ICOT Technical Report: TR-222

TR-222

An Object-oriented Programming Language based on A Parallel Logic Programming Language KL 1

by

M. Ohki, A. Takeuchi and K. Furukawa

December, 1986

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 Object-oriented Programming Language based on A Parallel Logic Programming Language KL1

Masaru Ohki, Akikazu Takeuchi * and Koichi Furukawa

ICOT Research Center,
Institute for New Generation Computer Technology,
Mita Kokusai Bldg. 21F, 1-4-28, Mita,
Minato-ku, Tokyo, 108, Japan

Abstract

We have been studying a knowledge programming language called Mandala, based on object-oriented programming. In this paper, we describe an implementation of an object-oriented programming language as part of our study of Mandala. Object-oriented programming is well-suited to parallel execution. However, for many parallel executable object-oriented programming languages, it is only possible to execute procedures for individual objects in parallel, but not procedures within objects. We propose a language which can execute procedures within objects in parallel. The language is implemented on KL1(Kernel Language Version 1) which is a parallel logic programming language based on GHC, and maintains the KL1 feature that a unit of parallel execution is small. However, it is difficult to implement the instance variables, the internal states of objects, because this involves multiple access to resources, i.e., to instance variables. First, we propose a method for implementing access of instance variables to avoid parallel multiple access, called the single assignment method. An instance variable can only be updated once while processing a message. We designed an object-oriented programming language using this approach so that this language would have the functions of an object-oriented subset of Mandala. This language has the instance variable, the is_a hierarchy and the part_of relation as language primitives.

*) current address : Central Research Laboratory, Mitubishi Electric Corp. 1-1, Tsukaguchi-Honmachi 8-Chome, Amagasaki, Hyogo, 661, Japan

1. Introduction

We have been studying the knowledge programming language Mandala [Furukawa 84] on KL1(Kernel Language Version 1) [ICOT 85, Tanaka 86]. KL1 is a parallel logic programming language based on the resolution mechanism of GHC [Ueda 86], and it is originally the language for the PIM(Parallel Inference Machine) hardware which is developed in ICOT. KL1 provides object-oriented programming using a perpetual process similar to Concurrent Prolog [Shapiro 83a]. Mandala has realized object-oriented programming using this mechanism. Since its first implementation aimed to verify the power of the language, the object-oriented part of Mandala was not entirely implemented on KL1 [Furukawa 83]. In this paper, we describe an implementation of an object-oriented programming language, a subset of Mandala, in KL1 alone as part of our study of Mandala.

Object-oriented programming is not only useful for writing programs but also suitable for parallel execution, and many parallel executable object-oriented programming language have been proposed [Yonezawa 86, Tokoro 84]. These programming languages can execute procedures on individual objects in parallel, but cannot execute procedures within objects in parallel, with some exceptions, e.g. [Kahn 86]. The language we are proposing can even execute procedures within objects in parallel. We do not want to restrict the executable parallelism unit of KL1. If procedures within an object are executable in parallel, the following advantages result.

(1) It becomes possible to improve the parallelism.
If procedures within objects cannot be executed in parallel, it is because the parallelism is less than or equal to the numbers of objects. The parallelism is the number of processes which can be executed in parallel. However, if the procedures within objects can be executed in parallel, the parallelism can exceed the number of objects. Users can naturally extract the entire parallelism of programs by considering only local parallelism within an object. But if procedures within objects could not be executed in parallel, the users could only extract the entire parallelism by increasing the number of objects.

(2) It becomes unnecessary to introduce special message sending primitives.
In our language, it is possible for programmers to control the synchronization of the execution between objects using the suspend mechanism of KL1 after sending messages to other objects. For the language whose procedures cannot be executed in parallel within an object, special message sending primitives are necessary to process objects in parallel. They are related to the control of execution after sending messages. Most parallel object-oriented programming languages have at least two types of message-sending primitives. Those of one type wait to return the reply, and the others do not. Moreover, a special primitive is necessary to wait for replies from other objects.

However, if we make procedures within an object executable in parallel and the syntax of the methods which programmers should describe is abstracted like Mandala, it becomes difficult to implement instance variables which are the internal states in objects. The reason is that parallel multiple access to a resource, an instance variable, occurs.

