TR-509

An Overview of ExReps System

by
J. Tanaka, Y. Ohta & F. Matano (Fujitsu)

September, 1989

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191－5
Telex ICOT J32964

Institute for New Generation Computer Technology

An Overview of ExReps System

Jiro Tanaka[†], Yukiko Ohta[‡] and Fumio Matono[‡]

[†]IIAS-SIS, FUJITSU LIMITED,
1-17-25 Shinkamata, Ota-ku, Tokyo 144, JAPAN

[‡]FUJITSU SOCIAL SCIENCE LABORATORY LIMITED,
1-6-4 Osaki, Shinagawa-ku, Tokyo 141, JAPAN

Abstract

An overview of an experimental reflective programming system (ExReps) is presented in this paper. ExReps consists of two layers, i.e., abstract machine layer and execution system layer. Both layers are constructed based on *enhanced metacall mechanism*. Actual program execution examples and reflective programming examples are also presented.

1. Introduction

Nowadays, various *parallel logic languages*, which are logic-based languages and can still create *concurrent processes* dynamically, has been proposed. PARLOG [1], Concurrent Prolog [2] and GHC [3] are examples of such languages. Since these languages possess the notion of *processes* and *synchronization* inside language, it is quite natural to try to describe an *operating system* in these languages. In fact, various works has been done for *systems programming* from the very beginning of *parallel logic languages* [4] [5]. PPS (PARLOG Programming System) [6] and Logix [7] are the examples of such systems.

ExReps, which stands for "Experimental Reflective Programming System," has been built up [8]. We try to describe the overview of ExReps in this paper. A *programming system* can be defined as a small *operating system* where one can input programs and execute goals. Our objective is not building up a practical programming system like PPS or Logix. Rather, our interest exists in expressing an *elegant* programming system in more systematic manner [9].

The organization of this paper is as follows. Section 2 describes the overall structure of ExReps. Section 3 describes *metacalls* which provide us the bases for our programming system. ExReps consists of two layers, i.e., the abstract machine layer and the execution system layer. Respective layers are described in Section 4 and Section 5. The examples of application program execution are shown in Section 6. The reflective programming examples are shown in Section 7.

2. Overall structure of ExReps

The overall structure of ExReps is shown in Figure 1. ExReps is implemented on

PSI-II Machine [10]. Since the current version of PSI-II only understands ESP [11] which is the object-oriented dialect of Prolog, we install GHC system first. This GHC system is the slightly modified version of Ueda's GHC run-time system [12] and executes the compiled GHC programs.

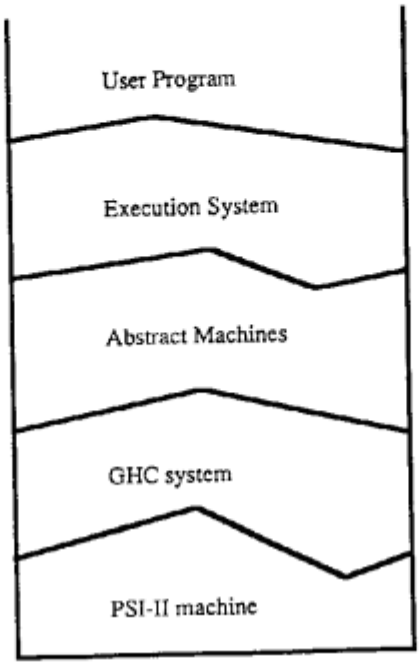

Figure 1 Overall structures of ExReps

Next layer is the *abstract machine layer*. Since various kinds of *multi-micro* distributed computers have become more popular, we expect ExReps should also be adaptable to distributed hardware. Our approach is constructing distributed abstract machine layer [13] on top of GHC system.

The execution system, which loads user programs and executes user jobs, is constructed on top of the abstract machine layer.

3. Stepwise enhancement of metacalls

User programs can be executed on a programming system. However, the programming system must not fail even if a user program fails. *Metacall* mechanism [4] works as a basic unit of *execution control* and *protection* in the programming system.

Various kinds of *metacalls* have already been discussed. Here, we briefly review how they work. The simplest metacall is the following single-argument metacall.

```
exec(G)
```

This metacall simply executes goal "G" and the result is exactly the same as the direct execution of "G." This form of metacall does not help much because the execution result is always same to the direct execution.

The first extension is the following three-argument metacall [4].

```
exec(G,In,Out)
```

Here, "In" is called *input stream* and "Out" is called *output stream*. Goal execution can be *suspended*, *resumed*, or *aborted* by sending appropriate messages from "In." When the execution of the metacall finishes successfully or failed, the metacall simply sends a *success* or *fail* message from Out. This metacal itself never fails, i.e., it has the capability of *failure protection*.

A goal "G" is executed in an interpreted manner in these metacalls. Therefore, we can separate two levels here, i.e., *meta-level*, where the top-level program execution is performed, and *object-level*, where the execution of programs is done inside a meta-interpreter.

Though this extension of the metacall aims at obtaining *object-level* information to *meta-level* world, there exists the other direction for extending metacall. It is realizing *reflective* capabilities [14] [15], where *object-level* program can obtain *meta-level* information.

This extension depends on resources we want to control. Since it is useful managing processes *dynamically*, we introduce a *scheduling queue* explicitly in our metacall. The *enhanced* metacall becomes as follows.

```
exec(H,T,In,Out)
```

The first two arguments express scheduling queue in difference list form. The use of difference list for expressing scheduling queue was originally proposed by Shapiro [2].

