harmonia's* experimental hardware consists of six ELISs connected by Ethernet (Fig. 1).

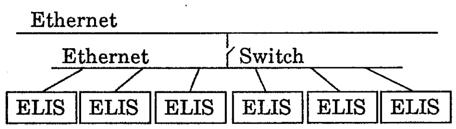

Fig. 1 Harmonia experimental hardware

Harmonia is based on the dynamic manager-solver hierarchy model, and permits communication not only between managers and solvers, but also between solvers. "Dynamic" means that an IA can change from manager to solver and vice versa. Figure 2 shows one possible relation among a number of IAs at one particular time. Thinking of the user at a terminal as a manager and the log-in process initiated by the user as a solver, the user can interactively input various messages from an ELIS terminal.

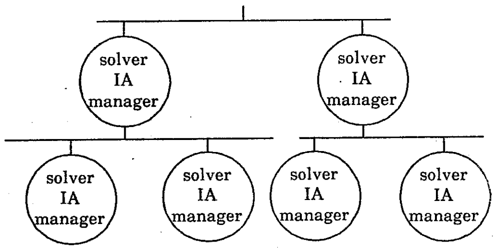

Fig. 2 Dynamic manager-solver hierarchy model

3 Basic Functions

This section describes the basic functions of *Harmonia*: communication, dynamic object-oriented programming and clock synchronization.

3.1 Communication function

This subsection describes inter-IA communication function in *Harmonia*. The communication is realized as a higher level of TCP (Transmission Control Protocol) [Post 81]. Since TCP is a reliably delivered, full-duplex, stream-oriented protocol that uses an adaptive retransmission algorithm and forms the foundation on which most of the higher-level protocols depend, it is chosen as the starting point of the *Harmonia* communication protocol. *Harmonia* provides two levels of communication using TCP and the communication is also used to operate the dynamic object, described in **3.2**, and to synchronize clocks, described in **3.3**.

3.1.1 A first step toward virtual communication

The future target is virtual communication among distributed IAs. First, instead of the IA name and the ELIS name in which it exists, the IA name only is required in the communication. A correspondence between an IA name and a physical ELIS name is given on metalevel and stored in the communication server (Comser in Fig. 3).

Syntax :

(start-cps
(<*IA-name*> <*initial-function*> <*initial-args*>
【<*physical-ELIS-name*>】 【<*exported-files*>】)
{(<*IA-name*> <*initial-function*> <*initial-args*>
【<*physical-ELIS-name*>】 【<*exported-files*>】)})

The syntax in this paper is as follows:

< and > denotes that the enclosed construct is a metasymbol.

【 and 】 denotes that the enclosed construct is optional.

{ and } implies that the enclosed construct occurs zero or more times.

When <*physical-ELIS-name*> is omitted, the IA <*IA-name*> is executed on the same ELIS as that on which this metalevel function start-cps is executed.

When <*physical-ELIS-name*> is given, the IA <*IA-name*> is exported to the <*physical-ELIS-name*> and <*initial-function*> is applied to <*initial-args*> in the environment of ELIS <*physical-ELIS-name*> .

When files <*exported-files*> are given, they are exported to <*physical-ELIS-name*>. Correspondences between IAs and ELISs are stored in the communication servers of both import-ELIS and export-ELIS.

3.1.2 Two communication levels (Fig. 3)

Harmonia provides two communication levels. One is for ordinary communication using mailboxes. The other is for emergency communication using inter-IA interrupt. Various kinds of communication functions are possible using a combination of the two communication levels [Onai 86].

There are two cases of inter-IA communication: within an ELIS, and between ELISs. The communication server (Comser-A, Comser-B) is responsible for correspondence between an IA (IA1, IA2, IA3, IA4) and an ELIS (ELIS-A, ELIS-B). The server uses an inter-ELIS communication mechanism which is realized as a higher level of TCP.

(1) Ordinary Communication

Syntax of ordinary communication:

(send-message <*IAs*> <*message*>)

(receive-message)

Each IA has its own general mailbox (G-mailbox in Fig. 3). The function receive-message takes a message from the mailbox and then returns it. When (send-message <*IAs*> <*message*>) is executed, <*IAs*> and <*message*> are first sent to the communication server.

There are two cases:

① The receiving IA exists in the same ELIS as the sending IA.
The communication server determines the receiver's general mailbox and executes the TAO function (send-mail <*G-box*> <*message*>). In this function, <*G-box*> is a general mailbox.

② The receiving IA exists in a different ELIS.
The communication server determines the ELIS of the receiving IA. It then sends the appropriate ELIS a packet consisting of the name of the receiving IA, and the message. The communication server in the receiving ELIS determines the general mailbox of the receiving IA and executes the TAO function (send-mail <*G-box*> <*message*>).

(2) Emergency Communication

In emergency communication the receiving IA is interrupted and the message is delivered (Interrupt in Fig. 3). This is achieved using the TAO functions process-interrupt and throw.

Syntax: (send-e-message <*IA*> <*message*>)

<*Message*> with <*IA*> is sent to the general mailbox of the communication server, which takes emergency messages from the mailbox before ordinary messages. The same two cases exist here as in ordinary communication. There is a catch tag in the car part of the message. The communication server then interrupts the receiving IA and delivers the message. When the interrupt succeeds, the sending IA receives t (true). When the interrupt fails, the sending IA receives nil. Hence, the emergency communication is synchronized.

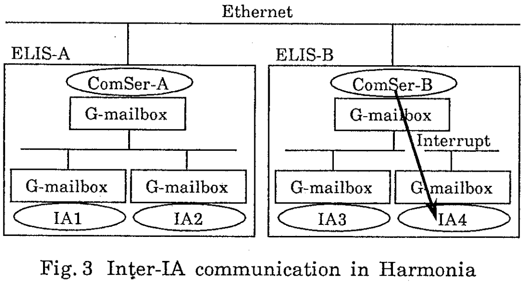

Fig. 3 Inter-IA communication in Harmonia

3.2 Dynamic Object-Oriented Programming Function

We have selected object-oriented description because of its modularity, hierarchy, easy development of large and complex programs, and maturing concept.

In a parallel coordinated problem-solving system [Filmann 84], a manager gives solvers knowledge which either the solvers require or the manager thinks they need. If the knowledge is perfect or does not require future modification, problems never occur. However, it is quite natural for knowledge to change and it is impossible for perfect knowledge to be given in the beginning. Therefore, dynamic modification function is requisite in the system.

However, the objects proposed in most object-oriented languages are not dynamic except CLOS [Bobrow 87-1][Bobrow 87-2] and new Flavors [Symbolics 86]. For example, when a class is redefined and new instance variables are added to it, all existing instances refer to the old class definition. Instances must be reconstructed to make them refer to the new class definition. Therefore, conventional objects are not suitable for knowledge representation in the system. Objects which can be modified dynamically, i.e. **dynamic objects**, are required. Dynamic objects allow users, manager processes, and solver processes to dynamically modify objects as knowledge. When part of classes, methods, and instances are modified, related classes, methods, and instances are redefined or modified automatically.

This subsection describes functions for the dynamic object-oriented programming using the dynamic object and shows that the dynamic object-oriented programming is more natural and more efficient than the ordinary object-oriented programming from the standpoint of knowledge modification [Onai 88-2].

