Translation of the IPSJ Best Paper Award papers

Object-oriented Real-time Programming in Concurrent Process Language

KATSUMI MARUYAMA* and NOBUYUKI WATANABE*

Object-oriented programming is a programming approach for improving switching program maintainability. A switching program is a severe real-time and multi-processing system, and therefore suitable concurrent object models and an appropriate language are necessary. This paper discusses a concurrent object model for real-time multi-processing applications, and presents object-oriented programming techniques using an accustomed concurrent process language. Compared with the development of a new and ideal object-oriented language for real-time applications, this approach may be provisional. However, this approach is advantageous in that developments cost and training time are insignificant, since existing environments can be used and programmer education shortened. In this paper, Chill is used as the base language, but these techniques may be applied to other concurrent process languages.

1. Introduction

Switching programs (the control program of telephone switching systems) have very long lives (more than 10 years), are very frequently updated (new service introductions), and are very large (Mega lines of codes) systems. Thus, improving the program maintainability has been strongly required. To answer this requirement, object-oriented programming is promising.

A switching program must provide severe real-time and multi-processing services, controlling many switching system resources. Therefore, conventional object-oriented programming, such as with Smalltalk-80 [1], are insufficient, especially with respect to concurrent processing capability and run-time efficiency.

We have been studying the object-oriented switching program, and have devised a "**concurrent object model**" for real-time multi-processing applications. To implement a program of this model, the development of a new ideal object-oriented language is desired. However, the introduction of a new language requires long development time as well as programmers to be educated. Object-oriented programming in an accustomed language, if possible, is very advantageous. This paper presents object-oriented programming techniques for the concurrent process language "Chill" [3]. Chill was designed by CCITT for real-time program developments, and has been widely used all over the world, including NTT. These techniques can also be applied to other concurrent process languages. A "very simple" pre-processor can improve readability and writability, almost to the same level of programs written in genuine object-oriented languages. A prototype switching program was developed using these techniques, and the obtained results were satisfactory.

2. Switching Programs and Object-oriented Programming

A switching program is a typical real-time and multi-processing system. It controls very many thousands of common resources, such as subscriber lines, trunk lines, switching paths, service circuits, etc. These resources have the following characteristics: (a) Each has its own state, (b) there are multiple instances of the same class, (c) they are allocated/freed from/to idle resource pools on request, and (d) their actions are activated by commands. Other components of a switching program, e.g., call controls, service analyzers, etc., also have similar attributes. This shows that most components of a switching program have "object-oriented" natures.

Therefore, logical switching components can be implemented as software objects, and a switching program can be systematically constructed as a collection of communicating objects. Because a switching program must provide several thousands of telephone switching services at the same time, a very efficient multi-processing capability is required. We designed the following "**concurrent object model**" which includes "**concurrent objects**" and "**sequential objects**". The former model has concurrent processing capability with some extent of overhead, while the latter model has no concurrent processing capability with little overhead. (See Fig. 1)

This is a translation of the IPSJ Best Paper Award paper that appeared originally in Japanese in Transactions of IPSJ, Vol. 31, No. 1 (1990), pp. 88-97.

*NTT Communication Switching Laboratories.

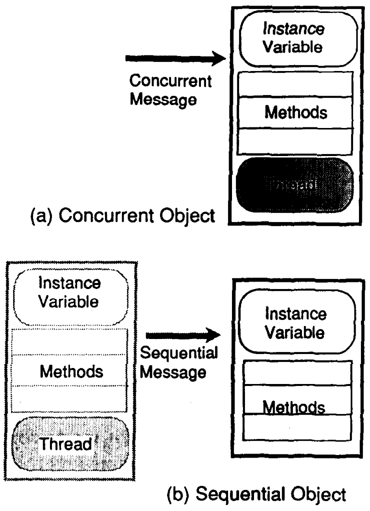

Fig. 1 Concurrent object and sequential object.

