ICOT Technical Memorandum: TM-0818

TM-0818

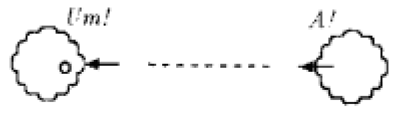
A'UM Introductory Guide

by
K. Yoshida

October, 1989

$\mathcal{A}'\mathcal{UM}$ Introductory Guide

(Version 1.0)



Institute for New Generation Computer Technology

Fourth Laboratory

Abstract

$\mathcal{A'UM}$ [1] is a concurrent object-oriented programming language which looks at concurrent computation and object-oriented abstraction from a microscopic viewpoint based on the notion of a stream. $\mathcal{XAS}$ is an experimental system to execute $\mathcal{A'UM}$ programs on top of PDSS (PIMOS Development Support System). This manual desribes the computational model and programming language of $\mathcal{A'UM}$ and guides how to use $\mathcal{XAS}$.

[1] $\mathcal{A'UM}$ is a Japanese word, derived from the Sanskrit "ahum" consisting of *A* and *Hum*, 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.

Contents

Chapter 1

Model and Language of $\mathcal{A}'\mathcal{UM}$

1.1 Introduction

A concurrent system is one in which more than one event can arise independently, where the word *independently* means that the events may arise in parallel or in any order.

For example, suppose that there is a system S which consists of three events a, b and c, between which the following ordering relations exist:

- a must arise earlier than b;
- c is independent from both a and b.

A set consisting of uniquely ordered elements is called a *totally ordered set* or a *chain*, while a set containing elements whose ordering cannot be uniquely determined is called a *partially ordered set* (*poset* for short).

Let us define three sets, $A = \{a\}$, $B = \{b\}$ and $C = \{c\}$, then the above system S is expressed as:

$$S = (A \oplus B) + C$$

where $\oplus$ is called an *ordinal addition* and $+$ a *cardinal addition*; $+$ is commutative, but $\oplus$ is not. Each operator implies a different kind of addition on posets and produces a poset as the operational result.

In a sequential system, since only one event can occur at a time, the entire system is defined as a chain of events. The ordinal addition is enough to define a sequential system. In a concurrent system, in contrast, some events may occur in a row and some occur independently at the same time. In addition to the ordinal sum, another operator is needed to define an indeterministic order, that is the cardinal addition. Thus, a concurrent system is defined as a poset of events.

Again, a chain is an ordinal sum of elements, and a poset is a cardinal sum of chains. This relation can be represented naturally with streams.

In general, there are several kinds of streams, such as streams of water (rivers), streams of electricity (electric currents), and streams of cars (traffic). We visualize the notion of streams illustrating a river in Figure 1.1.

The river consists of countless drops of water. The drops connect one after another to form a continuous stream. Moving from the mountains to the ocean, the tributaries converge into wider and wider streams until the river arrives at the ocean. One drop from one tributary may arrive at the ocean earlier or later than another drop from another tributary. The river has a direction of flow, downhill. Another example of stream is connected electrical wires. For each wire, the end toward the anode is attached with a red tag and the other end toward the cathode with a blue tag. We can make a long wire by connecting the red tag of one wire to the blue tag of another.

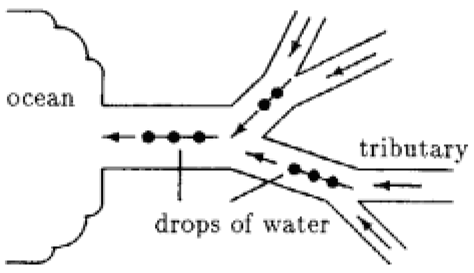


Figure 1.1: River and ocean

$\mathcal{A}'\mathcal{UM}$ is a concurrent programming language which deals with streams of messages [Yoshida88]. Let us regard the ocean as an object and each drop of water as a message flowing into the object. Here is the basic idea of $\mathcal{A}'\mathcal{UM}$.

The computation model is defined in sections 1.2 and 1.3. The programming language is explained in sections 1.4 and 1.5.

1.2 Stream Computation

1.2.1 Streams

A *stream* is a sequence of messages. The direction of a stream is depicted as an arrow with a pair of terminals: *inlet*, denoted by a variable name preceded by "^" (^X for example), and *outlet*, denoted by a variable name (X for example), as shown in Figure 1.2.

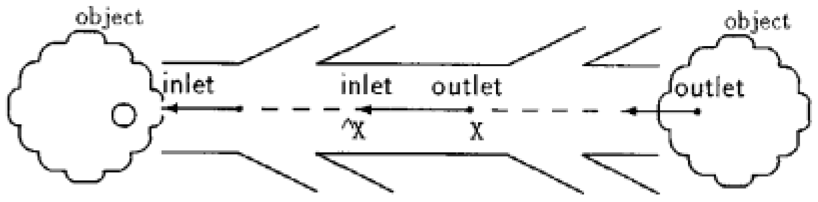


Figure 1.2: Stream representation

The terms, *inlet* and *outlet*, are given from the viewpoint of an *object* who receives a message from the stream or sends a message to the stream. So from the stream's point of view, inlet and outlet each mean the opposite of what they say. For a stream, messages go out of the inlet and come from the outlet. Remember that inlet and outlet are from the object's point of view.

1.2.1.1 Stream Operations

Basic operations to produce and consume a stream that is a sequence of messages are defined, as depicted in Figure 1.3.

send: sending a message to an outlet (X), when the outlet of the following stream is given as Y.

close: closing an outlet (Y), when a state, *nil*, is left, which indicates that the stream is no longer accessible.

receive: receiving a message from an inlet (^X), when the inlet of the following stream is given as ^Y.

is_closed: detecting that an inlet (X) is closed.

Thus, we explain stream computation using three terms, *inlet*, *outlet* and *nil*.

The most characteristic feature of this stream computation model is that there is no restriction or constraint on the stream production order. The destination object toward which a stream is being produced need not be created before the stream is produced. Before knowing which object a stream is connected to, messages can be sent to the stream. The object may be created after the stream has been produced. Also a stream can be produced from any part and in any order. The part closer to the destination object may be created later than the part further.

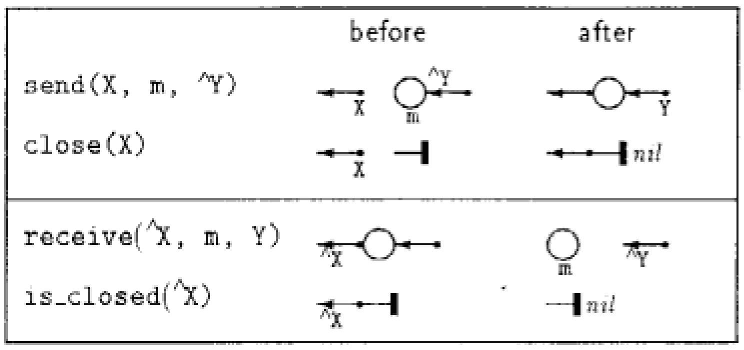


Figure 1.3: Basic operations

1.2.2 Joints

In order to build a stream tree, we define two kinds of binary operations, called *joints*, as depicted in Figure 1.4:

merge: merging messages from two streams, ^X and ^Y, in an indeterministic order, where the message order in stream X and that in stream Y are kept.

append: appending messages from one stream ^Y to any of the messages from the other stream ^X, where the message order in stream X and that in stream Y are kept.

The merge operation corresponds to a cardinal sum of posets, and the append operation to an ordinal sum of posets. In $\mathcal{A}'\mathcal{UM}$, indeterminacy is contained only in the merge operation. Hereafter, we refer to posets produced by joints as *channels* and chains as *streams* in a strict sense.

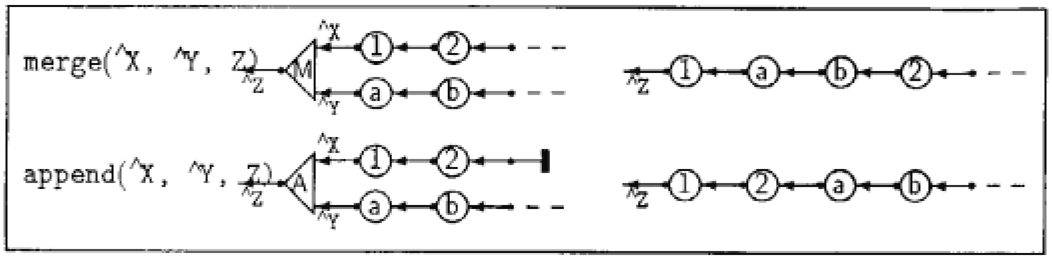


Figure 1.4: Joints

In case that one of the two incoming streams (Y for example) is immediately closed for either of the binary operations, merge and append, this means that the other incoming stream (X for example) is connected to the outgoing stream (Z for example).

connect: connecting an inlet (^X) to an outlet (Z);

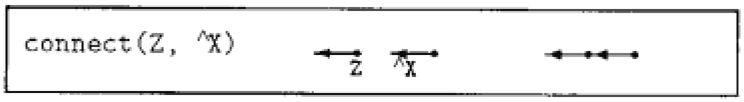


Figure 1.5: Stream connection

1.2.3 Messages

A *message* contains a message name and a tuple of stream terminals as its arguments, each of which is either an inlet or an outlet. Those messages which have only a message name but no argument are called *atomic messages*; those which have some arguments are called *compound messages*.

An individual message is identified by the message name, the number of the terminals, and their directions from the receiver's viewpoint. Even if the message names match, if either the number of arguments or their directions are different, messages are recognized as being different.

Each message works as a stream connector which connects streams given as actual parameters in the sender's scope, with streams given as formal parameters in the receiver's scope. Since two streams can be connected when the inlet of one and the outlet of the other are given, the sender and the receiver must specify complementary directions for each parameter.

