PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

$A'UM$

– A STREAM-BASED CONCURRENT OBJECT-ORIENTED LANGUAGE –

Kaoru Yoshida and Takashi Chikayama

Institute for New Generation Computer Technology
4-28, Mita-1, Minato-ku, Tokyo 108, Japan,
e-mail: {yoshida, chikayama}%icot.jp@relay.cs.net

ABSTRACT

This paper presents a computation model and its programming language, $A'UM$ [1], as a result of our pursuit of high parallelism and high expressivity for the development of a large scale software. By basing it on streams and integrating it with objects and relations, $A'UM$ realizes an elegant model, natural representation and efficient execution, all at once, that have never been done by any other approaches.

1 INTRODUCTION

In general, the larger a problem is, the harder it is to solve, exponentially proportional to the problem size. Our goal is to realize a computation system in which large or complicated problems can be solved quickly in terms of the total amount of time spent for the designing, programming, debugging, maintenance and extension. To achieve this goal, the system should satisfy two kinds of requirements at once: one is to offer natural and flexible user interfaces, including languages and design/program/debugging environments; the other is to realize an efficient implementation. Neither natural user interfaces nor efficient implementations can exist without any sound computation model.

First, we need a computation model that can extract maximum parallelism from the problem and load minimum overhead to the architecture, and provide high level abstractions. Furthermore, we don't want a model such that mixes several paradigms, since it is hard to formalize, understand and realize. At first, the model must be simple and uniform with a single fundamental notion. Abstractions should be built up based on this notion.

In this paper, we present a computation model for completely distributed systems and a language realizing the model, $A'UM$, which is characterized by the following three features:

- Stream-based computation,
- Object-oriented abstractions, and
- Relational representation.

2 BACKGROUND

There have been many theories, models and languages proposed for concurrent systems [Filman84].

[1] $A'UM$ is a Japanese word, derived from a Sanskrit "ahum" consisting of *Ah* and *Um*, which implies the beginning and the end, an open voice and a close voice, and expiration and inspiration. This name was given to symbolize stream communication which is the basic notion of this language.

2.1 Poset-Based Modelling

Concurrent systems are those in which there can exist independent events. Between these independent events, there is no precedence or constraint on which one should happen earlier or later.

While a totally ordered set (or *chain*) is a set in which any pair of elements is given some order, a partially ordered set (*poset*) is a set which may contain elements which have no order. Since posets reflect the concurrent situation naturally, concurrent systems have been formalized based on posets, while sequential systems have been formalized based on chains. The most notable work at this side is Scott's lattice theory [Scott72] and information system theory [Scott82].

[Pratt86] gave a formal language in which a system is represented as an algebraic formula of posets (strictly pomsets) of events. The representation is quite descriptive for programming.

There are other algebraic approaches, called *process algebras*, [Milner80, Milner83, Winskel84, Bergstra84, Winkowski87]. A process is defined to be a set of ports. Communication between processes is basically synchronous and global time and space are assumed to exist. Two systems are said to be equivalent if their sets of linearized events are the same. When a new system is added to some system, the prover has to re-generate all possible chains from all events in the existing system and in the newly added system. This would require an extravagant amount of computation. In other words, models based on the total ordering in global time and space are weak in modularization.

In contrast, the sheaf-theoretical model [Monteiro86] has high modularity. It is assumed that a system has a location and any activity takes place in some location. Communication between systems takes place in the intersection of their locations. Two systems are said to be equivalent if the set of intersections of locations in one system is equivalent to that in the other. The notion is clear but too formal.

The diagram of a poset is a graph. Petri nets [Peterson81] and dataflow graphs [Dennis69, Dennis75, Arvind77, Arvind86] are graphical representations. They are different from each other in representation form and dynamicity, but they have in common localization of functional processes and asynchronous communication which could support a high degree of parallelism. To keep the functionality, procedural languages [Ashcroft77] which adopt the *single assignment rule* [Tesler68], and functional languages [McGraw82] have been proposed as dataflow languages.

In the dataflow model, how to realize I/O, monitors managing some shared resource, as the most typical entities requiring side effects contrary to the single assignment rule, had been a problem. This problem was solved by the notion of streams [Dennis76, Arvind82].

Thus, it is clear that concurrent systems should be modelled based on posets. The problem is how to represent the ordering. Streams are the simplest media for this purpose. The computatioal model of $A'UM$ is based on posets and is represented by streams.

2.2 Stream Computation

A stream is a chain as is a list, but additionally a stream connotes synchronization.

The notion of a stream was first introduced in [Landin65], as a function: $() \rightarrow element \times stream$, that generates an element and another stream when receiving (), where () may be interpreted as the evaluation timing. Since the *lazy evaluation* mechanism that suspends evaluation until needed was introduced to realize the stream, many functional languages supporting streams have been designed [Ida83, Bellot85, Broy86].

G.Kahn's process network [G.Kahn74, G.Kahn77] was the first attempt to model a distributed system using streams. Processes are connected via streams. Each process defines relations between its input streams and output streams and also has its own memory.

2.3 Relational Programming

[Clark81] proposed a relational language for the purpose of expressing a process network, not by functions but by relations. This language provided a basic mechanism for concurrent logic programming (CLP), that is synchronization-embedded unification and commitment control. Stream communication was realized simply by list unification. Following this, several CLP languages [Shapiro83, Clark84, Ueda85] have been proposed. Their declarative nature can extract maximum parallelism and provide high expressivity.

2.3.1 Declarativity

Declarativity is the property that the program contains no constraint on the execution sequence. The following declarative features of CLP frees the programmer's mind from concern about the execution sequence.

- **Parallel Actions:** Goals are executable in parallel or in any order, and their arguments can be unified in parallel.
- **Causal Relations:** All that relates one goal to another is data dependency or causality, that is the relationship of their arguments.
- **Asynchronous Communication:** The writer goal does not wait for the variable to be read. Goals wait only when they read a variable.

Declarativity is one of the greatest advantages to be obtained from concurrent programming. $A'UM$ inherits the above declarative features from CLP.

2.4 Object-Oriented Abstractions

The notion of an object [Goldberg83] is natural. The object-oriented paradigm's specification of abstract computation by message-passing and encapsulation promotes top-down design/programming, systematic designing, and flexible modification and extension. Its modular programming support by class inheritance makes the program code strikingly compact.

2.5 Integration of Streams and Objects

An object or a process is the abstraction of iterative computation as formalized by the fixed-point theory [Staples83, Winkowski87, Milner83, Bakker87]. Objects can be integrated with streams naturally, just by regarding consumers and producers of streams as objects.

In $A'UM$, a system is composed of streams and objects. Computation consists of consumption and production of streams. As long as there exist messages, computation may proceed. When there is no message left, this is the completion of the computation.

In the program, each stream can be looked upon as an object itself. *Making acquaintance with an object* is getting a stream toward the object. *Introducing some acquaintance to another* is splitting the stream toward the former into two and passing one of the two streams to the latter.

[Shapiro83B] proposed an object-oriented programming style for CLP. This style shows the framework of stream communication and generation management of an object. An object is represented as a sequence of tail recursive goals, each of which has an interface stream to receive a message from, and carries arguments as its internal states. A clause is regarded as a generation of an object, a tail recursive goal as the creation of a new generation. Attempting to program in this style revealed several substantial problems as follows, which motivated us to design $A'UM$.

Semantic Representation: Deadlock, that is the state in which no goal is executable, is caused by very tiny bugs, such as misnaming and mispositioning logical variables that represent internal states of objects, much more often than by algorithmic errors. To solve large scale or complicated problems, program representation should be more semantic and flexible than syntactical.

$A'UM$ provides object-oriented abstractions, including message passing, name association of slots, and class inheritance, which will make the semantics clearer and program modification easier, and will also increase modularity.