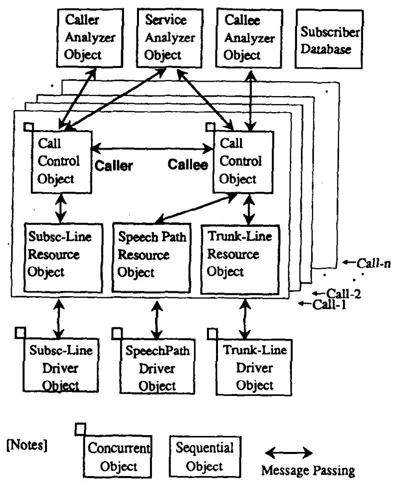

Fig. 2 Switching program model based on object-oriented concept.

(1) Concurrent Object

A concurrent object is a capsule comprising "instance variables", "methods" and a "thread". Here, the thread is a concurent execution unit. Concurrent objects can run concurrently by themselves. Message passing to concurrent objects is called "concurrent message". A concurrent message is an asynchronous communication, and a sender object is not blocked when a message is sent out. In a switching program, asynchronous communication is essential because there are many situations where the message sender can not be blocked waiting for the message acceptance by a receiver.

Concurrent objects, however, have time-and-space overheads because they require run-time context switching and their own stack memory spaces. Therefore it is not economical to implement thousands of resource objects as concurrent objects. Objects requiring no concurrency are to be implemented as sequential objects.

(2) Sequential Object

Sequential objects are capsules of an "instance variable" and "methods". They have no threads and cannot be executed by themselves. They are executed by the threads of message-sender-side objects. Therefore, message sender and message receiver are sequentially executed. Message passing to sequential objects is called "**sequential message**".

Fig. 2 shows a general view of the object-oriented switching program system, in which all components are implemented as objects [8, 9]: (a) Service analyzer objects determine the service from the dialed directory number, (b) service scenario objects describe the call handling procedure of each service (ordinary call, free call, etc.), (c) call control objects are allocated one per call basis, and carry out the call handling according to the service scenario, (d) resource objects such as subscriber line resource objects, switched path resource objects, and trunk line resource objects represent the resources of switching systems. These components are allocated/freed from/to idle resource pools. Components requiring concurrency (ex. call control objects) are implemented as concurrent objects, but most components (such as resource objects) are implemented as sequential objects. When a call is originated, and originating-side call control object (called "caller") is allocated. The caller receives dialed directory numbers and asks for the service analyzer object. The service analyzer object determines the service and the terminating line. Then, a terminating-side call control object (called "callee") is created. Callers control the originating-line-side, and callees control the terminating-line-side. In this way, call processing proceeds.

3. Concurrent Object Model for Real-time Applications

This chapter explains concurrent objects and sequential objects of the "concurrent object model" in detail.

3.1 Concurrent Object and Concurrent Message

A concurrent object has the following characteristics:

(1) A concurrent object is a capsule of "instance variables", "method" and a "**thread**".

(2) A thread is a concurrent execution unit, and consists of a ThCB (Thread control block) and a Stack area. A thread provides an environment for method execution. Concurrent objects can run by themselves using their own threads.

(3) Message passing to concurrent objects is called

concurrent messages and are expressed as follows,

$$\beta \lll \mu(\pi);$$

where, β: destination object ID; μ: message name; and π: message parameters. When this statement is executed, the message "$\mu(\pi)$" is sent to the destination object "β".

(4) Concurrent messages are asynchronous communication, and sender objects are not blocked when messages are sent out. A concurrent object has its own message queue, and delivered messages are put into this queue. Under the scheduler's control, a concurrent object gets a message from its message queue, and executes the method required by the message.

(5) If a reply value is returned, a pair of "request-message sending" and "reply-message receiving" is used. The following extended expression is provided for readability:

$$x := \beta \lll \mu(\pi);$$

here, "x" is a variable that receives the reply value.

3.2 Sequential Object and Sequential Message

A sequential object has the following characteristics:

(1) A sequential object is a capsule of "instance variables" and "methods".

(2) Having no threads, they cannot be executed by themselves. They are executed by the threads of message-sender-side objects. Therefore, message sender and message receiver are sequentially executed (in fact, this message passing is mapped to an indirect procedure call (see Chapter 5)).

(3) Message passing to sequential objects is called "**sequential message**", and is expressed as follows:

$$x := \beta \ll \mu(\pi);$$

here, x is a variable that receives the reply value.

