A Parallel Object Oriented Language FLENG++ and Its Control System on the Parallel Machine PIE64

Hidehiko Tanaka
Department of Electrical Engineering
University of Tokyo, Hongo 7-3-1, Bunkyo-ku, Tokyo, 113 JAPAN

Abstract

This paper describes the language set of PIE which is composed of three kinds of languages, Fleng++, Fleng, and Fleng--. Fleng++ is a user-level parallel language to which module facility is added through the object-oriented style. Fleng-- is a low level language used for the explicit control of execution. Fleng is the core language for this language set, on which Fleng++ and Fleng-- are based. Control features for parallel machine such as process allocation, load distribution and memory management are also discussed in relation to this language set.

1 Introduction

To realize the high level knowledge information processing, we need some nice knowledge representation languages, special processor system of super speed and memory system of large scale. We have been making our research to develop a prototype system for the large scale knowledge processing since 1987. As for the processor system of the prototype, we have been developing a parallel computer system called PIE64(Parallel Inference Engine) composed of 64 Inference Units[1].

As for the memory system, a relational database machine called SDC[2] of 8 units. We designed a committed choice language Fleng[3] as the kernel language for the prototype system. Fleng is a simple language derived from GHC[4] and easy to implement as all goals interacts only through shared variables, and implicit "AND" relation between goals is not assumed(when needed, it should be described explicitly by the form of AND predicate.

This paper describes the language set of PIE which is composed of three kinds of languages, Fleng++, Fleng, and Fleng--. Fleng++ is a user-level parallel language to which module facility is added through the object-oriented style. We describe the language features of Fleng++, and discuss its translation method into Fleng. Fleng-- is a low level language used for the target language of explicit control of execution. While the compiled code of WAM level is executed in the form of communicating parallel processes, control features for parallel machine such as process allocation, load distribution and memory management are also discussed.

2 Language set of PIE

2.1 Committed choice language, Fleng

As a parallel logic programming language, Ueda developed GHC and ICOT developed a language system KL1 based on it. However, it seems that the implementation of GHC is a little bit complicated and that the language level is too high for us to use it for the implementation of control system of parallel machine PIE. Accordingly, we developed a simplified language Fleng as the kernel language for our system.

Fleng is a kind of committed choice language, but has no guard goals. A Fleng program is a set of clauses with a head part and a body part of goals.

```
H :- G1, G2, ..., Gn.
```

A program is executed by giving it a set Q of goals. All goals are executed independently, in any order. A goal G is executed by removing it from Q, and matching it with the head of all clauses in the program. For the first clause with matching head, that clause is committed to, and the body goals of that clause are added to Q. Matching is like unification, but variable bindings is not allowed in G during the matching. If a match of a variable in G with a non-variable or an unbound variable from G is attempted, this matching is suspended, and resumes only if the variable becomes bound.

As opposed to GHC, the execution of a goal is not associated with success, failure, or any logical truth values. New goals are essentially spawned out and executed. If we do want logical truth values, they can be passed as parameters of the goals. For instance, if we have two goals `g(X,R1)` and `h(X,R2)`, where `R1` and `R2` are bound by predicates `g` and `h` to the appropriate truth values, we can define a new predicate `p(X,R3)` to be the logical conjunction of `g` and `h` by:

```
p(X,R3) :- g(X,R1), h(X,R2), and(R1,R2,R3).
```

where

```
and(true,true,R) :- R=true.
and(false,_,R)   :- R=false.
and(_,false,R)   :- R=false.
```

Though the simple structure of Fleng is suitable for the role of core language, it increases the load of programmer. So, we introduced macro and library for Fleng. For example, guard macro is used to represent conditional branch, and "`is`" function is realized by a library. Figure 1 shows an example of macro and library.