3.2.1 Object-Oriented Programming in TAO

The dynamic object is implemented using TAO and our improvement on it, hereafter called improved TAO. This **3.2.1** outlines TAO's object-oriented features.

TAO is a Lisp dialect with object-oriented programming and logic programming features implemented on ELIS. It incorporates features of Prolog, Smalltalk [Goldberg 83], and Flavors [Weinreb 83]. TAO is designed as an interpreter-centered programming language in order to support a fully interactive environment with comfortable speed. Indeed, the speed of the interpreter is becoming comparable to compilers of other commercial Lisp systems. TAO emulates Common Lisp in the Common Lisp mode. ELIS is equipped with 64K, 64-bit WCS. Almost all parts of the TAO language kernel are microcoded. ELIS can be equipped with maximum 128Mbytes main memory.

The TAO object-oriented programming paradigm is similar to ZetaLisp's Flavors. However, unlike Flavors, TAO has the syntactical flavor of Smalltalk-like message passing and treats all primitive data types as objects.

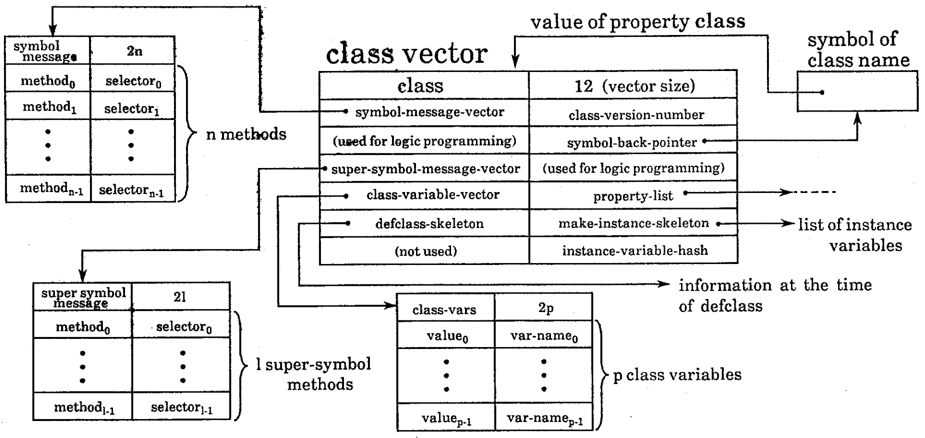

Fig. 4 Class vector and related tables in TAO

Next a class vector (Fig.4) is explained on various definitions of object-oriented functions.

(1) Class Definition

Syntax:

(defclass <*class*> <*class-variable-list*> <*instance-variable-list*> <*superclass-list*> {<*options*>})

defclass creates a vector called a class vector (Fig.4) and stores it in the property list of the symbol <*class*>, whose indicator is class. When subclasses are defined, they are stored in the property list of the symbol <*class*>, whose indicator is component-of-what. <*Superclass-list*> specifies a set of superclasses. TAO supports multiple inheritance.

A class inherits all the instance variables and all the methods of all its superclasses. That is, inherited instance variables and methods are registered to the class. By contrast, it does not inherit any class variables. However, the superclasses' class variables can be accessed from this class's methods through the class inheritance network.

<*Options*> may specify the default creation of some kinds of methods. For example,

`(:method-combination (combination-type1 selector11 selector12 ...) (combination-type2 selector21 selector22 ...) ...)` which declares the way the methods are combined.

When a class is defined by defclass, only the slots for class version number, symbol backpointer, defclass skeleton, and class variables is filled. 12 in a class vector, 2n in a symbol message vector, 2p in a class variable vector, and 2l in a super symbol message vector are the corresponding vector sizes.

Other slots are filled just when they are needed for the first time. This is because each defmethod after class definition changes the class structure, and some superclasses may not yet be defined when the class is defined. The class vector has property-lists and pointers to the lists are stored in the property-list slot (Fig.4).

(2) Instance Definition

Syntax:

(make-instance <*class*> {<*instance-variable*> <*initial-value*>})

make-instance returns a user-defined object, hereafter called "udo", of a class <*class*>. A udo is a vector (Fig. 5).

The make-instance-skeleton slot are filled and an instance variable hash table is built just when the first instance of the class is to be created. The symbol message vector is created just when one of the symbol messages is to be sent to an instance of the class for the first time. When a superclass is redefined, the corresponding subclass information slots for such as symbol message vector and make-instance skeleton are cleared. This lets the subclasses follow the change when needed. Every time a message is sent to an instance, the existence of the message vector is checked. However, this does not bring overhead into the actual microcode.

back pointer to class	2N (vector size)
value 1	instance-variable 1
value 2	instance-variable 2
• • •	
value N	instance-variable N

Fig. 5 User-defined object (udo)

TAO has no notion of metaclass. Classes need not be objects since the Lisp world itself is a metaconcept over the object-oriented world.

(3) Method Definition

Syntax:
(defmethod (<*class*> 【<*type*>】 <*selector*>) <*args*>
<*method-body*>)
(undefmethod (<*class*> <*selector*>) <*args*>
<*method-body*>)

defmethod lets users add new methods incrementally. When a new method is defined using defmethod, this method is registered temporarily in the class vector defclass skeleton and the symbol message vector is cleared. Symbol message vectors and super symbol message vectors in subclasses are also cleared. At the next message passing to an instance in the class, superclasses are traversed, and the symbol message vector is created. The option <*type*> is used for the method combination, similar to Flavors.

When undefmethod deletes a method, the method is deleted from the class vector defclass skeleton and the symbol message vector is cleared. Symbol message vectors and super symbol message vectors in subclasses are also cleared.

TAO function (super <*superclass*>.<*selector*> {<*args*>}) can be used in <*method-body*>. When a method call whose selector is <*selector*> is first executed, superclasses are traversed, and the method is added to a super symbol message vector. Using super in <*method-body*>, a particular superclass's method can be accessed directly.

3.2.2 Operaion of Dynamic Objects

This **3.2.2** describes the operations which are used for dynamic modification of object in *Harmonia*. These operations are written in <*message*> part of the two levels inter-IA communications as described in **3.1**. We have improved the structure of TAO's class vector to support the dynamic objects. A dynamic object is implemented in the improved TAO.

(1) Dynamic Class Creation and Modification

Syntax: (d-defclass <*class*> <*c-vars*> <*i-vars*> <*supers*>
{<*options*>})

D- represents "dynamic"; < *class* >, a class name; < *c-vars* >, class variables; < *i-vars* >, instance variables; and < *supers* >, superclasses.

First d-defclass creates a class vector and the related tables. When d-defclass is applied to the <*class*> next, dynamic modification happens. Dynamic class modification affects the class itself, its instances, its subclasses, and the subclasses' instances. The class itself and its subclasses are modified just when d-defclass is executed. Subclasses are gotten from the property list of the class.

However, an instance is modified on demand, that is, just when the first message after d-defclass execution is passed to the instance. Both a class and an instance have a version number (zero origin) for on-demand modification. Class version number is stored in the class vector (Fig. 4). Instance version number is stored in the property list of the instance name.

Instance names of a class and the corresponding udos are stored in the property list of the class vector under the indicator :i-list. It is an association list, the key is the instance name, and the datum is a udo corresponding to the instance name. A udo in the a-list is regarded as a pointer from a class to its instance. An instance is usually modified on demand. However, if ELIS is idling, instances can be modified by the pointer.

