ICOT Technical Report: TR-335

TR-335

A ´UM

—Parallel Object-Orieneted Language upon KL1 —

Kaoru Yoshida and Takashi Chikayama

January, 1988

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

$A'UM$

– KL1 上の並列オブジェクト指向言語 –

吉田 かおる ， 近山 隆

(財) 新世代コンピュータ技術開発機構

概要

本論文は並列オブジェクト指向言語 $A'UM$ について述べている。

現在 ICOT では並列推論マシン PIM を設計および開発しており、その核言語 KL1 の上位ユーザ言語として $A'UM$ は設計されている。

$A'UM$ の目的は、大規模なシステムおよび応用プログラムの開発を容易にするための高い記述力を提供することであり、並列推論マシン PIM のオペレーティングシステム PIMOS が第一の記述対象である。

$A'UM$ の特徴は、副作用を持たない並列オブジェクトに基づいた、その高い抽象化機構にある。なかでも、言語下でのストリームの自動併合、名前によるオブジェクトの連想管理、マクロ展開に基づく文法、そしてクラス継承によるモジュール化支援の機能は最も特徴的であり、これらについて詳細に述べる。

$A'UM$

– Parallel Object-Orieneted Language upon KL1 –

Kaoru Yoshida and Takashi Chikayama

Institute for New Generation Computer Technology (ICOT)

4-28, Mita 1-chome, Minato-ku, Tokyo 108 JAPAN

Abstract

This paper describes a parallel object-oriented programming language, $A'UM$.

$A'UM$ has been designed as a user's language upon KL1 which is the kernel language of the parallel inference machine, PIM, being developed at ICOT. The goal of $A'UM$ is to provide high description power for ease of writing large-scale parallel systems and applications, including the operating system, PIMOS, for PIM.

$A'UM$ is characterized with its high level abstractions based on pure parallel objects, of which most characteristic are implicit stream merging, object-name association, macro expansion based grammer, and modular programming support by class inheritance, that are described in detail.

1 Introduction

In the fifth generation computer systems project at ICOT, we have been designing and developing a parallel inference machine, PIM [5] and its operating system, PIMOS.

In general, the larger a problem, the more difficult to solve. For a large-scale problem such as an operating system, the entire function seems to be very complicated, but in most cases, it is the result of piling up many simple functions and basic concepts. The secret to develop a large-scale reliable system is to divide the system into modules, each of which has a simple function, and to make each module sound. The more information and control flow is localized, the easier it is to design and test.

For parallel systems, it is much harder to develop than sequential systems. Most of the difficulties in debugging a parallel program is to reconstruct the causal chain from actually happening events, that is, to analyze error reasons from phenomena when it results in failure. Although events are due to a single causal chain, when the whole events from root to leaf are actually happening, they seem to be happening independently at random. Namely, flatness of events makes analysis difficult. Therefore, hierarchical and modular design and test are indispensable to develop parallel large-scale systems.

For this purpose, the object-oriented paradigm is most effective. We have developed an object-oriented logic programming and operating system, SIMPOS [2], for the sequential inference machines, PSI and PSI-II [3,4]. SIMPOS is written in the language, ESP [1], which introduced the notion of object-orientation, which is to encapsulate data and operations, into a logic programming language. Through this experience, we have learned that the object-orientation contributes greatly to both the design and test of such a large-scale system.

The notion of object-orientation is natural to model a large-scale system and represent the programmer's intentions. Designing a system is directly writing programs without any intermediate process. Mainly because of its uniform and dynamic control by method call, and modularization support by multiple class inheritance and method combination, the entire system is made simple and compact, that shortens the development period at last. What forwards shortening the development period is not only the programming language itself, but also its high level programming and debugging environment.

As a result, for SIMPOS, it took few years to develop the entire system, though it is furnished with a rich set of functions.

We would like to make good and best use of the benefits, obtained by object-orientation, in parallel systems, too.

This paper describes a parallel object-oriented language, $\mathcal{A}'\mathcal{UM}$. $\mathcal{A}'\mathcal{UM}$ has been designed upon KL1, which is the kernel language of the parallel inference machine, PIM, for ease of writing large-scale systems and applications, mainly as a description language of the operating system, PIMOS.

The organization of the paper is as follows: Firstly, the object-oriented programming style in KL1, which originated $\mathcal{A}'\mathcal{UM}$, is shown. Secondly, after the main features of $\mathcal{A}'\mathcal{UM}$ are outlined, their detail are described with some examples. In addition, the implementation of $\mathcal{A}'\mathcal{UM}$ onto KL1 are described. Finally, $\mathcal{A}'\mathcal{UM}$ is compared with other related works.