Implicit Completion: Deadlock also often occurs due to failure to connect and close streams. In stream programming, the most important things are to be sure to connect a stream to its destination, and to close a stream. Failure to connect does not happen so often, but stream closing is often overlooked as it is both trivial and burdensome. We would like to offer some linguistic support to remedy this problem.

$A'UM$ implicitly closes open streams and connects orphan streams to *sink objects* that absorb messages.

Uniform Universe: Objects with stream interfaces and primitive data objects are treated differently: the former requires stream merging when they are shared, but the latter not. This decreases the ability of top-down programming or program reusability, since different programs for the top level must be prepared depending on the data type which might be hidden in a lower level.

$A'UM$ realizes a universal object space. There exist nothing but objects which communicate with each other via streams. Primitive objects, such as integers and atoms, also have the same stream interface. So, we can write generic programs in a top-down manner. [Aczel88] formalizes such infinite levels of abstraction.

Solid Abstraction: Several approaches to provide a macro package for this object-oriented programming style have been proposed [K.Kahn86]. But, syntax sugars melt at execution time.

$A'UM$ preserves its object-oriented abstractions at execution time, which will be a great help for debugging.

2.6 Nondeterminacy

G.Kahn's model was limited to deterministic (or functional) processes, whose semantics are determined only by the parameters appearing in the program.

[Brock81] pointed out that a nondeterministic process, which may accept either a single event or multiple serial events from a stream nondeterministically, cannot be defined only by the relations between streams.

Nondeterminacy in Stream Merging: $A'UM$ makes this problem simple. A stream is split into two when it is shared, so a stream eventually becomes a tree of streams as the amount of sharing increases. At a splitting (or merging) joint, messages from the two branches are nondeterministically ordered. Nondeterminacy exists only in stream merging and objects are deterministic.

2.7 Graph Grammar

In general, a language is defined by a grammar [Mandrioli87].

A sequential program is regarded as a string of symbols. A sequential language is defined by a string grammar which rewrites one non-terminal symbol to another by applying one production rule at a time.

In contrast, a concurrent program is regarded as a graph consisting of vertices as processes and edges as communication links. A concurrent language is defined by a graph grammar [Ehrig79], which rewrites non-terminal graphs to others by applying multiple production rules at one time.

[Kreowski86] made clear the correspondence between derivation in a graph grammar and computation in a Petri net. [Kaplan88] proposed a language which visualizes the graph derivation.

Derivation of Nils: In this paper, all pictures are drawn in a certain manner that messages should flow *from right to left* toward the leftmost object. We define a language for $A'UM$, with a grammar which keeps this manner, so that we can write a program like drawing a picture.

A stream is represented as a pair of terminals: an *inlet* and an *outlet*, to indicate the direction of message flow. Two streams are connected to be one, when the inlet of one is connected to the outlet of the other. The consumer object holds an *inlet* to receive a message from, and the producer object holds an *outlet* to send a message to. A closed stream is represented as *nil*.

The grammar is composed of expressions, which are categorized into *inlet*, *outlet* and *nil* expressions, depending on what they express. A method of an object is defined as a set of *nil* expressions, which is constructed from *inlet* and *outlet* expressions. The derivation of a set of nil's leads to the completion of computation.

2.8 Efficient Architecture

As mentioned earlier, the single assignment rule is essential to increase parallelism. In CLP, the single assignment rule is assured by logical variables, but logical variables can be unified in any direction and shared by multiple readers. A process may wait for any combination of multiple events.

First, it is so rare (and strange) for one process to nondeterministically read or write an identical variable, but the architecture always has to prepare for the bi-directional unification of any logical variable. Some CLP languages allow to specify variable modes, but some do not. The lack of variable modes although some CLP languages do support them, makes it invisible from where to where messages flow, so that the program readability and the potential ability for optimization and error-detection at compilation time decrease.

Second, garbage collection (GC) is one of the most critical problems to be solved for making parallel architectures practical. To increase memory usability and obtain continuous response, most of the discussions have been made on real-time GC [Cohen81, Chikayama87, Goto88]. The problem of multiple references makes this hard to realize.

The stream computation of $A'UM$ makes the architecture simpler and more efficient.

Single Assignment and Single Reference: A stream is consumed by a single reader and produced by a single writer. This is the single reference single assignment principle. The single assignment raises parallelism in the program, which can be supported by the architecture. The single reference expands the possibility of static optimization and error detection, makes GC simple, resulting in better memory usability.

Single Direction and Single Event: Each object waits for a single event, of either receiving a message or detecting that its interface stream is closed. Nondeterminacy that deals with multiple event waiting exists only in stream merging. The single direction single event rule makes the process scheduling simple.

3 COMPUTATION MODEL

In this section, we define the computation model of $A'UM$ with some interpretation for intuitive understanding.

3.1 Systems

A system is composed of streams and objects.

3.2 Streams

A *stream* is a chain which is represented as a pair $(\mathcal{E}, \prec)$ where $\mathcal{E}$ is a set of elements and $\prec$ is a total order between any pair of messages. For $e_1, e_2 \in \mathcal{E}$, $e_1 \prec e_2$ implies that e_1 should be received precedeing e_2.

Let us add two additional elements: *bottom* $\bot$ and *top* $\top$, such that $\bot \prec e \prec \top$ for any $e \in \mathcal{E}$.

A stream that is complemented with bottom and top is said to be *complete* and represented as a pair $(\overline{\mathcal{E}}, \prec)$, where $\overline{\mathcal{E}} = \mathcal{E} \cup \{\bot, \top\}$. The bottom $\bot$ is the initial state of the stream, *undefined*. It symbolizes the existence of a consumer

(or *target object*) that receives messages from the stream. There exists only one consumer for each stream. The top $\top$ is the final state of the stream, *nil*, that shows the stream is closed.

Production and Consumption: We define four operations on streams: `send`, `close`, `receive` and `is_close`.

As a relation, there is no distinction between `send` and `receive` and between `close` and `is_closed`.

`send`(X, m, Y), `receive`(X, m, Y): The least element of stream X is associated with message m and stream Y is the rest of the stream, i.e.

$$\exists e(f(e) = m,\ X \supset Y,\ \mathcal{E} = \{e\} \cup \mathcal{E}',\ \forall e' \in \mathcal{E}'(e \prec e')),$$

where $X = (\mathcal{E}, \prec)$, $Y = (\mathcal{E}', \prec)$, f is a function associating an element with a message.

`close`(X), `is_closed`(X): Stream X is completed by *nil*, i.e. $\mathcal{E} = \phi$, $\bot \prec \top$ where $\overline{X} = (\overline{\mathcal{E}}, \prec)$, $\overline{\mathcal{E}} = \mathcal{E} \cup \{\bot, \top\}$.

Operationally, they are different: `send` and `close` instantiate an undefined stream and `receive` and `is_closed` observe it. To distinguish their operational semantics, we specify the *stream direction* by $\triangledown$ for the stream where instantiation (either association or completion) takes place and $\triangle$ for the stream where the instantiation is observed. They can be regarded as *terminals*: $\triangledown$ is a terminal to send a message out to and $\triangle$ is a terminal to receive a message from, so we refer to $\triangledown$ as the *outlet* and $\triangle$ as the *inlet*, as shown in figure 1.

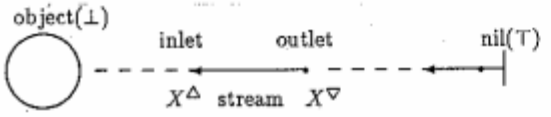

Figure 1: Stream

`Send`$(X^\triangledown, m, Y^\triangle)$ associates the least element of stream X with message m.

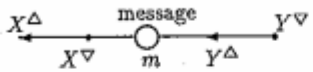

Figure 2: Message Sending

`Close`$(X^\triangledown)$ completes stream X by *nil*.

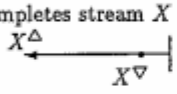

Figure 3: Stream Closing

`Receive`$(X^\triangle, m, Y^\triangledown)$ observes that the least element of stream X is associated with message m.