Next we introduce two more arguments, "MaxRC" and "RC," to control *reduction count* [16]. We assume that *reduction count* corresponds to the *computation time* in conventional systems. "MaxRC" shows the limit of the reduction count allowed in "exec." "RC" shows the current reduction count.

```
exec(H,T,In,Out,MaxRC,RC)
```

Implementing various *reflective operations* is not too difficult, once we get the enhanced metacalls. We consider four kinds of reflective operations, "get_rc," "put_rc," "get_q" and "put_q," of which *object-level* program can make use. Two kinds of meta-information are *reified* here, i.e., *scheduling queue* and *reduction count*. "get" operations obtain meta-information for the *object-level*. On the other hand, "put" operations return the information to the *meta-level*.

The meaning of each operation is as follows: "get_rc(MaxRC,RC)" gets "MaxRC" and "RC" from the meta-level, "put_rc(MaxRC)" resets "MaxRC" to the given argument, "get_q(H,T)" gets the current scheduling queue in difference list form, and "put_q(H,T)" resets the current scheduling queue to the given arguments.

In the programming system, this enhanced metacall is used in two ways. Since this metacall has *i/o channels*, a *scheduling queue* and a *reduction counter*, it can be considered as an abstract GHC machine. Therefore, it can be used for expressing an abstract machine. The other way is using it for *task management*. Since it provides us the unit of *execution control* and *protection*, we can use it for expressing a *user process*.

4. Abstract machine layer

In this section, we describe the conceptual image of the abstract machine layer first. Then the actual implementation of the abstract machine layer will be described.

4.1. Conceptual image of the abstract machine layer

We construct distributed abstract GHC machines on top of GHC system. On this layer, abstract GHC machines are connected by network and each GHC machine executes GHC goals.

For example, we can define the following ring-connected distributed computers by using "exec" and "nm," where each "exec" is the one described in the previous section, and "nm" is the network manager.

```
d_machine:-true|
        nm(Nm4,Nm1,In1,Out1),exec(T1,T1,In1,Out1,_,0),
        nm(Nm1,Nm2,In2,Out2),exec(T2,T2,In2,Out2,_,0),
        nm(Nm2,Nm3,In3,Out3),exec(T3,T3,In3,Out3,_,0),
        nm(Nm3,Nm4,In4,Out4),exec(T4,T4,In4,Out4,_,0).
```

Four "nm" processes are connected to the uni-directed ring. The output of one network manager is connected to the input of the other network manager. Each "nm" is also connected to a "exec." The scheduling queue of "exec" is initially empty. User goals can be entered in "exec" from the input stream.

Inside of each "exec," the ordinary GHC program runs. However each "exec" can throw goals which has *pragma* [17] to other "exec" through the output stream. Goal "A" which has pragma "@P" is expressed as "A@P." The kind of *pragma* depends on the topology of abstract machines. We assume that pragma "@forward" is used for uni-directed ring.

Each "nm" delivers the goal with *pragma*. If the goal has the pragma, "nm" simply peels off the *outermost* pragma and sends the remaining part to the next "nm." The goal which has no pragma is dropped to the "exec" connected to the "nm." Therefore, goal "A@forward@forward" will be dropped to "exec" located ahead by two.

However, these distributed GHC machines are isolated from the external world. The program of distributed machines with i/o becomes as follows:

```
d_machine:-true|
        window(O),
        keyboard(O1,I),
        nm(Nm4',Nm1,In1,Out1),exec(T1,T1,In1',Out1,_,0),
        nm(Nm1,Nm2,In2,Out2),exec(T2,T2,In2,Out2,_,0),
        nm(Nm2,Nm3,In3,Out3),exec(T3,T3,In3,Out3,_,0),
        nm(Nm3,Nm4,In4,Out4),exec(T4,T4,In4,Out4,_,0),
        dist(Nm4,Nm4',O2),
        merge(I,In1,In1'),
        merge(O1,O2,O).
```

The overall structure of this distributed machine is shown in Figure 2. Comparing to the previous program, a *window* and a *keyboard controller* are added for the interface to the outer world. The window takes care of all inputs and outputs at the abstract machine layer level. The keyboard controller is used to generate the read request so that we can input goals from the window. Two "merge" processes are added to join "I" to "In," and "O1" to "O2" We assume that goal "A@io," which is the i/o operation, simply passes through "nm." However these goals are captured by "dist" and sent to the window.