2 Object-Oriented Programming in KL1

2.1 GHC and KL1

The kernel language of PIM is a committed-choice parallel logic programming language, called KL1. KL1 is an extention of FGHC which is a subset of GHC [6].

The committed-choice parallel logic programming language family has been paid special attention. That is mainly because of their simple and atomic mechanism for communication and synchronization. Since a parallel construct is embedded at the atomic level, it is possible to solve the entire problem uniformly from top to bottom.

Above all, GHC [6] is simplest in this family.

A GHC procedure is a set of guarded Horn clauses of the following form:

$$H \ :- \ G_1,...,G_m \mid B_1,...,B_n. \quad (m>0, n>0)$$

where H, G_i's and B_i's are atomic formulas. H is called a *clause head*, G_i's *guard goals* and B_i's *body goals* respectively. The operator '|' is called a *commitment operator*, the left part before the operator a guard and the right part a body respectively.

Roughly speaking, the execution of a GHC procedure is explained as follows: When a procedure is invoked, all clauses defining the procedure can run in parallel, keeping the following suspension and commitment rules:

Suspension

- Unification invoked directly or indirectly in the guard of a clause C called by a goal G cannot instantiate the goal G.
- Unification invoked directly or indirectly in the body of a clause C cannot instantiate the guard of C until that clause is selected for commitment.

Commitment

If some of the clauses succeed in the execution of the guard part, one and the only one of them is nondeterministically selected. The selected clause continues execution of the body.

GHC realizes synchronization only with the guard construct. Unification in the guard is not allowed to instantiate the invoking goal, so it requires no multiple

environments for its execution. Such a simple mechanism of the guard is desirable for the architecture. It makes the practical implementation feasible, especially considering its implementation in a distributed environment.

FGHC is further given a limitation that only system-defined (or built-in) predicates can be invoked in the guard but no user-defined predicates. Since FGHC does not require nested guard control, it is simpler and more suitable for the hardware implementation of PIM.

2.2 Object-Oriented Programming Style

The notion of *object* has been spread widely. Basically, an object is an entity to encapsulate internal states and a set of operations [11]. Objects which have only this feature as capsules are called *static objects*. In contrast, by integrating the notion of process, which is an execution unit that can run in parallel, with that of capsule, another kind of objects, called *dynamic objects*, have been introduced. Each dynamic object has an independent execution environment, and communicates with others by message passing via communication media such as message streams and message boxes. Several languages to realize dynamic objects have been developed [12,13,14,15].

As mentioned earlier, the committed-choice parallel logic programming language family, including KL1, provides the basic framework for synchronization and communication at the base, which is required to realize dynamic objects.

Shapiro and Takeuchi [8] shows that CP [7] supports object-oriented programming style in the framework of *perpetual process* using stream communication, which can be applied to GHC and KL1, too.

A perpetual process is a causal chain of tail-recursive goals, regarding each goal as a process state at some stage. A clause waits for some particular event to hold. After commitment, it takes behaviors corresponding to the event, such as sending messages or modifying its internal states, and invokes an identical goal for the next stage.

Communication is performed through a message stream, which is recognized as an object from the outside. A message stream is represented using a list construct, of which the car part means a message and the cdr part a succeeding stream respectively.

For example, a stack is defined in the object-oriented programming style in KL1 as follows:

Example 1 *Stack in KL1*

```
stack(Stack) :- true |
    bottom(Bottom), stack(Stack, Bottom).

stack([push(X)|S], Top) :- true |
    element(Elmnt, X, Top), stack(S, Elmnt).
stack([pop(X)|S], Top) :- true |
    Top = [get(X, Y)], stack(S, Y).
stack([read(X)|S], Top) :- true |
    Top = [get(X, [])|Top1], stack(S, Top1).
stack([], Top) :- true | Top = [].

element([get(X, Y)|S], Data, Next) :- true |
    X = Data, merge(Y, Next1, Next),
    element(S, Data, Next1).
element([], _, Next) :- true | Next = [].

bottom([get(X, Y)|S]) :- true |
    X = '$error(end_of_stack)', bottom(S).
bottom([]) :- true | true.

merge([X|Xs], Ys, Zs) :- true |
    Zs = [X|Zs1], merge(Xs, Ys, Zs1).
merge(Xs, [Y|Ys], Zs) :- true |
    Zs = [Y|Zs1], merge(Xs, Ys, Zs1).
merge([], Ys, Zs) :- true | Zs = Ys.
merge(Xs, [], Zs) :- true | Zs = Xs.
```

This program can be read as follows: A `stack` object which holds a `bottom` object as the top element is created first. At each stage, the stack may receive a message, either `push/1` or `pop/1`, until it is closed with `[]`. For message `pop/1`, the stack sends a message, `get/2`, to the top to get its data and next element, and recurs with the next element as a new top element. For message `push/1`, it creates an `element` object that should hold the given data and the current top, and recurs with the new element as a top. When it is closed with `[]`, it terminates closing the top with `[]`.