`Is_closed`$(X^\triangle)$ observes that stream X is completed by *nil*.

Connection: Connecting an inlet to an outlet implies that two orderings in a row $(\prec\prec)$ results in one ordering $(\prec)$.

`Connect`$(X^\triangledown, Y^\triangle)$ unifies stream X and stream Y are identical, i.e. $X = Y$.

3.3 Channels

A *channel* is a poset composed of streams (chains), which is represented as a pair $(\mathcal{E}, \preceq)$ where $\preceq$ is a partial order. A channel (poset) is converted to be a stream (chain) by the following operation.

`Serialize`$(C^\triangle, S^\triangledown)$: Stream S is a chain of the messages in channel C, where incomparable messages in channel C are nondeterministically ordered, i.e.

$$\mathcal{E} = \mathcal{E}',\ \neg\exists e_1, e_2(e_1 \prec e_2 \in S, e_2 \prec e_1 \in C)$$

where $C = (\mathcal{E}, \preceq)$, $S = (\mathcal{E}', \prec)$.

Since posets generalize chains, streams are channels. We give the following definitions using channels.

3.4 Joints

A *joint* is an operation mapping two channels to another: *channel*$\times$*channel* $\rightarrow$ *channel*. We define two kinds of joints: *merge* and *append*.

`Merge`$(X^\triangle, Y^\triangle, Z^\triangledown)$: Channel Z is the union of all messages from two disjoint channels: X and Y; the orderings in X and Y are kept, i.e. $Z = X \cup Y,\ X \cap Y = \phi$.

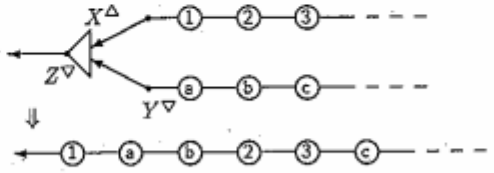

Figure 4: Stream Merging

`Append`$(X^\triangle, Y^\triangle, Z^\triangledown)$: Channel Z is the union of all messages in two disjoint channels: X and Y; the messages in X precede any message in Y and the orderings in X and Y are kept, i.e.

$$Z = X \cup Y,\ X \cap Y = \phi,\ \forall e \in X, e' \in Y(e \prec e' \in Z).$$

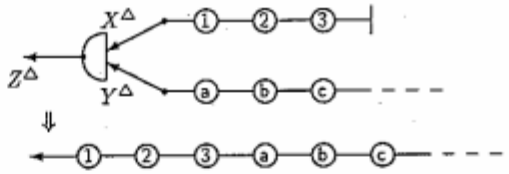

Figure 5: Stream Appending

3.5 Messages

A *message* is an element of a channel which is represented as a pair $(\nu, \mathcal{P})$ where ν is a message name and $\mathcal{P}$ is a tuple of stream terminals.

A message is called either an *atomic message* or a *compound message* depending on whether $\mathcal{P}$ is empty or not.

Each compound message is identified by the message name, the number of the terminals and their directions. Even if messages consist of the same message name and the same number of terminals, they are different if any of their terminals has a different direction.

Compound messages work as connectors. As mentioned above, one channel is connected to another when the inlet of the one and the outlet to the other are given. Thus, the sender and the receiver should specify complementary directions for each terminal.

EXAMPLE: Message $\mathtt{a}(X^\triangle, Y^\triangledown, Z^\triangle)$ has a name a and three terminals: two inlets $X^\triangle$ and $Z^\triangle$ and one inlet $Y^\triangledown$, so this message is identified as $\mathtt{a}(\triangle, \triangledown, \triangle)$. To receive this message, the target object should specify receiving a message like $\mathtt{a}(U^\triangledown, V^\triangle, W^\triangledown)$.

Inquiry Message "Who are you?": There are several generic messages which any object is supposed to receive

and answer. Among them, the most interesting one is the inquiry message, `who_are_you(`Who^{∇}`)`. When an object receives this message, it is supposed to send a message to outlet *Who*, which should reflect the object itself, such as an integer value for an integer object. "Who are you?" is just a message, no different from any other message.

3.6 Objects and Generations

An *object* is a chain of generations, whose overview is depicted in figure 6.

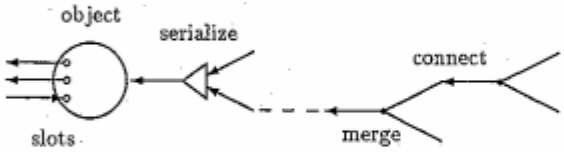

Figure 6: Overview of an Object

Creation: When message `new(`X^{Δ}`)` is sent to a class, the class creates the 0-th generation of an instance, by the following operation.

`New(`$Obj^{\Delta}, D^{\nabla}, S^{\nabla}$`)` creates an object Obj^{Δ} which is *undefined* ($\perp$), where S is a set of streams each of which represents an internal state or attribute of the object.

After sending an *initiation* message (`initiate`) to the 0-th generation, the class attaches a serializer which will serialize the messages from the outside channel, as shown in figure 7. Each generation observes the inlet of the serializer, that is called the *interface stream*.

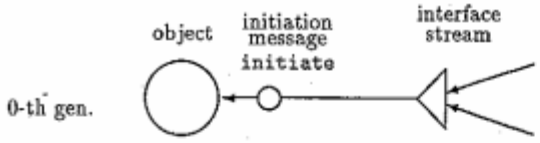

Figure 7: Creation

There are several *primitive objects*, such integers (e.g. `1`), an atoms (e.g. `a`), booleans (e.g. `'true`), strings (e.g. `''hi''`), classes (e.g. `##stack`), and some system defined objects, such as *sink objects*.

Sink Object: A *sink object* receives messages from its interface stream and, out of the terminals contained in the messages, it closes each outlet and connects each inlet to another sink object.

Routine: The behavior of each generation is defined by a *method*, represented as a pair $(e, \mathcal{A})$, where e is an *event* that is either `receive` or `is_closed`, and $\mathcal{A}$ is a set of *actions* which contains any number of `send`, `close`, `connect`, `merge`, `append`, `new` and the primitive or system-defined object creation, and one or fewer `descend` defined as follows:

`Descend(`$Self^{\Delta}, S^{\nabla}$`)` creates a new generation $Self^{\Delta}$ which is *undefined* ($\perp$).

A generation first waits for an *event* on its interface stream. When it observes an event, it serches an appropriate method for the event, then it takes the set of actions, as shown in figure 8.

The actions may be executed in parallel or in any order. The streams related with the actions may be connected independently. Since the generation descending action and other

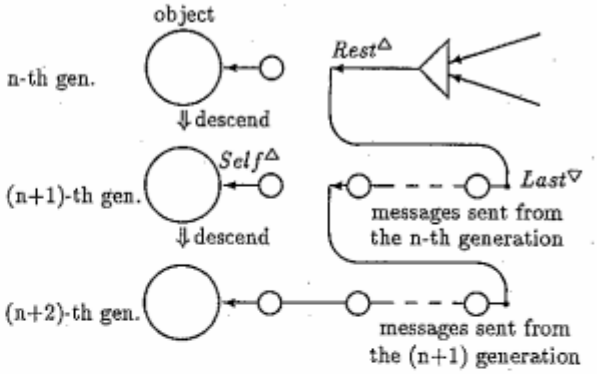

Figure 8: Succession

actions may be executed in any order, there can exist those actions taken by different generations at a time. When some generation sends a message to itself, *itself* means the next generation of the object.

Termination: An object descends generations until it receives a *termination message* (`'$terminate'`), as shown in figure 9. When an object receives the termination message, it *completes* all the streams it holds, such as slots, and terminates its life.