2.2 Language set

Based on the language Fleng, we developed a language set for our system. As the programming language of application level, we developed an object oriented language Fleng++. As for the implementation language of language processor and control systems, we developed a language Fleng-- which has the capability of expressing the details of control.

```
fact(N,R) :-
     N == 0  | R = 1;
     N  >  0 | N1 is N-1,
               fact(N1,R1),
               R is R1 * N.
```

Figure 1: Example program of macro and library

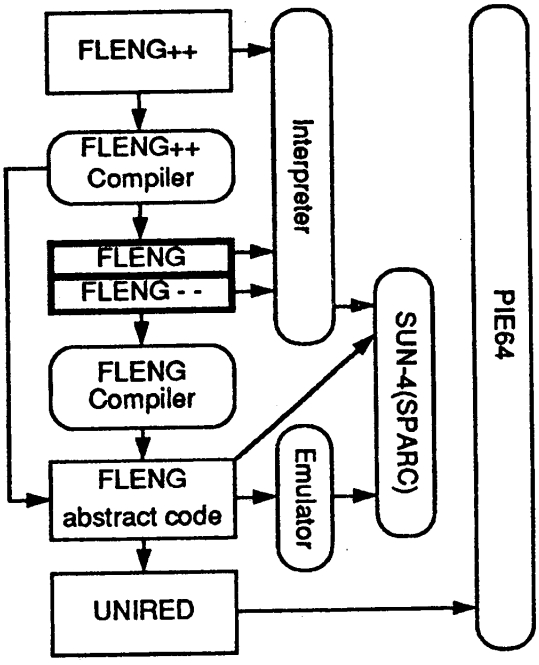


Figure 2: Language System of PIE64

The language system of PIE machine is shown in Figure 2. Application programs are written in Fleng++, translated into the programs written in Fleng--, which are compiled into programs of intermediate code. The intermediate code is of WAM(Warren Abstract Machine) level(but, not WAM itself), tuned to parallel processing and interpreted by PIE machine.

The control policy of parallel execution is not described explicitly at the level of Fleng++. Accordingly, the details of control which is expressed clearly at the level of Fleng-- should be decided by the compiler and/or programmer.

3 Fleng++

3.1 Special features of Fleng++

As a user level language, Fleng++ has special features as follows:

1. Method call of Fleng++ and predicate call of Fleng can be mixed in a program.
2. Control structure of conditional branch is the same as Fleng macro.
3. No explicit description of type and mode for variables.
4. Two kinds of variables : temporary variables and instance variables.
5. Multiple inheritance
6. Class library

3.2 Class definition and object

Program of Fleng++ is composed of class definitions. For example,

```
class ex.
  :p(Object,A) :- :q(A,1).
end.
```

ex is the name of the class. The second line is an expression of method definition, of which syntax is as follows.

```
:<method>(<object>,<arg-1>,...,<arg-n>)
        [ :- <body> ].
```

where <object> is the name of called object. :q(A,1) is a method call that message :q is sent to an object A. For example, a factorial program is described as follows.

```
class fact1.
  :do(Self,N,R) :-
        N == 0  | n = 1 ;
        N  >  0 | N1 is N-1,
                  :do(Self,N1,R1),
                  R is R1 * N.
end.
```

Execution of program is performed by sending a message to an object. An object is generated by a statement such as

```
T := #fact1.
```

An instance of class `fact1` is generated and assigned to T. In case of ESP, the constructor # is used to generate a class object. But, in Fleng++, # is for an instance object. To get the value of factorial 5, a statement

```
T := #fact1, :do(T,5,R).
```

or

```
:do(#fact1,5,R).
```

is used, where the result is gotten from R.

3.3 Temporary variables and instance variables

Variables of Fleng are single-assignment variables. Similarly, variables of `fact1` are generated whenever it receives a message, and discarded when the execution is finished. This kind of variable is called temporary variable. We introduced the other kind of variable called instance variable which exists for the same period as the object and can be rewritten any number of times.

The instance variable is used for designating the characteristics and the internal status of the object. For example, another factorial program can be written by using an instance variable as follows.

```
class fact2
  var $n := 1.
  :do(Self,N,R) :-
        N == 0  | R := $n;
        N  >  0 | $n is $n * N,
                  N1 is N - 1,
                  :do(Self,N1,R).
end.
```

Instance variables are declared by var and shown by a prefix $.

3.4 Serialization of messages

Each goal of Fleng is executed independently. The execution order is determined only by suspension/resume mechanism through shared variables and by goal-generation process of reduction. However, as objects in Fleng++ have side effects, the arrival order of messages should be counted.