The top element is an internal state to the stack, and the data and next element are those to the element. These internal states are represented using local variables; `Top`, `Data` and `Next`, each of which appears in the fixed position, and their new variables are carried by the tail goal for the next stage.

Another noticeable point is the way in which the element object passes its next object in return to the `get/2` message: the next object is not passed directly, but is merged with the output parameter, `Y`. Then, the element object recurs with the output variable, `Next1`, of the merger.

In comparison with other object-oriented languages and their implementations, one of the most characteristic features with this program is that the semantics of *updating internal states* is logically pure, that is, side-effect-free. A chain of logical variables placed at the same position is the history of an internal state.

As easily seen from such a small example, however, even with this programming style, programs are too primitive and verbose to develop a large-scale system, since KL1 is positioned as the kernel language. As a result, most of the deadlocks are brought by stream breaking attributing to tiny bugs such as misnaming and misplacing variables, rather than by algorithmic ones.

To represent the programmer's intention more directly and concisely and higher level abstraction, which makes the program semantics vivid, is needed. Also programming and debugging environment should be provided at this level or higher.

3 Parallel Object-Oriented Language $\mathcal{A}'\mathcal{UM}$

We propose a parallel object-oriented programming language, called $\mathcal{A}'\mathcal{UM}$ [1].

$\mathcal{A}'\mathcal{UM}$ has been designed as a user's language which is compiled into KL1, for ease of writing large-scale systems and applications. $\mathcal{A}'\mathcal{UM}$ is independent of KL1, and KL1 programs cannot be contained together in $\mathcal{A}'\mathcal{UM}$ programs.

This section summarizes major characteristic features of $\mathcal{A}'\mathcal{UM}$.

Firstly, $\mathcal{A}'\mathcal{UM}$ objects are characterized as follows:

Pure Parallel Object $\mathcal{A}'\mathcal{UM}$ is a pure parallel object-oriented language.

Each $\mathcal{A}'\mathcal{UM}$ object is a perpetual process which belongs to some class: it repeats the cycle of receiving a message, sending messages to itself or other objects in response. Sending messages to objects is the basic mechanism to execute an $\mathcal{A}'\mathcal{UM}$ program.

Name Association In $\mathcal{A}'\mathcal{UM}$, each object is associated with a name. Updating an object is performed in the side-effect-free foundation: a new version of object is created and associated with the same name.

Stream Merging The external interface to an object is a directional stream. When an object is referred to by more than one object, a merger is implicitly inserted to split a stream to the referred object.

In addition, $\mathcal{A}'\mathcal{UM}$ provides the following syntactic and modularization support:

Macro Expansion An $\mathcal{A}'\mathcal{UM}$ program is composed of macro expressions, each of which is evaluated to an object with a sequence of abstract $\mathcal{A}'\mathcal{UM}$ instructions expanded. With this feature, $\mathcal{A}'\mathcal{UM}$ programs can be written compactly.

Class Inheritance An $\mathcal{A}'\mathcal{UM}$ class can inherit multiple classes. Class inheritance only expands the method space applicable to an instance, but does not bring forth any other instances of the super classes.

The above stack example can be written in $\mathcal{A}'\mathcal{UM}$ as follows:

[1] $\mathcal{A}'\mathcal{UM}$ is a Japanese word, derived from a Sanskrit "ahum", which consists of $\mathcal{A}$ and $\mathcal{UM}$ and implies the beginning and the end, an open voice and a close voice, and expiration and inspiration.

Example 2 *Stack in* $\mathcal{A}'\mathcal{UM}$

```
class  stack.
    slot  top.
  :initiate -> #bottom :new(^B), !top = B.
  :push(^X) -> #element :new(^E),
               !top = E :set(X, !top).
  :pop(X) ->   !top :get(^X, ^Y), !top = Y.
  :read(X) ->  !top :get(^X, ^Y).
end.

class  element.
    slot  data, next.
  :set(^X, ^Y) -> !data = X, !next = Y.
  :get(!data, !next) -> .
end.

class  bottom.
  :get('$error'(end_of_stack), Y) -> .
end.
```

A stack is created by sending a message :new/1 as follows:

```
#stack :new(^S0),
^S1 = S0 :push(1) :read(^A) :pop(^B) :pop(^C)
```

4 Class and Object

4.1 Class

<class definition> ::=
    class *<class name>*
        <superclass definition>
        <slot definition>
        { *<method definition>* }
    end '.'