3.3 Class Definition, Instance Object and Class Object

In Smalltalk, concepts of **class definition, instance object** and **class object** are introduced. These concepts are very convenient and simplify the language model; therefore, we have adopted them. A class definition includes definitions of a class object and instance objects. One class definition accompanies one class object. A class object provides common features to the objects in the class, such as object creation. Instance objects are created by sending "Create" messages to the class object. A class object is a sequential object.

The class definition has the following contents:

```
ClassName:  CLASS;
            Export/Import Part
            CLASSVARS
            Definition-of-class-variables
            INSTANCEVARS
            Definition-of-instance-variables
            CLASSMETHODS
            Definitions-of-classe-methods
            INSTANCEMETHODS
            Definitions-of-instance-methods
            ENDCLASS;
```

4. Short Introduction of Chill

Chill is a suitable language for our concurrent object model implementation because of the following features:

[Notes: In the following examples, the Chill syntax is slightly modified/simplified for explanatory purposes.]

(1) A strongly typed language (In Chill, terminology "**mode**" means "data type"): A **MODE** statement defines mode names. Explicit mode conversion is possible using notations "modename (expression)" and "modename (variable)", The following are self-explanatory.

```
Ex.  MODE aRefM=REF a; MODE bRefM=REF
     b;
     DCL aRef aRefM; DCL bRef bRefM;
     aRef:=aRefM(bRef); bRefM(aRef):=bRef;
     /*Explicit mode conversion*/
```

(2) Module Construct: The **MODULE . . . END** construct encloses data definitions and procedure definitions, as in Modula-2 [7]. Module is a convenient capsuling facility. Grant and Seize statements correspond to Export and Import of Modula. The name "v" granted (exported) in the module "λ" can be referred to as "$\lambda!v$" in a seizing (importing) module.

```
Ex.  cc:  MODULE;
          GRANT x, y;
          FROM dd SEIZE s, t; /*"dd!s" means "s"
                                    of module "dd"*/
          . . . data definitions and procedure defini-
          tions . . . .
          END cc;
```

(3) Light-weight Process: PROCESS . . . END construct defines a light-weight-process. A **START** expression creates a process instance, and returns its process-ID values. (**INSTANCE** mode)

```
Ex.  ee:  PROCESS (i, j INT);
          . . . data definitions and action statements . . .
          END ee;
          VAR MyPID INSTANCE;
          MyPID:=START ee (10, 20);
```

(4) SIGNAL(Message): Signal is an asynchronous inter-process-communication message. Signal is sent to the specified target process directly. Signal can accompany parameters. Chill signal is suitable for implementing concurrent messages of our model.

Ex. **SIGNAL** Sigl=(INT), Sig2=(INT, BOOL);
SEND Sigl (i+j) **TO** DestinationPID;

Here, DesitinationPID is the process-ID of the destination process. Sigl has an integer parameter, and Sig2 has an integer and a Boolean parameters;

(5) Selective Message Receiving: Chill has a RECEIVE-CASE statement, which can receive one of the nominated messages. Here, a matched action is activated.

Ex. **RECEIVE CASE**
(Sigl **IN** x): . . . *action to be taken when Sigl is received . . .*
(Sig2 **IN** y, y): . . . *action to be taken when Sig2 is received . . .*
ESAC;

In this example, if "Sigl" is received, the parameter value is stored to the local variable "x", and following statements are executed.

(6) A Conditional Critical Region like Hoare's Monitor [5]: This is conveniently used to implement resource allocation objects which require exclusive accessing.

5. Internal Structures of Objects

In this chapter, we explain how the aforementioned concurrent object model is achieved in Chill.

5.1 Concurrent Object

Concurrent objects can be conveniently mapped to Chill processes: Instance variables are mapped to process local variables; methods are mapped to process statement-lists; and concurrent object definitions are mapped to Chill process definitions. A concurrent "instance" object is created by the "Start" expression. Each process instance is identified by the "**INSTANCE** mode" value, whih is returned by the **START** expression, and this is used as the "**concurrent object ID**".

