INFORMATION PROCESSING 86, H.-J. Kugler (ed.)
Elsevier Science Publishers B.V. (North-Holland)

A MULTILINGUAL MODULAR PROGRAMMING SYSTEM FOR DESCRIBING KNOWLEDGE INFORMATION PROCESSING SYSTEMS

Takeshi CHUSHO and Hirohide HAGA
Systems Development Laboratory, Hitachi Ltd.
1099, Ohzenji, Asao-ku, Kawasaki 215, Japan

For describing knowledge information processing systems, many new paradigms have been proposed and each of them has inherent features. This paper proposes a multilingual modular programming system supporting these paradigms for general use. An intermodule relation is described in an object-oriented programming language because it is suitable for the natural description of a global computation model. Each module is described in one of the conventional languages which is suitable for an exact description of the module function. A composite language, Super-LONLI is developed for supporting this system to mix an object-oriented language and conventional languages while clarifying the boundaries among them instead of unifying them with semantic ambiguity. In particular, this language overcomes such difficulties in mixing logic and object-oriented programming languages as backtracking vs. side effect, the close world assumption vs. object hierarchy with inheritance rules, and sequentiality vs. concurrency.

1. INTRODUCTION

The essence of modern programming methodologies for large-scale software development is considered as modularization techniques such as data abstraction, top-down development by stepwise refinement and structured programming which were proposed in the 1970's. These modularization techniques divide a program into modules so that each module has only one function. Therefore, a programming language must have the ability to express the various functions which those modules have.

There are two approaches for designing such a new language. One is to design a large language such as Ada* which provides many facilities. Such a language has an intermodule relation description facility in addition to a module description facility. However, programming in this language is not easy since the language specification becomes complicated.

The other approach is to prepare an intermodule relation description language and a set of small languages for the description of modules. That is, a different category of module function should be written in a different language which is sufficiently simple and suitable for its function. This approach was first proposed in 1979 and called MMP(5) (Mutilingual Modular Programming). However, it was uncertain how to describe an interface between modules written in different languages.

Recently, this approach has become more important as research in knowledge engineering is being activated. This is because many programming paradigms such as logic programming, functional programming, rule-based programming and object-oriented programming are considered useful for knowledge engineering but any one of them alone is not sufficient for general use(1).

Some languages mixing several paradigms have been already proposed. For example, Loops(2) and Tao(18) are Lisp-based composite languages. ESP(4) and Mandala(12) are Prolog-based composite languages. The design policies of these languages are similar to the aforementioned large language approach. That is, it is intended that all kinds of programs can be written in one language including many functions. However, it is considered difficult to unify different languages based on different paradigms into one language without conflicts or semantic ambiguity.

For avoiding these difficulties, this paper proposes multilingual modular programming. In particular, an object-oriented programming language is adopted as an intermodule relation description language. On the other hand, each module is described in one of the conventional languages such as Prolog, Lisp, a rule-based language, or a procedural language. This system is called MMP-83 and the old version of this approach is called MMP-79 in this paper.

The design policies of MMP-83 are as follows :
(1) A knowledge information processing system requires facilities for knowledge representation and inference in addition to conventional procedural description. However, since a variety of knowledge representation and inference can not be written in any one of the conventional languages, a composite language including them is necessary.
(2) The composite language should be composed of an intermodule relation description language and a group of module description languages in order to avoid the aforementioned difficulties in unifying different languages.
(3) An object-oriented programming language is adopted as an intermodule relation description language for the following reasons :
(a) A programming unit called an object can be regarded as a capsule including a category of knowledge and its operations. This implies that categorization of knowledge and modularization of functions are performed simultaneously in only one manner.
(b) An object hierarchy and its inheritance rules are useful for the expression of the static relation among objects and then for knowledge base construction.
(c) Message passing between objects gives a simple expression of the dynamic relation among objects.
(d) These features of (a), (b) and (c) are not lost even if each object is written in the conventional languages.