If the instance version number is different from the class version number when an instance receives a message, the class and its superclasses are traversed, methods or instance variables or both are gathered, a new udo frame is created, and the instance version number is set at the class version number. A udo in the property list under the indicator :i-list is updated. Of course, the old udo is handled by garbage-collection. When the instance variable name in the old udo is the same as that in the new defclass skeleton, we must choose a value to put in the new udo. A value in the old udo is called the old value; a value in the new defclass skeleton is called the new value. When a variable is undefined, it has the value (called the undefined value) which the TAO function undef returns. Then *Harmonia's* selection rules are as follows:

① When an instance variable in a new defclass skeleton has the property :set-all, a new value is chosen and the property is removed.

② An instance variable in a new defclass skeleton has the property :set-if-undef. When the old value is the undefined value, a new value is chosen. When the old value is defined, the old value is chosen. The property :set-if-undef is removed.

③ When the old value is different from the initial value in the old class vector's property list, i.e. when the value of the instance variable is changed during problem solving, the old value is selected.

④ When the old value is the same as the initial value in the old class vector's property list, a new value is selected.

When a user wants to choose a new value in a different way, he may specify a value in an instance by d-instance, which is described later. In modification, only the modified argument is specified. The other argument is nil. When a manager wants to add, delete, or change more than two items, d-defclass can be used.

There are several kinds of syntax to modify one item as follows.

• **Class Variable Modification**

Syntax: (d-class-var <*class*> <*modified-c-vars*>
{<*modified-c-vars*>})

Since class variables are not inherited in TAO, related slots in the class vector are modified and the class version number is augmented by 1 just when this message is received.

There are four kinds of <*modified-c-vars*> syntax.

① (:delete <*c-var*>) <*C-var*> is deleted. <*C-var*> in a class variable vector table is replaced by a sign for empty.

② (<*c-var*> <*val*>) <*C-var*> and its value <*val*> are added. The dynamic object mechanism puts <*c-var*> into a defclass skeleton (Fig. 4), creates a new class variable vector table which is two vector sizes larger than the old one, copies values & names of old variables, and puts <*c-var*> and <*val*> into the new vector table. If there are empty slots in the old vector, <*c-var*> is put into the slot and <*val*> is put into the next slot.

③ <*c-var*> <*C-var*> and the undefined value are added.

④ (:set <*c-var*> <*val*>) The old value of <*c-var* > is replaced by the new value <*val*>.

• **Instance Variable Modification**

Syntax: (d-instance-var <*class*> <*modified-i-vars*>
{<*modified-i-vars*>})

The dynamic object mechanism modifies first the class vector as follows.

⋆ Increases the class version number by one

⋆ Clears the instance variable hash table and the make-instance skeleton

The same common processing as that for the class vector is done for subclasses recursively. Subclasses are stored in the property list of the symbol <*class*>, whose indicator is component-of-what.

There are five kinds of <*modified-i-vars*> syntax.

① (<*i-var*> <*val*>) <*I-var*> and its value <*val*> are added.

The dynamic object mechanism puts <*i-var*> into the defclass skeleton and <*i-var*> & <*val*> into the property list of the existing class vector. At the next message passing to an instance in the class, the dynamic object mechanism traverses superclasses, creates a new udo frame (Fig. 5) for the instance. If old udo size is enough for new instance variables, the old udo frame is used again. The instance variables' values are written into slots in the udo using the selection rules.

② (:delete <*i-var*>) <*I-var*> is deleted.

<*I-var*> is removed from the defclass skeleton. If <*i-var*> has initial value, it is also removed from the property list, whose indicator is :init-plist. At the next message passing to an instance, <*i-var*> name in the udo is removed.

③ (:set-all <*i-var*> <*val*>) The old value of <*i-var*> is replaced by < *val* >.

<*I-var*> & <*val*> are put into the property list, whose indicator is :init-plist. The property :set-all is put into <*i-var*>. At the next message passing to an instance, <*val*> is written into the corresponding value vector slot in a udo. The property :set-all is removed from <*i-var*>.

④ (:set-if-undef <*i-var*> <*val*>) If the old value of <*i-var*> is the undefined value, <*val*> is set.

<*I-var*> & <*val*> are put into the property list, whose indicator is :init-plist. The property :set-if-undef is put into <*i-var*>. At the next message passing to an instance, if the value of <*i-var*> is the undefined value, <*val*> is written into the corresponding value vector slot in the udo. The property :set-if-undef is removed from <*i-var*>.

⑤ <*i-var*> <*I-var*> and the undefined value are added.

• **Super Class Modification**

Syntax:(d-hierarchy <*class*> <*modified-super*>
{<*modified-super*>})

The dynamic object mechanism first modifies the class vector as follows just when d-hierarchy is called:

⋆ Increase the class version number by one

⋆ Clears the symbol message vector, the super symbol message vector, the instance variable hash table, and the make-instance skeleton.

Class vectors of subclasses are recursively modified in the same way.

There are three kinds of <*modified-super*> syntax.

① (:delete <*super*>) the class <*super*> is no longer a super class.

<*super*> is removed from the defclass skeleton. At the next message passing to an instance in the class, instance variables inherited from the deleted superclass are removed from the new udo of the instance.

② <*super*> the class <*super*> is added as a new super class.

<*super*> is added to the defclass skeleton. At the next message passing to an instance in the class, a new udo frame is created. Values of instance variables are determined using the selection rules.

③ (:change <*old-super*> <*new-super*>)

Class <*old-super*> is replaced by class <*new-super*>. That is, class <*old-super*> is deleted and class <*new-super*> is added. <*old-super*> is removed from the defclass skeleton and <*new-super*> is added. At the next message passing to an instance in the class, a new udo is made using the selection rules.

• **Option Modification**

Syntax: (d-option <*class*> <*modified-option*>
{<*modified-option*>})

There are two kinds of <*modified-option*> syntax.

① (:delete <*option*>) <*Option*> is deleted.

② <*option*>　　　New <*option*> is added.

The specified option and related information are stored or deleted in a class vector. Related tables and vectors are updated.

(2) Instance Creation

Syntax:(d-make-instance <*class*> <*instance*> {<*option*>})

This function creates a dynamic instance which belongs to the class <*class*> and returns a udo.

The difference in syntax between d-make-instance and make-instance of TAO is the argument <*instance*>.

The only processing differences are the storing of the paired <*instance*> and the udo in the property list of the class vector under the indicator :i-list, and the setting of the instance version number at zero. When a manager IA sends knowledge of his instance object to a solver, since the manager does not know the udo in the solver, the instance name is passed instead of the udo itself. This is the reason that an instance name <*instance*> exists in d-make-instance syntax and the <*instance*> exists in the element corresponding to an indicator :i-list in the class vector property list.

(3) Instance Modification

There are two kinds of syntax.

•Syntax: (d-instance <*class*> <*instance*> <*modified-i-var*>)

This is used to modify a value of an instance variable. There are two kinds of <*modified-i-var*> syntax.

① (:set <*i-var*> <*val*>)　　<*Val*> is stored as the new value of an <*i-var*>.

② (:set-if-undef <*i-var*> <*val*>)　When the value of an <*i-var*> is the undefined value, < *val* > is stored as the value of <*i-var*>.