For example, message m(X, ^Y, Z) in Figure 1.6 has message name m and contains three terminals: one inlet ^Y and two outlets X and Z, so it is identified as m(-+-). The sender of this message is expected to specify such as m(^U, V, ^W) so that ^U is connected to X, ^Y to V, and ^W to Z.

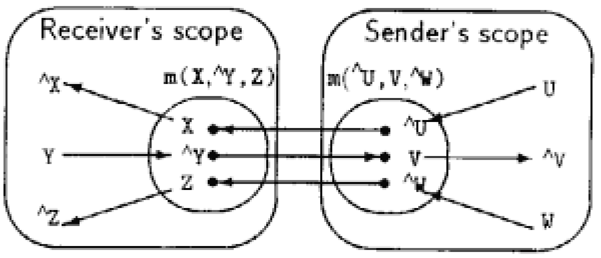


Figure 1.6: Message as a stream connector

1.3 Object-Oriented Abstraction

1.3.1 Objects

An *object* is the abstraction of iterative computation. When an object is created, it is given one stream, called the *interface stream*, through which the object will receive a message from the outside.

From the sender's viewpoint, each stream toward an object can be looked upon as the object itself. *Making acquaintance with an object* is obtaining a stream toward the object. *Introducing an acquaintance* (A) to another (B) is splitting the stream toward A into two and passing one of the two branch streams to B.

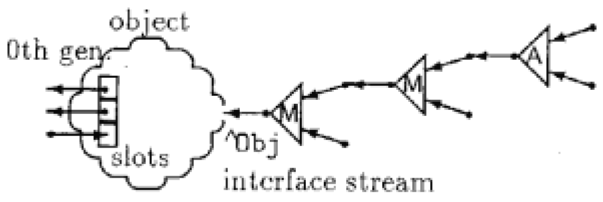


Figure 1.7: External view of an object

Creation of an object: When a message, `new(^Obj)` is sent to a class, one instance object of the class is created, which is called the *0-th generation* of the object. The inlet of the interface stream to this fresh object is given as `^Obj`.

1.3.2 Generations

Each object repeats a cycle, called *generation*, which consists of the following two phases:

1. **Passive phase:** An object waits for an event on the interface stream, which is either receiving a message from the stream or detecting that the stream is closed.

2. **Active phase:** When an event is detected, the object is activated and takes some actions according to the observed event. Among those actions are:

   - zero or more actions of sending, closing, connection, merging, appending and creation of primitive objects mentioned later, and
   - one or fewer actions of generation descending.

Each generation is a collection of stream terminals, including the inlet of the interface stream from which it receives a message and other stream terminals which represent the internal states of the objects.

1.3.2.1 Methods

A *method* is a script that defines the behavior of one generation. Each method consists of two parts: a *passive part* to specify what event should be observed, and an *active part* to specify what actions should be taken according to the event.

1.3.2.2 Generation Descending

The action to create the next generation of an object is called *generation descending*. The generation descending action may be executed in parallel with other actions. Since the current generation and the next generation are related only by the causality of their interface streams, it does not matter when the generation descending action is executed.

1.3.2.3 Self

For each generation of an object, the object *itself* means the next generation. Sending a message to itself is simply sending a message to a stream (outlet) toward the next generation. At generation descending, the rest of the interface stream follows the last of those messages sent by the current generation to itself, as depicted in Figure 1.8.

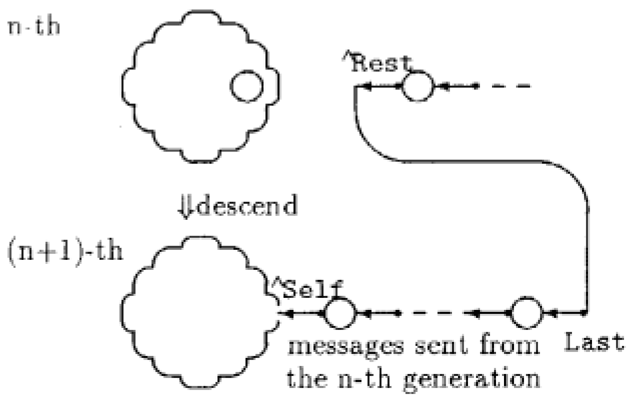


Figure 1.8: Generation descending and Self

Splitting of Self: If an object splits a stream toward itself and gives one of the branch streams to some other object, it will be able to receive messages sent by the other object in the future.

1.3.2.4 Succession and Termination

Each object descends its generations until it receives a termination message, `$terminate`, so that the life of an object is a chain of generations as depicted in Figure 1.9.

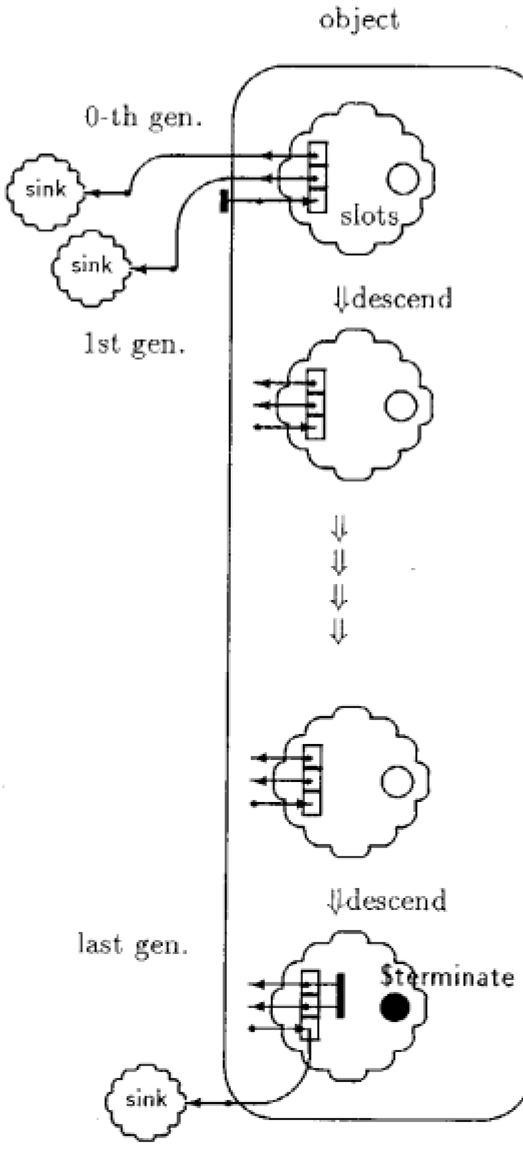


Figure 1.9: Object as a chain of generations

When to send the termination message can be specified in the program. Without any specification, it is regarded as when the closing of the interface stream is detected.

Termination of an object: When an object receives a termination message, `$terminate`, it *completes* all the stream terminals it holds. Stream completion is mentioned later more in detail.

1.3.3 Stream Completion and Computation Termination

For a poset which is composed of chains, if the least element under the ordering relation is determined and the least upper bound is given to each of the chains, the poset is called a *complete poset* (*cpo* for short). Defining the least element of the entire set and the least upper bounds of all the chains of a poset is called the *completion of a poset*.

The least element can be regarded as the initiation of computation, the entire set as the computation process, and the least upper bound as the termination of computation on the chain. Computation neither starts unless the least element is given, nor terminates unless all the least upper bounds are given.

Globally looking over a stream tree formed toward an object, the object corresponds to the least element of the entire stream tree, and the outlet of each branch stream to the least upper bound. Microscopically at the level of a stream, the inlet corresponds to the least element, and the outlet to the least upper bound. Therefore, missing either the inlet or the outlet of a stream causes the following problems:

Missing an outlet: If the outlet of a stream is left open, since there may be some object waiting for some event on the stream, a deadlock might occur.

Missing an inlet: If the inlet of a stream is left unconnected, at first, those messages coming into the stream will be garbages.

In addition, the inlets and outlets carried by these messages will be left unconnected. Those who sent these messages might be expecting the inlets to be connected to some objects and the outlets to be closed after some messages sent, so deadlock might occur.

For stream computation, we refer to determining both the inlet and the outlet of a stream as the *completion of a stream*. The missing inlet or outlet of an incomplete stream is processed as follows:

Discharging inlets: If there is an inlet which is left unconnected, it is connected to a *sink object* which is mentioned below. The sink object is thought to be the lower bound of the stream.

Closing outlets: If there is an outlet which is left open, it is closed. Nil is thought to be the upper bound of the stream.

As long as a message exists in a system, there may be some activity in that system. If there is no message left in a system, this is when the system is terminated. As long as all streams existing in a system are finite chains of messages, completion of the streams guarantee that all messages will be accepted by objects, so that a program will be terminated.

1.3.3.1 Sink Objects

A *sink object* is an object which can interpret messages and works as a message disposer. In each generation,

- if it receives a message from the interface stream, it completes all the arguments of the received message. It connects the inlets to other sink objects and closes the outlets.
- if it detects that the interface stream is closed, it terminates its life as an object.

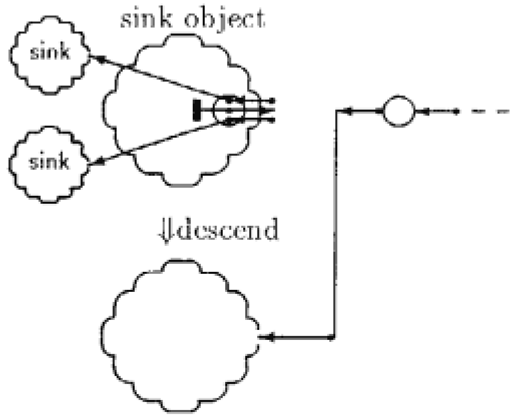


Figure 1.10: Sink object

1.3.4 Slots

An object may hold a set of *slots*, each of which holds a stream terminal associated with a name. There are two kinds of slots, *inlet slots* and *outlet slots*, according to the direction of the terminal that they hold.