Consequently, MMP-83 provides the following new paradigm :
(1) A global computation model and a global knowledge structure are written naturally in an

* Ada is a registered trademark of the Department of Defence, USA.

object-oriented language as the interobject relation description.
(2) Computation rules and knowledge of facts are written exactly in any suitable conventional language as the object description.

This paper describes the MMP-83 approach in Chapter 2, an object-oriented programming model in Chapter 3, a method to embed Prolog in the model and the newly developed language, Super-LONLI(S-LONLI)(14), in Chapter 4, and discussions for completion of MMP-83 in Chapter 5.

2. INTERMODULE RELATION DESCRIPTION LANGUAGE

2.1 Previous works

The most important technique for large-scale software system development is modularization. Since it is best that an intermodule relation is as simple and easy-to-understand as possible, the relation should be expressed explicitly in a source program. There are two approaches in previous works.

One is to include an intermodule relation description facility in a module description language such as Ada. The structured programming language, SPL(7), which the authors developed for supporting top-down development by stepwise refinement and data abstraction in 1976, took this approach also. However, this approach makes a programming language specification complex and limits an intermodule relation description to a module interface.

The other approach is to provide a proper language for intermodule relation description such as MIL(11) (Module Interconnection Language). However, since these languages are indepedent of a programming language for a module description, their utilization is limited to documentation and consistency check of module interfaces described in separation from a source program.

Therefore, the authors proposed the third approach called MMP-79 in 1979 as follows:
(1) A proper intermodule description language is provided as a programming language.
(2) Each module is described in any conventional language matched with the module function.

For practical use, however, this approach had to overcome such difficulties in different programming languges as different data types, different parameter mechanisms, different control mechanisms and different execution environments.

2.2 A object-oriented multilingual system

Recently, considerable effort has begun to be spent on research in knowledge engineering which intends to increase the ability of computer systems by embedding human knowledge. This knowlege engineering is expected to break through such a problem in software engineering as a productivity gap between expert and novice programmers also.

First of all, a language for describing knowledge information processing systems has been required. Then many candidates were proposed. Some of them are rule-based languages, frame-based languages, logic programming languages, functional programming languages, and object-oriented programming languages. However, none of them is sufficient to

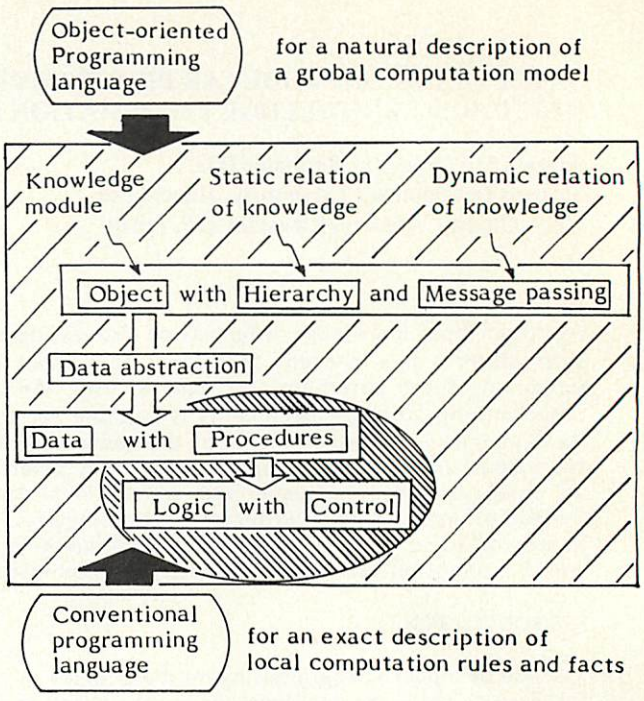


Figure 1. Design concept of MMP-83

describe a large-scale knowledge information processing system.