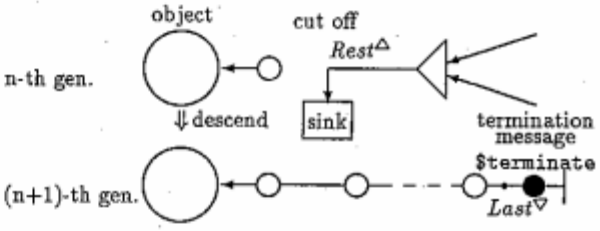

Figure 9: Termination

3.7 Slots

An object may hold a set of stream terminals. Each slot is associated with a slot name and accessible by the name beyond generations. There are two kinds of slots: *inlet slots* and *outlet slots* according to their directions. An object accesses each slot by sending a particular message to itself.

3.7.1 Inlet Slots

Reference: When message `get_inlet(`$Name^{\nabla}, Slot^{\nabla}$`)` is sent, the current inlet is given to the accessor and a new slot is created, which holds a closed stream.

Updating: When message `set_inlet(`$Name^{\nabla}, Slot^{\Delta}$`)` is sent, the current inlet is connected to a *sink* object and a new slot is created, which associates the given inlet with the slot name.

3.7.2 Outlet Slots

Reference: When message `get_outlet(`$Name^{\nabla}, Slot^{\Delta}$`)` is sent, the current outlet is split into two, the one is given to the accessor and a new slot is created, which holds the other.

Updating: When message `set_outlet(`$Name^{\nabla}, Slot^{\nabla}$`)` is sent, the outlet the current slot holds is closed and a new slot is created, which associates the given stream with the slot name.

3.8 Classes

A class is an object which defines a set of slot names, a set of methods and a set of super classes it inherits and creates instances according to the definition. There is no notion of meta classes whose instances are classes.

A class can inherit a set of methods from one or more classes. Class inheritance expands the set of applicative methods, but does not create any instance for the inherited classes. For every event, an object searches for an appropriate method by traversing the inheritance tree.

4 LANGUAGE

We define a language for the computation model, with a two-layerd grammar:

- the *basic grammar*, which represents the computation model exactly, and
- the *extended grammar*, which introduces several extensions and linguistic supports into the basic grammar for ease of writing compact and safe programs.

Before explaining the details, we give two examples: *Stack*, a typically object-oriented program, and *Tree Reverse*, a highly parallel program, both written in the extended grammar.

Example 1 (Stack) (see figure 10)
A stack holds the top of the stack, which initially points at the bottom of the stack.

For message `push(^Data)`, the stack creates a new element and asks it to set the given data, `Data`, and the current top, respectively in its slots `data` and `next`. Then, the stack holds the element after the sending as the new top.

For message `pop(Data)`, the stack asks the current top to get the current `data` and `next`, and makes the given outlet `Next` the new top.

Method `read(Data)` is the same as method `pop(Data)` except for that it does not update the current top.

Method `test` creates a stack, pushes an integer 1 onto it, checks if the data by `read` and `pop` are the same.

```
class  stack.                                          (1)
    out  top.                                          (2)
 :initiate    -> #bottom = !top.                       (3)
 :push(^Data) -> #element:set(Data,!top)= !top.        (4)
 :pop(Data) -> !top:get(^Data,^Next),Next= !top.       (5)
 :read(Data)  -> !top:get(^Data,^Next).                (6)
end.                                                   (7)
class  element.                                        (8)
    out  data,next.                                    (9)
 :set(^Data,^Next) -> Data= !data, Next= !next.        (10)
 :get(Data,Next) -> !data= ^Data, !next= ^Next.        (11)
end.                                                   (12)
class  bottom.                                         (13)
 :get(eos,Next) -> .                                   (14)
end.                                                   (15)
class  test_stack.                                     (16)
 :test((A==B)) ->                                      (17)
    #stack:push(1):read(^A):pop(^B).                   (18)
end.                                                   (19)
```

Figure 10: Program of Stack

Example 2 (Tree Reverse) (see figure 11)
Method `do` *(2-20)* reverses each node of the given tree `T` to obtain a reversed tree `RevT`. Simultaneously it obtains the maximum `Max` of the leaves and sifts the leaves into even integers `^Evens` and odd integers `^Odds`. If `T` is a leaf *(4-9)*, it is the reversed tree and the maximum. If `T` is a node *(10-19)*, its left and right trees are reversed.

Method `test` *(23-29)* creates a `reverse` object, makes it reverse a tree `{{1,{11,12}},{2,{21,22}}}`, and generates a stream of the `Answers`, such as `reverse:v(^R):max:n(22):evens:n(2):n(22):n(12):odds:n(21):n(1):n(11)::`, where `^R` is the reversed tree, `{{{21,22},2},{{12,11},1}}`.

```
class  reverse.                                        (1)
 :do(^T,RevT,Max,^Evens,^Odds) ->                      (2)
   (class_of T) ? (                                    (3)
     :integer ->                                       (4)
       T= ^RevT, T= ^Max,                              (5)
       ( T mod 2 == 0 ) ? (                            (6)
          :'true  -> Evens:n(T) ;                      (7)
          :'false -> Odds:n(T)                         (8)
       ) ;                                             (9)
     :vector  ->                                       (10)
       T:element(0,^L),                                (11)
       #reverse:do(L,^RevL,^MaxL,Evens,Odds),          (12)
       T:element(1,^R),                                (13)
       #reverse:do(R,^RevR,^MaxR,Evens,Odds),          (14)
       {RevR,RevL} = ^RevT,                            (15)
       ( MaxL > MaxR ) ? (                             (16)
         :'true  -> MaxL= ^Max ;                       (17)
         :'false -> MaxR= ^Max                         (18)
       )                                               (19)
   ).                                                  (20)
end.                                                   (21)
class  test_reverse.                                   (22)
 :test ->                                              (23)
   #reverse:do({{1,{11,12}},{2,{21,22}}},              (24)
               ^RevT,^Max,Evens,Odds),                 (24)
   Answers$1:reverse:v(RevT),                          (25)
   Answers$2:max:n(Max),                               (26)
   Answers$3:evens= ^Evens,                            (27)
   Answers$4:odds= ^Odds,                              (28)
   :nop(^Answers).                                     (29)
 :nop(Answers) -> .                                    (30)
end.                                                   (31)
```

Figure 11: Program of Tree Traverse

5 BASIC GRAMMAR

The basic grammar represents the computation model exactly and is characterized by the following two rules:

R1: For each stream, its two terminals: an inlet and an outlet *must* be specified in the method.

R2: Each action is represented as an expression, which denotes either an inlet, an outlet or *nil* as its result, and expressions can be constructed according to their result category. A method *must* be composed of nil expressions.

Missing a terminal and leaving inlet or outlet expressions might lead to deadlock. These rules are given for the pur-

pose of *completing* all the streams, i.e. for promoting the completion of computation.

5.1 Class Definition

A class definition consists of a class name, super classes, slot names and methods as follows:

```
<ClassDefinition> ::=
  class <ClassName> '.'
    [ super <ClassName> { ',' <ClassName> } '.' ]
    [ in <SlotName> { ',' <SlotName> } '.' ]
    [ out <SlotName> { ',' <SlotName> } '.' ]
    { <Method> '.' }
  end '.'
<Method> ::= <Event> '|' [ <Actions> { ',' <Actions> }]
<Actions> ::= <NilExpression>
```

5.2 Stream Variables

A variable represents a stream, which has two terminals: an *inlet* and an *outlet*.

Inlet is a terminal from which a message is received, or which is connected to the outlet of another stream. An inlet is specified by a variable name preceded by '^', e.g. `^X`.

Outlet is a terminal to which a message is sent, to which the inlet of another stream is connected, or which is closed. An outlet is specified by a variable name, e.g. `X`.

Nil is the state left after an outlet is closed.

5.3 Expressions

Events and actions are specified by expressions. An expression represents either an inlet, an outlet or nil as its result, and is correspondingly named *an inlet expression*, *an outlet expression* or *a nil expression*, as listed in table 1.

All pictures in this paper are drawn in a certain manner that messages should flow *from right to left* toward the leftmost object. Expressions have been designed to keep this manner.