• Syntax: (d-instance (:change <*old-class*> <*new-class*>) <*instance*> 【<*exp*>】)

This is used to change a class of an instance dynamically. <*Old class*> of the instance is changed to <*new-class*>. <*Exp*> is the expression specifying the relation between the variables in an old class and a new class. The values of the new variables are computed using <*exp*>. Since in a parallel coordinated problem solving system we want to suppress network traffic, i.e. reduce the number of message passing as much as possible, d-instance can contain an expression which specifies the relation among the variables.

If <*exp*> of a dynamic class change is used often, the <*exp*> is sent as a message with the syntax:

(d-class-relation <*old-class*> <*new-class*> <*exp*>)

d-class-relation is called by d-instance. When <*old-class*> and <*new-class*> have the same variable names, the <*old-class*> name is attached to an <*old-class*> variable. d-class-relation corresponds to CLOS's generic function class-changed [Bobrow 87-1]. Flavors [Weinreb 83][Symbolics 86] has no function corresponding to change-class and class-changed in CLOS or to :change of d-instance and d-class-relation in *Harmonia*. d-class-relation is explained using examples in **3.2.3**.

(4) Method Creation and Modification

Since method definition is dynamic in TAO, method creation and modification is the same as those of TAO.

(5) Method Call

Dynamic and ordinary objects have a common method call syntax. It is used, when a user does not know whether or not an instance is a dynamic object.

Syntax: (m-call <*instance*> <*selector*> {<*args*>})

3.2.3 Dynamic Object-Oriented Programming

Knowledge can be represented using the object-oriented concept. For example, Minsky's frame, which is a famous representation of knowledge, can be implemented in the object-oriented programming paradigm. A frame is a class or an instance. An AKO link is a relation between a class and a superclass. An Is-A is a relation between a class and an instance. Slots are class variables, instance variables, and superclasses. Procedures and slot operations are methods.

Knowledge represented by classes, instances, and methods has a long life and must be maintained. That is, knowledge is continually modified over a long time. For example, once the class `human-being` is created, it may not be deleted from the knowledge base. It is modified, e.g. instance variables are added, if necessary. As the knowledge base expands, there come thousands of instances or more which belong to the class `human-being`. Without dynamic objects, it may become impossible to modify class `human-being` and instances.

We call programming using dynamic objects **dynamic object-oriented programming.** It is more natural and efficient than the ordinary object-oriented programming for knowledge modification, as the following examples show. Programs, described below, are run for measurement on ELIS in *Harmonia's* experimental system.

(1) Gradually Increasing Instance Variables

Let us consider a simple example of gradually increasing knowledge involving a manager and a solver.

First, the manager gives the solver the knowledge that wedge has a triangular prism shape and that wedge wedge88 exists.

Second, the manager gives the solver the knowledge that wedge has color and wedge88 is red.

Finally, the manager gives the solver the knowledge that wedge has the material attribute and wedge88 is made of wood.

To describe this gradually increasing knowledge with object-oriented programming, there are three programming styles, described below.

①Redefining Class & Instance

```
(defclass wedge () ((shape 'triangular-prism))
  ())
(!wedge88 (make-instance 'wedge))
(defclass wedge () ((shape 'triangular-prism)
  color) ())
(!wedge88 (make-instance 'wedge color 'red))
(defclass wedge () ((shape 'triangular-prism)
  color material) ())
(!wedge88 (make-instance 'wedge color 'red
  material 'wood))
```

(!*<assignee>* *<new-value>*) is almost equivalent to (setq *<assignee>* *<new-value>*) in TAO. This programming style involves editing & loading and is tedious.

②Differential Programming

This is both an important concept and a useful technique in object-oriented programming.

```
(defclass wedge () ((shape 'triangular-prism))
  ())
(!wedge88 (make-instance 'wedge))
(defclass colored-wedge () (color) (wedge))
(!wedge88 (make-instance 'colored-wedge color
  'red))
(defclass material-wedge () (material)
  (colored-wedge))
(!wedge88 (make-instance 'material-wedge color
  'red material 'wood))
```

Class colored-wedge becomes the superclass of class material-wedge, because knowledge of color comes before knowledge of material. If knowledge of material had come before knowledge of color, class material-wedge might have become the superclass of class colored-wedge. The relation of superclass and class is unnatural and not essential in this case. Accidental knowledge change happens in the real world. Small classes are created successively and superclass traversing is required frequently.

Therefore, differential programming is not necessarily suitable for knowledge modification.

③Dynamic object-oriented programming

```
(d-defclass wedge () ((shape
  'triangular-prism)) ())
(!wedge88 (d-make-instance 'wedge 'wedge88))
(d-instance-var wedge color)
(d-instance wedge wedge88 (:set color 'red))
(d-instance-var wedge material)
(d-instance wedge wedge88 (:set material
  'wood))
```

Only one class wedge is created-and-modified. Class hierarchy is not created. We feel that the dynamic object-oriented programming, hereafter the dynamic o-o programming, is natural and suitable for knowledge modification.

When d-instance-var is executed, the dynamic object mechanism puts instance variables into the defclass skeleton and the instance variable& value (in this case, the value is undefined.) into the property list of the existing class vector. At the next message passing (d-instance) to the instance wedge88 in the class, the dynamic object mechanism traverses superclasses and creates a new udo for the instance.

Of course, a manager can give a solver the knowledge that wedge has shape, color and material at the outset. However, the real world always changes and knowledge of it also changes. Therefore, it is impossible to give all indispensable knowledge at the outset. Thus the dynamic o-o programming is natural and useful.

Let us compare the three programs in terms of cpu time and memory use. Table 1 compares the measurement results. CPU time is 14 ms, the number of used cells is 296, and the number of used vectors is 104 in the case of redefinition. The number of used vectors is the sum of used vectors' sizes.

	time	cell	vector
Redefinition	1	1	1
Differential programming	1	0.97	1
Dynamic object-oriented	0.81	0.72	0.86

Table 1 Ratio of CPU time and memory consumption

In redefinition and differential programming, three classes are created. Two of the classes created by differential programming are smaller than those created by redefinition. Thus differential programming requires fewer cells than redefinition. Since differential programming requires superclass traversing, cpu time is nearly the same.

The dynamic o-o programming creates the same three udos of wedge88 as do the other two types of programming. However, since only one class is created and modified, the required memory and cpu time are less than for the other two types of programming.

The above example involves only one instance wedge88. Suppose there are already one, ten, a hundred, or a thousand instances under the class wedge. Each instance has an instance variable shape. Let us measure the cpu time and memory needed to add two instance variables, color & material to wedge, and to set color red and material wood in one of the instances. Color and material of other instances are not yet set. Methods are used to set values of instance variables.

The dynamic o-o programming is:

```
(d-instance-var wedge color)
(defmethod (wedge set-color) (x) (!color x))
(m-call wedge88 set-color 'red)
(d-instance-var wedge material)
```

```
(defmethod (wedge set-material) (x)
  (!material x))
(m-call wedge88 set-material 'wood)
```

On the other hand, `defclass` and `make-instance` for all instances are re-executed in the redefinition programming.

Figure 6 compares the cpu time and required memory of the dynamic o-o programming with redefinition ([<*receiver*> <*message*> <*args*>] which is microcoded message passing in TAO is used instead of `m-call` in the redefinition programming.). In the dynamic o-o programming, instances are modified on demand. The udo of only one instance is newly created in this example. By contrast, udos of all instances are newly created in redefinition. Therefore, as instances increase, the required memory ratio becomes smaller. That is, the more instances, the better the dynamic o-o programming is. The measurement results of differential programming are similar to redefinition.