<superclass definition> ::=
    super *<superclass name>* { ',' *<superclass name>* } '.'

<slot definition> ::=
    slot *<slot name>* { ',' *<slot name>* } '.'

Each $\mathcal{A}'\mathcal{UM}$ object is an instance which belongs to some class.

A class is a module which defines a set of attributes and functions of instance objects. Classes are treated as immutable objects, which will be mentioned later, but belongs to no other classes: there is no notion of meta class.

Each class can inherit multiple classes. By inheriting a class, a set of attributes and functions applicable for an instance object is expanded, but no other instances of super classes are created.

4.2 Object

Each $\mathcal{A}'\mathcal{UM}$ object is a perpetual process which is represented with the following attributes:

Original Class which the object belongs to and is created from.

For an instance object, the original class is fixed through life.

Current Class which defines a method which is applied for each received message.

For an instance object, the current class is variable depending on the received message. When a method of some class is applied, the object is said to be *under* the class. At initiation or every time the object recurs, the current class is set to the original class.

External Interface Streams through which messages are sent to the object.

One or more interface streams can be offered, each of which is assigned a different priority and priority-merged into the internal input stream. From the outside, a given interface stream is regarded as the target object itself.

Internal Input Stream from which the object receives messages. The internal input stream is accessible with the name `$self`.

Slots each of which is associated with a slot name given in the `slot` definition.

Any slot is visible only within the class which defines it. Even if another slot is defined with the same slot name in either super or inferior classes, it is a different slot.

When an object is created, a global stream named `$system` is given, through which messages are raised to the underlying operating system. In the conceptual model, this global stream may be treated as one of the slots.

Supers which are the super classes that the original class inherits directly and indirectly. The inheritance tree is constructed from the `super` definition in the left-first depth-first order.

For an instance object, the supers are fixed through its life.

Delegates which are super classes positioning later than the current class on the inheritance tree.

A delegate class next to the current class is accessible with the name `$super`.

As well as the current class, the delegates are variable depending on the received message. At initiation or every time the object recurs, the delegates are set to the supers.

Example 3 *Object Attributes*

Given the following class definitions:

```
class c21
    super c11, c12.
    slot  s.
  :ma -> !s :ma.
end.

class c3
    super c21, c22.
    slot  s.
  :mb -> !s :mb.
end.
```

An instance of class c3 is created, which some messages are sent as follows:

```
#c3 :new(~C3),
C3 :ma :mb
```

1. For an instance of class `c3`, the original class is `c3`, and the supers are `c21` → `c11` → `c12` → `c22`, both of which are fixed through life.

2. The slot `s` in class `c21` is independent of that in class `c3`.

3. During execution, the current class and delegates are transitional depending on received messages.

   When executing a method for message `:mb`, the current class is `c3`, and the delegates are `c21` → `c11` → `c12` → `c22`.

   When executing a method for message `:ma`, the current class is `c21`, and the delegates are `c11` → `c12` → `c22`.

4.3 Object Life

The life of an $\mathcal{A}'\mathcal{UM}$ object is drawn as follows:

Creation When a message `:new/1` (or `:new_with_priority/2`) is sent to a class, an instance object of the class is created, and a message `:initiate` is sent to the object. An interface stream (or a set of interface streams) to the object, which positions after the `:initiate` message, is returned to the `:new/1` message.

Initiation Whenever an instance is created, it is implicitly sent a message `:initiate`. The programmer can overwrite the default method for the `:initiate` message, which is predefined as follows:

```
:initiate -> .
```

Generation Including internal states such as `$self`, slots and `$system`, any object is associated with a name. Updating an object is not giving any side effect on it, but creating a version of object and associating it with the name.

Each version of object is called *a generation*, and changing the name association is called *generation descending*.

Cycle After receiving an external message, an object behaves descending one generation to another. A sequence of generations derived from receiving one external message is called a *cycle*. A script of the cycle for one external message is called a *method*.

Termination At the end of a cycle, the object either recurs to the next cycle or terminates its life. The former is syntactically specified with '.', the latter with '..'.

When to terminate can be defined freely by the programmer: it may be when the internal input stream is closed, or at any time.

When the internal input stream is closed, the default closing method is defined to close all the slot objects as follows:

```
:: -> $slots :: ..
```

The programmer can overwrite this closing method.