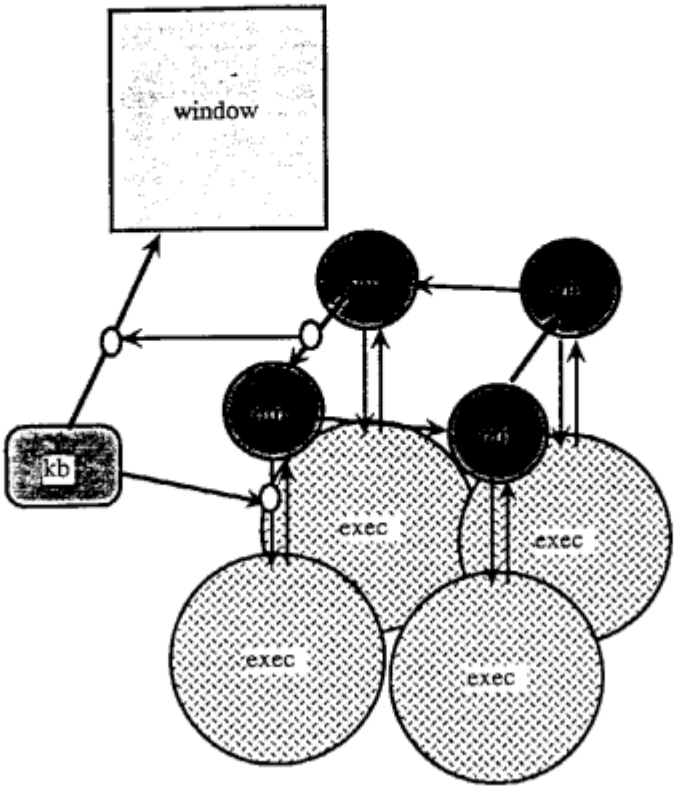

Figure 2 Distributed abstract machines

4.2. Actual implementation of abstract machine layer

In the actual system, we can build various distributed abstract machines, such as *linear array, ring, cube, square mesh, triangular mesh, hexagonal mesh* and *tree*. We can also specify the size of network, i.e., the number of processors, we want to construct. There exist different *pragmas* for different topologies. For example, "`@forward`" was used for uni-directed ring. However, "`@right`," "`@left`," "`@up`" and "`@down`"are used for square mesh.

An example of the actual program execution on ExReps is shown in Figure 3. After installing GHC system, we execute the *abstract machine construction* program which can *dynamically* create the network of abstract GHC machines following the user input. This program opens "`VM_window`" in PSI-II display (the upper-left corner). We can specify the topology and the size of the network from this "`VM_window`."

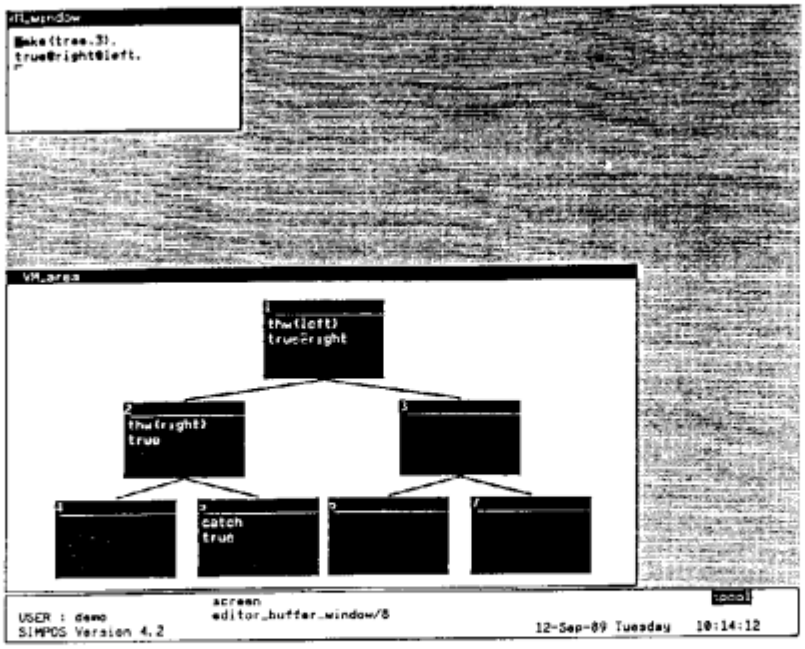

Figure 3 Abstract machine construction on ExReps

In this case, we typed in "`make(tree,3),`" which *dynamically* created the *tree* network of size 3. The structures of abstract machines will be displayed in "`VM_area`" window (lower-left corner). This window is used for displaying the execution state of each abstract machine. Each abstract machine, which is shown as a black square, is also a window and the executing goals are displayed through those windows.

When the abstract machines are constructed, "`VM_window`" is connected to the GHC machine #1. We need to use *pragmas* to throw goals to other abstract machines. Therefore, if "`true@right@left`" is typed in, where goal "`true`" corresponds to "`noop`" in conventional languages, that goal will be carried to the GHC machine #5 through network managers. (Note that pragmas are peeled off from the *outermost* level.)

5. Execution system layer

In this section, we first describe the structures of the execution system layer. Then the *shell* which plays the central role in the execution system is shown. The actual implementation of the execution system is also shown.

5.1. Structures of the execution system

It is possible to execute an application program directly on top of the abstract machine layer. However, application program are usually executed on the operating system. The execution system layer works such an *operating system* and provides the user the capability of entering *user programs* and execution control. The structures of the execution system can be be illustrated in Figure 4.

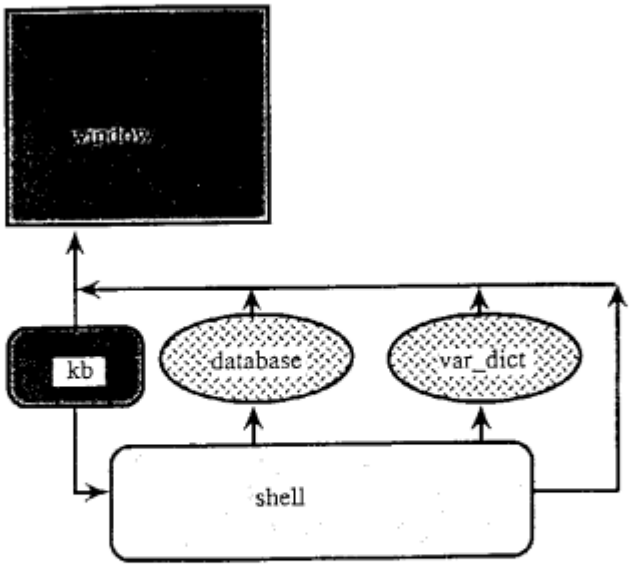

Figure 4 Execution system layer

The execution system layer consists of a *window*, a *keyboard controller*, a *shell*, a *database server* and a *variable dictionary*. The *window* is the *system window* of the execution system. It takes care of all inputs and outputs of the system. The *keyboard controller* always generates the read request to the system window. Therefore, we can always input goals or commands from the system window. The *shell* generates the user processes depending on the inputs from the user. The *shell* will be explained in the following section. The *database server* is the part which keeps the user program. We can *add*, *delete* and *check* the user program definitions. The *variable dictionary* provides us the facilities for defining *macro*. It can memorize the values of variables as its internal state and replace the user's query by its value.