One solution is MMP-83. The main idea of MMP-83 in comparison with MMP-79 is that an object-oriented programming language, referred to as OP, is adopted for intermodule relation description as shown in fig. 1. The design policies of MMP-83 has already been described in Chapter 1. In addition, OP overcomes difficulties in linkage among different languages, with which MMP-79 was faced, as follows :

(1) Message passing is adopted as a common control mechanism among objects. Various control mechanisms can be introduced by providing various reply methods of message passing. For example, one of them is whether a reply to a message sender returns via the channel in which the message was sent or whether the reply returns by sending another message from the receiver. The second is whether an address for the reply is limited to the message sender or whether the reply can be accepted by any other objects or by objects specified by the sender. The third is whether the message sender waits for the reply soon after sending the message or at an arbitrary point.

(2) A call-by-value method is adopted as a basic parameter mechanism corresponding with message passing. A call-by-value-and/or-result method is necessary also if the message is accompanied by the reply.

(3) A demon for data type conversion is provided for linkage between different data types. This demon is invoked in the message receiver when it receives the message with parameter values and when it returns the result. The demon is user-definable and is attached to an object.

3. OBJECT-ORIENTED PROGRAMMING LANGUAGE

An OP model is constructed as follows :
(1) The functional specification of an object is expressed in a set of methods. A method is invoked by

receiving a message. It may have input and/or output parameters.
(2) An object may include data declared as variable.
(3) From the viewpoint of data abstraction, only methods in objects including data can access the data.
(4) Many instances are dynamically created from the prototype to efficently produce many objects which include the same methods but different data values. The prototype is called a class object.
(5) Data belonging to a class or an instance is called a class variable or an instance variable, respectively.
(6) A method of accessing a class variable or an instance variable is called a class method or an instance method, respectively.
(7) Class objects are composed in hierachy to efficiently produce many objects which have sets of methods slightly different from each other. A descendant class inherits the methods of the ascendant classes.

A conceptual scheme of this model and examples of messages sent from instance I2 are shown in fig.2. An actual description of sending a message is included in a method in class C2. The use of the OP model is assumed for the remainder of this paper.

4. EMBEDDING LP in OP

4.1 A module description in LP

OP is adopted for an intermodule relation description in MMP-83 since OP is suitable for a natural description of a global computation model of the real world. However, it is still a problem to describe manipulation procedures for data. In Smalltalk-80(13), data manipulation procedures are described in the same manner as global description, that is, in the object-oriented style. This pure approach may not be suitable for a description of inference. On the other hand, in such a language as Ada which supports data abstraction, data manipulation is described in the conventional procedural manner. Programming in this style is inefficient because of the semantic gap between programs and the real world.

Consequently, a logic programming language(17), referred to as LP, is considered the first module description language in MMP-83 shown in fig.1, because LP reduces the above-mentioned disadvantages. Furthermore, LP provides inference ability.
However, it is not easy to mix several languages based on different paradigms or programming methodologies. The following is a brief survey of languages supporting OP and LP. First, there are ESP, Intermission(16), PARLOG(10) and Mandala based on Prolog. In these languages, OP is embedded in LP(LP-with-OP type). Next, TAO is based on Lisp and equally supports LP and OP (LP-and-OP type). These methods of inserting OP into such conventional languages as Prolog or Lisp, may obscure the OP framework. It is considered difficult to mix several languages based on different paradigms in designing a unique language.

The approach described here is to insert LP in OP (OP-with-LP type) and to clarify the boundary between the languages instead of unifying them. This paper discusses this OP-with-LP approach while assuming that LP is Prolog(3).

4.2 Problems with mixture of OP and LP

The above-mentioned OP features cause the following problems for LP.

First, an object has variables which may be "own variables". This implies that the state of the object varies with time. Because LP does not allow such "own variables", it is uncertain how to deal with the side effect on the variables.