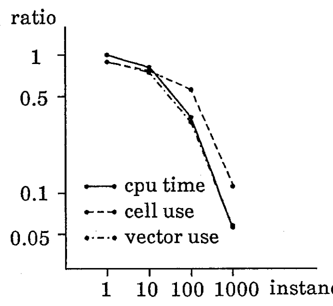

Fig. 6 Ratio of CPU time and memory consumption of dynamic o-o programming vs. redefinition programming at increasing instances in a class

The above examples involves only one class. Suppose there are 63 classes whose structure is a binary tree (Fig. 7) and there are already one, five, ten, or fifty instances in each class which has an instance variable shape. Let us measure the cpu time and memory needed to add two instance variables, color & material, to the root class `wedge1`, and to set color red and material wood in one of the instances in the leaf class `wedge63`. Color and material of other instances are not yet set.

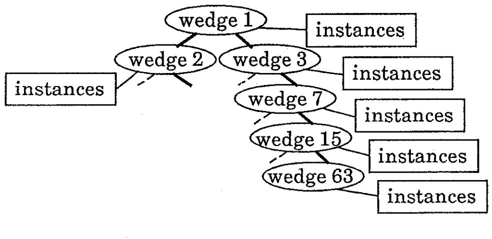

Fig. 7 Binary tree structure of classes

Figure 8 compares the cpu time and required memory of the dynamic o-o programming with redefinition. In the dynamic o-o programming, the udo of only one instance is newly created, and class vectors are modified, not newly created at superclass traversing. By contrast, udos of all instances and all class vectors are newly created in redefinition. Therefore, as instances increase, the required vector and cell ratios become smaller. The more instances in the class hierarchy, the better the dynamic o-o programming is.

The above example first involves an instance variable shape in an instance. The more instance variables and class variables, the better the dynamic o-o programming is.

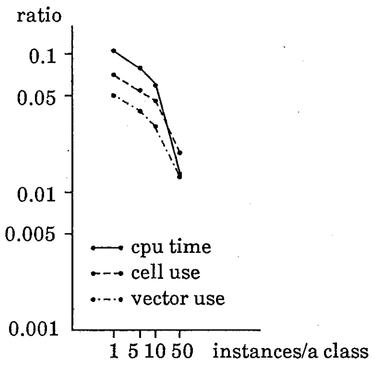

Fig. 8 Ratio of CPU time and memory consumption of dynamic o-o programming vs. redefinition programming at increasing instances in binary tree structured classes

(2) Dynamic Class Change

As knowledge continually changes, an instance may change its class. Therefore, a coordinated problem solving system must represent and support such dynamic class change.

For example, a class hierarchy is shown in Fig. 9. Mr. A is first a part-timer, but is hired as a regular engineer after several months. There are usually some relations between instance and class variables of an old class and those of a new class. d-instance and d-class-relation, described in **3.2.2**, are used for such dynamic class change.

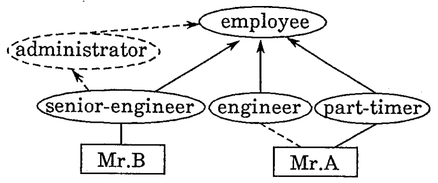

Fig. 9 Dynamic class change and dynamic superclass change

The dynamic o-o programming is:

```
(d-defclass employee () ((weekly-holiday 2))
  ())
(d-defclass engineer ()
  (salary (privilege 0)) (employee))
(d-defclass senior-engineer ()
  (salary (privilege 1)) (employee))
;This class is used in (3) of 3.2.3.
(d-defclass part-timer ()
  (average-of-monthly-working-time
  pay-by-the-hour (privilege 0)) (employee))
(!Mr.A (d-make-instance 'part-timer 'Mr.A
  privilege 1 pay-by-the-hour 25))
(d-class-relation part-timer engineer
  (!salary (* pay-by-the-hour (max  150
  average-of-monthly-working-time))))
(d-instance (:change part-timer engineer)
  'Mr.A)
```

Harmonia takes 6 ms, 78 cells, and 33 vector sizes to execute (d-instance (:change part-timer engineer) 'Mr.A).

Mr.A's privilege is 1 and it is different from the initial privilege value 0 in class part-timer. Therefore, selection rule ③, described in **3.2.2**, becomes valid and the privilege is set at 1 in the new udo.

Dynamic class change function is useful in parallel coordinated problem solving. It is difficult, although not impossible, for redefinition to represent it.

(3) Dynamic Superclass Change

When knowledge goes into details in problem solving, a new class may be inserted into the existing class hierarchy. At this time, related classes must change their superclasses.

For example, suppose the class mammal is a superclass of class dog and Tinker is an instance of a dog (Fig. 10). When the new class carnivore is inserted, class dog must replace superclass mammal for new superclass carnivore.

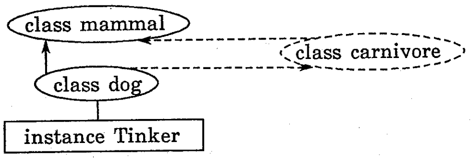

Fig. 10 Dynamic superclass change

This can be done by either redefinition or the dynamic o-o programming:

①Redefinition

```
(defclass mammal () ((respiration 'lungs)) ())
(defmethod (mammal respiration?)  ()
  respiration)
(defclass dog () ((leg 4)) (mammal))
(defmethod (dog get-leg) () leg)
(!Tinker (make-instance 'dog))
(defclass carnivore () ((food 'meat))
  (mammal))
(defmethod (carnivore food?)  () food)
(defclass dog () ((leg 4)) (carnivore))
(defmethod (dog get-leg) () leg)
(!Tinker (make-instance 'dog ))
```

When [Tinker food?] is executed, returned value is meat.

②The dynamic o-o programming

```
(d-defclass mammal () ((respiration 'lungs))
  ())
(defmethod (mammal respiration?)  ()
  respiration)
(d-defclass dog () ((leg 4)) (mammal))
(defmethod (dog get-leg) () leg)
(!Tinker (d-make-instance 'dog 'Tinker))
(d-defclass carnivore () ((food 'meat))
  (mammal))
(defmethod (carnivore food?)  () food)
(d-hierarchy dog (:change mammal carnivore))
```

When (m-call Tinker food?) is executed, returned value is also meat.

The redefinition programming requires the class dog redefinition, the class dog's method redefinition, and remake-instance for an instance Tinker in the class dog for the superclass change. By contrast, the dynamic o-o programming requires only d-hierarchy.

	time	cell	vector
Redefinition (no [Tinker food?])	1	1	1
Dynamic object-oriented (no (m-call Tinker food?))	0.4	0.4	0.2
Redefinition	1	1	1
Dynamic object-oriented	0.98	0.94	0.95

Table 2 Dynamic superclass change

The two types of programming are compared in terms of cpu time and memory use in Table 2. The upper half measurement in Table 2 does not contain [Tinker food?] or (m-call Tinker food?). Since the dynamic o-o programming creates new udos on demand, the udo of Tinker is still old. In redefinition, a new udo is created by (!Tinker (make-instance 'dog)) and class dog and its methods are redefined too. Thus the dynamic o-o programming requires less memory than does redefinition. Superclasses are not yet traversed in either programming case.