In this paper, we propose an implementation method of instance variables to avoid parallel multiple access, called the single assignment method. This method is restricted to updating an instance variable only once during processing a message. Using this approach we design an object-oriented programming language with the functions of an object-oriented part in Mandala. Our language has

(1) instance variable,
(2) is_a hierarchy, and
(3) part_of relation

as language primitives.

There are some differences between Vulcan and our language. The methods of Vulcan [Kahn 86] are different from its implementation language Concurrent Prolog [Shapiro 83a], but they are rather similar to Smalltalk [Goldberg 83]. The methods of our language are exactly KL1 clauses. We think that KL1 is a powerful language. Vulcan introduces some sequentialities to its language specification to implement instance variables and sending messages, but our language avoids sequentiality in implementing instance variables and sending messages. The inheritance mechanism of our language is implemented by calling methods in superworlds, but Vulcan's implementation involves copying methods in superclasses or delegation of parts. We cannot use both methods, because the copying method may generate enormous translated codes, and it is impossible to call methods in superworlds from the guard in the delegation method.

We present the above problem and implementation methods in Section 2. We have considered three methods, but chose the single assignment method, which is more suitable for our purpose than the other two. Section 3 contains a language specification and description of its implementation, and Section 4 presents an example of a simple programming environment.

2. Implementation Issues for Parallel Execution in Objects

2.1 KL1

KL1 is a parallel logic programming language based on GHC [Ueda 86]. It is similar to Concurrent Prolog and PARLOG [Clark 83] in that its program consists of Horn clauses with guards in the following form.

```
H :- G1,.., Gm | B1,.., Bn. (m>0, n>0).
```

where H, Gi and Bi are atomic formulas. H is called a head, the Gi's are called guard goals and the Bi's, body goals. "|" is called a commit operator, the left part (H,G1,..,Gm) of the commit operator is called the guard part, and the right part (B1,..,Bn) is called the body part.

To execute a KL1 program is to resolve a KL1 goal according to the resolution rule. The resolution can be performed under the suspension rule [Ueda 86] as shown below. That is, several goals may be executed in parallel, and several clauses whose predicate names are the same as the goal are tried in parallel.

(1) Unification invoked directly or indirectly in the guard of a clause "C" called by a goal "G" (i.e., unification of "G" with the head of "C" and any unification invoked by solving the guard goals of "C") cannot instantiate the goal "G".
(2) Unification invoked directly or indirectly in the body of a clause "C" called by a goal "G" cannot instantiate the guard of "C" or "G" until "C" is selected for commitment.

A piece of unification that can succeed only by causing such instantiation is suspended until it can succeed without causing such instantiation.

2.2 Approach to an object-oriented programming language

KL1 is well-suited to object-oriented programming [Shapiro 83b]. We can perform object-oriented programming without introducing special primitives to KL1. When writing a program in KL1 in object-oriented programming, we can implement an instance (we often use the word "instance" instead of "object" to emphasize that we are dealing with an object not a class, which is a template of an object) using recursive call as a perpetual process, and regard a set of clauses having the same predicate name as templates of an instance. The name corresponds to the name of a class to which the instance belongs. Here is an example of object-oriented programming using KL1.

```
counter([up|I],State) :-
        New_State := State + 1, counter(I,New_State).
counter([down|I],State) :-
        New_State := State - 1, counter(I,New_State).
counter([show(X)|I],State) :-
```

```
            State = X, counter(I,State).
    counter([],State).
```

This is a simple counter. The predicate name "counter" indicates the class name. A variable "State" is the internal state that indicates the value of the counter, and a variable "New_State" contains the value of the modified new internal state. An instance is a perpetual process. A predicate "counter" is recursively called in the first three clauses. An instance of "counter" counts up when it receives an "up" message, it counts down when it receives a "down" message, and it replies with its value when it receives a "show" message. The "up" and "down" messages modify the internal states "State" to the new internal states "New_State". The last clause is used to terminate a perpetual process when the final message "null" arrives.

The above example shows that KL1 can easily realize object-oriented programming, but we want to describe programs briefly, as in Mandala. The following features are necessary for this.