Second, functions of each object are not self-

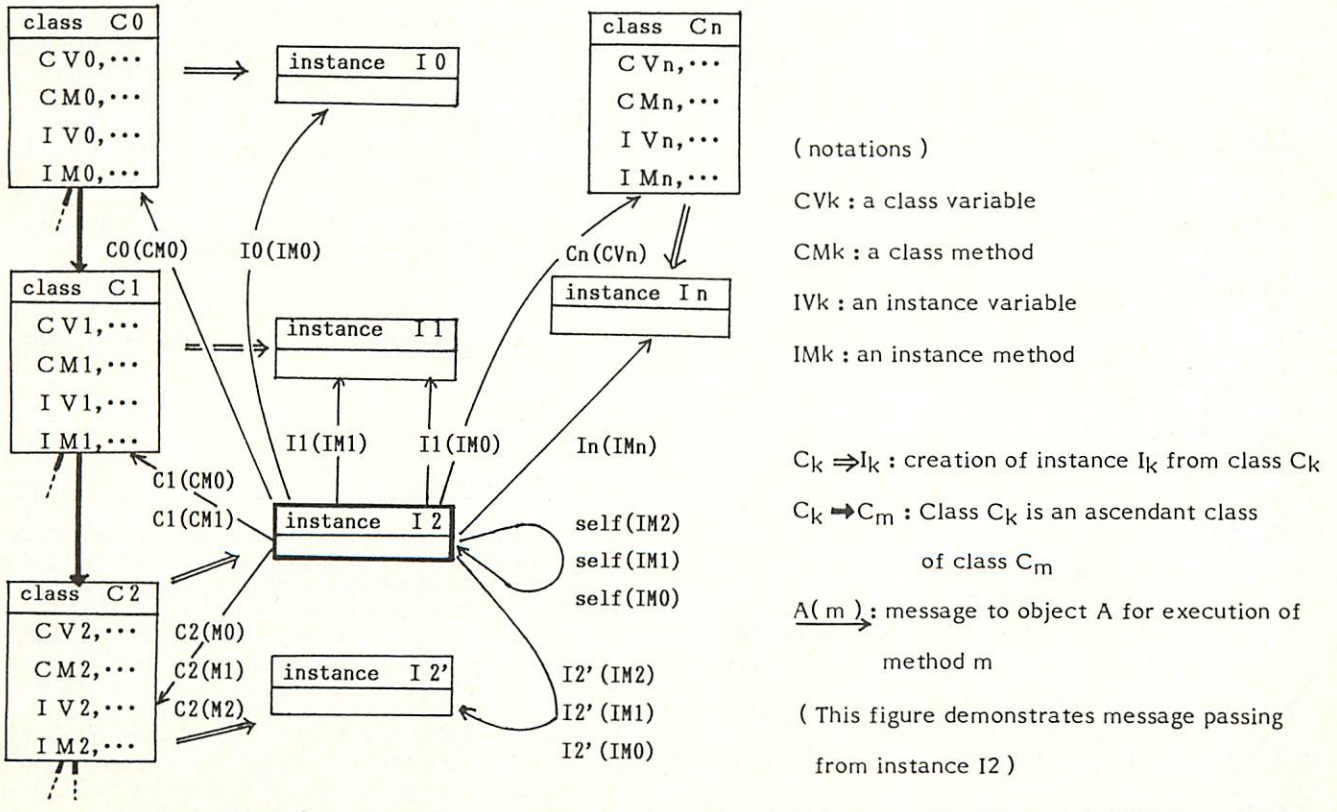


Figure 2. An example of a model of an object-oriented Programming language .

contained because of inheritance rules in the object hierarchy. It is uncertain how to understand these inheritance rules in the closed world assumption(9) that the knowledge base knows everything there is to know.

Third, instance objects are dynamically created from a class object. There are no such facilities in LP.

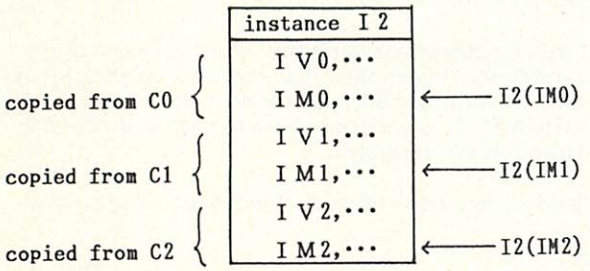
Last, OP assumes that objects are concurrently executed in principle(15). It is uncertain how to deal with message passing in sequential LP.