We set a principle of sequential interpretation and parallel execution. Though program can be read deterministically from right to left and in the order of depth-first, it is executed in parallel except the case where serialization is required. For example, when a method

```
:p(S) :- ..,$n is $n+1,..,$n := 0,..
```

```
class one.
  :test(Self,R) :- R = 1.
  :result1(Self,R) :- :test(Self,R).
end.

class two inherit one.
  :test(Self,R) :- R = 2.
end.
```

Figure 3: Example program of inheritance

is activated on the condition that the value of $n is 2, the value 2 is used for the value of $n in the goals at the left-side of "$n is $n+1", value 3 is between "$n is $n+1" and "$n := 0", and value 0 is at the right-side of "$n := 0".

Execution order of goals is controlled for the two kinds of method call as follows. When a method

```
:alpha(Self) :- ..,:beta(Self),.. .
```

is executed, the processing of :beta takes precedence over the reception of another message from the other object. The other is the case when multiple messages are sent in one method to an object. The messages are sent in the order of right to left and depth-first. The receive is in the same order.

3.5 Execution control facility

The syntax of conditional branch is the same as Fleng.

```
:m1(S,A,B) :-
        A > B - 1 | :m2(S);
        B > A - 1 | :m3(S).
:n1(S,A,B) :-
        A > B - 1 -> :n2(S);
        B > A - 1 -> :n3(S).
```

Conditional parts in guarded commands(|) have no priority for execution. The part whose condition is met at first is selected and executed. On the other hand, in the case of if-then-else(->), the first conditional part(A > B - 1) is tested first and the second one is tested only when the result of the first part is false.

3.6 Inheritance

Figure 3 is an example of inheritance facility of Fleng++.

Class two in Figure 3 inherits class one. All of the classes in Fleng++ inherit class "object" by default. The class "object" has methods such as :super(Object,Super) which returns a super object of Object.

Fleng++ adopts multiple inheritance. The sequence of method-traverse is determined by "*left to right up to join*". We chose this sequence because this is natural from the view point of description.

However, for the compatibility to ESP[5], we can change the traverse order to "*depth first left to right*" by indicating it at the compile time. Instance variables can be inherited as well.

3.7 Library

We have a library to which user defined classes can be registered, from which user can pick up some objects and use them.

Other than the user defined class, we registered some system defined classes such as I/O management class and object management class.

Class which manages multiple objects is called "pool". All operations to pool are done through messages. The operations are store, take out and delete of object, and inquiry about storage status. We provided the following classes as pool.

1. list: Elements in the list have sequences. Operations are based on first-in last-out.
2. index: Same as list except that key is attached to each object and used for object-access.
3. array: The element number is fixed. Access time to each object is constant.
4. fifo: Operations are based on first-in first-out. fifo can be used to synchronize the operations of two objects by sharing a fifo pool.

4 Implementation of Fleng++

Fleng++ programs are translated into Fleng programs which in turn are compiled into an intermediate code and executed. Objects described in parallel logic programming language are implemented by recursive goal call. Internal status such as instance variables is represented explicitly by goal arguments. Followings are the major consideration points of the implementation.

4.1 Message sending

An expression of message sending :msg(A)(only an argument of destination, for the simplification of explanation) is expanded into a Fleng program,

```
A=[msg(A1)|A2], merge(A1,A3,A2)
```

where A is a stream which designates the destination object, A1 which is one of the end point of the stream is passed to the destination object, A3 is a new stream at sending side, and A=[] means a unification.

When the destination of the message is the object itself, the merge predicate above is exchanged to append predicate as the message handling sequence must be controlled.

When some object is passed to a destinated object as an argument of message, number of references to the object increases as both sending and receiving objects refer it. Procedure for this increase differs depending on the type of the object, primitive object and instance object. Primitive object is such object as atom and complex term that can be represented by literals. Instance object is such object that is generated from class definition. In case of primitive object, unification is the everything for the increase. For the instance object, "merge" predicate is used to make another branch on the stream which points the object.

So, Fleng++ processor is required to check the type of the value. However, as variable of Fleng++ has no type, we used $s-list structure to represent a stream to send messages of which arguments are instance objects. That is, message sending expression S=[msg(A,B)|S1] is divided into two unifications,

```
S=[MSG|S1], MSG = msg(A,B)
```

where first one is to form a stream by list cell, and second one is to unify the message itself.