5.2. Shell

The shell creates the user task, accesses to the *database server* or sends messages to the *variable dictionary*, depending on messages from the user. The following is the program for the *shell*.

```
shell([],Val,Db,MaxRC,Out):-true|
        Val=[],Db=[],Out=[].
shell([goal(Goal)|In],Val,Db,MaxRC,Out):-true|
        Val=[record_dict(Goal,NGoal)|Val1],
        window(WOut),
        keyboard(KOut,EIn),
        exec_server(run,NGoal,EIn,EOut,I,O),
        exec([NGoal|T],T,I,O,MaxRC,O),
        shell(In,Val1,Db,MaxRC,Out),
        merge(KOut,EOut,WOut).
shell([db(Message)|In],Val,Db,MaxRC,Out):-true|
        Db=[Message|Db1],
```

```
        shell(In,Val,Db1,MaxRC,Out).
shell([binding(Message)|In],Val,Db,MaxRC,Out):-true|
        Val=[Message|Val1],
        shell(In,Val1,Db,MaxRC,Out).
```

The "shell" has five arguments; the first is the input stream, the second is the stream to the *variable dictionary*, the third is the stream to the *database server*, the fourth is the internal state which specifies the maximum reduction count allowed for the user process, and the fifth is the output stream.

This program works as follows:

1. If the input stream of "shell" is "[]," it means the end of input. All streams will be closed in this case.

2. If "goal(Goal)" is in the input stream, "Goal" is sent to the variable dictionary. The variable dictionary checks the bindings of every variable in "Goal" and creates "NGoal" where every variable is bound to the current bindings. Then "exec_server" and "exec" are created. The "exec_server" works as a back-end process of "exec." It keeps the internal state and provides the user various kinds of services. A *user* window and a *user* keyboard controller are also created.

3. If "db(Message)" or "binding(Message)" is in the input stream, "Message" is sent to the appropriate stream. Actually, an application program can be registered to the *database server* by message "db(assert(Program))." "binding(Message)" is used for *registering* and *checking* the current bindings of variables.

Figure 5 shows the snapshot where processes are created in accordance with the user input. Each *exec* has their own window and keyboard. Therefore, we can enter commands from the *user* window. Once created they run independently from the *shell*.

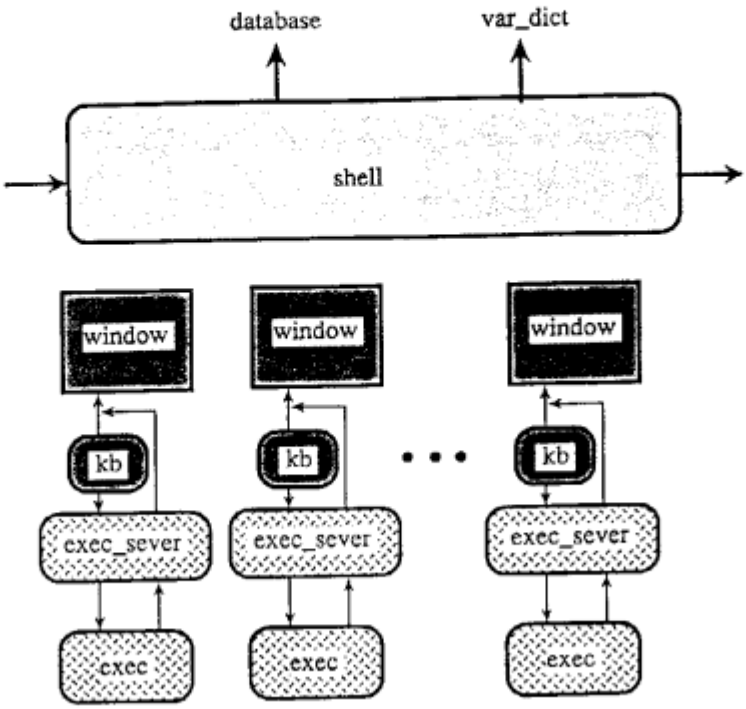

Figure 5 The creation of processes in *shell*

5.3. Actual implementation of the execution system layer

In the actual ExReps system, the execution system will be created by typing in "ps" from "`VM_window`." When it is created, "`PS_window`" is also opened. We can input user programs and goals from "`PS_window`."

An actual program execution example is shown in Figure 6. In this case, we created abstract machines of 2 by 2 mesh and installed *primes* program which computes the list of prime numbers lower than the given number.