On the other hand, LP also causes some problems for OP. The main features of LP such as Prolog are unification and backtracking. Unification can be regarded as a manner for matching a received message to a target method in the receiver object. However, backtracking is not acceptable for OP. This is the first problem for OP. The second is the same as the aforementioned problem, that is, sequentiality vs. concurrency.

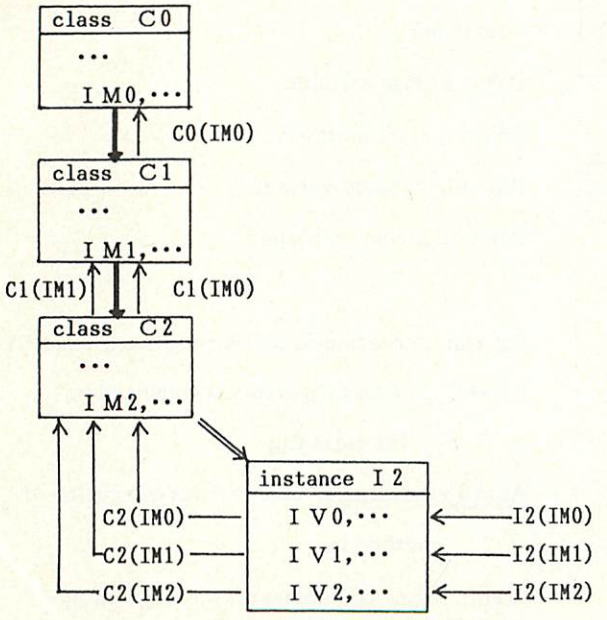
4.3 OP predicates

S-LONLI is a new language for describing knowledge information processing systems. This language is designed based on the OP model. Methods are described in LP. The following built-in predicates are provided for OP:

(1) gen(a) : A class receiving this message creates an instance of itself and names it 'a'.



(1) An example of application of the copy rule



(2) An example of application of the pass rule

Figure 3. Semantics of inheritance rules in object hierarchy

(2) change(x , v): The value v is assigned to the variable x.
(3) send(a , m): The message m is sent to the object a .

In the remainder of this paper, explanations of any other OP predicates are omitted because they do not relate to the aforementioned problems with the mixture of OP and LP.

4.4 Semantics of inheritance rules in hierarchy of classes

The semantics of inheritance rules in a hierarchy of classes should be defined from the viewpoint of the closed world assumption. In general, the following two definitions are considered:

(1) Copy rule
Instance variables and instance methods in a class and its ascendant classes are copied in the instance created from the class. An example of I2 shown in fig.2 is demonstrated in fig.3(1). Class variables and class methods in the ascendant classes of a class are copied in the class itself.

(2) Pass rule
If no definition of the method corresponding to a message can be found in the class receiving the message or in the class of the instance receiving the message, the class passes the message to the ascendant class. The process of the message received by the instance I2 shown in fig.2 is demonstrated in fig.3(2). A message for the IM0 method is passed from I2 to C0 via C2 and C1. However, instance variables are copied into an instance by this rule just as in the copy rule.

These two rules have the same semantics in the OP model. However, the copy rule is preferable for this new language because it corresponds with the closed world assumption. The following grammatical rules are introduced in accordance with the copy rule:
(1) When a method invokes other methods which are copied into the former method in accordance with the copy rule, the former can do so directly in the same syntactic form as in invoking an ordinary predicate.
(2) In the other case, a method must invoke other methods by sending messages to the objects including the latter methods.

For example, when the IM2 method of C2 invokes the IM0 method in fig.2 , it can be expressed as

im2:- ・・・ , im0, ・・・

instead of

im2 :- ・・・ , send(self, im0), ・・・ .