As the first one can be done at the instance object side, the list cell can be used to distinguish instance objects from primitive objects. That is, to send some instance object we use the special $s-list structure such as '$s'(msg1,'$s'(msg2,..)) that uses complex term headed with '$s' of which use is limited for system use only. Using this structure, we defined a predicate refer/3 to increase the reference number as follows.

```
refer(A,B,C) :-
        C == '$s'(_,_) -> merge(A,B,C);
                          A = C, B = C.
```

where C is the referred object of which type is checked.

On the other hand, as arguments of message are passed by unification in Fleng, input/output mode of variable should be checked before using the refer/3. We provided a mode-check predicate.

The type check and mode check stated above are done at runtime only for such a case that the type or mode can not be determined at compile time.

4.2 Type/mode analysis

As the type/mode check at run time is not efficient, type/mode analysis is incorporated in compiler. This analysis is also effective for reading and debugging of programs as well as for the compiler optimization. We developed an analyzer for Fleng, on which analyzer for Fleng++ is built. When we limit our attention to the compiler optimization,

it is enough for us to distinguish such types as primitive objects, instance objects and any type(unknown or undefined), and such modes as output(marked by -), input(+) and any(?, unknown or undefined).

The Fleng++ compiler is made of two pathes. The first path analyzes the type/mode of arguments of method-head through checking the class definition. For example, the type of the first argument of message-sending expression becomes instance object, and the mode is input-mode. The first argument of method head is of instance type and input mode. The non-temporary variables in the head is of primitive type and input mode ... and so on. For example, when we have a method definition,

```
:p(X,Y) :- :q(Y).
```

the result of analysis is

```
:p(+ins(X), +ins(Y)).
```

The second path compiles the method definitions based on the result of first path and the type/mode table of methods retrieved from library which is already analyzed.

4.3 Access of instance variables

As described earlier, the method is interpreted from left to right and depth-first for the definition. Instance variables are implemented by introducing two special methods and inserting them in appropriate locations of method definition following the scope of instance variable defined by the interpretation sequence. The methods, which are used to refer and rewrite instance variables are

```
:'get$n'(S,Obj), and :'put$n'(S,Obj).
```

For example, consider a method such as

```
:alpha(S) :-
     ...,$n is $n + 1,...,:beta(T,$n),...
```

This is rewritten by inserting the two methods as follows.

```
:alpha(S) :-
     ...,'get$n'(S,N1),:'put$n'(S,N2),
        N2 is N1 + 1,
     ...,'get$n'(S,N3),:beta(S,N3),...
```

4.4 Conditional branch

Conditional branch is macro expanded into two parts. One is to evaluate the condition. The other is to call the subgoal using the result. As the stream is changed into another variable when a message to an object pointed by the stream is sent, transformation is required in such case that there are some message-sending expression inside the conditional part.

For example, when we have a conditional branch

```
..,(A > 0 -> :write(Obj,ok)),...
```

this is transformed into

```
...,greater(A,0,Res),
    sub_0(Res,Obj0,Obj1),...

sub_0(true,Obj0,Obj1) :-
    Obj0 = '$s'(write(Obj1,ok),Obj1).
sub_0(false,Obj0,Obj1) :-
    Obj1 = Obj0.
```

The nesting level of conditional part is managed by compiler.

4.5 Optimization by partial evaluation

Fleng programs which are converted from Fleng++ programs in the unit of every method include redundancy a lot because the interrelationship among methods is not taken into account. This can be reduced by evaluating such predicates as built-in predicates, `merge/3` and `append/3` whose definitions are known to compiler.

For example, message-sending, which is the operation sequence of using `$s`-list and instanciation of message, can be optimized by developing almost all of the related unifications into a few one. This method omits most of `merge/3` or `append/3`.

In some cases, static binding of method can be used instead of dynamic binding through partial evaluation.