(1) Omission of recursive call
We abstract template clauses for instances, like Mandala, to the following:

```
instance([Message|Input],State) :-
        simulate(Message,State,New_State),
        instance(Input,New_State).
```

In this clause, the section to solve messages is concentrated in the "simulate" predicate. Now it is unnecessary to describe recursive call in each clause.

(2) Omission of arguments to describe instance variables
This makes an instance variable a special primitive. Programmers are no longer able to deal with them like ordinary logical variables, but can only access them by special methods. An instance variable becomes a shared resource held in common by several parallel processes. Multiple access to shared resources is a difficult problem. The difficulty results from parallel multiple access to one resource without specifying synchronization of the accesses. For example, suppose predicate "get" is a command to read from an instance variable and predicate "put" is a command to write to an instance variable. Let us consider the following example to access an instance variable, whose name is "state".

```
,..., put(state,X1),..,put(state,X2),..,get(state,X3),...,
```

Since all goals are executed in parallel, the two "put" predicates can be executed in parallel. If parallel multiple access to the same logical variable which implements the instance variable occurs, the "put" predicate executed later may fail because variable "X1" may not be unifiable to variable "X2". Moreover, whether the "get" predicate reads the value of either "X1" or "X2", or the value before both "put" predicates are executed, may be left undetermined.

(3) The introduction of inheritance
Mandala has a mechanism of property inheritance. Property inheritance is also introduced in our language.

2.3 Implementation methods for instance variables

If we omit arguments to describe instance variables and make procedures within an object executable in parallel, parallel multiple access to instance variables has to be avoided. There are three ways to do this.

(1) Database method

The instance variables are stored in a database managed by a database manager implemented by a perpetual process. Messages to the database manager are used for access to the instance variables. The merit of this method is that more than one access to an instance variable, serialized by using a stream, is possible while a message is being processed. But, it has the defect that it is impossible to access instance variables in the guard part.

(2) Bucket relay method

Moving instance variables from left goals to right goals in a clause is regarded as bucket relay. If no access to an instance variable occurs, the unmodified instance variable is passed to the next goal; if it is modified, the modified value is passed to the next goal. This method has the same advantage as the database method that more than one access to an instance variable is possible while processing a message to the object. It does not have the defect of the database method described above, but if goals must be executed from right to left, the program deadlocks.

(3) Single assignment method

This method is restricted in that it can only modify an instance variable once while processing a message, and the modification does not become effective until processing the next message. It does not have the defects of the other two methods. However, it restricts programming, though we do not think the restriction is very strong.

Since the single assignment method involves the least restrictions on considering methods in objects as KL1 clauses, we chose it even though it does not allow modification of an instance variable more than once when a message is being processed. The next section describes implementation of the object-oriented programming language on KL1 using the single assignment method.

3. Implementation.

3.1 Language specification

We designed an object-oriented programming language on KL1 according to Mandala specifications. There are two basic components: worlds and instances. Each instance is associated with one world. A world represents the knowledge required to solve goals, and an instance is a goal solver for the associated world. When an instance receives messages from other instances, the instance regards the messages as goals to be solved and tries to prove them using a set of clauses stored in the associated world or the superworlds which the associated world inherits.

It is possible to declare clauses, relations between worlds, and instance variables in a world. Each clause is a KL1 clause. We want to add object-oriented features to, and describe methods in, KL1. We introduce the local method in addition to ordinary clauses. By describing a KL1 program as local clauses, the program can be executed without modification. Local clauses can only be called from their own world, but ordinary clauses cannot be called from local clauses.

There are two types of declaration that relate worlds. They are super and part declarations. A super declaration represents a conceptual hierarchical relation between worlds. A lower world can inherit all declarations of its immediately upper worlds and their upper worlds. But if there are several declarations of instance variables with the same name in inherited worlds, the declaration in the lowest world is used. A world may have several immediately upper worlds, that is, multiple inheritance is possible. When a goal is called, clauses are tried from lower world to upper world in turn, and clauses related by multiple inheritance are tried in parallel. In our language, like Mandala, if the guard part of a method is failed, other methods are tried, and if the guard part is successful, that method is selected. Part declaration is used to define a composite instance having lower level instances as its parts. If a world name is specified in a part declaration, its part instance is automatically generated when a composite instance is generated.