EXAMPLE: `#bottom = !top` at *(3)* in example 1 means that messages flow from the streams which will be merged into slot `top`, to an object `#bottom`.

Expressions can be constructed according to their result category. Connection `connect(X, ^Y)` is represented as a combination of `merge` and `close`, i.e. `X = ^Y::`.

EXAMPLE: `X:m1:m2 = ^Y ::` means that outlet `X` is sent two serial messages: `m1` and `m2` and connected by inlet `Y`.

Non-nil expressions can be specified for parameters contained in messages, since they express stream terminals.

EXAMPLE: method `test` in example 1 is equivalent to:

```
:test(TF) -> #stack:push(1):read(^A):pop(^B)::,
             (A == B) = ^TF ::.
```

5.4 Descending Action

Generation descending is an action which can be taken in parallel with the other actions. The following two expressions: *succession* and *termination* are provided to abstract the generation descending.

Succession (<*Event*> '->') merges the rest of the interface stream ($Rest^{\triangle}$) into the last Self ($Last^{\triangledown}$), as shown in figure 8.

EXAMPLE: `:m -> :do.` is equivalent to:

```
:m = Rest | <== ^Self, Self:do = ^Rest.
```

Table 1: Expressions

relation	*expression*	*result*
`receive(^X,m,Y)`	':' <*Message*> '=' <*Out*> `:m = Y`	<*Nil*>
`is_closed(^X)`	`::`	<*Nil*>
`send(X,m,^Y)`	<*Out*> ':' <*Message*> `X : m`	<*Out*> `Y`
`close(X)`	<*Out*> '::' `X ::`	<*Nil*>
`merge(^X,^Y,Z)`	<*Out*> '=' <*In*> `Z = ^X`	<*Out*> `Y`
`append(^X,^Y,Z)`	<*In*> '\' <*In*> `^X \ ^Y`	<*In*> `^Z`
`descend(^X,S)`	'<==' <*In*> `<== ^X`	<*Nil*>

Termination (<*Event*> '-|') connects the rest of the interface stream ($Rest^{\triangle}$) to a sink object, sends a termination message to the last Self and then closes its tail, as shown in figure 9.

EXAMPLE: `:m -| :do.` is equivalent to:

```
:m = Rest | <== ^Self, Self:do:'$terminate'::,
            ##sink:new(^Rest)::.
```

5.5 Self

For each generation, the object *itself* means the next generation, which is accessed by name `$self` in same way as slots.

Causality on Self: A method is expanded in the top-to-bottom, left-to-right inner-to-outer order. The order of messages being sent to *itself* is determined in accordance with the method expansion order.

If sending-to-self expressions are specified as parameters of a message to be received or sent, the Self referred to is the one just after the receiving or just before the sending respectively.

EXAMPLE: `:get(!data, !next) ->.` is equivalent to:

```
:get(Data,Next) -> !data= ^Data, !next= ^Next.
```

that is expanded in the basic grammar as follows:

```
:get(Data,Next)= Rest | <== Self,
    Self:get_outlet(data,^Data)
        :get_outlet(next,^Next)= ^Rest.
```

5.6 Volatile Objects

Each generation is (1) activated by a certain event and takes some actions according to the event, and (2) descends to the next generation in parallel. The former means conditioning and the latter looping. Originally, objects were themselves condition handlers.

If a class was defined for handling each condition, however, many small classes would have to be defined and the program context would be geographically scattered into pieces. We introduce the notion of a *volatile object*, which is defined within a method and is realized in the object framework.

Volatile Classes: A *volatile class* is temporarily defined within a method. Any number of volatile classes can be defined within a method and they can be nested, i.e. in the definition of some method of some volatile class, another volatile class may be defined. There is no distinction between

external classes and *volatile classes* except that the former have *external names* but the latter do not. External classes can be accessed and inherited using their external names by any other classes including volatile classes, but volatile classes cannot.

EXAMPLE: Method do in example 2 contains three volatile classes: `$v1`:*(4-20)*, `$v11`:*(7-8)* and `$v12`:*(17-18)*, where `$v1`, `$v11` and `$v12` are temporary names given for explanation.

Volatile Objects: A volatile object is an instance of some volatile class, and is executable in parallel with its creator.

Creator-Bound Stream: Each volatile object is given a stream to its creator at initiation, and can access the stream as a outlet slot named `$creator`. The creator-bound streams passed to multiple volatile objects are appended according to the method expansion order.

5.6.1 Creation of Volatile Objects

The creation of a volatile object is defined with an interface stream and a class definition specified as follows:

<ImmutableVolatileObjectCreation> ::=
    <Interface> '?' *<ImmutableVolatileClassDefinition>*
<MutableVolatileObjectCreation> ::=
    <Interface> '=>' *<MutableVolatileClassDefinition>*
<Interface> ::= *<In>* | *<Out>*

Volatile object creation expressions are nil expressions.

Basic: Inlet as Interface If an inlet was specified for the *<Interface>*, the volatile object takes the inlet as its interface stream.

EXAMPLE: A volatile object created by:

```
^Hunger ? (:'true -> :eat; :'false -> :sleep)
```

receives a message from inlet `^Hunger`.

Extension: "Who are you?" to Outlet If an outlet was specified for the *<Interface>*, the message `who_are_you(Who)` is implicitly sent to the outlet and the volatile object takes inlet `^Who` as its interface stream.

EXAMPLE: `(T mod 2 == 0) ? (...)` is equivalent to: `(T mod 2 == 0):who_are_you(Who)::, ^Who ? (...)`.

5.6.2 Immutable and Mutable Volatile Objects

There are two kinds of volatile classes: *immutable volatile (IV) classes* and *mutable volatile (MV) classes*, which differ from each other in their variable scope and relationship with their creator, e.g. the `$v1`, `$v11` and `$v12` are IV classes.

Scope: Slot names are *permanent names* which are valid outside generations. In contrast, variable names are *temporary names* which are valid only within a generation. The space in which a name is visible is called its *name scope*.

Immutable Volatile (IV) Objects:

- *Transparent Scope:* An IV object shares the same scope as its creator. The same variable names in an IV object and its creator represent an identical stream/channel,
- *Creator as Its Next Generation:* For an IV object, the Self and its slots are those of its creator, i.e. `$self` and `$creator` are identical.

IV classes are used mainly for conditioning.

Mutable Volatile (MV) Objects:

- *Independent Scope:* An MV object has an independent name scope. The same variable names in an MV object and its creator represent different stream/channels.
- *Creator as a Slot:* An MV object is another chain of generations, whose Self and slots are independent from those of the creator, i.e. `$self` is different from `$creator`.

MV classes are used mainly for looping.

Example 3 (Prime) (see figure 12)
A program for the pipe-lined prime number generator is given as follows, where lines *(19-25)* defines the creation of a mutable volatile object.

```
class  primes.                                              (1)
 :primes(^Max, ^Ps) ->                                      (2)
   :generate(3,Max,Ns), #sift:do(3,^Ns,Ps:n(2)).           (3)
 :generate(^N, ^Max, ^Ns) ->                                (4)
   ((N+2 = ^NewN) < Max) ? (                                (5)
     :'true -> :generate(NewN, Max, Ns:n(NewN));            (6)
     :'false -> % end %                                     (7)
   ).                                                       (8)
end.                                                        (9)
class  sift.                                                (10)
 :do(^V, Ns, ^Ps) ->                                        (11)
   S:initialize(V, Ps)= ^Ns,                                (12)
   ^S => ( out me, next, to_next, primes;                   (13)
     :initialize(^V,^Ps) ->                                 (14)
       V= !me, O= !next, Ps= !primes.                       (15)
     :n(^X) ->                                              (16)
       (X mod !me == 0) ? (                                 (17)
         :'true -> % do nothing %                           (18)
         :'false ->                                         (19)
           (!next == 0) ? (                                 (20)
             :'true -> % there is no next yet %             (21)
                X= !next, Ns= !to_next,                     (22)
                #sift:do(X, ^Ns, !primes:n(X));             (23)
             :'false -> !to_next :n(X)                      (24)
           ) ) )                                            (25)
end.                                                        (26)
```

Figure 12: Program of Prime Number Generator

6 EXTENDED GRAMMAR

The basic grammar is extended for the purpose of ease of writing natural, compact and safe programs. The following extensions are made:

- The meaning of a variable name is extended from a stream to a channel, so that streams toward the same object can be identified only by variable occurrences.
- Non-nil expressions can be specified for actions, which the language system will implicitly complete. This linguistic support protects us from unexpected deadlock.
- Macro expressions and abbreviations are provided, so that we can write simple and compact programs even without paying attention to the notions of message-passing or stream.