Since the lower half measurement in Table 2 contains [Tinker food?] or (m-call Tinker food?), the message is passed to the instance Tinker and a new udo is created too in dynamic o-o programming. Cpu time and memory use for check on dynamic modification in dynamic o-o programming should be compared with those for redefinition of class and its methods in the redefinition programming. The cpu time and memory use of the dynamic o-o programming is similar to redefinition in the most disadvantageous case (there is only one instance `Tinker` in the class `dog`). In the dynamic superclass change, the more instances, the better the dynamic o-o programming is.

Tinker has four legs. This is the same as the initial leg value of the class dog in the above example. Suppose Tinker lost one leg in an accident and dynamic superclass change from `mammal` to `carnivore` occurs for `dog`. In this case, the old value of leg is different from the initial leg value 4 in the old class vector of `dog`. Therefore, selection rule ③ becomes valid and leg is set at 3 in the new udo. By contrast, in redefinition, a user must remember that Tinker has lost a leg in an accident and remake a new instance as follows: `(!Tinker (make-instance 'dog leg 3))`

Suppose there is an instance Mr.B in class `senior-engineer` made as follows:

```
(!Mr.B (d-make-instance 'senior-engineer 'Mr.B))
```

and new class `administrator` is introduced into the class hierarchy (Fig. 9). And suppose the value of weekly-holiday in `administrator` is 1.

The dynamic o-o programming for the dynamic superclass change is only:

```
(d-defclass administrator ()
   ((weekly-holiday 1)) (employee))
(d-hierarchy senior-engineer
   (:change employee administrator))
```

In this case, weekly-holiday's old value of Mr.B is the same as the initial weekly-holiday's value 2 of old superclass `employee`. Therefore, selection rule ④ becomes valid and the weekly-holiday is set to 1. When a user wants to choose a new value in a different way, he may specify a value in an instance by `d-instance`.

It takes 0.3 ms, 5 cells, and 0 vectors to execute `(d-hierarchy senior-engineer (:change employee administrator))`. The comparison with redefinition in terms of cpu time and memory use shows that the tendency is the same as the example of `mammal`, `carnivore`, and `dog` shown in Fig. 10.

3.3 Function of Synchronizing Clocks

In parallel coordinated problem solving using distributed computers, computer clocks are used to compare the time when messages are sent, when events occur, and how long knowledge is held. For these reason, differences in computer clocks need to be minimized.

This subsection proposes a general method for synchronizing real clocks (also referred to as 'clocks') of distributed computers. It proves that this method finishes within a finite time and suppresses clock differences to network delay times and also describes experiments in synchronizing real clocks of ELIS AI workstations in *Harmonia* [Onai 88-1].

3.3.1 Synchronizing clocks of two computers

Two computers, M and N, are connected by a network. The times of M and N are represented by $T_{M\sim}$ and $T_{N\sim}$ respectively (Fig. 11).

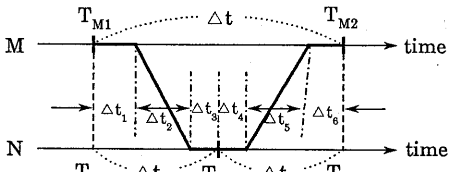

Fig. 11 Space-time diagram of two computers M and N

This method consists of several procedures (refer to Fig. 11).

Procedures for M

Proc-M1

M records T_{M1} and sends a message to N. Δt_1 is the time between the recording of T_{M1} and the message's output from M.

Proc-M2

M records T_{M2} when a message with the timestamp T_{N1} is returned from N. Δt_5 is network delay time and Δt_6 is the time between the reception of a message from N and the recording of T_{M2} .

Proc-M3

M compares T_{M1}, T_{N1}, and T_{M2} and sends a command, which depends on the result, to N.

3.1 When $T_{M1} < T_{N1} < T_{M2}$, M sends an end command to N.

3.2 When $T_{N1} \leq T_{M1}$, M sends N a command to set N's clock(T_N) ahead by $T_{M1} - T_{N1} + \Delta t_s$. Δt_s is the minimum unit of time that a computer clock can be set back or ahead.

3.3 When $T_{M2} \leq T_{N1}$, M sends N a command to set N's clock(T_N) back by $T_{N1} - T_{M2} + \Delta t_s$.

Proc-M4

When **3.1** is met in **Proc-M3**, all M's procedures are completed. Otherwise, **Proc-M1** begins again.

Procedures for N

Proc-N1

When N receives a message from M, N returns a message that includes T_{N1} to M. Δt_2 is the network delay time of a message sent from M to N. Δt_3 is the time between the reception of a message and the recording of T_{N1}. Δt_4 is the time between the recording of T_{N1} and the sending of a message that includes T_{N1}.

Proc-N2

N receives a command from M and accordingly sets its clock (T_N) ahead or back. Δt_m is the time required to adjust the clock.

Proc-N3

When the command is not an end command, **Proc-N1** begins again.

When $|T_{M1} - T_{N0}| < \Delta$, the time difference between M and N is less than Δ.

Fig. 11 shows that $\Delta t_a = \Delta t_1 + \Delta t_2 + \Delta t_3$,

$\Delta t_b = \Delta t_4 + \Delta t_5 + \Delta t_6$, and $\Delta t = \Delta t_a + \Delta t_b$.

3.3.2 Assumptions

There are four assumptions.

Assumption1

Time ticks are quantumized and represented as t_q. Since time ticks of computer clocks are quantumized by clock frequency, this assumption is reasonable.

Assumption2

$t_q \times n$ (n is a positive integer) passes in M and N at the same time. When $\Delta\tau$ is the time needed for synchronizing clocks, κ is the variation from the correct time rate, and Δt_s is the minimum unit of time the computer clock can be set back or ahead. If $\kappa \times \Delta\tau \ll \Delta t_s$, this assumption is reasonable. κ is less than 10^{-6} in typical crystal-controlled clocks.

Assumption3

$\Delta t_a \geq \Delta t_s$ and $\Delta t_b \geq \Delta t_s$

Assumption4

$\Delta t_q \ll \Delta t_s$ and $\Delta t_m < \Delta t_s$.

Δt_m is the time required to set the clock ahead or back.

Assumption3 and **assumption4** depend on the network and computers used.

3.3.3 Proof: The time difference between two computers is suppressed less than $\Delta t - \Delta t_s$

If $T_{M1} < T_{N1} < T_{M2}$ is reached using this method, the following paragraph proves that $|T_M - T_N| < \Delta t - \Delta t_s$.

When T_N is behind, we assume that $T_{M1} - T_{N0} \geq \Delta t - \Delta t_s$.

Assume that $T_{N0} \leq T_{M1} - \Delta t + \Delta t_s$. (1)

Since $T_{N0} + \Delta t_a = T_{N1}, T_{N0} = T_{N1} - \Delta t_a$. (2)

Since$T_{M1} < T_{N1} < T_{M2}$, $0 < T_{N1} - T_{M1} < \Delta t$ (3)

From (1) and (2), $T_{N1} - T_{M1} \leq \Delta t_a - \Delta t + \Delta t_s$.

Because of $\Delta t - \Delta t_a = \Delta t_b \geq \Delta t_s$, $T_{N1} - T_{M1} \leq 0$. However, this contradicts (3).

Therefore $T_{M1} - T_{N0} < \Delta t - \Delta t_s$.

When T_N is ahead, $T_{N2} - T_{M2} < \Delta t - \Delta t_s$ can be proved similarly.