Declarations of instance variables specify internal states of an instance. If an initial value is specified, the value is stored in the instance variable when the instance is created. Since we use the single assignment method to implement instance variables, only one modification of an instance variable is allowed during processing of a message. Thus, the modified value of an instance variable cannot be referred to while processing the current message.

The language syntax specified in extended BNF is shown below. The extensions are: (1) "X" indicates a terminal symbol X; (2) { X } indicates arbitrarily many (possibly zero) repeated appearances of X; (3) [X] indicates X or void, i.e., X is optional.

```
<world declaration> ::=
        "world" <world name>
                ["super" <world name> {"," <world name>} ";"]
                ["part"  <part declaration> {"," <part declaration>} ";"]
                ["variable"
                   <variable declaration> {"," <variable declaration>} ";"]
                ["method"
                   <clause> ";" {<clause> ";"}]
                ["local"
                   <clause> ";" {<clause> ";"}]
        "end."

<part declaration> ::= <part name> ["-" <world name>].
<variable declaration> ::= <variable name> [":=" <initial value>].
<clause> ::= <head> | <head> ":-" <body> | <head> ":-" <guard> "\" <body>.
```

Here "\" is used in place of the commit operator "|". An example program is given in Section 4.

Here are the main system predicates in this language.

(1) new(Instance_variable,Variable)
The "new" predicate is used to set a new value of an instance variable. It unifies a variable "Variable" to a value of an instance variable "Instance_variable". Even if a new value of an instance variable is set by the "new" predicate, it is impossible to refer to the value using the "old" predicate while processing the current message. It is possible to refer it while processing the next message.

(2) old(Instance_variable,Variable)
The "old" predicate is used to refer to instance variables, and unifies a variable "Variable" to the current value of an instance variable "Instance_variable".

(3) send(Destination,Message)
The "send" predicate sends a message "Message" to an instance whose name is "Destination". It is possible to send more than one message to (possibly) different instances while a message is being processed.

(4) add_channel(Instance_Channel_Pair)

The "add_channel" predicate enters a list "Instance_Channel_Pair", consisting of a pair of an instance name and its channel, in a channel list. Every instance has a channel list, which contains pairs of instance names and the channels to them like a telephone directory. A channel is a logical variable connecting to an input variable of an instance. If there is no entry of an instance in a channel list, sending of messages to the instance is suspended.

(5) get_channel(Name,Channel)

The "get_channel" predicate fetches the channel "Channel" for the instance name "Name" from the channel list.

3.2 Implementation

We developed a translator of an object-oriented programming language on KL1. The translator transforms programs with the syntax shown above to translated KL1 codes. A translator has the advantage that translated programs can execute fast, but it has the drawback that it is difficult to handle programs as data. We developed a translator first because we consider that efficiency is more important.

The single assignment method is used to implement instance variables, but the method cannot be used to implement channels for message sending to other instances, because it is necessary to send more than one message while processing a message. We use the database method described in the previous section to implement channels. The database manager distributes a message to the channel connected to its destination using its channel list. The database managers are called the instance distributors.

Translated KL1 codes consist of the "create" predicate to create an instance, the "instance" predicate to implement an abstract instance, the procedures to call global methods, and the translated code of methods.

(1) Creation of instances

The "create" predicate is used to create an instance. The predicate is common to all application programs. Thus, it is enough that only one "create" predicate exists in translated code. The definition of the "create" predicate is as follows.

```
create(World_name,Instance_Name,Input,Initial_Values,Initial_Directory) :-
    world_template(World_name,Instance_Parts,Instance_Variables),
    set_initial_values(Initial_Values,Instance_Variables,State),
    create_parts(Instance_Parts,Initial_Directory,Directory),
    instance_distributor(Channel,Directory),
    instance(World_name,Instance_Name,Input,Channel,State).
```

Among the arguments, "World_name" is the name of the world used to create an instance, "Instance_Name" is a name of an instance, "Input" is an input channel for messages to the instance, "Initial_Values" is a list of initial values of instance variables, and "Initial_Directory" is used to set a list of pairs of names of instances and their channels, known from the start, to the channel list. The "world_template" predicate returns the names of the parts and the instance variables given in the declaration of the world. The "set_initial_values" predicate initializes an internal state "State" of the instance using the initial values and declarations of instance variables. The "create_parts" predicate makes instances for parts and the directory to send messages to them. The "instance_distributor" predicate delivers messages to destinations using the directory which is a list of pairs of destination names and channels to them. "Channel" is a channel from the created instance to its instance distributor. The "instance" predicate corresponds to the created instance itself.