Slot access is done by sending a particular message to the object *itself*. In the following, we explain what action is taken for each slot access message.

1.3.4.1 Inlet Slots

Initiation: When an object is created, each of its inlet slots holds nil.

Reference: When an inlet-slot-reference message `get_inlet(Name, Slot)` arrives at an object, the current inlet slot is connected to the second argument of the message. The new generation of the inlet slot holds nil.

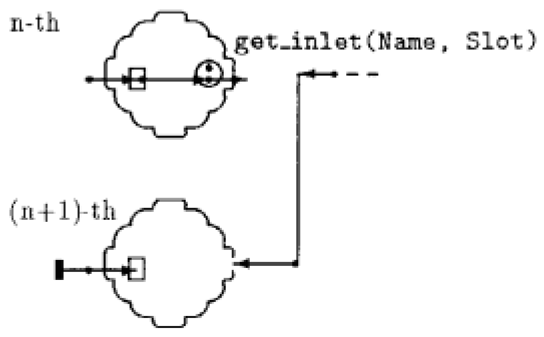


Figure 1.11: Reference to an inlet slot

Updating: When an inlet-slot-updating message `set_inlet(Name, ^Slot)` arrives at an object, the current inlet slot is connected to a sink object. The second argument of the message is connected to the next generation of the inlet slot.

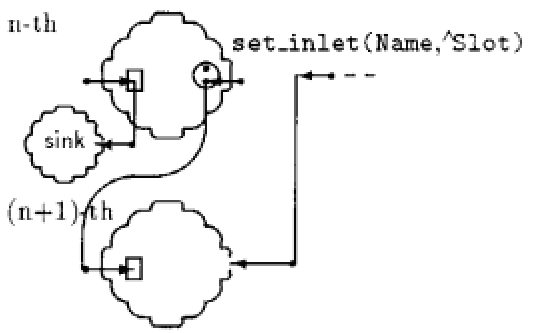


Figure 1.12: Updating an inlet slot

Termination: When an object is terminated, each of its inlet slots is connected to a sink object.

1.3.4.2 Outlet Slots

Initiation: When an object is created, each of its outlet slots is connected to a sink object.

Reference: When an outlet-slot-reference message `get_outlet(Name, ^Slot)` arrives at an object, the current outlet slot is split into two, one of the two branches is connected from the second argument of the message, and the new generation of the outlet slot is connected to the other branch.

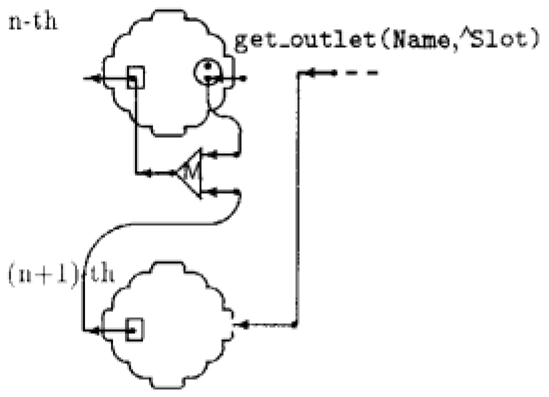


Figure 1.13: Referring to an outlet slot

Updating: When an outlet-slot-updating message `set_outlet(Name, Slot)` arrives at an object, the current inlet slot is closed. The new generation of the outlet slot is connected to the second argument of the message.

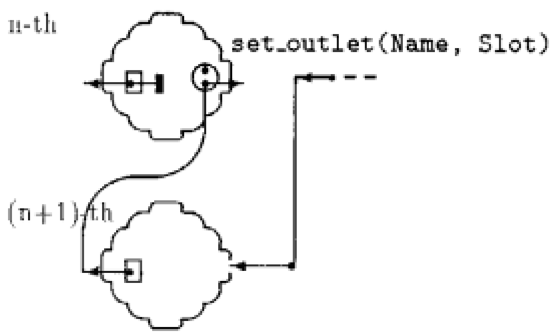


Figure 1.14: Updating an outlet slot

Termination: When an object is terminated, each of its outlet slots is closed.

1.3.5 Primitive Objects

Those which are usually categorized into primitive data, such as integers (5 for example), atoms (`abc` for example), booleans (`'true` for example) and strings (`"hi"` for example), are also objects that communicate with one another by message-passing via streams as well as abstract objects. These objects are called *primitive objects*. Classes, mentioned later, are among the primitive objects.

For example, specifying integer `5` in a program implies an outlet toward an integer object `5`. If an addition message `add(1, ^Sum)` is sent to this outlet, an integer object `6` is created as the operation result ahead of `Sum`.

1.3.6 Classes

A *class* defines a template of its instances, which consists of the following:

- a set of super classes,

- a set of inlet and outlet slot names, and
- a set of methods.

Classes are primitive objects with stream interface. When a class receives an instance creation message, it creates an instance according to its own template. There is no notion of a meta-class which is a class of classes.

1.3.7 Inheritance

A'UM supports multiple class inheritance for the purpose of minimizing the size of program code. One class can inherit a set of methods from any number of classes. Class inheritance expands the set of applicable methods and the set of accessible slots, but does not create a ıy extra instance for any of the super classes. Note that class inheritance is a matter orthogonal to stream computation.

1.3.7.1 Inheritance Tree

For each class, its super class definition forms an inheritance tree. An inheritance list is generated by traversing this inheritance tree in left-to-right and depth-first order. If one class appears on different paths in an inheritance tree, only the first occurrence is taken and the rest are ignored, so that duplication of a class is eliminated.

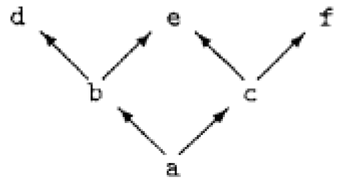


Figure 1.15: Inheritance tree

For example, as shown in Figure 1.15, let seven classes, a to g, keep an inheritance relation such that a inherits b and c; b inherits d and e; c inherits e and f.

For this inheritance tree, the following inheritance list is generated:

$$\texttt{a} \rightarrow \texttt{b} \rightarrow \texttt{d} \rightarrow \texttt{e} \rightarrow \texttt{c} \rightarrow \texttt{f}$$

To any generated inheritance list, we supplement two generic classes: *extrasuper class* (`$object`) at the top and *infra class* (`$OBJECT`) at the bottom as follows:

$$\texttt{\$OBJECT} \rightarrow \texttt{a} \rightarrow \texttt{b} \rightarrow \texttt{d} \rightarrow \texttt{e} \rightarrow \texttt{c} \rightarrow \texttt{f} \rightarrow \texttt{\$object}$$

The resulting list, called the *complete inheritance list*, will be the entire space for searching methods. Even if a class has no super class definition, it is supplemented by the extrasuper class and the infra class, so any class inherits some class.

1.3.7.2 Extrasuper Class ($object)

The extrasuper class is positioned at the end of the inheritance list to define a set of common methods which may be overwritten, such as a method to terminate an object when its interface stream is closed, and methods for error-handling.

1.3.7.3 Infra Class ($OBJECT)

The infra class is positioned at the beginning of the inheritance list to define a set of common methods which cannot be overwritten, such as methods for slot access and a method for the termination message, `$terminate`.

1.3.7.4 Method Search

Just after an object observes some event in the passive phase, it searches a method appropriate for the event, and it then enters the active phase. Message senders can specify from which class in the complete inheritance list the destination object should search a method. Without any specification, method search is done from the infra class.

1.4 Basic Grammar

Since a concurrent system is modeled as a poset of events, concurrent programming is drawing a graph of events. Making it easy to draw a graph is the most important role of concurrent programming languages. Although a graph representation itself may be one kind of language, we define a symbolic language here, since it is easier.

First in this section, we define a basic grammar in which the above computation model can be represented in a straightforward way. Later in the next section, the grammar is extended with several linguistic supports, so that we can write safe and compact programs more easily.

1.4.1 Class Definition

A program consists of classes. Each class is defined by a class name, super classes, inlet and outlet slot names, and methods.

```
<ClassDefinition> ::=
    class <ClassName> "."
        [ <SuperclassDefinition> "." ]
        [ <InletSlotDefinition> "." ]
        [ <OutletSlotDefinition> "." ]
        { <Method> "." }
    end "."

<SuperclassDefinition> ::=
    super <SuperClassName> { ","<SuperClassName> }
<InletSlotDefinition> ::=
    in <SlotName> { ","<SlotName> }
<OutletSlotDefinition> ::=
    out <SlotName> { ","<SlotName> }
```

1.4.2 Method Definition

A method consists of two parts: the *passive part* to define what event should be observed, and the *active part* to define what actions should be taken for the event.

<Method> ::= *<Event>* "|" *<Actions>* { ","*<Actions>* }

<Actions> ::= *<NilExpression>*

1.4.3 Stream Variables

The direction of a stream is specified as occurrences of a pair of stream variables defined as follows:

Input variable: a variable name preceded by "^" (`^X` for example) denoting the inlet.

Output variable: just a variable name (`X` for example) denoting the outlet.

1.4.4 Functional Grammar

The passive and active parts of a method are both defined using functional expressions listed in Table 1.1.

Table 1.1: Basic 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)`	*<Out>* "\" *<In>* `Z \ ^X`	*<Out>* `Y`
`descend(^X,S)`	"<==" *<In>* `<== ^X`	*<Nil>*

Each expression represents either an inlet, an outlet or a nil as its result; it is called either an *inlet expression*, an *outlet expression*, or a *nil expression* according to the result. Any complicated graph can be drawn simply by constructing these expressions.