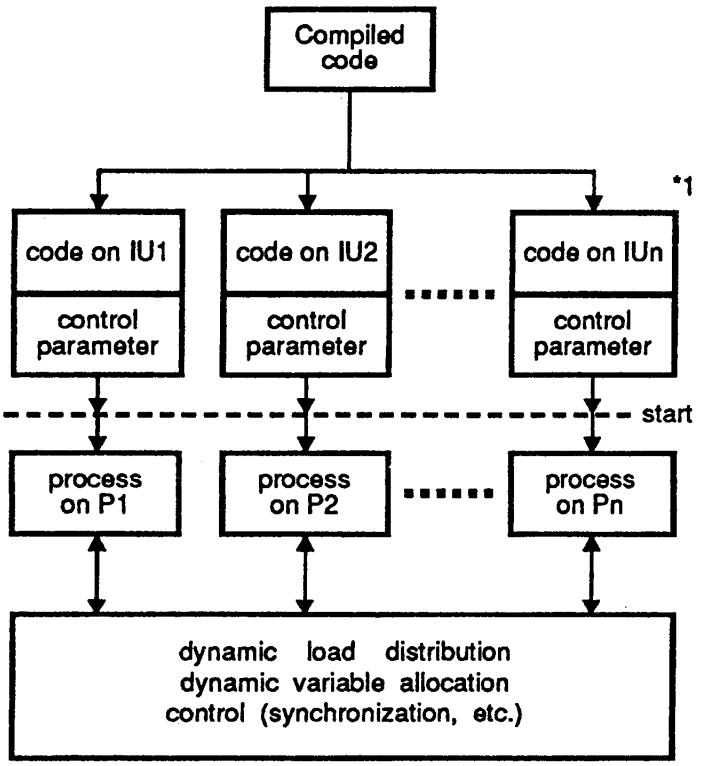
5 Control of parallel processing

5.1 Static analysis and dynamic control

Fleng is a very simple language and has a role of kernel language for the PIE-language system. However, the language level of Fleng is a little bit too high to be used for the machine language of PIE and for the implementation language of PIE operating system. For this kind of language, we designed a low level language Fleng−−, by which programmer can describe some characteristic aspect of code execution which is important from practical point of view.

While programs in Fleng++ are translated into Fleng−−, module information derived from the unit of objects is extracted in terms of control information from Fleng++ by compiler and used for load distribution. So the role of compiler is two fold, one is to generate object code, and the other is to extract the control information.

Figure 4 shows the procedure of code execution. The compiled code in the figure is the intermediate code in Figure 2. The code is distributed in parallel processing units(Inference Unit:IU) with such control information as parameter for load distribution and variable allocation by loader. At the time, the loader is given such resource-information as allocated number of Inference Units by operating system. Each loaded



*1 Duplicated allocation is allowed.

Figure 4: Procedure of Code Execution

code is a part of the compiled code and may be duplicated with the code in the other unit for efficient code execution.

Given a start command, each code is executed as a process. Processes communicate each other through message passing for load distribution and execution control among parallel goals.

So far as the load distribution is concerned, dynamic control as well as static allocation are used. The static analysis is used to allocate the compiled code to each IU at the beginning of runtime and to give the suggestive information to dynamic controller at runtime. Dynamic controller which exists on each IU dispatches its local load to some other idle IUs when the local load is heavier than others. The monitoring of other IUs' load is done with the aid of multi-stage interconnection network which has the capability of finding the minimum load IU. Accordingly, each IU can decide locally whether it should distribute the load or not.

We have two choices for load distribution strategy:

1. load minimum

2. designation of a specific IU by a statement such as "To the IU which has the variable `X` locally."

When a goal refers the variable `X` many times, (2) is selected. Otherwise, strategy (1) is selected, which is supported by network hardware.

As a matter of fact, dynamic load dispatching is decided with the variables location(in which IU) as a parameter. That is, the specific IU to which the load should be dispatched should be decided by taking the locality of variable access of the dispatched load into account.

Generally speaking, variable address is allocated dynamically to such IU that some activity which needs the variable is running. However, as the load is dispatched to the other IU, remote access becomes needed. Getting the IU address is a simple procedure as variable address is global in PIE machine.

The granularity of parallel execution is controlled in terms of the number of consecutive reductions. Execution is done sequentially until reduction count reaches the number.

5.2 Fleng--

Fleng-- is a low level language by which programmer can designate his intention regarding the behavior of program execution. That is, we added a few features to Fleng. Using these features, we can expect the following results.