Figure 3 shows a concurrent object example in Chill. A consumer has an endless loop (**DO FOR EVER**), in which there is a **RECEIVE_CASE** statement. Arriving messages are received and matched actions are executed in the RECEIVE-CASE statement.

5.2 Concurrent Message Sending

The "signal" of Chill is an asynchronous inter-process-communication mechanism, and is very convenient as a concurrent message. Concurrent messages are directly mapped to Chill Signals. A message is sent by the SEND statement, and is received by the RECEIVE CASE statement. Concurrent message sending is mapped to the SEND statements as follows:

```
SIGNAL  Msgl = (INT,BOOL); /* Message (=signal) definition */
SIGNAL  Msg2 = (INT,INT);
DCL  PObj, CObj  INSTANCE;

Producer:PROCESS( );  /* Concurrent object (=process) definition */
     ....
   DO  FOR  EVER;  /* Endless loop */
       ....
       SEND Msgl(i,j) TO CObj;  /* Send "Msgl" messgae to CObj*/
       ....
       SEND Msg2(i,K) TO CObj;
    OD;
       ....
END Producer;

Consumer:PROCESS( );  /* Concurrent object (=process) definition */
         ....
       DO FOR EVER:  /* Endless loop */
         RECEIVE CASE  /* receive nominated messages */
         (Msgl IN x,y): ...actions to be taken when Msgl is received...
         (Msg2 IN x,z): ...actions to be taken when Msg2 is received...
         ESAC;
       OD;
       ....
END Consumer ;
/*** Concurrent object creation ***/
PObj := START Producer( ); /* Create a concurrent object instance "Pobj" */
CObj := START Consumer( ); /* Create a concurrent object instance "Cobj" */
```

Fig. 3 Concurrent object implementation in Chill process.

```
 Concurrent ObjectID<<<MessageName (Parameters);
                          ⇓
SEND MessageName (Parameters)
                              TO ConcurrentObjectID;
```

(where, MessageName: Chill Signal Name; and ConcurrentObjectID: Chill Process ID)

5.3 Sequential Object and Class Definition

As explained in sub-chapter 3.3, sequential objects are defined by class definitions; we have also adopted the concepts of class objects. The class definition includes definitions of (a) class variables, (b) instance variables, (c) class methods and (d) instance methods. They can be mapped to Chill features as follows:

(a) Class variables can be mapped to module-level variable declarations.

(b) Instance variables definition can be mapped to a structure-mode definition. Instance object creation corresponds to the variable allocation of the specified "mode", and the ID of the allocated variable is used as the **sequential object ID**.

(c) Class method definitions and (d) instance

```
CircleM :MODULE; /*** A Class definition in a Chill module ***/
      GRANT Circle, Methods, New, GetTotalNum, Enlarge, Move;

  /*** Instance Variable Definition ***/
      MODE Circle =
           STRUCT(E REF Methods,  /* Link to a Method Table */
                 Center STRUCT(x, y INT), r INT);

  /*** Class Variable Definition ***/
      DCL   TotalNum INT  INIT:=0;

  /*** Class Method Definition ***/
  New :PROC( )RETURNS(REF Circle); /* Creates an instance object */
       DCL   p REF Circle;
        p := ALLOC(Circle);
        p-> :=[ADDR(MethodTable),[0,0],1];
        RETURN p;
       END;
  GetTotalNum:PROC( )RETURNS(INT);
              RETURN TotalNum;
             END ;

  /*** Instance Methods Definition ***/
  Move :PROC  (ObjP REF Circle, x, y INT); /* Moves the object */
             ObjP->.Center.x += x;
             ObjP->.Center.y += y;
        END ;
  Enlarge :PROC(ObjP REF Circle, n INT); /* Enlarge the object */
             ObjP->.r *= n;
        END;

  /*** Method Table ***/
      MODE  Methods =
            STRUCT(Move  PROC(ObjP REF Circle, x,y INT),
                   Enlarge PROC(ObjP REF Circle, x,y INT));
      DCL  MethodTable Methods INIT = [Move, Enlarge];

END CircleM ;

/*** Object creation and message passings  ***/
DCL c1 REF CircleM!Circle ;
c1 := CircleM!New( );  /* Instance object "c1" is created. */
CALL  c1->.E->.Enlarge(c1,10); /* Enlarge the object "c1". */
CALL  c1->.E->.Move (c1,15,7); /* Move the object "c1". */

[Notes]  "var += exp;" means "var:=var+(exp);"
```

Fig. 4 Sequential object implementation in Chill module.

method definitions can be mapped to procedure definitions.