For example, `C:up:up:up:show(^U)` is composed of message sending expressions. The first `C:up` sends message `up` to outlet `C` and represents the outlet (`C1`) of the following stream as its result, so the original expression is rewritten to `C1:up:up:show(^U)`. By repeating this, we see that the outlet left after sending message `show(^U)` be the result of the original expression.

Merge and append expressions are designed so that a stream tree can be written easily. For example, `A = ^B = ^C = ^D` is composed of three merge expressions. For the first merging joint, `A = ^B`, `A` is the outlet of the outgoing stream, `^B` is the inlet of one of the two incoming streams, and the expression represents the outlet of the other incoming stream, here temporarily named `W`. Then the original expression is rewritten to `W = ^C = ^D`. By repeating this, the original expression represents the outlet of one stream incoming to the last merging joint. One may imagine that there is a cable along the equal operators toward `A` and that three other lines plugged into the cable and messages are flowing from each line toward `A`. Append expressions are similar except that messages from a plug to the left flow before any of those from a plug to the right.

Since stream connection is a special case in that one of the incoming streams is closed for either `merge` or `append`, it is represented as a combination of a merge (or append) expression and a close expression. For example,
`Z = ^X ::` means connecting an inlet `^X` to an outlet `Z`, which will be simplified to `Z = ^X` by a linguistic support of implicit completion mentioned later.

1.4.4.1 Right-to-Left Principle

All the pictures in this manual are drawn in a manner that *messages should flow from right to left* toward the leftmost object. Changing the viewpoint to the arriving time of messages, the manner can be taken as that *time should go by from left to right.* Expressions are designed to keep this manner, so that we can write a program like drawing a picture.

1.4.4.2 Derivation of Nils

To promote the completion of computation, the grammar prescribes the following two rules:

R1 (Pairing of stream variables): Each stream variable must make a pair, that is, an inlet variable cannot appear alone without its corresponding outlet variable and vice versa.

R2 (Allowing only nil expressions at the top): It allows only an *nil expression* to be specified for each top-level expression in the < *Action* > field, so that no outlet be left open and no inlet be left unconnected.

In the extended grammar, described later, these rules will be abolished.

1.4.4.3 Primitive Objects

The occurrence of a lexicon corresponding to a primitive object denotes an outlet to the object. For example, `5` means an outlet to an integer object, 5, and `##counter` means an outlet to a class object, `counter`.

1.4.4.4 Common Messages

There are several kinds of *common messages* which can be accepted by any objects including primitive objects. For abstract objects, default methods for these common messages are defined either in the extrasuper class or in the infra class. Here, we introduce the most characteristic one among them.

"Who are you?" message: `who_are_you(Who)` is a message to ask an object of its *denotational image*. When an object receives this message, the object will send back to the terminal, `Who`, a message representing its own denotational image. For the *denotational image*, anything is good if it can be recognized as the object itself.

For example, an integer object, `5`, will send back a message, "5", to the terminal `Who` and close the end. Note that the latter "5" is a message, not an object. For an abstract object like a counter object, the default method is defined so that the object should send back a message telling "I am I", that is `i_am(^Obj)` where `^Obj` is an inlet toward the object.

Example 1 (Counter)

Figure 1.16 shows a program of a counter written in the basic grammar.

A counter is an object which increments or decrements its counting value every time it receives a message, *up* or *down*. The counting value is updated or referred to by a message `set` or `show`. We try two kinds of test using the counter: `testM` and `testA`. Both methods:

1. create an instance of class `counter`,
2. send message `set(5)` to the instance,
3. split into two the stream remaining left after sending the set message,
4. send two up messages and one `show(^U)` message in a row to one branch, and
5. send two `down` messages and one `show(^D)` message in a row to the other branch.

The difference between the two methods is on the ordering of up and `down` messages as follows:

- In `testA`, the two branch streams, `C1` and `C2`, are appended, so it is sure that the two up messages from `C1` arrive at the counter earlier than any of the two `down` messages from `C2`. Therefore, counting result `^U` will be exactly `7` and `^D` be `5`.
- In `testM`, messages from the two branches are merged, so it is indeterministic how many `up` messages will arrive at the counter earlier than `down` messages. Therefore, counting result `^U` will be either `5`, `6` or `7`, and `^D` be either `3`, `4` or `5`.

Figure 1.17 shows the situation that the first generation of the counter object received a message `set(5)` and has just entered the active phase. Figure 1.18 shows the same program written in the extended grammar, which will be described in the next section.

```
class   counter.
     out n.
  :up = Rest       | <== ^Self,
                     Self:get_outlet(n, ^N)
                          :set_outlet(n, N1) = ^Rest ::,
                     N:add(1, ^N1):: .
  :down = Rest     | <== ^Self,
                     Self:get_outlet(n, ^N)
                          :set_outlet(n, N1) = ^Rest ::,
                     N:sub(1, ^N1):: .
  :set(^N) = Rest  | <== ^Self,
                     Self:set_outlet(n, N) = ^Rest :: .
  :show(N) = Rest  | <== ^Self,
                     Self:get_outlet(n, ^N) = ^Rest :: .
  ::               | <== ^Self,
                     Self:'$terminte':: .
end.

class   test.
  :test = Rest         | <== ^Self,
                         Self:testM(^Um,^Dm):testA(^Ua,^Da)
                              :nop(^Result) = ^Rest ::,
                         Result\ ^WUm\ ^WDm\ ^WUa\ ^WDa ::,
                         Um:who_are_you(WUm)::,
                         Dm:who_are_you(WDm)::,
                         Ua:who_are_you(WUa)::,
                         Da:who_are_you(WDa):: .
  :testM(U, D) = Rest  | <== ^Rest,
                         ##counter:new(^Counter)::,
                         Counter:set(5) = ^C ::,
                         C1:up:up:show(^U)::,
                         C2:up:up:show(^D)::,
                         C = ^C1 = ^C2 :: .
  :testA(U, D) = Rest  | <== ^Rest,
                         ##counter:new(^Counter)::,
                         Counter:set(5) = ^C ::,
                         C1:up:up:show(^U)::,
                         C2:up:up:show(^D)::,
                         C \ ^C1 \ ^C2 :: .
  :nop(Result) = Rest  | <== ^Rest,
                         ##sink:new(^Result)::.
end.
```

Figure 1.16: Program of a counter in the basic grammar

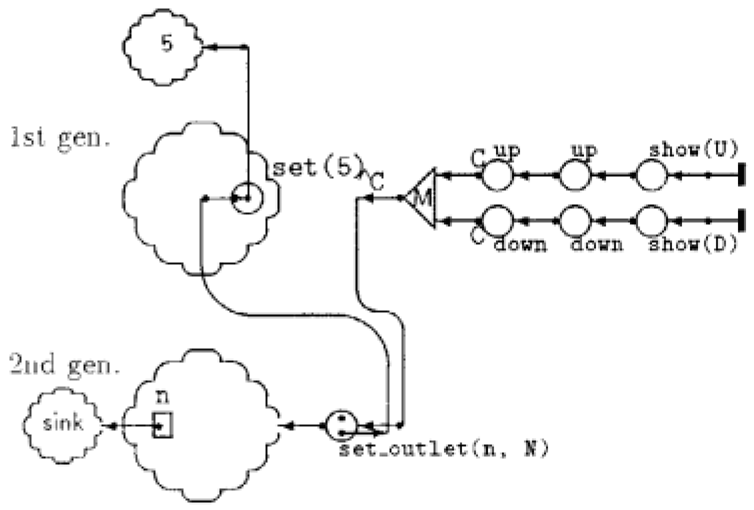


Figure 1.17: First generation of a counter

```
class   counter.
     out  n.
  :up    -> !n + 1 = !n.     % X+Y => S ; X:add(Y,^S)
  :down -> !n - 1 = !n.      % X-Y => D ; X:sub(Y,^D)
  :set(^N) -> N = !n.        % !n => $self:set_outlet(n,N)
  :show(N) -> !n = ^N.       % !n => $self:get_outlet(n,^N)
end.

class  test.
  :test -> :testM(^Um, ^Dm):testA(^Ua, ^Da):nop(^Result),
           Result$1 = (Um?), Result$2 = (Dm?),
           Result$3 = (Ua?), Result$4 = (Da?).

  :testM(U, D) ->
           #counter:set(5) = ^C, C:up:up:show(^U),
                                 C:down:down:show(^D).
  :testA(U, D) ->
           #counter:set(5) = ^C, C$1:up:up:show(^U),
                                 C$2:down:down:show(^D).
  :nop(Result) -> .
end.
```

Figure 1.18: Program of a counter in the extended grammar

1.5 Extended Grammar

Although the stream computation model can be directly represented in the basic grammar, in order to write large scale concurrent problems, simpler and more abstract description is needed. For ease of programming, we introduce the following four main extensions:

- implicit completion of streams,
- introduction of macro expressions,
- extension of the meaning of a variable from a stream to a channel, and
- introduction of volatile objects.

1.5.1 Implicit Completion of Streams

Section 1.4.4.2 prescribed two rules to promote the termination of computation. Rule R1 is abolished by the introduction of channel variables described later. For rule R2, it is laborious for the programmer to carefully complete all the streams, and in particular to be sure to close outlets. So we abolish rule R2 but instead extend the grammar as follows:

Allowing inlet and outlet expressions at the top: Inlet and outlet expressions may be specified in the $<Action>$ field, at the top level of a method.

Implicit completion: Non-nil expressions left alone are implicitly completed. Outlet expressions are closed, and inlet expressions are discharged. Also a default method for termination is given.