(2) The definition of an instance

An instance is implemented by recursive call. In the definition below of the "instance" predicate, it calls itself as the last goal in its definition.

```
instance(World_name,Name,[Goal|Input],Channel,State) :-
  counter_global(Goal,Name,Channel1,State,Update_Transactions,(succ,ok),_),
  update_states(Update_Transactions,State,New_State),
  merge(Channel1,New_Channel,Channel),
  instance(World_name,Name,Input,New_Channel,New_State).
```

Note that definitions of instances are different in each world. The above definition defines an "instance" for the world "counter". If a goal "Goal" comes to an instance of "counter", the global method in the world "counter" is tried first. The "counter_global" predicate calls the global method in "counter". "counter" in "counter_global" originates from the name of the world "counter". For other worlds, "counter" appearing afterwards in translated code should be changed to the name of the world. This predicate essentially executes an input goal. "Channel1" is a channel for sending messages to outer objects through the "instance_distributor" during execution of the goal. Updating the instance variables during execution is reported using a stream through "Update_Transactions". An element of the stream is a pair of names of instance variables and values to be updated. The last two arguments in the "counter_global" predicate are used for multiple inheritance. 'succ' in the last but one argument means that the goal must be successful, and 'ok' means that if a guard part is successful, its body part may be executed. The "update_states" predicate updates the instance variables according to the stream. When the stream ends, all instance variables which are not updated are passed to "New_State". The "merge" predicate merges the channel from the "counter_global" predicate and the new channel of the recursively called "instance" predicate into the channel to the "instance_distributor".

If a null message arrives, an instance sends a null message to its instance distributor through "Channel" and terminates itself.

```
instance(World_name,[],Name,Channel,State) :-
  Channel = [].
```

(3) Procedures to call global methods

Goals to an instance are first executed using global methods. There is a priority of methods in our language, because it has an inheritance mechanism. Local methods of the world associated with the instance have the highest priority, with the restriction that local methods cannot be called by messages from outer objects. The global methods in the world associated with the instance have next highest priority. Global methods in the superworlds have the lowest priority. The lower the superworlds, the higher the priority of the global methods. Methods are separately executed as a guard part and a body part. The "counter_global_guard" predicate executes only a guard part of a method. If a guard part of the method fails, other guard parts of methods are tried in order of the priority of the method. The code for the global method is shown below.

```
counter_global(Goal,Name,Channel,State,Update_Transactions,Result,_) :-
```

```
counter_global_guard(Goal,Name,Channel1,State,Update_Transactions1,Body) |
Result = (succ,Ack),
counter_global_body0(Body,Name,Channel1,Channel,
            State,Update_Transactions1,Update_Transactions,Ack).
```

If the guard part is successful, the body part of the method will be executed. The argument "Body" is used to specify the body part corresponding to the guard part of the method. "Result = (succ,Ack)" returns the result which means that the guard part is successful, and "Ack" is a variable for acknowledgment. If the "counter_global" predicate is called from the "instance" predicate, "Ack" is set 'ok' in the "instance" predicate. The "counter_global_body0" predicate checks the value of "Ack".

Next, we describe the code for inheritance. The translated codes for multiple inheritance, i.e., that the world "counter" inherits two superworlds "super1" and "super2", are used as an example. In this implementation, control of execution to the superworld is passed by "otherwise" when the guard parts in the current world fail. "otherwise" is a special system predicate in KL1, and its function is that a goal "otherwise" succeeds when the guard part of all other clauses whose predicate have the same name have failed. It is easy to implement the inheritance mechanism from the lower worlds to the superworlds using "otherwise". But it is not easy to implement multiple inheritance. The inheritance rule for multiple inheritance in our language is that the methods of multiple superworlds are tried in parallel. The method whose guard part first succeeds is selected; other methods are abandoned even if their guard parts succeed. Multiple inheritance is implemented using the metacall. The syntax of the metacall is "call(Goal,Result,Control)", with three arguments, "Goal", "Result" and "Control". Its specification is as follows [Miyazaki 85] (This specification is tentative in KL1):

Wait until "Goal" becomes a non-variable and call it. If it succeeds, "Result" is bound to 'success'. If it fails, "Result" is bound to 'fail'. If "Control" is bound to 'stop' while solving "Goal", "Result" is bound to 'stopped' and the execution of the metacall terminates, but all bindings made by "Goal" remain.