4.4 Mutable and Immutable Objects

A'UM objects are categorized into two; *mutable objects* and *immutable objects*, depending on whether they have changeable internal states or not.

Class objects are immutable. Some primitive classes such as `true`, `false`, `integer`, `vector` and `string`, are also immutable.

Both mutable and immutable objects are treated uniformly in their message passing. For example, a stack element has two slots, of which slot `data` is immutable and slot `next` mutable. There is nothing different in sending messages or making accesses to either of them.

5 Basic Notions

5.1 Name Association

A'UM objects are associated with names. Names are categorized into two: *temporary names* and *permanent names*, depending to their name scope, that is, their life time.

Permanent Names The name scope of a permanent name is one generation.

Among permanent names are system-defined names, such as `$self`, `$system` and `$super`, and user-defined slot names.

Temporary Names The name scope of a temporary name is one cycle.

Among temporary names are variables, including parameter variables which are carried in messages, and temporary variables which are generated in the cycle.

5.2 Stream Merging

Given a stream, if it accepts any message applicable to an object, the stream can be regarded as the object itself from the outside.

As shown in Section 2, when an identical object is referred to by more than one object, it requires for a stream to the referred object to be split into two, so that each referring object can hold one stream through which it can send messages independently. In other words, branches are merged into the stem, that is, the internal input stream of the target object.

In *A'UM* , non-determinicity exists only in the stream merger. Stream merging has no logical meaning other than sending messages to the target object. A stream merger is inserted in the following two cases:

- when more than one input terminal of a temporary name (or variable) occurs.
- every time a permanent name except `self` occurs.

5.2.1 Variable Mode

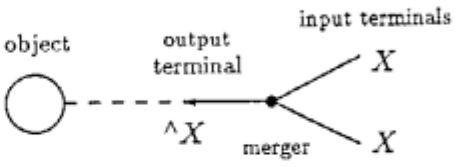
Each stream has a direction which is toward the target object. To specify the stream direction, each variable occurrence has its terminal *mode*, which is either *input* or *output*.

- Variables have only one occurrence with '^', called an output terminal, and one or more occurrences without '^', called input terminals.
- An object is somewhere ahead of the output terminal.
- A stream is connected to the output terminal.
- Messages can be sent to the input terminals.
- The messages sent to the input terminals are merged and sent to the target object ahead of the output terminal.

Example 4 *Variable Mode*

```
:consult(^A, ^B, ^C) ->
    A :try(X), B :try(X), C :select(^X).
```

Variables `A`, `B`, `C` and two `X`s are input terminals, while `^A`, `^B`, `^C` and `^X` are output terminals. The two `X`s are merged into `^X`.



5.2.2 Slot Access

Referring When a slot is referred to, it opens a stream to be returned and a new generation slot, both of which are merged into the current generation slot.

Updating Slots are updated when they are specified as the destination of stream connection and message sending.

Updating a slot is changing the name association in a side-effect-free manner: the specified new value is associated with the slot name, and the old value is closed.

5.3 Macro Expansion

An $\mathcal{A}'\mathcal{UM}$ program is composed of *macro expressions*, each of which is evaluated to be an object with a sequence of abstract $\mathcal{A}'\mathcal{UM}$ instructions expanded. With this feature, $\mathcal{A}'\mathcal{UM}$ programs can be written compactly and clearly.

For example, a message sending expression is evaluated to a new object after the message is sent.

Example 5 *Macro Expansion*

```
:create(A0:initialize(^IL), IL) ->
    #a :new(^A0).
```

is equivalent to:

```
:create(A, IL) ->
    #a :new(^A0),
    ^A = A0 :initialize(^IL).
```

6 Method

<method definition> ::=
    <message> '->' *<cycle>* *<terminator>*

<cycle> ::=
    <generation> { ',' *<generation>* }

A script of the cycle for one external message is called a *method*. One cycle consists of generations, for each of which one of the following four behaviors can be defined: stream connection, message sending, message delegation, and volatile object creation.

6.1 Stream Connection

<connection> ::=
    <output terminal> '=' *<input terminal>*

<output terminal> ::=
    <output variable> | *<slot>*

<input terminal> ::=
    <input variable> | *<expression>*

Through a stream, messages are sent from its input terminal to its output terminal, that is, the input terminal is a message source, and the output terminal is a destination.

An expression specified as the source, is evaluated to be an input terminal with a sequence of abstract instructions expanded. The input terminal is connected to the output terminal specified as follows:

<output variable> `^Y = !top`

Slot `top` is referred to, to which output terminal `Y` is connected.