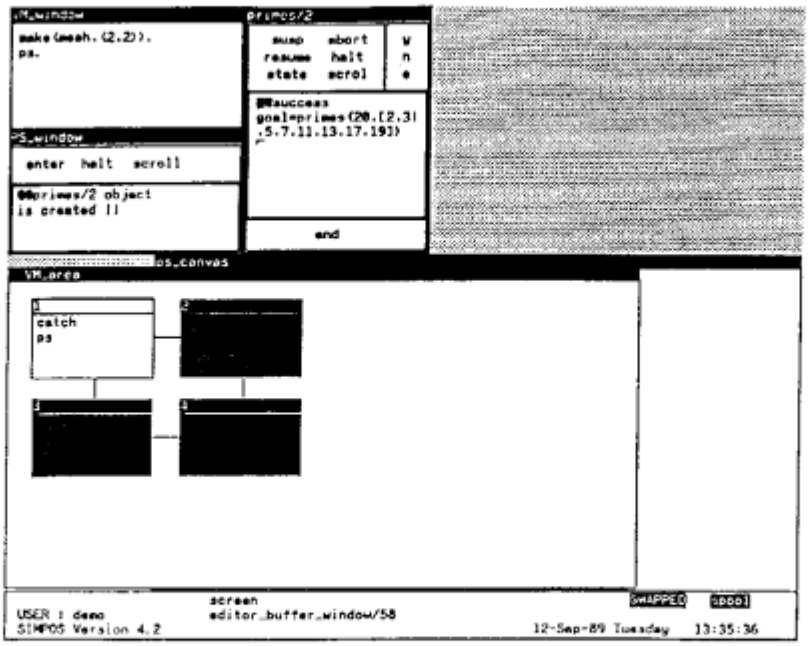

Figure 6 User program execution on ExReps

When we type in "`primes(20,X)`" from "`PS_window`," user process "`primes/2`" will be created. It has its own user window "`primes/2`" and we can "`suspend`," "`abort`" or "`resume`" the process *dynamically* by sending appropriate messages from the window. We can execute other commands such as "`halt`," "`state`" and "`scrol`." "`halt`" is used for closing the user window. "`state`" shows the current variable bindings of the input goal. "`scrol`" is used for scrolling up/down the i/o window. The result of computation is shown in the i/o sub-window of "`primes/2`" as the *bindings* of the input goal.

6. Application program execution on ExReps

In this section, we show two examples of application program execution. The first is *four queen* problem. This example shows the distributed execution of a user program on abstract machines. The second is *odd-number-learning* example. This example is for showing the interactive execution capabilities of ExReps.

6.1. Four queen program

As mentioned before, we can execute *user program* in a distributed manner by adding *pragmas* to the application program. As an example, we chose the well-known *n queen* problem. This problem is to find the solution of locating *n queens* in *n by n* grid non-overlapping with vertical, horizontal and diagonal directions.

We tried to execute *four queen* problem in a distributed manner. Our solution is generating 4 by 4 processes which correspond to 4 by 4 grid and solving the problem by stream communications between processes.

The actual program execution is shown in Figure 7. In this case we created 4 by 3 grid abstract machines.

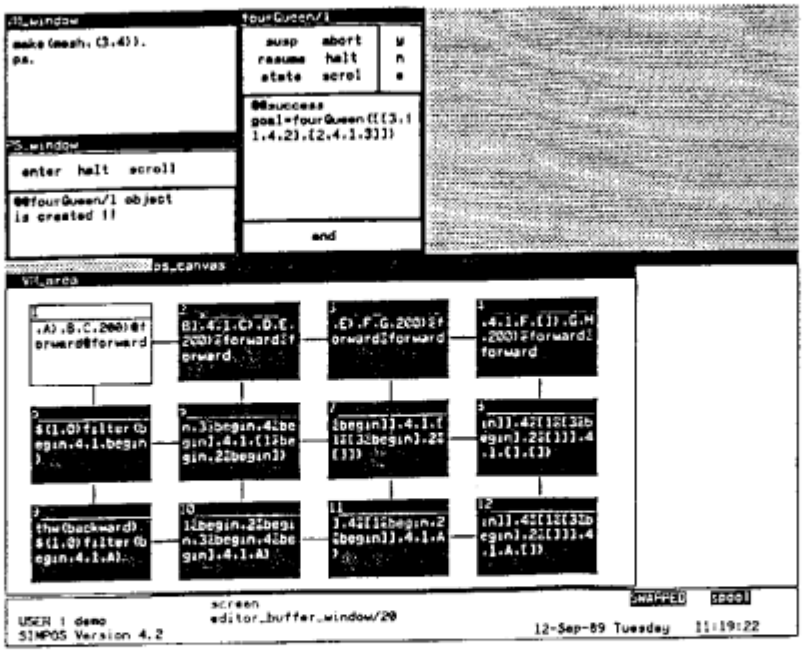

Figure 7 Four queen program execution on ExReps

The reason we created such 4 by 3 machines is for showing the folding capabilities of abstract machines. Though this program creates 4 by 4 processes, they must be folded to the 4 by 3 abstract machines. Folding is realized by extending the interpretation of *pragma* to the opposite direction. In this case, pragma "`@down`" at the bottom row of the grid is interpreted as *up*.

There are two solutions for four queen problem and the computation result is shown in "`fourQueen/1`" window.

6.2. Odd-number-learning

The other application is *odd-number-learning* example. This program is made for demonstrating the interactive program execution capabilities of ExReps.

As explained before, we can *suspend*, *abort* and *resume* the execution of user program from *user window*. However, we sometimes need more *fine-grain* execution control.

GHC program has the notion of *processes* and *streams*. (A *process* is considered as a tail-recursive *goal*. A *stream* is considered as a *variable* which is instantiated gradually.) Therefore, we would like to control the execution of each *process* or each *stream*.