A metacall is used to avoid wasteful execution, because there is a possibility that all the guard parts of the methods that can possibly be used to execute the goal are tried. Let us consider the counter program example. If the "counter_global_guard" goal in the current world fails, the two metacalls in the following clause first try to execute the guard part of the global methods of the two superworlds. Their results are returned to "Result1" and "Result2". If the guard part of a method is successful, the result is "(succ,Ack)" in which "Ack" is a variable for acknowledgment, otherwise, the result is "(fail,no)". The results are checked by the "commit_world" predicate.

```
counter_global(Goal,Name,Channel,State,Update_Transactions,Result,Cont) :-
  otherwise |
  call(
   super1_global(Goal,Name,Channel,
           State,Update_Transactions,Result1,Cont1),_,Cont),
  call(
   super2_global(Goal,Name,Channel,
           State,Update_Transactions,Result2,Cont2),_,Cont),
  commit_worlds(Result,Cont,Result1,Result2,Cont1,Cont2).

commit_world(Result,Cont,(succ,Ack),X2,_,Cont2) :-
  X2=(_,no),Result=(succ,Ack),Cont2=stop.
commit_world(Result,Cont,X1,(succ,Ack),Cont1,_) :-
  X1=(_,no),Result=(succ,Ack),Cont2=stop.
```

```
commit_world(Result,_,(fail,no),(fail,no),Cont1,Cont2) :-
  Result=(fail,no),Cont1=stop,Cont2=stop.
commit_world(Result,stop,X1,X2,Cont1,Cont2) :-
  X1=(_,no),X2=(_,no),Cont1=stop,Cont2=stop.
```

The "commit_world" predicate selects the method which first reported the success, whose form is "(succ,Ack)". If successes are reported from the two superworlds, the "commit_world" predicate selects either of the two and stops execution of other methods by instantiating 'stop' to the control variable of the metacall. The result may be also reported to the lower worlds by the unification of "Result" and "(succ,Ack)", because the possibility that this world is a superworld of the lower worlds exists. If the lower world is on the way, the result is tested like this world. Finally, the result is passed to the caller of the goal, and the acknowledgment is set to 'ok'. The caller is the "counter_global" predicate in the "instance" predicate or the "call_method" predicate described below. The acknowledgment "Ack" is used to decide which body part can execute in the "counter_global_body0" predicate. If the acknowledgment is 'ok', the body part is selected; if it is 'no', the method is not selected. If the results of the two superworlds fail, the third definition of "commit_world" clauses is selected and a fail is reported to its lower worlds (Result = (fail,no)). If another method is selected, the control variable is instantiated to 'stop' and tries to cancel the execution of the methods of the super worlds.

Execution of the body part waits until the acknowledgment is returned. The "counter_global_body0" predicate checks the acknowledgment. If the acknowledgment is 'ok', the body part is executed. The "counter_global_body" is called to execute the body part of the method.

```
counter_global_body0(Body,Name,Channel1,Channel,
              State,Update_Transactions1,Update_Transactions,ok) :-
  merge(Update_Transactions1,Update_Transactions2,Update_Transactions),
  merge(Channel1,Channel2,Channel),
  counter_global_body(Body,Name,Channel2,State,Update_Transactions2).
counter_global_body0(_,_,_,_,_,_,_,no).
```

(4) Translated code of a method

Now let us take a look at the compiled code of methods. A method is separately translated to a guard part and a body part. We translate the method below contained in the world "counter".

```
up(X) :- X > 0 \ add1(X,X1), new(state1,X1) ;
```

There is a goal "X > 0" in the guard part. In the body, "add1" is a user-defined predicate, and predicate "new" is a system predicate. The translated code of the guard part of the method is as follows:

```
counter_global_guard(up(X),Name,Channel,State,Update_Transactions,Body) :-
  X > 0 | Body = body1(X), Update_Transactions = [], Channel = [].
```