However, when invoking IMn, it can not be expressed as

im2 :- ・・・ , imn, ・・・

instead of

im2 :- ・・・ , send(in, imn), ・・・ .

4.5 Side effect and backtracking

Execution of built-in predicates for OP may cause side effects, which should be undone when backtracking is triggered. That is, backtracking must kill an instance created by 'gen', replace the value of a variable assigned by 'change' with the previous value, and undo the results of message execution caused by 'send'. Furthermore, message passing to a created instance and access to a changed value of a variable should also be undone. This is called "distributed backtracking" because this OP model assumes concurrency of object execution. Since such complicated behavior must manage a large amount of execution records

increasing with time, backtracking must be restricted to some extent for the practical use of a programming language.

4.6 Success and failure of OP predicates

OP predicates will fail in the following cases when used with LP:
(1) The predicate has grammatical errors.
(2) A variable specified by 'change' is not found or does not correspond with the data type of the value.
(3) An instance specified by 'gen' has already been created.
(4) An object or a method specified by 'send' is not found.
(5) A method specified by 'send' is executed but then fails.

The following alternatives for these cases are considered:
(A) They are regarded as failures.
(B) They are regarded as successes.
Rule A requires sequential execution for practical use. The sequentiality causes no problems on items (1) - (4) since these items can be instantly checked. In the case of item (5), however, since a sender of a message must wait until the message is executed, OP concurrency is obstructed. On the other hand, rule B is not as advantageous in its manner of describing methods in LP since confusing side effects occur.

Consequently, the following policy is taken:
(a) Rule A is applied to items (1) - (4) since it does not conflict with OP.
(b) Rule B is applied to item (5) since concurrency of OP seems to be very important.

However, it may sometimes be necessary for a message sender to wait for the result of success or failure. Therefore, in addition to 'send', the new predicate, 'sendw' (send and wait), is provided as follows:
(1) send : After execution of this predicate, a program continues execution without waiting for the result.
(2) sendw : After that, the program halts until results return.

For example:
```
appoint(Date):-
sendw(secretary,schedule(Date, Item)),
( Item == vacant, answer(ok) ;answer(no)).
```
This program waits for the result after sending 'secretary' a message of inquiring about the schedule for the date specified by 'Date'. Then this program replies 'ok' if 'Item' is vacant and otherwise replies 'no'.

4.7 Restriction on backtracking

Strictly speaking, when backtracking follows side effects accompanying successful execution of 'gen', 'change', and/or 'send', these side effects should be undone. For example, this problem is caused by failure of 'r' in the following clause:
```
p(X) :- send(c1, gen(X)), change(last, X),
        send(monitor, register(X)), r.
```
However, such backtracking cannot help neglecting side effects since undoing side effects confuses the OP world model because of the occurrence of distributed backtracking and makes implementation difficult.

In compensation for users assuming full responsibility for side effects, it is desirable for users to be able to control such side effects. The primary side effect of 'gen' can be canceled by 'kill' extinguishing an instance object. That of 'change' can be canceled by 'change' itself assigning the previous value. But that of 'send' cannot be undone by any predicate.

Consequently, the following predicate, 'sendq' (send via queue), is provided for lazy evaluation of 'send':
sendq : when this predicate occurs during execution of a method, the predicate is stored in a queue without being executed. If backtracking occurs, the related 'sendq' is omitted from the queue. When the method has been successfully executed, all predicates in the queue are executed. That is, all messages by 'sendq' are sent to their receivers.

The message to 'monitor' in the aforementioned example should be modified as follows:
```
sendq(monitor, register(X))
```
This predicate enables LP to be embedded into OP without conflict between message passing and backtracking while clarifying the boundary between OP and LP.

Generally speaking, this facility implies that the following control primitives are introduced in Prolog:
queue : When atomic formula x is written in the form of queue(x), x is stored in a queue at its occurence and is omitted from the queue by backtracking. When the next-mentioned 'dequeue' occurs and/or when the goal is attained, x in the queue is executed in the first-in-first-out order.
dequeue :All atomic formulas in a queue are executed.