A Chill module is very convenient as a class definition capsule. Figure 4 shows a Chill module used as a class definition capsule. This includes definitions of (a) an instance variable mode (Circle), (b) a class variable (TotalNum), (c) class methods (New, GetTotalNum) and (d) instance methods (Enlarge, Move). The instance object represents a "circle" characterized by a center position (x, y) and a radius (r). When a message arrives at an object, a relevant method is selected in the mechanism explained in the next sub-chapters.

Chill has a special "conditional critical region" module: REGION . . . END construct. The class object implemented using this region is called a "critical region module". A critical region object is convenient for resource allocation algorithm because of its exclusive access nature. (A concurrent object is also applicable for resource allocation algorithm, because received messages are serialized in its message queue and processed one by one.)

5.4 Sequential Message

Let's consider the following message sending:

$$x:=\beta \ll \mu(\pi);$$

When a message arrives at an object, a relevant method must be selected and executed. Real time processing requires high run-time efficiency; the run-time method search mechanism, as in Smalltalk-80, is too heavy to use. Therefore, we designed the following mechanisms.

(1) Compile-time Method Binding for Class Methods

As a class definition is mapped to a Chill module, and Chill is a strongly typed language, the mode and the defining module of the destination object "β" can be analyzed at compile time. From this mode and module information, the related method having the name "μ" can be uniquely analyzed at compile time. In this way, a method can be analyzed and fixed at compile-time.

This mechanism is very efficient (no overheads are added to procedure calls), but polymorphism flexibility of object-oriented programming is limited (as will be explained later). Therefore, this compile-time method binding is applied to class methods. A messge passing to a class object is mapped to a Chill procedure call as follows:

```
Var:=ClassName«MessageName (Parameters);
                    ⇓
 Var:=ClassName!MessageName (Parameters);
```

As you can see, the message-name is prefixed by a module-name. Thus, even the same message name is used in different class definitions, no name clash occurs.

(2) Indirect Method Binding for Instance Methods

Let's assume that there are classes "A", "AB" and "AC", where "AB" and "AC" both inherit "A", and that all of them have the same name method "Print". (Inheritance mechanism is explained in Chapter 7.) Let's also assume that a pointer variable "ObjP" can point to any object of classes "A", "AB" or "AC". When a "Print( )" message is sent to the object pointed by "ObjP", it is requested the right method is selected and executed. (This feature is called polymorphism.) To attain this purpose, the following mechanism has been devised (See Fig. 5):

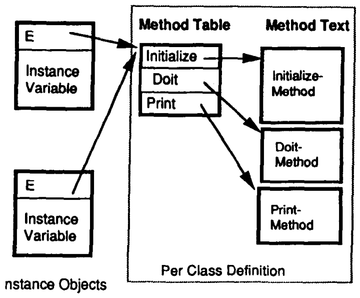

Fig. 5 Internal structure of sequential object.

(a) A STRUCT-mode table "**Method Table**" is allocated for each class definition as shown below. Each field is accessed by a "message name" and contains a "procedure mode" value (pointer to procedure) pointing to the relevant method text.

```
MODE Methods=STRUCT(MessageName PROC(. . .),
                    . . . . . . . .);
DCL MethodTable Methods
                    INIT=[. . . method-name . . ];
```

(b) Each instance variable (i.e. instance object) shall have a pointer to the "Method Table".

(c) When a message arrives at the object at run-time, the "Method Table" is accessed, and then the relevant method is accessed.