<slot> `!top = Y`

An input terminal, `Y`, is connected to the new generation of slot `top`, that is, slot `top` is updated with `Y`.

Thus, the semantics of *<slot>* is different depending on which side it appears: referring slot on the right and updating slot on the left.

6.2 Message Sending

<message sending> ::=
    <input entry> { *<message>* } *<last>*

<input entry> ::=
    {} | *<input variable>* | *<slot>* | *<system>*

<last> ::=
    { *<message>* | '::' }

A message sending expression is evaluated to be a new generation object after the message is sent. By repeating this evaluation, a sequence of message can be sent to an identical object.

An message sending expression can be specified wherever expressions are allowed, for example, as a parameter of another message or as the source of stream connection.

The semantics of a message sending expression is variable depending on the input entry as follows:

{} *(default)* `:m(P)`

prepends a `m(P)` message to the current self, that is, inserts it before the next external message.

<input variable> `E :set(X,!top)`

appends a `set(X,!top)` message to the input variable `E`.

<slot> `!top :get(^X,^Y)`

appends a `get(^X,^Y)` message to the `top` slot and updates the slot with a new stream following the message.

<system> `$system :m(X)`

raises a `m(X)` message to the system stream.

6.3 Message Delegation

<message delegation> ::=
    <delegate class> '<-' { *<message>* } *<last>*

<delegate class> ::=
    <direct super> | *<class>*

When a class inherits one or more super classes, a sequence of messages can be *delegated* to any of the super classes.

In $\mathcal{A}'\mathcal{UM}$, class is an index to categorize the method space applicable for an instance with. Class inheritance expands the method space applicable to an object, but does not bring forth any other instances of super classes. Therefore, delegating a message to some class is asking the object itself to apply a method defined in the class, that is, to receive the message *under* the class.

Message delegation is performed by sending an indirect message, which encloses the target message and the delegate class which should receive the target message, to the object itself. The delegate class is specified in the following two ways:

<direct super> `:m(^X) -> $super <- :mm(X).`

With `$super` specified, messages are delegated to the direct super.

<class> `:m(^X) -> #some_super <- :mm(X).`

Messages can be delegated to a certain class on the inheritance tree by specifying its class name.

When an object receives a delegation message, `delegate(Class,Message)`, it checks the delegate class `Class`. If it is equal to the current class, the object sends the target message `Message` under the current class. Otherwise, the object further delegates the delegation message to its direct super.

6.3.1 Default Message

```
:$default -> $super <- :$default.
```

When a method to the received message is not defined in a class, the received message is delegated to the direct super.

6.4 Volatile Object Creation

<volatile object creation> ::=
    <volatile immutable object definition>
    <volatile mutable object definition>

<volatile immutable object definition> ::=
    <object source>
        '[' { *<method definition>* } ']'

<volatile mutable object definition> ::=
    <object source>
        '{' *<slot definition>*
            { *<method definition>* } '}'

<object source> ::=
    <input terminal> | *<output terminal>*

A'UM introduces a notion of *volatile object* to realize conditioning and looping in the same notion of object.

Firstly, ordinary object, whose classes are defined as mentioned earlier, are already condition handlers: they receive a particular set of messages and behave differently for each received message.

If a class is defined for each condition, many classes will be required for one program. It will make the program size unreasonably large, and loose the readability, writability and maintainability of programs.

In addition, such an condition handler object should have a short life, since it should terminate just after conditioning. All the main object wishes to do is change its behaviors depending on the condition result. Therefore, condition handlers should be defined with the main object.

Volatile objects are those which are defined in a method, without any class name given. Applying this notion, volatile objects can be nested.

Thus, volatile objects keep programs from fragmentation and raise their modularity.

6.4.1 Basic

The object source is an external interface to the volatile object, from which messages are sent. This is the basic.

Output Terminal If an output terminal (with `^`) is specified as the object source, there must be one or more input terminals which are merged into the output terminal. Messages are flown from the input terminals to the output terminal.

```
^T [ :p(X, Y) -> P.
     :q(U, V, W) -> Q. ]
```

6.4.2 Extension

If an input terminal (without `^`) or an expression which is evaluated to be an input terminal is specified as the object source, it implies there already exists some output terminal into which messages should be flown. The semantics is extended with a notion of *reflection*.