1.5.1.1 Implicit Closing of Outlets

Outlets left open are implicitly closed. Among those implicitly closed are results of outlet expressions left at the $<Action>$ field, missing outlets of channels, current outlet slots to be updated, outlet slots at termination, and outlets of inlet slots at initiation.

1.5.1.2 Implicit Discharging of Inlets

Inlets left unconnected are implicitly discharged. Among those implicitly discharged are results of inlet expressions left at the $<Action>$ field, missing inlets of channels, the interface stream at termination, inlet slots at termination, and inlets of outlet slots at initiation.

1.5.1.3 Implicit Termination of Objects

Most objects are supposed to terminate when closing of their interface stream is detected, but we often forget to specify this termination. Therefore,

```
:: -> .
```

is defined as a default method for termination, in the extrasuper class which is inherited by any user-defined classes.

1.5.2 Macro Expressions

Various macro expressions are provided; they represent either an inlet, an outlet or a nil as their result.

1.5.2.1 Arithmetical/Logical Operation Macros

Arithmetical and logical operation macros are provided to handle primitive objects easily. Each macro of this kind represents the outlet of a stream toward the operation result, so that any complicated operational expression can be represented simply by constructing these arithmetical and logical operation macros similar to basic expressions.

For example, expression `3 + 5 == 8` is expanded to

```
3 :add(5, ^Sum), Sum:eq(8, ^True),
```

and represents the outlet `True` of a stream toward a boolean object, `'true`.

1.5.2.2 Instance Creation Macro

Instance creation macro sends an instance creation message to the specified class and represents the outlet of a stream toward the newly created instance.

For example, `#counter` is expanded to

```
##counter:new(^Counter),
```

and represents the outlet, `Counter`, to the instance of class `counter`. Where `##counter` denotes the class `counter` which is a primitive object.

1.5.2.3 Slot Access Macro

For case of accessing to slots, macro "`@`"< *SlotName* > is provided for inlet slot access, and macro "`!`"< *SlotName* > for outlet slot access. Their meanings are context dependent.

The outlet slot access macro means slot-reference in the outlet position, and slot updating in the inlet position. For example,
`!n = ^N` meaning outlet-slot-reference is expanded to:

```
$self:get_outlet(n, ^Slot), Slot = ^N
```

`N = !n` meaning outlet-slot-updating is expanded to:

```
$self:set_outlet(n, Slot), N = ^Slot
```

In contrast, the inlet slot access macro means slot reference in the inlet field, and slot-updating in the outlet field. For example,
`N = @n` meaning inlet-slot-reference is expanded to:

```
$self:get_inlet(n, Slot), N = ^Slot
```

`@n = ^N` meaning inlet-slot-updating is expanded to:

```
$self:set_inlet(n, ^Slot), Slot = ^N
```

1.5.2.4 Pseudo Variable `$self`

The object *itself*, that is a stream to its next generation, is accessed by a pseudo variable `$self`, whose meaning is also context dependent and similar to that of the outlet slot macro as follows:

- `$self` appearing in the outlet position means referring to the object *itself*.

  When the destination is omitted in message-sending, it is assumed that `$self` would have been specified, so the message is sent to the object *itself*.

- `$self` appearing in the inlet position means updating the object *itself*.

Causality on Self: Which generation the pseudo variable `$self` denotes is determined in accordance with the order of occurrences of the pseudo variable `$self` (and macros whose expansions include `$self`). Causality on the object *itself* depends on the order of expanding expressions, as follows:

- Methods are expanded *from the event to the actions*.
- A set of actions, separated by commas, are expanded *from left to right*.
- Parameters of each message are expanded from *left to right*.
- Macro expressions are expanded *from left to right* and *from inner to outer*.

For example,

```
:foo(!a, ^X) -> :foo1(!b, !c, ^Y) = $self,
                $self:foo2(Y, ^Z) = $self,
                X + Z = !d.
```

is equivalent to the following method:

```
:foo(A, ^X) -> $self:get_outlet(a, ^A)
                    :get_outlet(b, ^B)
                    :get_outlet(c, ^C)
                    :foo1(B, C, ^Y)
                    :foo2(Y, ^Z)
                    :set_outlet(d, ^Sum) = $self,
               X:add(Z,^Sum).
```

1.5.2.5 Generation Descending Macros

For ease of writing generation descending, the following two macros: *succession macro* and *termination macro* are provided.

<*Method*> ::= <*DescendingMacro*> <*Action*> { "," <*Action*> }
<*DescendingMacro*> ::= <*Succession*> | <*Termination*>
<*Succession*> ::= <*EventOnly*> "->"
<*Termination*> ::= <*EventOnly*> "-|"
<*EventOnly*> ::= ":" <*MessagePattern*> | "::"

Succession macro: Succession macro means descending a generation: connecting the inlet of the interface stream following the received message to the latest outlet to the next generation.

For example, method `:m -> do.` is expanded as follows:

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

Termination macro: Termination macro means descending a generation; sending a termination message, `$terminate`, to the object *itself* as the last message from this generation; connecting the inlet of the interface stream following the received message to the outlet left after the previous sending.

For example, method `:m -| :do.` is expanded as follows:

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

1.5.2.6 Destination Updating Macro

For message-sending expressions whose destination is related to the object *itself*, such as slots and `$self`, their destination is implicitly updated. For example,

```
:foo -> !a:m(^X).
```

is supplemented by the updating of an outlet slot named a as follows:

```
:foo -> !a:m(^X) = !a.
```

Owing to this macro, the generations of an object and slots are kept in accordance with the order of their occurrences, even if those self-related expressions are separated into pieces within a context. For example,

```
:foo -> !a:m1(^X), ... (other actions) ..., !a:m2.
```

is expanded as follows:

```
:foo -> !a:m1(^X) = !a, ... (other actions) ..., !a:m2 = !a.
```

so that the message `m1` is sent to the outlet slot named a earlier than the message `m2`.

Note that the destination update macro is applied not only for self-related message-sending expressions explicitly specified in the program, but also for some macro expressions which should be expanded to self-related message-sending expressions.

For example, `!n + 1 = !n` is expanded first to:

```
!n:add(1, ^Sum), Sum = !n
```

then, by applying the destination updating macro, it becomes:

```
!n:add(1, ^Sum) = !n, Sum = !n
```

1.5.3 Channel Variables

In the above sections, we used variables to represent streams that are chains of messages. Here, we extend the meaning of a variable from a stream to a channel which is a stream tree, so that a channel can be represented just by occurrences of variables, rather than by explicitly specifying merge and append expressions.

A channel is represented by channel variables defined as follows:

One or fewer inlet variables: The occurrence of a variable name preceded by "^" (`^X` for example) denotes an inlet.

Zero or more non-ordered outlet variables: The occurrence of a variable name only (`X` for example) denotes one of the outlets of a merge tree, whose root has the above as its inlet.

Zero or more ordered outlet variables: The occurrence of a variable name succeeded by "`$`" and an ordering number (`X$1` for example) denotes an outlet of an appending joint, where suffixes may be any positive numbers and do not have to begin with `1` such as `X$1`, nor to be continuous such as `X$2`, `X$3`, `X$4`.

Multiple occurrences of an ordered outlet variable with the same ordering number, such as two occurrences of `X$3`, make a merge tree for themselves.

When non-ordered outlet variables and ordered outlet variables appear together, the stream tree formed by the ordered outlet variables is taken as a subtree plugged into the entire merge tree.

For example, the occurrences of one inlet variable `^X`, two non-ordered outlet variables `X`s, and four ordered outlet variables, `X$1`, `X$2` and two `X$3` represent a stream network expressed by `X = ^P1 =^P2 \ ^S1 \^S2 \^S3, S3 = ^S31 = ^S32` as shown in Figure 1.19.

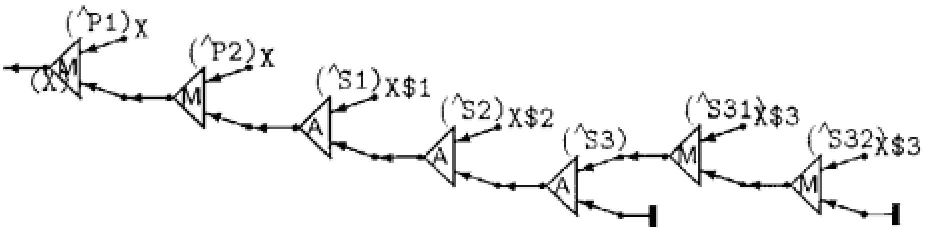


Figure 1.19: Channel variables

Owing to this extension, most of the stream programming features are hidden and we can concentrate on what to do with objects rather than how to connect streams, so that the programming paradigm of $\mathcal{A}'\mathcal{UM}$ gets close to that of other object-oriented languages.

1.5.4 Volatile Objects

Each generation of an object (1) is activated by a certain event, then takes actions according to the event, and (2) may descend to its next generation simultaneously. The former means conditional selection and the latter conditional repetition. As objects are natively condition testers, conditional selection and repetition can be written by defining a class for each condition.

If there are a number of conditional selections and repetitions, however, many small classes will have to be defined and the program context will be scattered into pieces. In addition, it must be quite burdensome for the programmer to pass a variable environment to each condition tester.

Here, we introduce the notion of a *volatile object* which is provided as a linguistic support so that conditional selection and repetition can be programmed easily.

Volatile classes: A *volatile class* is defined within a method. There is no distinction between *external classes*, such as class `counter`, and *volatile classes* except for the following points:

- External classes have their own class names, so any other classes including volatile classes can access and inherit them by specifying their names.

- Volatile classes do not have their own class names, so they cannot be referred to or inherited from anywhere.

Any number of volatile classes can be defined within a method and they can be nested.