Therefore, if $T_{M1} < T_{N1} < T_{M2}$ is reached using this method, $|T_M - T_N| < \Delta t - \Delta t_s$ (Q.E.D.).

3.3.4 Proof: Method convergence

Prove that T_{N1} converges between T_{M1} and T_{M2} ($T_{M1} < T_{N1} < T_{M1}$) by the repetition of procedures.

When T_N is behind, namely $T_{N1} \leq T_{M1}$, T_N is set ahead by $T_{M1} - T_{N1} + \Delta t_s$.

In the following proof, the time of the next cycle is represented with a prime symbol $'$. (ex. T'_{M1})

ΔT_M is the time between the beginnings of two consecutive procedures.

$T'_{M1} - T_{M1} = \Delta T_M$.

Since $T'_{N0} - T_{N0} = \Delta T_M + T_{M1} - T_{N1} + \Delta t_s - \Delta t_m$,
$T'_{M1} - T'_{N0} = T_{N1} - \Delta t_s + \Delta t_m - T_{N0} = \Delta t_a - \Delta t_s + \Delta t_m$.

Therefore $T_{M1} + \Delta t'_a - T'_{N1} = \Delta t_a - \Delta t_s + \Delta t_m$ (because $T'_{N0} + \Delta t'_a = T'_{N1}$)

Therefore $T'_{N1} - T'_{M1} = \Delta t'_a - \Delta t_a + \Delta t_s - \Delta t_m$. (4)

Since $\Delta t_s > \Delta t_m$ and the time tick is quatumized from the assumption, $\Delta t'_a = n' \times t_q$ (n' is a positive integer).

Therefore, finite repetition of the procedure cycle makes $\Delta t'_a \geq \Delta t_a$, namely $T'_{N1} > T'_{M1}$.

On the other hand, $T'_{M2} - T'_{N1} = T'_{M1} + \Delta t' - T'_{N1}$
(because $T'_{M2} - T'_{M1} = \Delta t'$)
$= \Delta t_a - \Delta t_s + \Delta t_m + \Delta t' - \Delta t'_a$ (because of (4))
$= \Delta t_a - \Delta t_s + \Delta t_m + \Delta t'_b$
(from the assumption, $\Delta t_a \geq \Delta t_s$)
> 0

Therefore finite repetition of the procedure cycle can make $T_{M1'} < T'_{N1} < T'_{M2}$.

When T_N is ahead, it can be similarly proved that finite repetition of the procedure cycle can make $T'_{M1} < T'_{N1} < T'_{M2}$.

Therefore, whether T_N is ahead or behind, finite repetition of the procedure cycle can make $T'_{M1} < T'_{N1} < T'_{M2}$ (Q.E.D.).

3.3.5 Experiments on *Harmonia*

Experiments were conducted using this method to set ELIS clocks in *Harmonia*. Messages used in the method were sent and received using TCP.

First, let us describe the experiments of synchronizing the clocks of two ELISs.

When the switch in Fig. 1 is turned off, *Harmonia* is isolated from other computers. Therefore, when procedures are executed in this condition, only packets for synchronizing clocks on Ethernet in *Harmonia* are transferred.

Δd is the initial time difference between the two ELISs clocks. When Δd is positive, an ELIS clock, on which the N procedures were executed, is ahead. When Δd is negative, the clock is behind. Before the procedure begins, Δd is set by sending messages between the two ELISs. The slight error in Δd did not affect the experiments. Δt_s is 20 ms in *Harmonia*.

Tables 3 and 4 show experiment results when the switch in Fig. 1 is off. The synchronization method is repeated one-hundred times for each Δd.

According to Tables 3 and 4, the average, max., min., and standard deviation of the number of clock adjustments, and the average, max., min., and standard deviation of Δt are independent of Δd.

$\triangle d$ (sec)	Number of Clock Adjustments			
	Average	Max	Min	Standard Deviation
+100	1.33	3	1	0.53
+10	1.36	3	1	0.59
+2	1.38	4	1	0.64
−2	1.22	3	1	0.50
−10	1.26	3	1	0.54
−100	1.25	3	1	0.50

Table 3 Initial time difference ($\triangle$d) and number of clock adjustments (switch off)

$\triangle d$ (sec)	$\triangle t$ (ms)			
	Average	Max	Min	Standard Deviation
+100	177.4	240	160	16.2
+10	176.8	220	160	16.4
+2	176.8	220	160	15.9
−2	176.6	220	160	17.4
−10	179.6	260	160	20.0
−100	177.6	240	160	16.3

Table 4 Initial time difference ($\triangle$d) and $\triangle$t (switch off)

Since the time needed for synchronizing clocks ($\Delta\tau$) is (cycle-time of the procedures × repetition number) and the tables show that the average Δt is 180 ms and the average repetition number is 2.3, the average $\Delta\tau$ is 400 ms. Since 400 ms ($\Delta\tau$) $\times 10^{-6} \ll$ 20 ms (Δt_s), **assumption2** is true.

As a result of message packet analysis using a network protocol analyzer, $\Delta t_a > 20$ ms and $\Delta t_b > 20$ ms. Therefore **assumption3** ($\Delta t_a \geq \Delta t_s$ and $\Delta t_b \geq \Delta t_s$) is fulfilled.

Since in *Harmonia*, Δt_s is 20 ms, t_q is 180 ns, and Δt_m is less than 100 μs, **assumption4** ($\Delta t_s > \Delta t_m$ and $t_q \ll \Delta t_s$) is fulfilled.

This experiment shows that this method can reduce the average of time difference ($\Delta t - \Delta t_s$) of two ELISs clocks in *Harmonia* to about 160 ms.

When the switch turns on, Δt_a and Δt_b, which comprise Δt in the synchronizing clocks, depend on network traffic. Experiment results in the normal time zone ($\Delta d = +2$, 100 times of synchronization) show that clock adjustments have an average of 1.36, a max. of 3, a min. of 1, and a standard deviation of 0.56. In addition, Δt (ms) has an average of 193, a max. of 260, a min. of 160, and a standard deviation of 17.5.

With the switch on, Δt is a little larger than when the switch is off. The less network traffic, the smaller Δt.

When there are n computers, M and N_i ($1 \leq i \leq n-1$), the time difference of clocks between N_i and N_j ($i \neq j, 1 \leq i,j \leq n-1$), that is $|T_{Ni} - T_{Nj}|$, is suppressed less than $\Delta t_i + \Delta t_j - 2\Delta t_s$ using our method. (Δt_i is Δt in the case of M and N_i.)

Next we described the synchronizing clock experiment results for six ELISs in *Harmonia*. One ELIS was a computer M, and each of the other five ELIS was a computer N. In addition, $\Delta d = +2$, the clocks were synchronized 100 times, and the switch was turned off. As expected from the experiment results with two ELISs, the average time difference between N_i and N_j ($i \neq j, 1 \leq i,j \leq 5$) was ab out 320 ms.

Next this method is compared with Lamport's method [Lamport 78]. Using Lamport's method,

$\varepsilon \approx d \times (2\kappa\tau + \xi)$.

(ε is the time difference between two clocks, d is 1 for Ethernet, κ is the variation from the correct time rate, τ is the time between the sending of synchronizing clock messages, and ξ is the unpredictable message delay.)