Input Terminal If an input terminal (without `^`) is specified, a primitive message `:who_are_you(Who)` is sent to the input terminal. An volatile object is created so that it should take an output terminal `^Who` as its external input stream. For each message from `Who`, a method is defined.

```
T [ :p -> P.
    :q -> Q. ]
```

is expanded to:

```
T :who_are_you(Who),
^Who [ :p -> P.
       :q -> Q. ]
```

6.4.3 Applications

Pattern Matching Message `:who_are_you(Who)` transforms an immutable object to a message stream which contains the frozen image of the object as a message. Using this mechanism, pattern matching can be represented.

```
N mod 3 [ :0 -> P1.
          :1 -> P2.
          :2 -> P3. ]
```

If-then-else Construct If a conditional expression, which is evaluated to be either a `true` or a `false` object, is specified in the volatile object field, it means the if-then-else construct.

```
X == Y [ :true -> Then.
         :false -> Else. ]
```

6.4.4 Volatile Mutable Objects and Volatile Immutable Objects

Volatile objects are categorized into two; volatile immutable objects and volatile mutable objects.

Example 6 *Number Generator*

```
class  numbers.
    slot  max, n.
  :initialize(IL) ->
      ^IL {          % volatile mutable %
          :set_max(^M) -> !max = M.
          :set_n(^N) -> !n = N.
      }.
  :do(^Ns) ->
      (!max > !n) [  % volatile immutable %
          :true -> !n = !n + 1,
                   :do( Ns:prime(!n) ).
          :false -> :terminate.
      ].
  :terminate -> .
end.

  #numbers :new(^Numbers),
  Numbers :initialize(^IL) :do(Ns),
  IL :set_max(M) :set_n(0) ::
```

Volatile Immutable Object A volatile immutable object may neither have any internal state nor recur after receiving one message, that is, it is supposed to terminate at once after receiving the message.

The name scope in a volatile immutable object is the same as that in the outside object. Temporary names such as parameter and temporary variables used in the outside object are also visible in the volatile immutable object.

In Example 6, when a message `:do/1` is sent, an immutable object is created for the the condition of `(!max > !n)`. The volatile object accepts a message, either `:true` or `:false`. When receiving `:true`, the volatile object sends a message `:add/2` to slot `n` and updates slot `n` with the computation result. sends a message `:do/1` to the `numbers` object and then terminates. For message `:false`, the volatile object sends message `:terminate` to the outside object and then terminates.

Volatile Mutable Object A volatile mutable object may have internal states and recur in the same way as an ordinary mutable objects do.

The name scope in a volatile mutable object is one level *deeper* than that in the outside object. Temporary names used in the outside object are not visible in the volatile mutable object.

In Example 6, when message `:initialize/1` is sent, a volatile mutable object is created for the output terminal `^IL`. The volatile object accepts messages of either `:set_max/1` or `:set_n/1`, and sets the corresponding slot and recurs until its input stream is closed.

7 Implementing $\mathcal{A}'\mathcal{UM}$ onto KL1

7.1 Message Sending

$\mathcal{A}'\mathcal{UM}$ has been firstly designed on top of KL1, in which a message stream is implemented as a list.

For example, the expression of message sending

```
^NewX = X :add(Y, Z)
```

is translated in KL1 to:

```
X = [add(Y, Z)|NewX]
```

As mentioned before, in $\mathcal{A}'\mathcal{UM}$, both mutable and immutable objects are treated uniformly in message passing.

7.2 Unification Failure Handling

In the above example, let `X` be an integer `1` and `Y` be `2`. Then the following unification must be made true:

```
1 = [add(2, 3)|1]
```

In order to realize it, some extensions have been introduced into KL1.

In the original KL1 language, such a unification normally *fails*. For a certain goal and all subgoals of the goal, a predicate for handling such failure can be specified, which is called in stead of simple failure. It is called the *unification failure handler*. The unification failure handler receives two original arguments, of the unification. If the unification was between two structures and the unification failed for certain elements of them, then these elements are passed as the argument of the unification failure handler. The execution of the unification handler takes place of the execution of the unification itself.

If integers should understand add messages, the unification failure handler should have clause such as the following:

```
handler(Int, [add(Addend, Sum)|Rest]) :-
    integer(Int), integer(Addend) |
    add(Int, Addend, Sum), Int = Rest.
```

The unification failure handler mechanism is harmless to KL1. In the above, it is defined to be appropriate for execution of $\mathcal{A}'\mathcal{UM}$. Users who prefer KL1 can define his/her own unification failure handler which simply fails, keeping the original semantics of KL1.