The aforementioned example is described by using these control primitives as follows:
```
p(X) :- send(c1, gen(X)), change(last, X),
    queue(send(monitor, registor(X))), r,dequeue.
```
Furthermore, side effects of input or output predicates in Prolog become controllable by applying these primitives to them as follows:
```
queue(write( ··· ))
```

4.8 Program examples in S-LONLI

An example program of S-LONLI is shown in fig.4. This defines the class "washing_machine". The washing_machine has two timers, one for washing time control and the other for drying time control. Users of this machine can select washing time from long or short. This program includes two class definitions. Class "washing_machine" has class "electric_equipment" as its superclass. This represents an "is_a" relation between these two classes. The instance of class "timer" is declared as an instance variable of class "washing_machine" by the "instance_of" attribute for the variable. This represents a "part_of" relation between "washing_machine" and "timer".

5. DISCUSSIONS AND CONCLUSIONS

5.1 Multilingual System

S-LONLI was designed to embed LP in OP as the first language based on MMP-83 philosophy. As described in Chapter 4, it is important to retain features of both languages and to clarify the boundary between both for avoiding semantic ambiguity when a composite language is designed by mixing different languages.

A rule-based programming language is selected as the second language for module description in MMP-83 since it is indispensable to compact knowledge

```
define_frame washing_machine;
  attribute class;
  supers electric_equipment;
  i_vars water_state(initial(0)),
         timer1(instance_of(timer)),
         timer2(instance_of(timer));
  i_method set_washing_time;
    set_washing_time(long) :-
      send(timer1,set_time(15,00)),
      send(timer2,set_time( 2,30));
    set_washing_time(short) :-
      send(timer1,set_time(10,00)),
      send(timer2,set_time( 1,30));
  end;
  i_method washing;
    washing(long) :-fill_water(high),washing_start;
    washing(small) :-fill_water(low),washing_start;
    washing_start :-
      motor_on,sendw(timer1,start),
      motor_off,drain,
      motor_on,sendw(timer2,start),
      motor_off;
  end;
  i_method fill_water;
    fill_water(high) :-change(water_state,20);
    fill_water(low) :-change(water_state,10);
  end;
  i_method drain ;
    drain :- change(wvater_state,0);
  end;
end.

define_frame timer;
  attribute class;
  supers root;
  i_vars second(initial(0)),
         minute(initial(0));
  i_method set_time;
    set_time(M,S):-change(minute,M),change(second,S);
  end;
  i_method start;
    start :- minute=0,second=0;
    start :- second=0,New_minute is minute-1,
             change(minute,New_minute),
             change(second,59),start;
    start :- New_second is second-1,
             change(second,New_second),start;
  end;
end.
```

Fig.4. An example program in S-LONLI

processing. Embedding this language in OP is easier than embedding LP in OP since an object hierarchy and inheritance rule are considered as a modularization mechanism. Any other language will also be embedded in S-LONLI in accordance with the same approach.

5.2 Implementation

The current version of S-LONLI was implemented in the logic programming language, LONLI, and is executed on HITAC M Series computers. S-LONLI programs are transformed to LONLI programs by a preprocessor and are executed with a message interpreter on the LONLI processor.

This process is general in MMP-83. Each object is transformed to a program written in the module description language in which the methods of the object are written. This transformed program is processed by each language processor and is executed under the control of the message interpreter.

5.3 Programming environment

Usability of a language system is dependent on its programming environment. In the first step, the same language-adaptive programming environment(8) as the authors developed for conventional procedural languages, is also developed. That is, a structure editor, a debugger coupled to the editor and a source-to-source transformer(6) for optimization. In the next step, an expert system for S-LONLI programming will be necessary for knowledge engineers, which system supports knowledge acquisition, error diagnosis, etc.