6.1 Channel Variables

Here, let us think of a simple example, *Consult*, that is let two persons (X and Y) solve some problem (P) by their own strategy and collect their answers (A); each person may give more than one answer.

(1) In case of collecting their answers in any order, for the program in the basic grammar:

```
:consult(^P, ^A, ^X, ^Y) ->
    X:solve(P1, A1)::, Y:solve(P2, A2)::,
    P = ^P1 = ^P2 ::, A = ^A1 = ^A2 ::.
```

let us represent the merging joints only by the occurrences of outlet variables as follows:

```
:consult(^P, ^A, ^X, ^Y) ->
    X:solve(P, A), Y:solve(P, A).
```

(2) In case of collecting their answers in the order that the answers from X should precede any answer from Y, for the program in the basic grammar:

```
:consult(^P, ^A, ^X, ^Y) ->
    X:solve(P1, A1)::, Y:solve(P2, A2)::,
    P = ^P1 = ^P2 ::, A = ^A1 \ ^A2 ::.
```

let us represent the appending joint by the occurences of outlet variables with ordering numbers as follows:

```
:consult(^P, ^A, ^X, ^Y, ^Z) ->
   X:solve(P, A$1), Y:solve(P, A$2).
```

Our programmers' concern is what to do an object, rather than how to connect streams toward it. With the extention of the meaning of a variable name from a stream to a channel, the programmer's intention can be represented directly. Generally, a channel may be any combination of merge and append joints. Thus, a variable name represents a channel which may have the following occurrences:

One or Fewer Inlets, which is specified by the variable name preceded by `'^'`, e.g. `^X`,

Zero or More Unordered Outlets, each of which is specified only by the variable name, e.g. `X`, and merged into the inlet, and

Zero or More Ordered Outlets, each of which is specified by the variable name succeeded by `'$'` and some ordering number, e.g. `X$2`, and is appended according to the number into the inlet.

EXAMPLE: channel `X= ^P1= ^P2= ^S1\ ^S2\ ^S3::`, can be represented by one inlet `^X`, two unordered outlets, `X`'s, and three ordered outlets, `X$1`, `X$2` and `X$3`, as shown in figure 13.

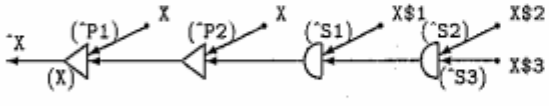

Figure 13: Channel Variables

6.2 Implicit Completion

In stream programming, the most important thing is ensuring the closing and connecting of streams; otherwise, an unexpected deadlock might occur. Our programmers, however, would like to be kept away from such detailed logic as much as possible. The grammar is extended in both directions toward flexibility and reliability as follows:

(1) Non-nil expressions, that leave outlets open or inlets unconnected, are allowed to be specified for the <*Actions*>.

(2) Non-nil streams are implicitly *completed*. Implicit completion includes: (a) autoclosing outlets, (b) autodischarging inlets, and (c) autotermination objects.

6.2.1 Autoclosing Outlets

Leaving one outlet open might cause deadlock, because there might be some object waiting for messages sent from the outlet. Outlets left open are implicitly closed. Among those autoclosed are:

(1) The result of an outlet expression, left open,
(2) The missing outlet of a channel,
(3) The current outlet slot on updating, and
(4) The outlet slots at termination of an object.

6.2.2 Autodischarging Inlets

Leaving one inlet unconnected also might cause deadlock, because there might be some object waiting for messages from some stream contained in some message running toward the inlet. An unconnected inlet is connected to a sink object. This is called *discharging an inlet*. Among those autodischarged are:

(1) The result of an inlet expression, left unconnected,
(2) The missing inlet of a channel,
(3) The interface stream cut off at termination, and
(4) The inlet slots at initiation of an object.

6.2.3 Autoterminating Objects

Most objects are terminated when their interface stream is closed, but we often fail to specify the termination, and that will cause deadlock. When no method for termination is defined, the object is implicitly terminated when its interface stream is closed.

6.3 Macros

Many kinds of macro expressions are provided for ease of writing compact programs as follows:

Arithmetic/Logical Operation Macros,
each of which represents the outlet to the operation result, e.g. `T mod 2 == 0` represents `TorF` with `T:mod(2,^M)::,M:eq(0,^TorF)::`.

Instance Creation Macro (`'#'`<*ClassName*>),
which represents outlet `Obj` with message `new(^Obj)` sent to the class,
e.g. `#stack` represents `S` with `##stack:new(^S)::` .

Outlet Slot Access Macro (`'!'`<*SlotName*>),
whose semantics depends on the context.
Reference: In the *may-be-outlet* field, it represents `Slot` with message `get_outlet(Name,^Slot)`,
e.g. `!data= ^Data` merges `^Data` into the current `data`.
Updating: In the *must-be-inlet* field, it represents inlet `^Slot` with message `set_outlet(Name,Slot)`,
e.g. `Data= !data` makes `Data` the new `data`.

Inlet Slot Access Macro (`'@'`<*SlotName*>),
whose semantics depends on the context.
Reference: In the *may-be-inlet* field, it represents inlet `^Slot` with message `get_inlet(Name,Slot)`,
e.g. `Cur= @ans` connects the current `ans` to `Cur`.
Updating: In the *must-be-outlet* field, it represents outlet `Slot` with message `set_inlet(Name,^Slot)`,
e.g. `@ans= ^New` makes `^New` the new `ans`.

6.4 Abbreviations

"Self" as Default Destination: When no destination is specified in message sending or closing, the destination is assumed to be the the object itself, e.g. `:nop(^Answers)` in example 2 is equivalent to: `$self:nop(^Answers)`.

Updating Destination: In sending a message to a named stream, such as an outlet slot and the object itself, the outlet left after the sending is assumed to be the new generation of the destination, e.g. `!top:get(^Data, ^Next)` is equivalent to: `!top:get(^Data, ^Next) = !top`.

7 IMPLEMENTATION ISSUES

We have implemented an experimental system for $\mathcal{A}'\mathcal{UM}$ on KL1 or Flat GHC (a subset of GHC), including its compiler and execution environment. Using this system, several application programs are being tested to investigate and improve the expressive power and performance of $\mathcal{A}'\mathcal{UM}$. Its debugging environment is now under development.

In this section, we describe several implementation issues which are particular to $\mathcal{A}'\mathcal{UM}$.

7.1 Implementation of Stream Joints

A channel consists of stream joints. If each joint were implemented as a process, a great number of processes would be created. For a stream programming language, one of the most critical points for the performance is how to implement stream joints effectively.

To minimize the number of processes, we took the following strategy:

- *Joint as a Vector:* Each joint is represeneted as a static vector which consists of a tag `T` indicating either `merge` or `append`, and two streams `X` and `Y`, i.e. `Z={T,X,Y}`.
- *Serializer as a Vector Consumer:* In front of the first joint, which interfaces an object with the outside, a dynamic process, *serializer*, is created, so there exist serializers as many as objects. The serializer consumes a nested vector according to the tag and generates a stream.

Since the number of joints can be much larger than that of objects, this strategy is very effective for performance.