- Compiled code becomes more efficient.
- Program can be controlled explicitly for load distribution and efficient execution.

The features introduced are the followings.

1. Active unify annotation (!)

   This annotation is used to designate that the terms attached with this annotation does not cause suspension when head unification is done. Accordingly, the terms can be replaced by variables and the unifications can be moved to the body part. For example, an append program:

```
append([H|X],Y,![H|Z]) :- append(X,Y,Z).
append([],X,!X).
```

   where the third terms of heads are the annotated one, is completely equivalent to the following program.

```
append([H|X],Y,Z1) :-
        append(X,Y,Z), Z1 = [H|Z].
append([],X,Y) :- Y = X.
```

   Through this annotation, we can improve the code execution efficiency by optimizing the unification procedure as the mode and structure of arity is partly known.

2. Non-variable annotation (#)

   This annotation is used to suspend the head unification when the variables in the head attached with this annotation is tested to be unified with variables in a goal. For example, a program

```
a(#X) :- b(X).
```

is suspended its execution when X is tried to be unified with some variable in the goal. However, such unification as between X and f(Y) is not suspended, as f(Y) is not a variable itself but a structure(though it includes a variable Y). This annotation is useful for waiting input and controlling parallelism by specifying additional data dependency.

3. Single reference annotation (')

This annotation is used to designate that the terms attached with this annotation are referenced only once. Generally, memory consumption rate of committed-choice language such as GHC and Fleng is relatively high compared with conventional languages. Accordingly, this annotation is very effective to reduce the frequency of garbage collection by telling to the language system that the memory cell can be collected and reused just after the head unification. It also reduces copying overhead.

For example, in a program

```
append('[H|X],Y,Z1) :-
            append(X,Y,Z), Z1 = [H|Z].
```

the list cell of [H|X] can be reused for generating [H|Z].

4. Load control

A special statement 'local_to' is introduced. This statement is used to designate the locality of activity. For example

```
a :- b,c,d.                                    (1)
a :- local(b,c),d.                             (2)
a(X,Y) :- (b(Y),local\_to Y),c(Y,Z).           (3)
```

(1) and (2) have the same semantics. However, local(b,c) in (2) means that literal b and c should be executed locally each other. (3) means that the execution of b(Y) should be done in the IU where the value of Y exists. When the value of Y is of some structure, and the internal part of the structure is tested by such clause as

```
b(Y1) :- ....,
```

the processing of b(Y) had better be executed at the IU that has the value of Y in its local memory, as many access to the elements of Y arises for unification.

5.3 Adding annotations automatically

Though the annotations are very effective, it can be a burden for application programmer to make use of them. Using the mode/type check result, we can attach the single reference annotations automatically. Following is the algorithm.

1. Macro expand all the active unify annotations into the body parts.
2. For all clauses, MR(Multiple Reference) marking is done as follows, where only such variables of which value-type is structure are paid attention.
   - Count the reference times for all variables of which usage mode is input in the body. When the variable appears in the head also, the count is added by one. When a variable appears more than one time in a body-goal, corresponding definition-clauses are checked as well.
   - When the count-number exceeds 3, MR mark is attached at the location of appearance for the variable in both the body-goal and the definition clauses.
   - For MR-marked variables in heads, MR mark is attached to the location of appearance in the definition clauses which correspond to the body-goal in which the variables appear.
3. When some structure is defined in the head without MR, a single reference annotation is attached to the structure.

Though the algorithm above can be used when the variable appears only once in the top-level goal, some provision is required for multiple appearance as follows.

1. Use copy scheme for such structure shared among goals of top-level by providing copy mechanism of structure in the language processor.
2. When the above algorithm decides that the single reference annotation can be attached in a predicate, copy the definition and rename the predicate in the copied program. Of which programs, original or copied one, is used is decided by the analysis of goals.

6 Discussion

6.1 Comparison with other parallel object-oriented languages

In the parallel object-oriented programming language such as ABCL/1[6] and Concurrent Smalltalk[7], consistent state transition is guaranteed by accepting next message only after the completion of a method execution. This resulted in a simple model.