ACKNOWLEDGEMENTS

The authors wish to express their gratitude to Dr. Jun Kawasaki and Yoshihiko Aoyama for their valuable suggetions and advice. They are indebted to Yoshiko Oto, who implemented S-LONLI with the authors, for her invaluable discussions. They also wish to thank Prof. Robert Kowalski and Dr. Steve Gregory for helpful comments.

REFERENCES

(1) D. G. Bobrow, If Prolog is the answer, what is the question?, Proc. Int. Conf. FGCS'84, Nov. 1984,138-145.
(2) D. G. Bobrow and M. Stefik, The LOOPS manual, Xerox Co., Jan. 1983.
(3) D. Bowen, DEC system-10 Prolog user's manual : Dept. of Artificial Intelligence, Univ. of Edinburgh, Dec. 1981.
(4) T. Chikayama, ESP reference manual : Technical Report TR-044, ICOT, Feb. 1984.
(5) T. Chusho, Conceptual design of multilingual modular programming and its processer, the 21st Annual Conf. of IPSJ, 2c-6, May 1980, 287-288 (Japanese).
(6) T. Chusho, A good program = a structured programming + optimization commands, Proc. IFIP'80,269-274.
(7) T,Chusho et al., A language with modified block structure for data abstraction and stepwise refinement, Proc. RIMS Simposium on the third Mathematical Methods in Software Science and Engineering, June, 1981, 156-173.
(8) T. Chusho. et al., A language-adaptive programming environment based on a program analyzer and a structure editor, Proc. IFIP'83, Sep. 1983, 621-626.
(9) K. Clark, Negation as failure, Logic and Databases (Ed. H. Gallaire, and J. Minker), Plenum Press, 1978, 293-322.
(10) K. Clark and S. Gregory, PARLOG : parallel programming in logic, Research Report DOC 84/4, Imperial College, Apr. 1984.
(11) F. DeRemer and H. Kron : Programming-in-the-large vesus programming-in-the-small : IEEE Trans. SE, vol.SE-2, no.4, June 1976, 80-86.
(12) K. Furukawa, et al., Mandala - A concurrent prolog based knowledge programming Language/System : IPSJ SIG on KE and AI 32, 32-1, Nov. 1983.
(13) A. Goldberg, and D. Robson, Smalltalk-80 the language and its implementation, Addison-Wesley, 1983.
(14) H. Haga and T. Chusho, S-LONLI - the language for description of knowledge information processing systems : Proc. the First Annual Conf. of Japan Society for Software and Technology, 2D-2, Dec. 1984, 167-170 (Japanese).
(15) C. Hewitt and H. Baker, Law for communicating parallel processes : Proc. IFIP'77, 1977, 987-992.
(16) K. Kahn, Intermission-actors in Prolog : Logic programming (Ed. K. Clark S. and Tarnlund), Academic Press, 1982, 213-228.
(17) R. Kowalski, Logic Programming : Proc. IFIP'83, Sep. 1983, 133-145.
(18) I. Takeuchi, et. al., TAO - a harmonic mean of Lisp, Prolog and Smalltalk, ACM SIGPLAN Notices, vol.18, no.7, July 1983, 65-74.

REPRINTED FROM:

INFORMATION PROCESSING 86

Proceedings of the IFIP 10th World Computer Congress
Dublin, Ireland
September 1-5, 1986

Edited by

H.-J. KUGLER
Department of Computer Science
Trinity College
Dublin, Ireland

PROGRAMME COMMITTEE

D. Bjørner *(Chairman)*
D. C. Tzichritzis *(Past Chairman)*, H.-J. Kugler *(Editor)*,
J. G. Byrne *(Organising Committee Liaison)*, V. Kotov, U. Montanari, B. Dömölki,
H. Hünke, R. Mori, H. Gallaire, A. Furtado, A. Danthine, J. Vlietstra,
A. I. Wasserman, R. Narasimhan

PARTICIPANTS EDITION

1986

NORTH-HOLLAND
AMSTERDAM • NEW YORK • OXFORD • TOKYO