Our approach is as follows. We declare *processes* and *streams* explicitly in the user program to distinguish them from ordinary *goals* and *variables*. When a *process* or a *stream* is created, the system asks *dynamically* whether he wants to open a window in "`ps_canvas`." If answered yes, a *process* or *stream* window will be created. We can *see* and *control* the current state of a *process* or a *stream* from there. (Here, "`ps_canvas`" window is considered as a kind of a blackboard in which we can freely write figures and communicate.)

The *odd-number-learning* example is shown in Figure 8.

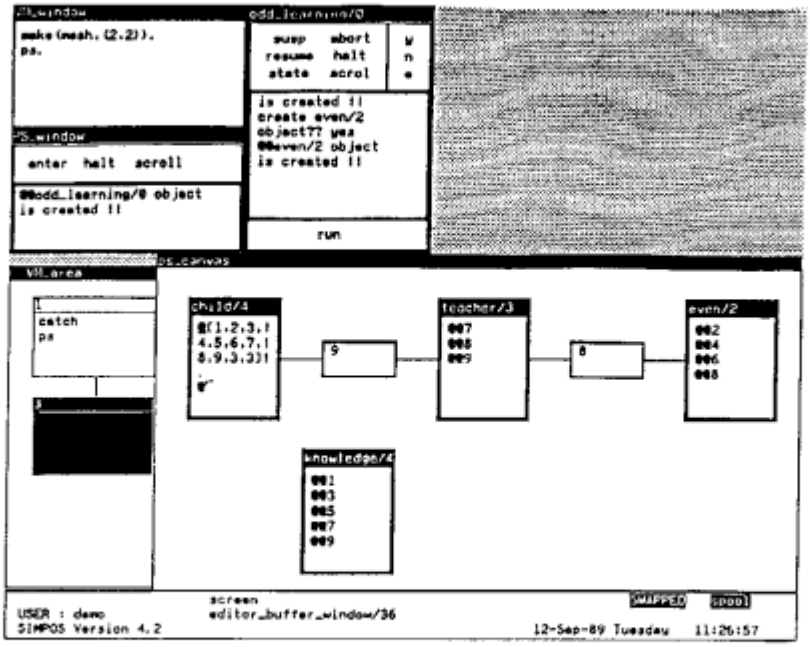

Figure 8 Odd-number-learning example

This example consists of four processes, i.e., *child*, *knowledge*, *teacher* and *even* processes.

We can input *integer* numbers from the *child* process. The *child* process can access to the *knowledge* process and is expected to throw only the even numbers to *teacher* process.

The *knowledge* process can contain *odd* numbers. However, it is initially empty. Initially, *child* does not recognize the given number whether it is even or odd. Therefore, it simply passes the given number to the *teacher*. The *teacher* knows whether it is odd or even. If it is odd, *teacher* teaches the *child* that it is odd and is entered to the *knowledge* process. If it is even, is is simply passed to the *even* process.

Next time the same odd number is entered from the *child* process, *child* can recognize that the number is odd by consulting the *knowledge* process and does not pass it to the *teacher*.

This *odd-number-learning* example is the interactive system which models the learning processes of a child at the very superficial level.

In Figure 8, four processes and two streams are explicitly declared and opened windows. We can see the current state of processes or streams from these windows. The execution of processes and streams can be *suspended*, *aborted* or *resumed* separately by entering commands from these windows.

7. Reflective programming on ExReps

We can write *reflective* programs by using *reflective operations* described in Section 3. In this section, we show two examples of *reflective programming*. The first example is the *load balancing* program, the second is the *dynamic reduction count control* program.

7.1. Load balancing

The first example is the *load balancing* program which is executed directly on top of abstract machines. The *load balancing* program can be written as a *reflective* program by using *reflective operations.*

Reflective operations must be executed *urgently.* Therefore, we introduce the notion of *express goals*, which have the form "G@exp." We assume that *express goals* are executed *urgently* in "exec." While *express goals* being processed, the normal execution of goals are *frozen.* When *express goals* are decomposed to subgoals, they are also processed as *express goals.*

The *load balancing* program is shown below. If we enter "load_balance@exp" as a goal executed on the abstract machine, it automatically moves among abstract machines and performs load balancing.

```
load_balance:-true|
        get_q(H,T),
        length(H,T,N),
        balance(N,H,T).

balance(N,H,T):-N>100|
        N1:=N-100,
        separate(N1,H,T,NH,NT,X),
        goals(X)@exp@down,
        load_balance@exp@right,
        put_q(NH,NT).
balance(N,H,T):-N=<100|
        load_balance@exp@right.
```

When "load_balance@exp" is executed inside an abstract machine, it goes into the *express* state. The current scheduling queue of the abstract machine is taken out and the length of the queue is computed. If it is longer than 100, "N1" excessive goals are separated from the scheduling queue and thrown out. "load_balance@exp" goal is also thrown out to the *right* direction to invoke load balancing on other abstract machines. (If the abstract machine is at the right end, it is thrown to the *left.*) If the length of the queue is shorter than 100, it simply throws the "load_balance@exp" goal to the *right* (or *left*) abstract machines.

Figure 9 shows the snapshot of executing this *load balancing* program.