In this mechanism, objects having the same message interfaces (this means that both objects have the same mode "Method Tables") can be treated in the same way, even though they belong to different class definitions. This feature provides flexibility and advantages to program development. For example, a switching system has several kinds of trunk lines (lines to carry inter-office calls), such as MF (Multi-Frequency-signaling) lines, DP (Dial-Pulse-signaling) lines, and CCS7(Common-Channel-Signaling-No. 7) lines. They are implemented as resource objects. They require different method texts (i.e. different class definitions), but can have the same messge interface. When a trunk line is allocated from the idle trunk line pool, it may be of any class, but regardless of the class it can be used in the same way as another.

Therefore, the indirect method binding is applied to instance methods.

```
     ObjectID « MessageName (Parameters);
                    ⇓
CALL ObjectID→.E→. MessageName
                    (ObjectID, Parameters);
```

Here, "ObjectID" is a Pointer variable to the instance variable, "E" is a field name of the instance variable which points to the MethodTable, and "MessageName" is a procedure-mode field name of the MethodTable.

6. Simple Pre-processor

The object-oriented programming technique in Chill is explained in Chapter 5. However,program readability and writability are not as good as a genuine object-oriented language program's. To improve them, a "very simple" pre-processor is introduced. An intelligent pre-processor can provide more skillful and complex features [6], but a simple pre-processor is welcomed from the viewpoints of debugging, education, etc. Figure 6 shows the effect of the pre-processor:

```
     [Pre-processor input]               |    [Pre-processor    output]
                                         |
   CircleM :CLASS                        | CircleM:MODULE
     EXPORT Circle,  New,                |    GRANT  Circle, Methods, New,
         GetTotalNum, Enlarge, Move;     |          GetTotalNum, Enlarge, Move;
                                         |
  INSTANCEVARS                           |
      MODE Circle =                      |     MODE Circle =
        STRUCT(METHODLINK,               |       STRUCT(E  REF  Methods,
         Center STRUCT(x,y INT),         |          Center STRUCT(x,y INT),
         r INT);                         |          r INT);
    CLASSVARS                            |
         DCL  TotalNum INT INIT:=0;      |
    CLASSMETHODS                         |
    New:PROC( )RETURNS(REF Circle);      |
         DCL   p REF Circle;             |
         p := ALLOC(Circle);             |
         p->:=[METHODLINK,[0,0],1];      |       p->:=[ADDR(MethodTable),[0,0],1];
         RETURN p;                       |
      END;                               |
    GetTotalNum:PROC()RETURNS(INT);      |
                 RETURN TotalNum;        |
                END ;                    |
    INSTANCEMETHODS                      |
    Move:PROC(SELF, x, y INT);           |  Move:PROC(SelfP REF  Circle,
                                         |            x,y  INT);
                 SELF.Center.x += x;     |        SelfP->.Center.x += x;
                 SELF.Center.y += y;     |        SelfP->.Center.y += y;
          END ;                          |      END  ;
    Enlarge:PROC(SELF,                   |  Enlarge:PROC(SelfP REF Circle,
                n INT);                  |                 n INT);
                 SELF.r *= n;            |         SelfP->.r *= n;
             END;                        |      END;
                                         |
                                         |    /***** Method Table is allocated ****/
                                         |     MODE  Methods = STRUCT(
                                         |        Move PROC(ObjP REF Circle, x,y INT),
                                         |        Enlarge PROC(ObjP REF Circle, x,y INT));
                                         |     DCL  MethodTable Methods
                                         |         INIT = [Move, Enlarge];
  ENDCLASS CircleM ;                     |  END CircleM;
                                         |
  DCL cl REF CircleM!Circle ;            |
  cl := CircleM << New( );               |  cl:= CircleM!New( );
  cl << Enlarge(10);                     |  CALL cl->.E->.Enlarge(Cl,10);
  cl << Move(15, 7);                     |  CALL cl->.E->.Move(Cl,10);

[Notes] Only converted parts by a pre-processor are shown on the right-hand side.
```

Fig. 6 A simple pre-processor and a class definition.

The left-hand-side and right-hand-side show input program and output program of the pre-processor, respectively. The key word "SELF" means the instance variable of itself. Process definitions are used as concurrent object definitions directly.

The pre-processor performs conversion as follows:

(a) A CLASS . . . ENDCLASS is converted to a "MODULE . . . END".