Since there is the unpredictable value ξ in ε, Lamport's method can not guarantee that the time difference of two clocks is less than a value to be detected. On the other hand, our method can guarantee that the time difference of two clocks is less than Δt - Δt_s.

When this method is executed every τ seconds, the maximum time difference between the two clocks is $\Delta t - \Delta t_s + 2\kappa\tau$. Since κ is less than 10^{-6} for typical crystal-controlled clocks, when τ is 20000 sec and $\Delta t - \Delta t_s$ is 160 ms, the time difference between the two clocks is a maxmum of 200 ms.

4 Conclusions

Our aim in developing *Harmonia* is to establish the fundamental technology for parallel coordinated problem solving. This paper described the basic functions of *Harmonia*, in particular communication, dynamic object-oriented programing for knowledge modification, and real clock synchronization. This computing system consists of six AI work-station ELISs connected by Ethernet. It's basic functions are implemented using the multiple-paradigm language, TAO.

Various kinds of communication were made possible by combining in *Harmonia* the two levels of communication functions: ordinary communication using mailboxes and emergency communication using inter-IA interrupt.

The target was virtual communication among distributed IAs. First, instead of the IA name and the ELIS name in which it exists, the IA name only was required in the two communication levels. A correspondence between an IA name and a physical ELIS name was given on metalevel and stored in the communication server.

Since the manager's and solvers' knowledge continually changes in a parallel coordinated computing system, when knowledge is represented using objects, the objects must be modified dynamically. This paper introduced the dynamic object-oriented programming function using such modifiable objects, i.e. **dynamic objects**, into *Harmonia*. The function was implemented in the improved TAO. This improvement took into account the coexistence of dynamic objects and ordinary TAO objects, and the preservation of the TAO functions.

Harmonia provides various kinds of functions such as d-defclass, d-make-instance, m-call, d-class-var, d-instance-var, d-hierarchy, d-option, d-class-relation, and d-instance to operate the dynamic objects. Classes are modified just when these functions are executed. By contrast, instances are modified on demand.

Some knowledge modification programs are run on ELIS in *Harmonia*. The measurements show that cpu time and memory use in the dynamic object-oriented programming are less than that in the ordinary object-oriented programming, e.g. redefinition and differential programming. The more classes, instances and variables are, the less cpu time and memory use are. Even as the program becomes more complex, this tendency does not change.

Thus, it is concluded that the dynamic objects are requisite and the dynamic object-oriented programming is more natural and more efficient than the ordinary object-oriented programming from the standpoint of knowledge modification. The dynamic objects are also useful for interactive program development. Since human beings are excellent managers in problem solving, they can use dynamic objects comfortably at *Harmonia* terminals.

This paper also proposed a general method for synchronizing clocks of distributed computers. It proved that this method finishes within a finite time and suppresses clock differences to network delay times. It also described experiments on ELIS AI workstations in *Harmonia*, which has a clock-synchronizing function.

Using this method, the average time difference of two ELIS clocks in *Harmonia* can be reduced to about 320 ms, and the average clock time difference between an ELIS on which M procedures are executed and that on which N procedures are executed can be reduced to about 160 ms.

When this method is executed every τ seconds, the maximum time difference between the two clocks is $\Delta t - \Delta t_s + 2\kappa\tau$. Since κ is less than 10^{-6} for typical crystal-controlled clocks, when τ is 20000 sec and $\Delta t - \Delta t_s$ is 160 ms, the time difference between the two clocks is a maximum of 200 ms. Therefore in coordinated problem solving using *Harmonia*, when the time difference between event occurrences is more than the maximum time difference of clocks $\Delta t - \Delta t_s + 2\kappa\tau$, the earlier event can be determined.

This method can only be applied when

$\Delta t_a \geq \Delta t_s, \Delta t_b \geq \Delta t_s$, and $\Delta t_m < \Delta t_s$.

Therefore, the specifications of the computers and networks determines the viability of this method, and if viable, the time difference of the clocks ($\Delta t - \Delta t_s$) can be estimated.

Finally the basic functions described in this paper can be used effectively and pleasantly in parallel coordinated problem solving using *Harmonia*.

Acknowledgment

Discussions with and comments by Ikuo Takeuchi, Nobuyasu Ohsato, Ken-ichiro Murakami, Yoshiji Amagai, Ken-ichi Yamazaki, Yukio Tsuruoka (NTT Software Labs.), Yoji Kogure (Toho System, Inc.), and Yasushi Hibino (NTT Human Interface Labs.) were of great benefit.

References

[Bobrow 87-1] Bobrow, D.G., DeMichiel, L.G., Gabriel, R.P., Keene, S., Kiczales, G., and Moon, D.A.: "Common Lisp Object System Specification", Draft X3 Document 87-002, 1987.

[Bobrow 87-2] Bobrow, D.G. and Kiczales, G.: "Common Lisp Object System Specification", Draft X3 Document 87-003, 1987.

[Goldberg 83] Goldberg, A. and Robson, D.: "Smalltalk-80: The Language and Its Implementation", Reading, Massachusetts, Addison-Wesley, 1983.

[Erman 80] Erman, L.D., Hayes-Roth, F., Lesser, V.R., and Reddy, D.R. : "The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty", Comput. Surv., vol.12, No.2, p.213-253, 1980. Computing", MacGraw-Hill, Inc., 1984.

[Filman 84] Filman, R.E. and Friedman, D.P.: "Coordinated Computing", MacGraw-Hill, Inc., 1984.

[Kornfeld 81] Kornfeld, W.A., and Hewitt, C.E. : "The Scientific Community Metaphor", IEEE Trans. Syst. Man Cybern., vol.SMC-11, No.1, p.24-33, 1981.

[Lamport 78] Lamport, L. : "Time, Clocks, and the Ordering of Events in a Distributed System", CACM, vol.21, No.7, p558-565, 1978.

[Smith 85] Smith, R.G., and Davis, R. : "Distributed Problem Solving: The Contract Net Approach", Proc. 2nd Natl. Conf. Canadian Soc. Comput. Stud. Intell, Toronto, p278-287, 1985.

[Symbolics 86] "Symbolics Common Lisp : Language Concepts", Symbolics, Inc. ,1986.

[Onai 86] Onai, R. and Takeuchi,I.: "Parallel Problem Solving Programming in TAO" (in Japanese), 信学技報, COMP86-58, 1986.

[Onai 88-1] Onai, R.: "Synchronizing Clocks of Distributed Computers" (in Japanese), 信学技報, CPSY88-52, 1988.

[Onai 88-2] Onai, R. and Tsuruoka, Y.: "Proposal and Evaluation of Dynamic Object-Oriented Programming" (in Japanese), 信学論(D), vol.J71-D, No.12, 1988.

[Post 81] Postel, J.: "Transmission Control Protocol", DARPA Internet Program Protocol Specification, September 1981.

[Takeuchi 86] Takeuchi, I., Okuno, H. and Ohsato, N.: "A List Processing Language TAO with Multiple Programming Paradigms", New Generation Computing Vol.4, No.4, pp.401-444, 1986.

[Watanabe 87] Watanabe, K., Ishikawa, A., Yamada, Y. and Hibino, Y.: "A 32b List Processor", IEEE International Solid-State Circuits Conference, pp.200-201, 1987.

[Weinreb 83] Weinreb, D., Moon, D. and Stallman, R.: "Lisp Machine Manual", Fifth Edition, System Version 92, LMI, 1983.