Volatile classes are advantageous not only for making programming easy, but also for giving more chances of optimization in the implementation, since they are not accessed in public.

Volatile objects: A *volatile object* is an instance of a volatile class.

Creator objects: An object which creates a volatile object (calls for execution of a method defining the volatile class) is called a *creator object.* When definitions of volatile classes make a nest, every volatile object will be the creator of its inner volatile objects, if any. Each volatile object can access its creator object via a stream named `$creator`.

Causality on creator: Section 1.5.2.4 ruled that the causality on the generation of an object should be kept in accordance with the order of occurrences of self-related actions. When volatile object creation expressions are contained, the causality on the creator's generation is kept as follows:

- When a volatile object is created, a stream to the object itself is split into two to make an append joint; one branch is given to the volatile object and accessed by the name, `$creator`; the other branch is given to the creator object.

- When the volatile object is terminated, it closes the stream, `$creator`, as well as other slots it may hold.

Thus, messages which the volatile object sends to `$creator` will arrive at the creator object earlier than any of those which the creator sends to itself syntactically after the volatile object creation expression.

There are two kinds of volatile objects: *immutable volatile objects*, used mainly for conditional selection like an IF statement, and *mutable volatile objects*, used mainly for conditional repetition like a LOOP statement. Before going into detail, we explain how to define volatile classes, as this is common to both kinds.

1.5.4.1 Creation of Volatile Objects

Volatile creation expression consists of the specification of the interface stream of a volatile object and the definition of a volatile class. This expression is a nil expression.

< *VolatileObjectCreation* > ::=
  < *ImmutableVolatileObjectCreation* > | < *MutableVolatileObjectCreation* >
< *ImmutableVolatileObjectCreation* > ::=
  < *Interface* > "?" < *ImmutableVolatileClassDefinition* >
< *MutableVolatileObjectCreation* > ::=
  < *Interface* > "=>" < *MutableVolatileClassDefinition* >

< *Interface* > ::= < *InletExpression* > | < *OutletExpression* >

< *ImmutableVolatileClassDefinition* > ::=
  "(" [< *SuperclassDefinition* > ";"]
    < *Method* > { ";"< *Method* > } ")"
< *MutableVolatileClassDefinition* > ::=
  "(" [< *SuperclassDefinition* > ";"]
    [< *InletSlotDefinition* > ";"]
    [< *OutletSlotDefinition* > ";"]
    < *Method* > { ";"< *Method* > } ")"

Basic: (Inlet as the interface) If an inlet is specified in the < *Interface* > field, the volatile object takes the inlet as its interface stream.

For example, a volatile immutable object created by

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

receives a message, `'true` or `'false`, from inlet `^Hunger`, and correspondingly sends a message, `eat` or `sleep` to the creator.

Extension: ("Who are you?" to Outlet) Most macros, especially arithmetical and logical operation macros, represent an outlet as their result, and most conditional selections are written for arithmetical or logical operation results. Therefore, the above rule is extended for ease of writing conditional selection as follows:

If an outlet is specified in the < *Interface* > field, the message `who_are_you(Who)` is implicitly sent to the outlet and the volatile object takes inlet `^Who` as its interface stream. For example,

```
(X > Y) ? ( :'true -> X = ^Max ;
            :'false -> Y = ^Max )
```

is for getting the maximum of `X` and `Y`. For the comparison, `X > Y`, a boolean object, either `'true` or `'false`, is produced, which is asked "Who are you?" as follows:

```
(X > Y):who_are_you(Who),
^Who ? ( :'true -> X = ^Max ;
         :'false -> Y = ^Max )
```

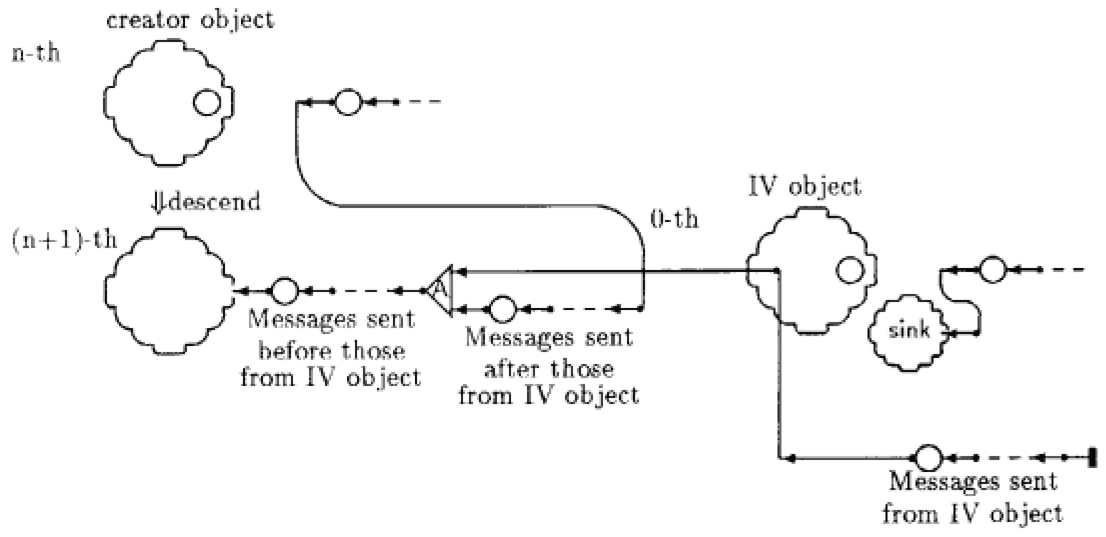

creator object
n-th
⇓descend
(n+1)-th
0-th
IV object
A
sink
Messages sent before those from IV object
Messages sent after those from IV object
Messages sent from IV object

Figure 1.20: Immutable volatile object and its creator object

1.5.4.2 Immutable Volatile Objects

As shown in Figure 1.20, an *immutable volatile object* (*IV object* for short) is an object which has a single generation and shares the same name scope of variables with its creator object.

Single generation: An IV object terminates its own life after being activated by an event on the interface stream. The rest of the interface stream is connected to a sink object.

Trasparent scope: For an IV object, the name scope of variables is the same as that for its creator object. The same variable names appearing both in an IV object and in its creator object represent an identical stream or channel.

Creator as itself: In addition, the creator object seems to be identical to the IV object *itself*. Pseudo variables, `$self` and `$creator`, are used for the same meaning; those messages which an IV object sends to *itself* are sent to its creator; slots which an IV object regards as its own are those of its creator object.

1.5.4.3 Mutable Volatile Objects

As Figure 1.21 shows, a *mutable volatile object* (*MV object* for short) is an object which may have multiple generations for it own and has its own scope. MV objects are almost the same as external objects except that their classes do not have their own class names.

Multiple generations: An MV object has its own life, which may consist of multiple generations.

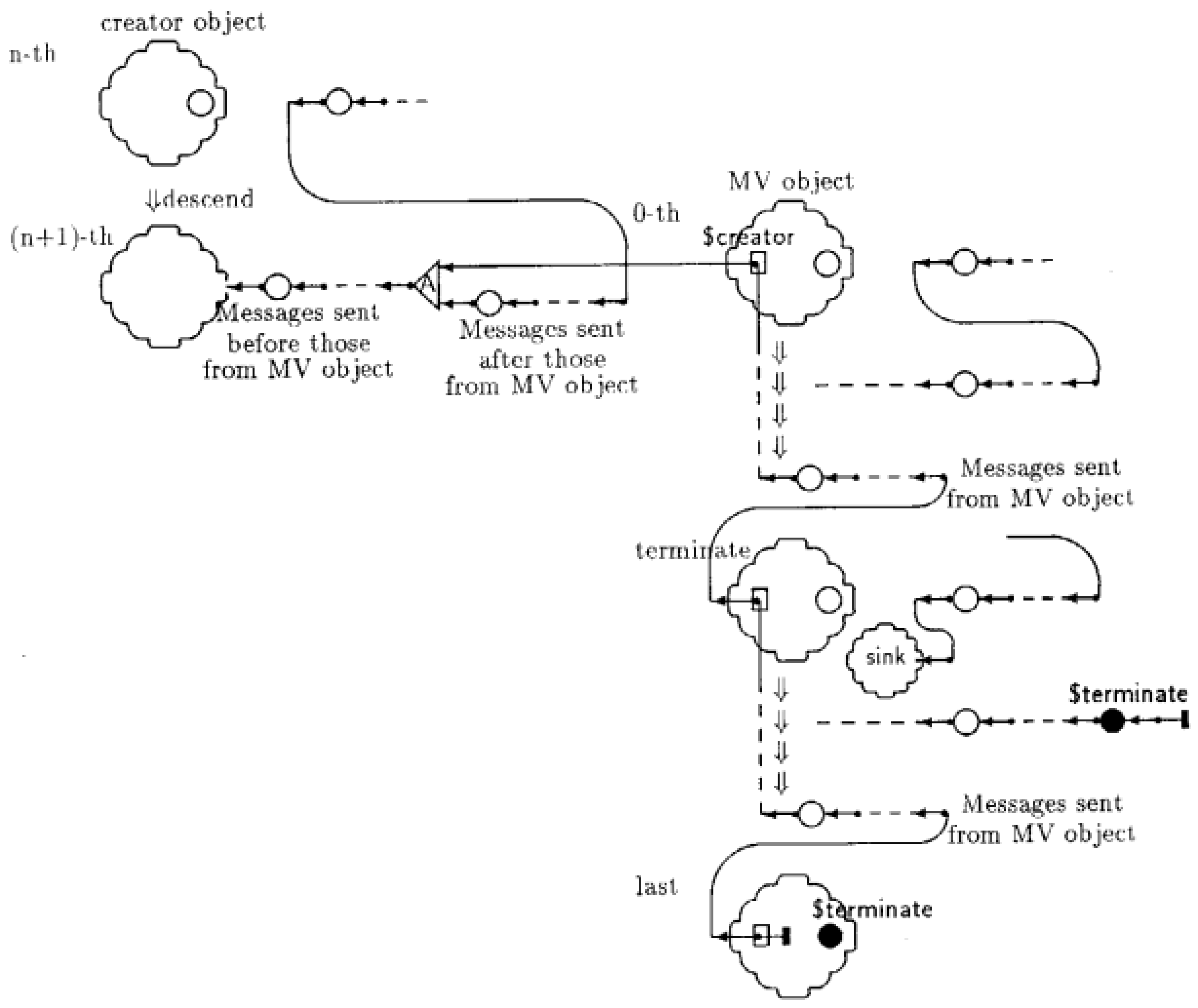

creator object
n-th
⇓descend
(n+1)-th
Messages sent before those from MV object
Messages sent after those from MV object
0-th
MV object
$creator
Messages sent from MV object
terminate
sink
$terminate
Messages sent from MV object
last
$terminate