"body1(X)" is constructed for the identification to link it to its body part. The identification must be unique in every method. The variables in a guard part are passed to the body part as the arguments in the identification. "Update_Transactions" and "Channel" are instantiated null, because no state is updated and no message is sent in the guard part of this method. The body part of the above method is as follows:

```
counter_global_body(body1(X),Name,Channel,State,Update_Transactions) :-
  call_method(add1(X,X1),Name,Channel,State,Update_Transactions1),
```

```
  new(state1,X1,Update_Transactions2),
  merge(Update_Transactions1,Update_Transactions2,Update_Transactions).
```

User-defined predicate "add1" is translated to the goal for the method call interface, the "call_method" predicate. Because states may be updated and messages may be sent while executing the "add1" predicate, the stream "Update_Transactions1" for updated states and the stream "Channel" for the channel to the instance distributor must be passed to the "call_method" predicate. The "merge" predicate is used to merge the streams from the "new" predicate and the "call_method" predicate to "Update_Transactions".

The "call_method" predicate first tries to evaluate the guard part of the local methods in the current world. If the guard part is successful, its body part is selected, otherwise, the global methods are tried.

```
  call_method(Goal,Name,Channel,State,Update_Transactions) :-
    counter_local_guard(Goal,Name,Body) |
    Update_Transactions = [], Channel = [],
    counter_local_body(Body,Name).
  call_method(Goal,Name,Channel,State,Update_Transactions) :-
    otherwise |
    counter_global(Goal,Name,Channel,State,Update_Transactions,(succ,ok),_).
```

4. Example

The following example is a very simple program environment. Four worlds are declared: "distributor", "object", "terminal_manager", and "counter". The distributor has two important functions. One is creation of instances and the other is distribution of messages from one instance to another. If the distributor creates an instance, it registers the name and the channel of the instance in its channel list. The world "object" is the top-level world, and has a common method among instances. This is the "send_to" method for sending a message "Message" to a destination "Destination" through the distributor. The world "terminal_manager" has a method to display a message to a terminal. The methods in the world "counter" are similar to the KL1 clauses of the counter shown in Section 2. The world "counter" has one superworld, "object", and an instance variable, "state". The "add1" method is a local method. In the top-level goal "test", the "create" predicate is called to create a distributor, and the message to create a counter and a terminal manager as well as the messages to the counter are sent to the distributor.

```
world distributor
        method
          send_to(Name,Message) :- send(Name,Message) ;
          create(Class,Instance) :-
                get_channel(distributor,Dis),
                create(Class,Instance,Mes,[],[(distributor,Dis)]),
                add_channel([(Instance,Mes)]) ;
        end.

world object
        method
          send_to(Destination,Message) :-
                send(distributor,send_to(Destination,Message)) ;
        end.

world terminal_manager
        super
          object ;
```

```
            method
              display(X) :- write(X) ;
            end.

    world counter
            super
              object ;
            variable
              state ;
            method
              set(X) :- new(state,X) ;
              up :- old(state,X), add1(X,X1), new(state,X1) ;
              show :- old(state,X), send_to(terminal_manager,[display(X)]);
            local
              add1(X,X1) :- X1 := X + 1 ;
            end.

    test :-
            Mes=[create(counter,ct1),
                    create(terminal_manager,terminal_manager),
                    send_to(ct1,[set(1),show])|Dis],
            create(distributor,distributor,Mes,[],[(distributor,Dis)]).
```

5. Discussion

We described an object-oriented language on KL1. This language is similar to Vulcan with some important differences.

(1) Instance variable

We use the single assignment method to implement instance variables. But Vulcan uses a method corresponding to the bucket relay method to implement instance variables. We selected the single assignment method because it imposes fewer restrictions on considering methods in objects as KL1 clauses than does the bucket relay method.

(2) Description of methods

The methods in our language are described as KL1 clauses. We selected KL1 for description of the methods because we believe that KL1 is a powerful language. The methods of Vulcan are not taken from Concurrent Prolog, but are rather similar to Smalltalk. Vulcan uses Concurrent Prolog as an implementation language, which means it has lost the parallel control mechanism of Concurrent Prolog. It is difficult for programmers to control programs in parallel.