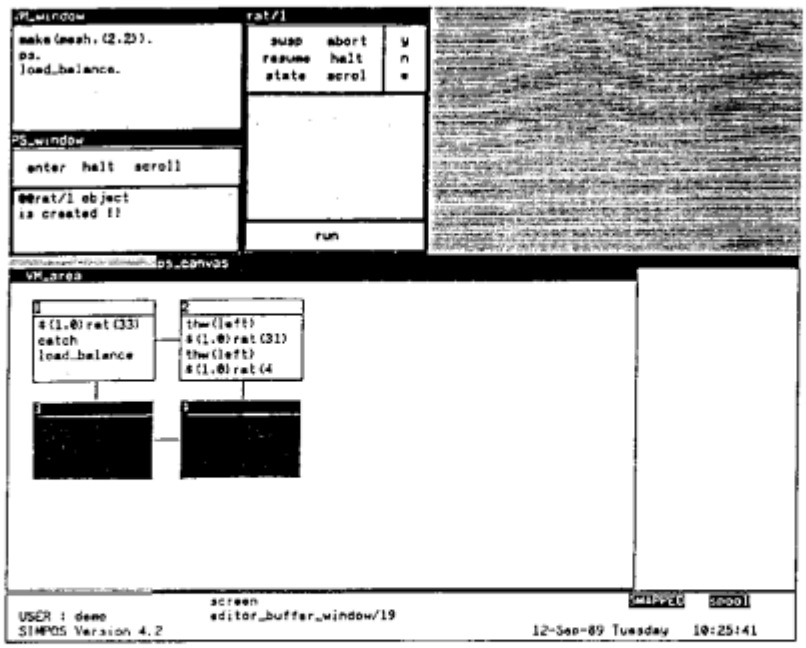

Figure 9 The execution snapshot of *load balancing* program

We assumed 2 by 2 mesh abstract machines. This *load balancing* program is entered from "`VM_window`." It means that this program is executed in parallel with the execution system program. As a user program, we execute *rat* programs only on machines #1 and #2. These *rat* programs produce children incessantly and the load of abstract machines becomes busy as time goes on. The *load balancing* program is also moving between machine #1 and #2. When the load of the machines #1 and #2 exceed the limitation, the excessive goals are thrown out to the machines #3 and #4.

7.2. Dynamic reduction count control

The second example is *dynamic reduction count control* program which is executed at the *user program level*. The following program shows how to define the "`check_rc`" predicate, which asks the user whether it wants to change the remaining reduction count of the user task.

```
check_rc:-true|
        get_rc(MaxRC,RC),
        RestRC:=MaxRC-RC,
        output([rc_rest=,RestRC])@io,
        input([change_rc??],Ans)@io,
        check(Ans,MaxRC,RC).

check(yes,MaxRC,RC):-true|
        input([add_rc>>],AddRC)@io,
        NMaxRC:=MaxRC+AddRC,
        put_rc(NMaxRC).
check(no,_,_):-true|true.
```

We assume that the input and output operations have the format of "`input(Message_list,X)`" and "`output(Message_list)`," respectively. In the case of the input, "`Message_list`" is printed first, then the user's input is instantiated to X.

We insert this "`check_rc@exp`"goal in the application program. Whenever this goal is executed, it it gets "`MaxRC`" and "`RC`" and computes the remaining reduction count. After displaying the remaining reduction count, it asks whether he wants to change the reduction count. If answered "`yes`," it asks how many reduction count he wants to increase. Then it computes the new maximum reduction count and stores it as the new "`MaxRC`" of the user task. If answered "`no`," it will do nothing.

The execution snapshot of this program is shown in Figure 10.

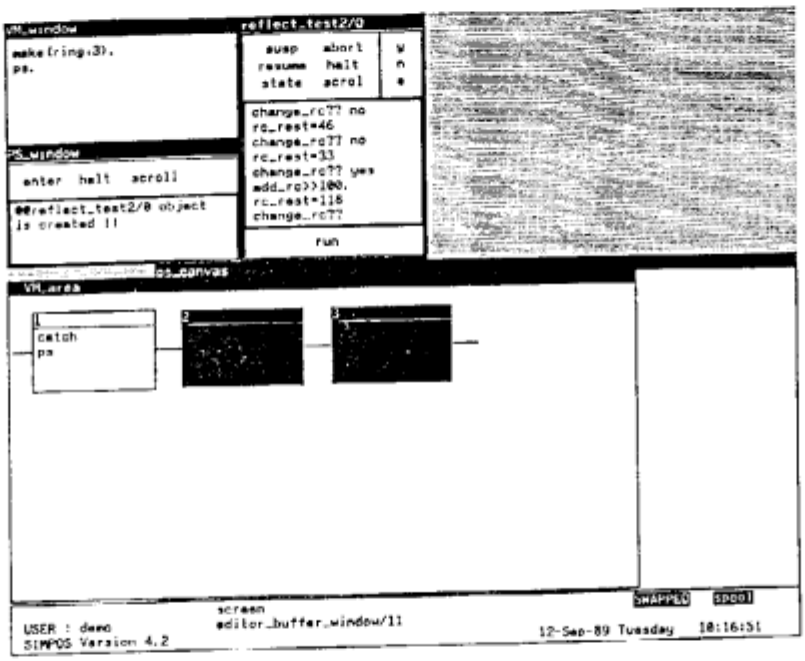

Figure 10 Dynamic reduction count control example

Though we created the ring abstract machines of size 3 here, the structure of abstract machines are irrelevant here. In this example, the current remaining reduction count is decreasing gradually. When it became "33," we added "100." However, since we *consume* reduction count before we come across the next "`check_rc@exp`"goal, the next display of the remaining reduction count is "118."