7.2 Optimizations Owing to Streams

The single assignment single reference property makes several static optimizations possible, among which are the following:

1. *Combining Merge, Append and Close:* If one stream of a joint is closed, the other stream can be unified with the result, e.g. `merge(X,Y,Z)` and `close(X)` leads `Y=Z`. Thus, no wasteful joint is created for connecting streams.
2. *Shortening Dereference Chains:* Applying the above elimination repeatedly might generate many transitive unifications, e.g. `X=Y, Y=Z`. Such an intermediate variable like `Y` can be reduced since it is assured that there is no other `Y`. Then, the above pair results in `X=Z`. This minimizes the length of dereference chain that is traversed at execution time.

7.3 Implementation of Variable Name Scope

When IV classes defined in a method, which IV objects are created and which channels they access are determined at execution time. In any case, all the related channels must be *completed*. It is impractical to generate a different code for every possible case statically. The physical code must be minimized. Variable's scope control is one of the most difficult problems for stream programming languages.

Our solution is simple: *what is determined dynamically should be solved dynamically.*

Scope Objects: A *scope object* manages name association of shared variables within a generation of an object, while other objects manage name association of their slots throughout their lives. An IV object shares (holds a stream to) the scope object of its creator, while an MV object creates an independent scope object for itself.

Scope objects are terminated similarly as other objects do: when its interface stream is closed, the scope object completes all the terminals it holds, i.e. autocloses the outlets and autodischarges the inlets.

Global and Local Variables: It is not necessary to create a scope object every time an IV object is created. The scope object is needed only when one or more variable names are shared among the IV objects and their creators.

The compiler categorizes variables into two: *local variables* and *global variables.*

- *Local variables* are those which appear only in one object. Their name association is statically resolved.
- *Global variables* are those which are shared by more than one object within a scope. If there exist one or more global variables, the compiler generates codes for creating a scope object and for sending a message to the scope object to access every global terminal by its variable name.

7.4 Implementation of Primitive Objects

In the current system, each primitive object is realized as a process which receives messages from its interface stream as the other objects are. How to implement primitive objects is another critical point for the performance. The following mechanism, *unification failure handler*, makes it possible to represent primitive objects as their exact values.

Unification Failure Handler: For example, the message sending expression, `X:add(Y,Z) = ^NewX`, is translated in KL1 to: `X=[add(Y,Z)|NewX]` .

If we represent primitive objects as themselves, for example, integer objects just as integer values, e.g. `X=1, Y=2`, then the unification `1=[add(2,3)|1]` must be made *true*. KL1 would make such a unification *fail*. In order to make it *true*, some extensions must be introduced into KL1.

For a certain goal and all of its subgoals, a predicate for handling such failure can be specified, which is invoked instead of simple failure. It is called the *unification failure handler*. The unification failure handler receives the two original arguments of the unification. If two structures are unified and the unification failed for certain of their elements, then these elements are passed as the arguments to the unification failure handler. The execution of the unification handler takes the place of that of the unification itself.