(a) A "Method Table" is allocated and initialized.

(b) A Method Table link is put into the instance variable mode definition.

(c) The method formal parameters "SELF" are converted to "SelfP PEF mode-name".

(d) Instance variable references "SELF" are converted to "SelfP→".

(e) Sequential message sendings are converted as aforementioned.

(f) Concurrent message sendings are converted as aforementioned.

7. Inheritance Technique

Let's assume that the class definition (Chill module) "A" has the instance variable definition ("a") and method definitions ("New", "m1"). Other modules can seize and inherit these definitions. Using this technique, class definition "AB" inherits from class definition "A" in Fig. 7.

This inheritance technique requires the programmer to understand the inheritance-relationship in the instance variable definition; however, it is very simple and easily extended to multiple inheritances. Practical merit is large.

Figure 7 includes mode errors, however. An "explicit mode conversion" is necessary when a "REF ab" value is assigned to a "REF a" variable, but they are omitted here for simplification. From the viewpoint of inheritance, the Chill mode-check rule on pointers is too strict, and explicit mode conversions are necessary to apply the inheritance. It is hoped that the mode check rule will be modified so as to allow the assignment of a sub-class pointer value to a super-class pointer variable.

Ex. **MODE** a=**STRUCT** (f INT), ab=**STRUCT** (f INT, g BIT(32)); **DCL** Pa **REF** a, Pab **REF** ab;
Pa:=Pab:/*It is hoped that this assignment will be allowed.*/
Pab:=Pa; /*This is of course illegal*/

8. Related Languages

From the viewpoint of modeling, our concurrent object model has some similarities to object-oriented research languages: "ABCL/1" [4] is an Actor model based language, and "Concurent Smalltalk" [4] is a Smalltalk-80 upper-compatible language. Here, we abbreviate them as ABCL and CS talk, respectively.

(1) ABCL objects are of the concurrent type. However, ABCL objects, which receive only "NOW" type messages, whose methods (scripts) exit when Reply statements are executed, and whose messages do not arrive concurrently, can be mapped to our sequential objects. This improves the effeciency.

(2) CStalk has Non-Atomic objects and Atomic Objects. The former are Smalltalk compatible objects. They can run concurrently sending asynchronous messages. The latter are exclusively accessed objects; here, arrived messages are serially processed. They correspond to our critical region objects.

(3) The "Past-type" message in ABCL and the asynchronous method call in CStalk correspond to our concurrent messages.

(4) The "Present-type" message in ABCL and the synchronous message in CStalk correspond to our sequential messages.

9. Evaluations

An object-oriented prototype switching program was implemented using this technique [8, 9]. The results are as follows:

(1) This concurrent object model is suitable for switching program structuring. Logical components of a switching system can be appropriately implemented as software objects.

(2) Proper use of concurrent objects and sequential objects lead to simple multi-processing and good real-time efficiency.

(3) The run-time overheads of sequential objects are not so large. Compared with Smalltalk-like languages, the strong typing allows a processing shift from run-time to compile-time, the non-automatic garbage collection eliminates overhead, and the large object granularity improves efficiency. Overheads, comparing with procedure-oriented programming, are caused by (a) message passings by indirect procedure-calls, (b) instance variable access via methods, and (c) increase of procedure calls (i.e., small and numerous methods). Overheads for procedure-oriented programming may be 20%.

(4) The time-and-space overheads of concurrent objects are those of concurrent process systems. The Chill process is very efficient and favorable for our applications. Concurrent message passing requires only about 100 dynamic steps.

(5) The combination of Chill and a very simple pre-processor enables good program readability and understandability. Inheritance is also possible (with some limitations). However, it is hoped that the pointer mode compatible rule of Chill will be extended as explained in Chapter 7 to eliminate explicit mode conversions for secure asssignments.