Figure 1.21: Mutable volatile object and its creator object

```
:sum(Ns, Sum) ->                                          % 1
     Loop:initialize(^Sum) = ^Ns,                         % 2
     ^Loop => (  in  sum;  out  temp;                     % 3
         :initialize(Sum) -> @sum = ^Sum, 0 = !temp ;     % 4
         :n(^N) -> !temp + N = !temp ;                    % 5
         :: -> !temp = @sum  ).                           % 6
```

Figure 1.22: An example of an MV object

Independent scope: An MV object has its own scope, which is independent of that of the creator object. Even if the same variable names appear inside and outside of the MV object definition, they denote different channels.

It is easy to see why the name scope of an MV object is independent by remembering how an external object is defined. The same variable names appearing in different methods have no relation, since variable names are valid only in one generation and each method implies an independent generation.

Creator as a slot: An MV object can hold a set of slots for its own. For an MV object, the creator object is an outlet slot among these slots, and is associated with the pseudo variable name, `$creator`. The pseudo variable, `$self`, denotes the MV object *itself*. Therefore, `$creator` and `$self` give different meanings.

For example, Figure 1.22 shows a method to sum up all numbers received from `Ns`, to `Result`. The MV object (lines 3 to 6) has one inlet, `sum`, which is thought of as the total sum, and one outlet, `temp`, which keeps a temporary sum during computation. The MV object first receives a message, `initialize`, to initialize these slots, and then receives one number (`X` contained in a message `n(^X)`) after another until no more message is coming, when the temporary sum, `temp`, is the desired total sum, `sum`.

Example 2 (Prime Number Generator)

Figure 1.23 shows a program using volatile objects which generates prime numbers up to the given maximum number according to the following algorithm:

1. Send 2 as the first prime, then generate a sequence of odd numbers starting with `3`. What is obtained by passing this sequence through a pipeline of sieves is the prime numbers.

2. Every time a new prime number is found, one sieve for the prime number is created and appended to the pipeline. At first, a sieve for `3` is created.

3. Every sieve tries to divide each incoming number by its own prime number.

    - If the incoming number is a multiple of the prime number, do nothing.
    - If not, check if the sieve's prime number is the biggest at the moment.
        - If so, the incoming number is found to be a prime number. Create a sieve for this number and take it as the *next* sieve.
        - If not, pass the incoming number to the next sieve.

    Repeat until there is no more incoming number.

An MV object, temporarily named *M1*, is defined between line 4 and line 20 of class `sift`. Inside, an IV object, temporarily named *I1*, is defined between line 9 and line 19. Further inside, another IV object, temporarily named *I2*, is defined between line 12 and line 18. Variables `V`, `Ns`, `Ps` appearing between line 2 and line 3 outside M1 are independent of those appearing between line 4 and line 20 inside M1, so they are passed to the inside via message `initialize`. Slot accesses at lines 7, 9, 14, 15 and 17 are on the slots of the outermost object sharing the scope: the MV object, M1.

```
class   prime.                                                    % 1
  :primes(^Max, ^Ps) ->                                           % 2
      3 = ^X,                                                     % 3
      :generate(X, Max, Ns),                                      % 4
      #sift:do(X, ^Ns, Ps:n(2):n(X) ).                            % 5
  :generate(^X, ^Max, ^Ns) ->                                     % 6
      ( (X+2 = ^NewX) < Max ) ? (                                 % 7
          : 'true  ->                                             % 8
               :generate(NewX, Max, Ns:n(NewX) ) ;                % 9
          : 'false ->         % end %                             %10
      ).                                                          %11
end.                                                              %12

class   sift.                                                     % 1
  :do(^V, Ns, ^Ps) ->                                             % 2
      S:initialize(V, Ps) = ^Ns,                                  % 3
      ^S => (                                                     % 4: (M1
              out  me, next, to_next, primes ;                    % 5
        :initialize(^V, ^Ps) ->                                   % 6
            V = !me, 0 = !next, Ps = !primes ;                    % 7
        :n(^X) ->                                                 % 8
            ( (X mod !me) == 0 ) ? (                              % 9: (I1
                : 'true  ->  ;                                    %10
                : 'false ->                                       %11
                     ( !next == 0 ) ? (                           %12: (I2
                          : 'true  ->                             %13
                               X = !next, Ns = !to_next,          %14
                               #sift:do(X, ^Ns, !primes:n(X));    %15
                          : 'false ->                             %16
                               !to_next:n(X)                      %17
                     )                                            %18: I2)
            )                                                     %19: I1)
      ).                                                          %20: M1)
end.                                                              %21

class  test.
    :test -> #prime:primes(20, Ps), :nop(^Res).
    :nop(Res) -> .
end.
```

Figure 1.23: Program of a prime number generator

Figure 1.24 shows the state immediately after a sieve for prime number 5 has been created.

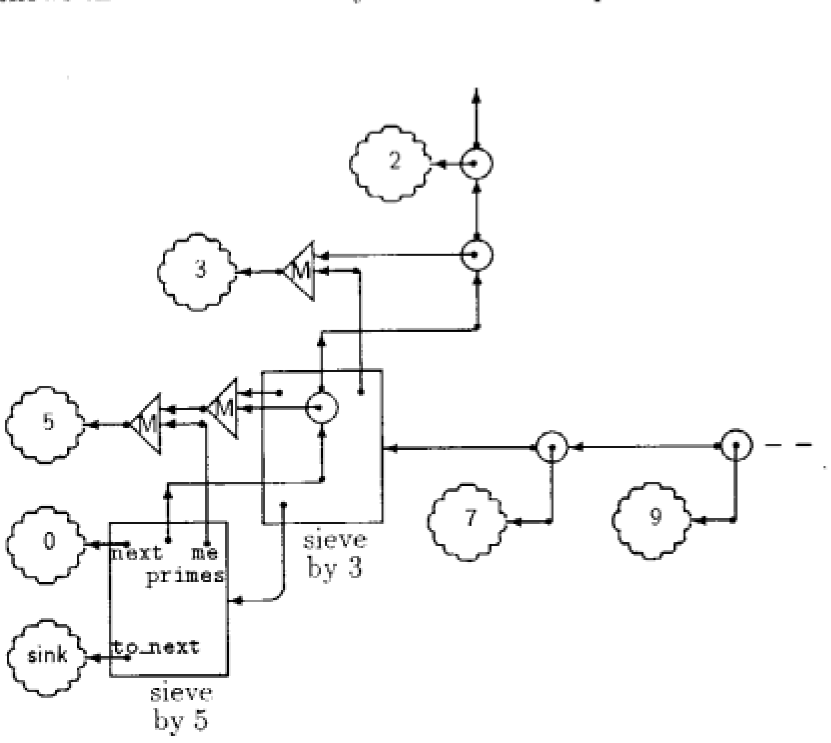


Figure 1.24: Sieves generating prime numbers

Chapter 2

Experimental $\mathcal{A}'\mathcal{UM}$ System: $\mathcal{XAS}$

2.1 Introduction

$\mathcal{XAS}$ is an experimental system developed to to execute $\mathcal{A}'\mathcal{UM}$ programs on top of PDSS (PIMOS Development Support System)[PDSS89] which is a KL1 system installed onto various UNIX[1] systems.

As Figure 2.1 shows, $\mathcal{XAS}$ consists of the following facilities:

- **Tiny shell:** provides commands to set up a directory environment and to compile and execute $\mathcal{A}'\mathcal{UM}$ programs.
- **Compiler:** compiles $\mathcal{A}'\mathcal{UM}$ programs through KL1-C code into KL1-B code and loads the generated KL1-B code into memory.
- **Program executer:** executes an $\mathcal{A}'\mathcal{UM}$ program.
- **Runtime environment:** provides a runtime environment necessary to execute $\mathcal{A}'\mathcal{UM}$ programs, including $\mathcal{A}'\mathcal{UM}$ primitive classes.
- **PDSS interface:** transfers PDSS commands issued from $\mathcal{XAS}$ to PDSS.

$\mathcal{XAS}$ is entirely written in KL1, runs on top of PDSS, and uses for its I/O functions those I/O facilities provided by PDSS.

$\mathcal{XAS}$ has no debugging environment for its own yet. Debugging is available only at the level of KL1 by using the KL1 debugger provided by PDSS.

[1]UNIX is a trademark of Bell Laboratories.