8 Related Works

In comparison of $\mathcal{A}'\mathcal{UM}$ with other related works, Vulcan [10] is one of the closest approaches. Vulcan is designed as a preprocessor on top of CP and is based on perpetual processes connected via streams. Vulcan supports a variety of functions as $\mathcal{A}'\mathcal{UM}$ does, but both are different from each other as follows: Unlike in $\mathcal{A}'\mathcal{UM}$, name space is flat in Vulcan. Temporary and parameter variables are treated in the same way as those representing the internal states, so it is hard for the programmer to grasp the transition of each internal state. On the way of class inheritance, they are also different. Vulcan supports two ways of inheritance; method copy and delegation, while $\mathcal{A}'\mathcal{UM}$ does only delegation. For developing large systems, the amount of copied method cannot be ignored. Basically, while Vulcan is a preprocessor, $\mathcal{A}'\mathcal{UM}$ is an independent language rather than a preprocessor, and supports message sending as a primitive instruction.

Mandala [9] was also designed on CP like Vulcan. Mandala supports the association of objects with their names, but names are managed globally by a name server, which must bring a bottleneck in performance. In $\mathcal{A}'\mathcal{UM}$, the name association is solved locally in each object, so such a centralization problem is not brought. Another difference is that message receiving in $\mathcal{A}'\mathcal{UM}$ is based on one-at-a-time principle: no external message is received until all the internal behaviors are taken to the previous external message. Mandala allows multiple messages to be received, so it makes its implementation difficult.

These languages explore to realize object-oriented programming with clean semantics. As another approach toward object-oriented programming with clean semantics based on side-effect free foundation, FOOPS [16] should be listed even though it is a functional programming language. FOOPS distinguishes objects from abstract data types and the basic construct is much more restrictive and complicated.

9 Current and Future Works

We are now implementing an experimental version of $\mathcal{A}'\mathcal{UM}$ compiler into KL1. We will write a variety of sample programs to investigate the expressive power of $\mathcal{A}'\mathcal{UM}$ and start writing the operating system PIMOS in this version.

In the future, we are planning to explore the better implementation in which primitive objects should work more effectively. The development of programming and debugging environment will be another work.

References

[1] Takashi Chikayama: *ESP Reference Manual*, Technical Report TR-044, ICOT 1984

[2] Toshio Yokoi: *Sequential Inference Machine: SIM -Its Programming and Operating System*, Proc. of FGCS'84, Tokyo 1984

[3] K. Nakajima, H. Nakashima, M. Yokota, K. Taki, S. Uchida, H. Nishikawa, A. Yamamoto and M. Mitsui: *Evaluation of PSI Micro-Interpreter*, Proc. of IEEE COMPCON-Spring'86, March 1986

[4] H. Nakashima, K. Nakajima, *Hardware Architecture of the Sequential Inference Machine: PSI-II*, Proc. of IEEE Symp. on Logic Programming, Sept. 1987

[5] Atsushi Goto, Shunichi Uchida: *Toward a High Performance Parallel Inference Machine - The Intermediate Stage Plan of PIM*, Technical Report TR-201, ICOT 1986

[6] Kazunori Ueda: *Guarded Horn Clauses*, Technical Report TR-103, ICOT, 1985

[7] Ehud Shapiro: *A Subset of Concurrent Prolog and Its Interpreter*, Technical Report TR-003, ICOT, 1983

[8] Ehud Shapiro and Akikazu Takeuchi: *Object Oriented Programming in Concurrent Prolog*, New Generation Computing, 1 (1983), OHMSHA Ltd. and Springer-Verlag

[9] Koichi Furukawa, Akikazu Takeuchi, Susumu Kunifuji, Hideki Yasukawa, Masaru Ohki and Kazunori Ueda: *Mandala: A Logic Based Knowledge Programming System*, Proc. of the International Conference on FGCS 1984

[10] Kenneth Kahn, Eric D. Tribble, Marks S. Miller and Daniel G. Bobrow: *Vulcan: Logical Concurrent Objects*, Technical Report, Xerox Palo Alto Research Center, 1986

[11] Adele Goldberg and David Robson: *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, 1983

[12] D. Theriault: *A Primer for the Act-1 Language*, MIT AI Memo No. 672, April 1982

[13] Akinori Yonezawa, Jean Pierre Briot and Etsuya Shibayama: *ObjectOriented Concurrent Programming in ABCL/1*, Proc. of OOPSLA'86, 1986

[14] Yutaka Ishikawa and Mario Tokoro: *A Concurrent Object-Oriented Knowledge Representation Language Orient84/K: Its Features and Implementation*, Proc. of OOPSLA'86, 1986

[15] Yasuhiko Yokote and Mario Tokoro: *The Design and Implementation of ConcurrentSmalltalk*, Proc. of OOPSLA'86, 1986

[16] Joseph A. Goguen, Jose Meseguer: *Extensions and Foundations of Object-Oriented Programming*, Internal Memo, SRI & CSLI, 1986