10. Closing Remarks

Object-oriented programming is a suitable approach for improving switching program maintainability. A

```
       [Pre-processor input]           |     [Pre-processor    output]
                                       |
A:CLASS ;                              |     A:MODULE;
       EXPORT  a,a_ ... ;              |      GRANT  a,a_ ... ;
   INSTANCEVARS                        |
       MODE  a_ =                      |
         STRUCT(a1 INT, a2 INT );      |
       MODE  a =                       |     MODE  a =
          STRUCT(METHODLINK,           |       STRUCT(E REF Methods,
                 A  a_);               |            A  a_);
   CLASSMETHODS                        |
     New:PROC( )RETURNS(REF a);        |
                ....                   |
                END;                   |
   INSTANCEMETHODS                     |
     m1:PROC(SELF,...);                |     m1:PROC(SelfP REF a ...);
                SELF.A.a2 :=  ... ;    |            SelfP->.A.a2 := ... ;
                ...                    |
              END m1;                  |
                                       |     ....Method Table (omitted)....
  ENDCLASS   A ;                       |         END A ;
---------------------------------------|--------------------------------------
                                       |
AB:CLASS ;                             |     AB:MODULE ;
   SUPER  A;                           |            From A SEIZE ALL;
   EXPORT  ab,ab_ ... ;                |            GRANT  ab,ab_ ... ;
   INSTANCEVARS                        |
     MODE  ab_ =                       |
        STRUCT( b1 INT, b2 CHAR);      |
       MODE  abM =                     |     MODE ab =
          STRUCT(METHODLINK,           |     STRUCT(E REF Methods,
                A   A!a_,              |         A   A!a_,
                AB  ab_);              |         AB  ab_);
  CLASSMETHODS                         |
    New:PROC( )RETURNS(REF ab);        |
                ...                    |
                END;                   |
   INSTANCEMETHODS                     |
     m2:PROC(SELF,...  );              |     m2:PROC(SelfP REF ab ... );
                SELF.AB.b1 := ... ;    |            SelfP->.AB.b1 := ... ;
                SELF.A.a1 := ... ;     |            SelfP->.A.a1 := ...;
                ...                    |
              END;                     |
                                       |     ....Method Table (omitted)....
   ENDCLASS  AB ;                      |         END  AB;
---------------------------------------|--------------------------------------
DCL  ObjA  REF A!a;                    |
DCL  ObjA  REF AB!ab;                  |
                                       |
   ObjA := A  << New ( );              |  ObjA := A! New( );
   ObjAB := AB << New ( );             |  ObjAB := AB! New( );
                                       |
   ObjA << m1(i);                      |  CALL ObjA->.E->.m1(ObjA, i);
                                       |
   ObjAB << m2(j);                     |  CALL ObjAB->.E->.m2(ObjAB, j);
                                       |
   ObjAB << m1(i);                     |  CALL ObjAB->.E->.m1(ObjAB, i);
```

Fig. 7 Inheritance technique in Chill.

switching program is a severe real-time and multi-processing system, and therefore appropriate concurrent object models and an efficient language are necessary. This paper discussed a concurrent object model for real-time multi-processing applications, and presented object-oriented programming techniques using an existing concurrent process language. Compared with the development of a new and ideal object-oriented language for real-time applications, this approach may be provisional. However, this approach is advantageous in that the development costs and training time are insignificant, since existing environments can be used and programmer education shortened.

In this paper, Chill is used as the base language, but these techniques may be applied to other (concurrent process) languages, such as Modula.

References

1. GOLDBERG, R. *Smalltalk-80: The language and its implementation*, Addison-Wesley (1983).
2. DAHL, J. et al. *The Simula 67 Common Base Language*.
3. CCITT recommendation: Z.200 CCITT High Level Language Chill, CCITT, 1988.
4. YONEZAWA, A. and TOKORO, M. (eds.) *Object oriented concurrent programming, MIT Press.*
5. HOARE, C. A. R. *Monitors: an Operating System Structuring Concept, CACM*, **18**, 9 (1975).
6. STROUSTRUP, B. *The C+ +Programming Language*, Addison Wesley.
7. WIRTH, N. *Programming in Modula-2*, Springer-Verlag.
8. MARUYAMA, K. et al. *A Concurrent Object-Oriented Switching Program in Chill, IEEE Communication Magazine*, **28** 1 (Jan. 1991).
9. MARUYAMA, K. et al. *A Concurrent Object-Oriented Switching Program in Chill, Proceedings of ISS'90.*