(3) Implementation of inheritance

Inheritance in Vulcan is implemented by the copying method that involves creating subclasses with source copies of all methods inherited from their superclasses, or the delegation method in which that superclasses are regarded as parts, and methods in superclasses are called by sending messages to them. In contrast, inheritance in our language is implemented by calling methods in superworlds. We think that the copying method is not realistic, because the number of copied methods become enormous when we write the operating system. We cannot use the delegation method either, because methods in superworlds cannot be called in guard parts.

We have implemented an object-oriented program language on KL1. But the current implementation is naive, so there is room for significant improvement of performance. Inheritance is one of the main overheads. The overheads may be decreased by calling the necessary method directly. Another main overhead is execution of the "update_states" predicate. It may be possible to reduce the overheads by specifying instance variables explicitly as arguments in translated code. For that purpose, it is necessary to determine which instance variables are updated and which are not in each method by analyzing programs.

We should evaluate this language from several standpoints, not just performance. One is usefulness. We intend to describe various applications to investigate the influence of the restriction to single assignment on programmers. Another issue is whether it is possible to check the violation of single assignment in programs. The violations are bugs, but the check is not easy. We also need to investigate a language based on the database method, because we think that the database method is a good alternative. It may even turn out to be better than the single assignment method, if it is possible to produce translated code while avoiding its defect, i.e., the possibility of accessing instance variables in guard parts.

Acknowledgment

This research is based on discussions in the Mandala group at ICOT, and we wish to express our thanks to the members, Susumu Kunifuji (Fujitsu Limited), Kazunori Ueda, Tosihiko Miyazaki, Akira Okumura (ICOT) and Hideki Yasukawa (Matusita Electric). We would also like to thank Kazuhiro Fuchi, Director of ICOT Research Center, who provided us with the opportunity of doing this research in the Fifth Generation Computer Systems Project.

References

[Clark 83] K.L.Clark, S.Gregory: PARLOG: A Parallel Logic Programming Language, Research Report DOC 83/5, May (1983).

[Furukawa 83] K.Furukawa, A.Takeuchi, S.Kunifuji: Mandala: A Concurrent Prolog Based Knowledge Programming Language/System, ICOT TR-29, (1983).

[Furukawa 84] K.Furukawa, A.Takeuchi, S.Kunifuji, H.Yasukawa, M.Ohki, K.Ueda: Mandala: A Logic based Knowledge Programming System, Proc. of FGCS'84, (1984).

[Goldberg 83] A.Goldberg, D.Robson: Smalltalk-80, The language and Its Implementation, Addison Wesley, (1983).

[ICOT 85] ICOT KL1 Group: Explanation material for Kernel Language 1, ICOT internal report, in Japanese (1985).

[Kahn 86] K.Kahn, E.D.Tribble, M.S.Miller and D.G.Bobrow: Vulcan:Logical Concurrent Objects, Knowledge Systems Area, Intelligent System Laboratory, Xerox Palo Alto Research Center, (1986).

[Miyazaki 85] T.Miyazaki: Guarded Horn Clause Compiler User's Guide, ICOT, (1985).

[Shapiro 83a] E.Shapiro: A Subset of Concurrent Prolog and Its Interpreter, ICOT Technical Report TR-003 (1983).

[Shapiro 83b] E.Shapiro, A.Takeuchi: Object Oriented Programming in Concurrent Prolog, New Generation Computing Vol.1, No.1 (1983)

[Tanaka 86] J.Tanaka, K.Ueda, T.Miyazaki, A.Takeuchi,Y.Matsumoto and K.Furukawa: Guarded Horn Clauses and Experiences with Parallel Logic Programming, Proc. of Fall Joint Computer Conference, (1986).

[Tokoro 84] M.Tokoro, Y.Ishikawa: An Object-oriented Approach to Knowledge Systems, Proc. of FGCS'84, (1984).

[Ueda 86] K.Ueda: Guarded Horn Clauses: A parallel Logic Programming Language with the Concept of a Guard, ICOT Technical Report TR-208 (1985).

[Yonezawa 86] A.Yonezawa, E.Shibayama, T.Tanaka, Y.Honda: Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1, in Object Oriented Concurrent Programming, edited by A.Yonezawa and M.Tokoro, MIT Press, (1986).