8. Conclusion

We have presented an overview of ExReps system. After describing *enhanced metacall mechanism*, we described abstract machine layer and execution system layer of ExReps system. The enhanced metacall has been used in two ways. One is for expressing an abstract machine, and the other is for managing user task. Program execution examples and reflective programming examples are also presented.

Our approach can be classified as a *software-oriented* approach. Our "exec" can be written in GHC. In contrast, PIMOS [18] tries to implement their "exec" directly as a built-in predicate. PIMOS tries to realize various features of distributed operating system in *machine-dependent* and *hard-wired* way.

On the other hand, *reflective operations* work as *wires* which connect *meta-level* and *object-level* in our approach. The *object-level* world can obtain *meta-level* information through these *wires*. User can write *object-level* programs which handle *meta-level* information and the *modified met-level* information can be *reflected back* to the *meta-level*. These result the flexible and powerful system which consists of small core.

Though *reflective* operations are defined as an ad hoc way, other resources, such as *variable environment*, can also be controlled in a similar manner [19]. Defining *reflective operations* in more sophisticated way [15] [20] is also possible.

The current version of ExReps is implemented on PSI-II. However, we imagine that the extension to distributed hardware, such as Multi-PSI [21], will not be difficult.

Also note that the programs shown here are the extremely simplified version. The more complete version of ExReps, running on PSI-II, has already been demonstrated at FGCS'88 and is available from the authors.

9. Acknowledgments

This research has been carried out as a part of the Fifth Generation Computer Project. I would like to express my thanks to Youji Kohda, Hiroyasu Sugano and Susumu Kunifuji for their discussions and encouragements.

References

[1] Clark, K. and Gregory, S.: PARLOG, Parallel Programming in Logic. Research Report DOC 84/4, Department of Computing, Imperial College of Science and Technology, Revised 1985.

[2] Shapiro, E.: A Subset of Concurrent Prolog and Its Interpreter. ICOT Technical Report, TR-003, 1983.

[3] Ueda, K.: Guarded Horn Clauses. ICOT Technical Report, TR-103, 1985.

[4] Clark, K. and Gregory, S.: Notes on Systems Programming in Parlog. Proc. International Conference on Fifth Generation Computer Systems 1984, ICOT, 1984, pp.299-306.

[5] Shapiro, E.: Systems programming in Concurrent Prolog. Proc. 11th Annual ACM Symposium on Principles of Programming Languages, ACM, January 1984, pp.93-105.

[6] Foster, I.: The Parlog Programming System (PPS). Version 0.2, Imperial College of Science and Technology, 1986.

[7] Silverman, W., Hirsch, M., Houri, A. and Shapiro, E.: The Logix System User Manual. Version 1.21, Weizmann Institute, Israel, July 1986.

[8] Tanaka, J.: Experimental Reflective Programming System "ExReps." Demonstration material of the International Conference on Fifth Generation Computer Systems 1988, ICOT, November 1988.

[9] Tanaka, J.: A Simple Programming System Written in GHC and Its Reflective Operations. Proc. The Logic Programming Conference '88, ICOT, Tokyo, April 1988, pp.143 149.

[10] Nakashima, H. and Nakajima, K.: Hardware Architecture of the Sequential Inference Machine: PSI-II. Proc. 1987 Symposium on Logic Programming, San Francisco, 1987, pp.104-113.

[11] Chikayama, T.: Unique Features of ESP. Proc. International Conference on Fifth Generation Computer Systems 1984, ICOT, 1984, pp.292-298.

[12] Ueda, K. and Chikayama, T.: Concurrent Prolog Compiler on Top of Prolog. Proc. 1985 Symposium on Logic Programming, Boston, 1985, pp.119-126.

[13] Taylor, S., Av-Ron, E. and Shapiro, E.: "A Layered Method for Process and Code Mapping." *Concurrent Prolog: collected papers,* Shapiro, E., ed., The MIT Press, 1987, pp.78-100.

[14] Maes, P.: Reflection in an Object-Oriented Language. Preprints of the Workshop on Metalevel Architectures and Reflection, Alghero-Sardinia, October 1986.

[15] Smith, B.C.: Reflection and Semantics in Lisp. Proc. 11th Annual ACM Symposium on Principles of Programming Languages, ACM, January 1984, pp.23-35.

[16] Foster, I.: Logic Operating Systems: Design Issues. Proc. Fourth International Conference on Logic Programming, Vol.2, The MIT Press, May 1987, pp.910-926.

[17] Shapiro, E.: Systolic programming: A paradigm of parallel processing. Proc. International Conference on Fifth Generation Computer Systems 1984, ICOT, 1984, pp.458-471.

[18] Chikayama, T., Sato, H. and Miyazaki, T.: Overview of the parallel inference machine operating system (PIMOS). Proc. International Conference on Fifth Generation Computer Systems 1988, ICOT, November 1988, pp.230-251.

[19] Tanaka, J.: Meta-interpreters and Reflective Operations in GHC. Proc. International Conference on Fifth Generation Computer Systems 1988, ICOT, November 1988, pp.774-783.

[20] Watanabe, T. and Yonezawa, A.: Reflection in an Object-Oriented Concurrent Language. Proc. ACM Conference on Object-Oriented Programming Systems, Languages and Applications, San Diego, September 1988, pp.306-315.

[21] Uchida, S., Taki, K., Nakajima, K., Goto, A. and Chikayama, T.: Research and development of the parallel inference system in the intermediate stage of the FGCS project. Proc. International Conference on Fifth Generation Computer Systems 1988, ICOT, November 1988, pp.16-36.