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

```
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. The above is defined to be appropriate for the execution of $A'UM$, but users who prefer KL1 can define another unification failure handler that simply fails, keeping the original semantics of KL1. Therefore, it is very general and effective for the implementation of primitive objects.

8 Comparisons with Other Works

8.1 $A'UM$ vs. Other COOP Languages

There have been proposed several concurrent object-oriented programming (COOP) languages [Yonezawa86, Ishikawa86, Yokote86, America85, Zhong87, Hur87]. Most of these langauges realize parallism by introducing a parallel mechanism onto sequential control. Objects are runnable in parallel, but inside each object, control is sequential. Synchronous communcation makes programming harder. Most of the programmer's attention is on how to draw the execution thread rather than on what to solve. With such control dependent programming, it is hard to expect high expressive power or high parallelism.

$A'UM$ is most different from and superior to these COOP languages mainly at the following two points:

(1) *Declarative Definition:* The object behavior is defined declaratively: an object takes actions in parallel, which are related with each other only by causality, and its communication is asynchronous. Declarativity is one of the most important clues on how to make concurrent programming simple and practical.

(2) *Conditioning by Volatile Objects:* Most of object-oriented languages realize a condition handler using the higher-order notion, *continuation*, that is the rest of a program. For example, the *block* scheme in Smalltalk is to create a *True* object and a *False* object and pass the program context to be executed after conditioning, to both of them. The *True* and *False* objects must be meta interpreters that can interprete the given program code.

$A'UM$ realizes condition handlers in the framework of objects, by volatile objects. The volatile object scheme neither creates more than one condition hander object nor requires any meta control, so it is uniform in the concept and efficient in the execution.

8.2 $A'UM$ vs. Actor

The Actor model [Hewitt77A, Hewitt77B, Agha86] is the closest in point of view to our model. In both models, the basis is on posets, objects and causality, communication is asynchronous, and actors (objects) can be created dynamically. The differences are as follows:

In the Actor model,

(1) The arrival order of events at an actor is uncontrollable in the program. The system implicitly inserts an arbiter for each object, which serializes events.

(2) Addressing is direct. Messages are sent directly to the target actor, so the programmer must notice that an actor should be created before messages are sent to it.

(3) Computation is functional. The actor *returns* the computation result to the destination actor that is specified in the request message as a continuation.

In $A'UM$,

(1) The ordering is explicitly representable in the program. We can specify orders between multiple messages toward an object, using streams.

(2) Addressing is indirect. A stream may be connected to either the target object or another indirect stream *some time in the future* and the connetion may happen *from anywhere*, so we can send a message to any stream without making sure if it is connected to the target object.

(3) Computation is relational. The object defines only relations between incoming streams and outgoing streams, which are symmetric.

8.3 Streams vs. Channels

[Tribble87] defined a channel to be a poset in contrast to a stream which is a chain. The main purposes of introducing channels are the following two:

(1) to regard a stream tree as a single entity,

(2) to reduce the number of serializer processes for high performance.

In addition, multiple writers and multiple readers were allowed for one channel and

(3) the writing order between multiple writers was controlled nondeterministically,

(4) multiple readers were allowed to read different chains from a single poset.

The former (1) and (2) are well-understood: (1) is a matter of representation and (2) is that of implementation. We realized (1) by regarding a variable name as a channel and (2) by vectorizing stream joints.

However, it is hard to understand that the latter (3) and (4) make any logical sense. In $A'UM$., a channel is composed of streams. Only single reader is allowed for each channel and each writer holds a component stream of a channel. This makes semantics more sound and implementation simpler.

9 FUTURE WORK

By basing it on streams and integrating it with objects and relations, $A'UM$ has satisfied most of our requirements: elegancy in the model, naturalness in representation and efficiency in execution, but not well enough.

The language represents the model well, but is not abstract enough for programming yet. More expressive power is needed. We would like to supplement abstractions and linguistic supports to the language.

In the experimental system, we utilized the parallel computation and communication mechanism of KL1, including unification and commitment control, but the basic mechanism required for $A'UM$ is much lighter. We plan to design and develop an independent system that provides the basic mechanism sufficient and well-suited for $A'UM$ by itself. With this system, we will prove that $A'UM$ is practical as much as elegant.

ACKNOWLEDGEMENTS

This research has been done as part of the FGCS project in ICOT. We would like to thank Kazuhiro Fuchi and Shunichi Uchida for giving us this research opportunity. Our

thanks are also sent to all those who gave us valuable comments and suggestions on $A'UM$ and the paper.

REFERENCES

[Ashcroft77] E. A. Ashcroft and W. W. Wadge: *Lucid, a Nonprocedural Language with Iteration*, CACM 20(7), 1977.

[Aczel88] P. Aczel: *Non-well-founded Sets*, CSLI Lecture Notes 14, Stanford University, 1988.

[Agha86] G. A. Agha: *A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.

[America85] P. America: *Definitions of the programming language POOL-T*, ESPRIT project 415, Doc.91, Philips Research Laboratories, Eindhoven, the Netherlands, 1985.

[Arvind77] Arvind and K.P. Gostelow: *A Computer Capable of Exchanging Processors for Time*, Proc. IFIP 77, 1977.

[Arvind82] Arvind and J. D. Brock: *Streams and Managers*, LNCS 143, pp.452-465, Springer-Verlag, 1982.

[Arvind86] Arvind and D. E. Culler: *Dataflow Architecture*, Ann. Rev. Computer Science, 1:225-53, 1986.

[Bellot85] P. Bellot and B. Robinet: *Streams are not Dreams*, LNCS 242, Springer-Verlag, 1985.

[Bakker87] J. W. de Bakker, J. J. Ch. Meyer, and E. R. Orderog: *Infinite Streams and Finite Observations in The Semantics of Uniform Concurrency*, Theoretical Computer Science 49, North-Holland, 1987.

[Bergstra84] J. A. Bergstra and J. W. Klop: *Process Algebra for Synchronous Communication*, Informantion and Control 60, 1984.

[Brock81] J. D. Brock and W. B. Ackerman: *Scenarios:A model of Non-determinate Computation*, LNCS 107, Springer-Verlag, 1981.

[Broy86] M. Broy: *A Theory for Nondeterminism, Parallelism, Communication, and Concurrency*, Theoretical Computer Science 45, North-Holland 1986.

[Chikayama87] T. Chikayama and Y. Kimura, *Multiple Reference Management in Flat GHC*, Proc. of 4th ICLP, 1987.

[Clark81] K. Clark and S. Gregory: *A Relational Language for Paralle Programming*, Concurrent Prolog Vol.1, MIT Press, 1987.

[Clark84] K. Clark and S. Gregory: *PARLOG: Parallel Programming in Logic*, Concurrent Prolog Vol.1, MIT Press, 1987.

[Cohen81] J. Cohen, *Garbage Collection of Linked Data Structures*, ACM Computing Surveys 13(3), 1981.

[Dennis69] J. B. Dennis: *Programming Generality, Parallelism and Computer Architecture*, Proc. of IFIP 69, 1969.

[Dennis75] J. B. Dennis: *Preliminary Architecture for a Basic Data-Flow Processor*, IEEE Symp. on Comp. Arch., 1975.

[Dennis76] J. B. Dennis: *A Language Design for Structured Concurrency*, LNCS 54, Springer-Verlag, 1976.

[Ehrig79] H. Ehrig: *Introduction to the Algebraic Theory of Graph Grammars*, LNCS 73, Springer-Verlag, 1979.

[Filman84] R. E. Filman and D. P. Friedman: *Coordinated Computing - Tools and Techniques for Distributed Software*, McGraw-Hill, 1984.

[Goldberg83] A. Goldberg and D. Robson: *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.

[Goto88] A. Goto, Y. Kimura, T. Nakagawa and T. Chikayama, *An Incremental Garbage Collection Method for Parallel Inference Machines*, Proc. of LP88, Seattle, 1988.

[Hewitt77A] C. Hewitt: *Viewing Control Structures as Patterns of Passing Messages*, Artificial Intelligence 8(3), 1977.

[Hewitt77B] C. Hewitt and H. Baker: *Laws for Communicating Parallel Processes*, Proc. of IFIP Congress 77, pp.987-992, North-Holland, 1977.

[Ida83] T. Ida and J. Tanaka: *Functional Programming with Streams*, Proc. IFIP 83, pp.265-270, 1983.

[Ishikawa86] Y. Ishikawa and M. Tokoro: *A Concurrent Object-Oriented Knowledge Representation Language Orient84/K: Its Features and Implementation*, Proc. of OOPSLA'86.

[Hur87] J. H. Hur and K. Chon: *Overview of a Parallel Object-Oriented Language CLIX* LNCS 276, Springer-Verlag, 1987.

[K.Kahn86] K. Kahn, E. D. Tribble, M. S. Miller, and D. G. Bobrow: *Vulcan: Logical Concurrent Objects*, Concurrent Prolog Vol.2, MIT Press, 1987.

[G.Kahn74] G. Kahn: *The Semantics of a Simple Language for Parallel Programming*, Proc. of IFIP 74, pp.471-475, 1974.

[G.Kahn77] G. Kahn and D. MacQueen: *Coroutines and Networks of Parallel Processes*, Proc. IFIP 77, pp.993-998, 1977.

[Kaplan88] S. M. Kaplan and G. E. Kaiser: *Garp: Graph Abstractions for Concurrent Programming*, LNCS 300, Springer-Verlag, 1988.

[Kreowski86] H.-J. Kerowski and A. Wilharm: *Net Processes Correspond to Derivation Processes in Graph Grammars*, Theoretical Computer Science 44, North-Holland, 1986.

[Landin65] P. J. Landin: *A Correspondence between Algol60 and Church's Lambda-Notation: Part I*, Comm. ACM 8(2), pp.89-101, 1965.

[McGraw82] J. R. McGraw, *The VAL Language: Description and Analysis*, ACM Transaction on Programming Languages and Systems 4(1), 1982.

[Mandrioli87] D. Mandrioli and C. Ghezzi: *Theoretical Foundation of Computer Science*, John Wiley & Sons, 1987.

[Milner80] R. Milner: *A Calculus of Communicating Systems*, LNCS 92, Springer-Verlag 1980.

[Milner83] R. Milner: *Calculi for Synchrony and Asynchrony*, Theoretical Computer Science 25, North-Holland, 1983.

[Monteiro86] L. F. Monteiro and F. C. N. Pereira, *A Sheaf-Theoretic Model of Concurrency*, Symp. on Logic in Computer Science, 1986.

[Peterson81] J. L. Peterson: *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, 1981.

[Pratt86] V. Pratt: *Modeling Concurrency with Partial Orders*, International Journal of Parallel Programming 15(1), 1986.

[Scott72] D. S. Scott: *Continuous Lattices*, LNM 274, Springer-Verlag, 1972.

[Scott82] D. S. Scott: *Domains for Denotational Semantics*, LNCS 140, Springer-Verlag, 1982.

[Shapiro83] E. Shapiro: *A Subset of Concurrent Prolog and Its Interpreter*, Concurrent Prolog Vol.1, MIT Press, 1987.

[Shapiro83B] E. Shapiro and A. Takeuchi: *Object Oriented Programming in Concurrent Prolog*, Concurrent Prolog Vol.2, MIT Press, 1987.

[Staples83] J. Staples and V.L. Nguyen: *Computing the Behavior of Asynchronous Processes*, Theoretical Computer Science 26, North-Holland, 1983.

[Tesler68] L. G. Tesler and H. J. Enea: *A Language Design for Concurrent Processes*, Proc. of SJCC, 1968.

[Tribble87] E. D. Tribble, M. S. Miller, K. Kahn, D. Bobrow and C. Abbott: *Channels: A Generalization of Streams*, Concurrent Prolog Vol.1, MIT Press, 1987.

[Ueda85] K. Ueda: *Guaded Horn Clauses*, Concurrent Prolog Vol.1, MIT Press, 1987.

[Winskel84] G. Winskel: *Synchronization Trees*, Theoretical Computer Science 34, pp.33-82, North-Holland, 1984.

[Winkowski87] J. Winkowski: *An Algebra of Processes*, Journal of Computer and System Sciences 35, pp.206-228, 1987.

[Yokote86] Y. Yokote and M. Tokoro: *The Design and Implementation of ConcurrentSmalltalk*, Proc. of OOPSLA'86.

[Yonezawa86] A. Yonezawa, J.-P. Briot and E. Shibayama: *Object-Oriented Concurrent Programming in ABCL/1*, Proc. of OOPSLA'86.

[Zhong87] Y. Zhong and M. Sowa: *Towards an Implicitly Parallel Object-Oriented Language* Proc. of COMPSAC'87, 1987.