In Fleng++, the next message can be accepted even before the completion of a method execution. This is realized safely through expressing states by single-assignment variables and serialized messages. While the introduction of single-assignment variables brings the

needs of efficient garbage collection usually, we can analyze statically almost all of the single reference for Fleng++ and decrease the frequency of garbage collection through collecting and reusing the cell just after the usage.

The objectives of Fleng++ is similar to Vulcan[8] in the meaning that they try to lighten the load of programmer for logic programming. A'UM[9] is a object oriented language based on the stream computation model and uses parallel logic programming language as a mean of implementation. However, Fleng++ supports the argument passing mechanism by unification without the designation of type/mode same as logic programming. Hence, we can get much freedom for argument passing. Moreover, Fleng programs can be rewritten easily into Fleng++ because both the syntax and semantics of Fleng++ are the extension of Fleng.

6.2 Programming environment

As for the programming environment of Fleng++, we are designing following tools.

1. Browser
2. Window programming interface
3. Parallel programming debugger
4. Object library

For (2), we are implementing an interface on X-window. And for (3), the usefulness of conventional trace is limited for parallel programming. On the other hand, declarative model for the debugging of logic programs is not enough for parallel logic programming which uses some committed choice languages, because they are not pure logic programming languages and they need to consider the timing of input/output and causal relationship. Accordingly, it is required to add some operational model. We designed a execution model which can represent the status of parallel execution in terms of communicating processes. Using the model, we are now designing the extension of algorithmic debugging.

7 Conclusion

The language set of Parallel Inference Engine PIE64 is presented. User level language, Fleng++, is a parallel object-oriented programming language based on logic. Fleng-- which is a low level language and has the facility of explicit control feature for parallel execution is also discussed.

8 Acknowledgement

The author would like to thank the members of SIGIE (Special Interest Group of Inference Engine): Hanpei Koike, Eiichi Takahashi, Hiroaki Nakamura, Minoru Yoshida, Xu

Lu, Kentaro Shimada, Takeshi Shimizu, Takeshi Shimoyama, Junichi Tatemura, Yasuo Hidaka, and Akiyoshi Katsumata. This work is supported by Grant-in-Aid for Specially Promoted Research of the Ministry of Education, Science and Culture of Japan (No. 62065002).

References

[1] H.Koike and H.Tanaka: *Multi-Context Processing and Data Balancing Mechanism of the Parallel Inference Machine PIE64*, Proc. Intl. Conf. Fifth Generation Computer Systems 1988, ICOT, pp.970-977, 1988.

[2] M.Kitsuregawa, M.Nakano and M.Takagi: *Query Execution for Large Relations on Functional Disk System*, IEEE Fifth Intl. Conf. Data Engineering, Feb. 1989.

[3] M.Nilsson and H.Tanaka:*Fleng Prolog-The Language which turns Supercomputers into Prolog Machines*,Proc. Japanese Logic Programming '86, in Wada,E.(Ed.), Springer LNCS 264, pp.170-179, 1987.

[4] K.Ueda: *Guarded Horn Clauses*, In Wada,E.(Ed.),Proc.Japanese Logic Programming Conference '85, pp.148-167, Springer LNCS 221, 1986.

[5] T.Chikayama: *Unique Features of ESP*, Proc. Intl. Conf. Fifth Generation Computer Systems, ICOT, pp.292-298, 1984.

[6] A.Yonezawa,J.P.Briot and E.Shibayama: *Object-Oriented Concurrent Programming in ABCL/1*, Proc. Conf. Object Oriented Programming, Systems, Languages and Applications, 1986.

[7] Y.Yokote and M.Tokoro: *The Design and Implementation of Concurrent Smalltalk*, Proc.Conf.Object Oriented Programming,Systems,Languages and Applications, 1986.

[8] K.Kahn, E.D.Tribble, M.S.Miller and D.G.Bobrow: *Vulcan: Logic Concurrent Objects*, In B.Shriber and P.Wegner(Ed.): Research Directions in Object Oriented Programming, MIT Press, 1987.

[9] K.Yoshida and T.Chikayama: *A'UM - A Stream-Based Concurrent Object-Oriented Language* -, Proc. Intl. Conf. Fifth Generation Computer Systems, ICOT, pp.638-649, 1988.