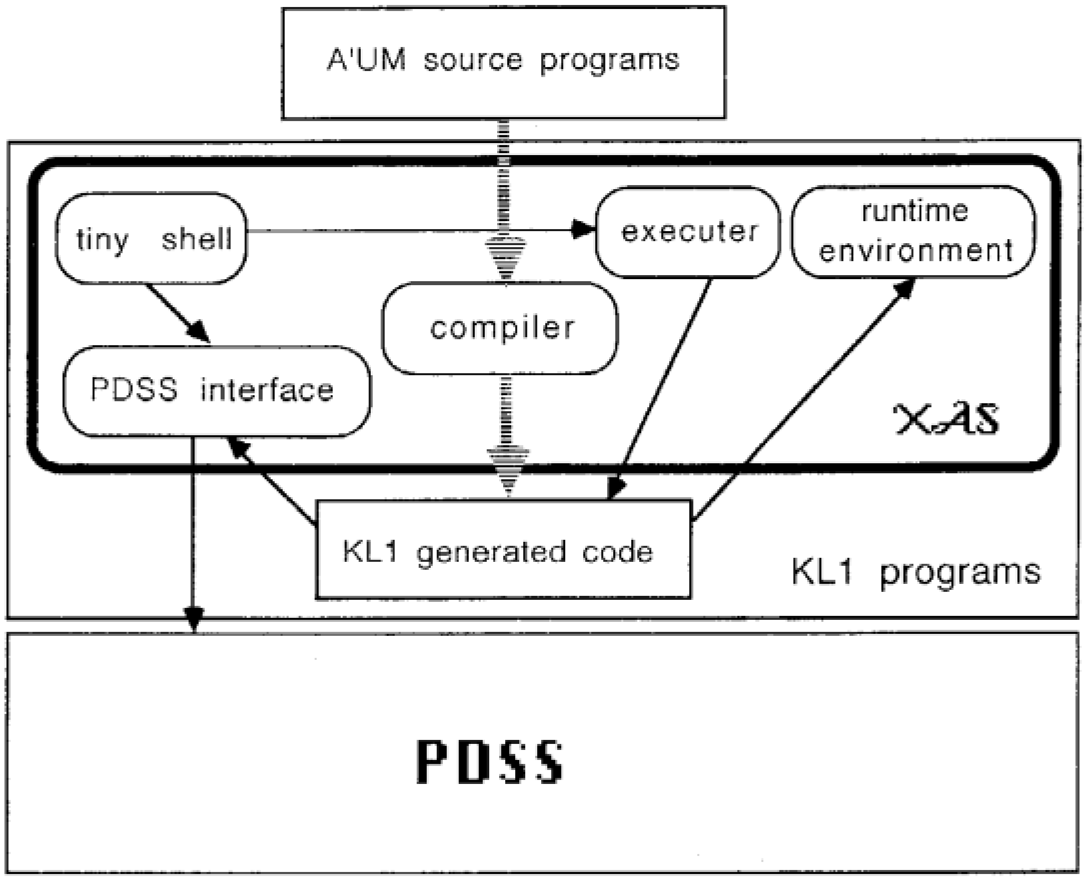


Figure 2.1: *XAS* configuration

2.2 Installation

𝒳𝒜𝒮 is distributed as a package of KL1 source programs stored in tree-structured directories whose top level directory contains the following files and directories:

1. Files:

`MYPDSS`	csh command file to invoke stand-alone PDSS for *𝒳𝒜𝒮*
`Install`	csh command file to install *𝒳𝒜𝒮*
`Startup`	PDSS command file to load up *𝒳𝒜𝒮* onto PDSS
`IPL`	PDSS command file to initiate *𝒳𝒜𝒮*

2. Directories:

`shell`	modules for the tiny shell
`cmp`	modules for the compiler
`kernel`	modules for the program executer
`primitive`	modules for the runtime environment (primitive classes etc.)
`@sample`	*𝒜′𝒰ℳ* sample programs

𝒳𝒜𝒮 can be installed as follows:

1. Check if PDSS has been already installed on your machine.
2. Restore the distributed tape into your desired directory which is called `XASDIR` hereafter for convenience.
3. Set the current directory to `XASDIR` and execute `Install` as follows:

```
% cd XASDIR  return
% Install  return
```

Now, *𝒳𝒜𝒮* is ready to run on top of PDSS.

2.3 Initiation

2.3.1 Initiation of PDSS

$\mathcal{XAS}$ Version 1.0 runs on top of PDSS Version 2.51 with 500 Kword heap area and 1 Mbyte code area.

!!Caution!! The entire $\mathcal{XAS}$ system cannot be loaded up if the code area is less than 1 Mbyte. Since the default size of code area is much less than 1 Mbyte, PDSS must be invoked with these options modified.

Although PDSS can be invoked both from GNU-Emacs and directly from top-level shell, but the former is preferable since all PDSS functions are available under GNU-Emacs.

2.3.1.1 Initiation of PDSS under GNU-Emacs

From GNU-Emacs, PDSS is initiated by the following command:

```
C-U M-X pdss  return
```

Since a prompt "`PDSS Option?:`" appears in the GNU-Emacs echo area, set the sizes of heap area and code area as follows:

```
PDSS Option?: -h2000000 -c2000000.  return
```

2.3.1.2 Initiation of Stand-Alone PDSS

To initiate stand-alone PDSS with the above options, a csh command file, `MYPDSS`, is prepared under `XASDIR` . With this csh command file, stand-alone PDSS can be invoked from any level of shell as follows:

```
% MYPDSS return
```

For more detail about PDSS options, see [PDSS89].

2.3.2 Initiation of $\mathcal{XAS}$

When PDSS is invoked, a couple of windows are created:

- **PDSS-SHELL window:** to accept PDSS commands
- **PDSS-CONSOLE window:** to display system messages and allow interactions to trace KL1 programs.

Now you are in a PDSS-SHELL window.

1. Change the current directory to the top-level directory, `XASDIR` , as follows:

```
:- cd( XASDIR ).  return
```

2. Load up $\mathcal{XAS}$ as follows:

   :- take("Startup"). *return*

   Note that this procedure is required only for the first time you invoke $\mathcal{XAS}$ in a PDSS session.

3. Enter $\mathcal{A}'\mathcal{UM}$ shell as follows:

   :- take("IPL"). *return*

   Then the following **A'UM-SHELL** window appears:

```
*************************************************************
*                   Welcome to A'UM World!                  *
*                                                           *
*             A'UM Shell: Version 1.0 (01/09/89)            *
*           A'UM Compiler: Version 1.0 (01/09/89)           *
*************************************************************

:-
```

   You are now in $\mathcal{A}'\mathcal{UM}$ shell.

2.4 $\mathcal{A}'\mathcal{UM}$ Shell

Once you enter `A'UM-SHELL` you don't need to go back and forth between `A'UM-SHELL`, `PDSS-SHELL` and `*shell*` windows. All the operations necessary to execute your $\mathcal{A}'\mathcal{UM}$ program can be issued from the `A'UM-SHELL` window. For example, if you compile your $\mathcal{A}'\mathcal{UM}$ program, it will be compiled through KL1-C to KL1-B and loaded into the underlying PDSS system automatically.

Table 2.1 shows the commands available in `A'UM-SHELL` for now.

Table 2.1: $\mathcal{A}'\mathcal{UM}$-shell commands

Directory Commands	
`cd(DirNameStr)`	change directory
`cd_home`	go back to home directory (`XASDIR`)
`pwd`	show current directory
`ls`	listing all
`ls(WildCardStr)`	listing files
$\mathcal{A}'\mathcal{UM}$ Support Commands	
`aumcmp(ListOfAumFileNameStrs)`	compile and load $\mathcal{A}'\mathcal{UM}$ programs
`load_class(ListOfAumClassNameStrs)`	load already compiled $\mathcal{A}'\mathcal{UM}$ classes
`trace_class(ListOfAumClassNames)`	turn on trace-switches of $\mathcal{A}'\mathcal{UM}$ classes
`notrace_class(ListOfAumClassNames)`	turn off trace-switches of $\mathcal{A}'\mathcal{UM}$ classes
`start`	invoke the program executer
KL1 Support Commands	
`kl1cmp(ListOfKL1FileNameStrs)`	compile KL1 programs
`load_kl1(ListOfKL1FileNames)`	load KL1 modules
`trace_kl1(ListOfKL1ModuleNames)`	turn on trace-switches of KL1 modules
`notrance_kl1(ListOfKL1ModuleNames)`	turn off trace-switches of KL1 modules
Others	
`halt`	halt A'UM-SHELL and go back to PDSS-SHELL

2.5 Compilation and Execution of $\mathcal{A}'\mathcal{UM}$ programs

2.5.1 Compilation

The $\mathcal{A}'\mathcal{UM}$ compiler compiles $\mathcal{A}'\mathcal{UM}$ programs through KL1-C code to KL1-B code, and then loads the generated KL1-B code into memory. Here, let us compile sample programs shipped in the $\mathcal{XAS}$ package.

1. Just after you enter the A'UM-SHELL, you are in the home directory, `XASDIR` . You can check where you are as follows:

   :- **pwd.** *return*

   Then the full path name of the current directory is displayed as follows:

   ```
   world = /login/olive/aum
   ```

   In case that you were moving around different directories, you can go back to the home directory as follows:

   :- **cd_home.** *return*

2. Enter directory `@sample/counter`.

   :- **cd("@sample/counter").** *return*

   Then you have moved to:

   ```
   world = /login/olive/aum/@sample/counter
   ```

3. Compile the counter program, shown in Example 1, as follows:

   :- **aumcmp(["counter.aum", "test.counter"]).** *return*

   Then the following sequence of messages will be displayed:

```
... compiling A'UM (/login/olive/aum/@sample/counter/counter.aum) ...
<<< compiled to KL1-C (/login/olive/aum/@sample/counter/@counter.kl1) >>>
<<< compiled to KL1-B (/login/olive/aum/@sample/counter/@counter.asm) >>>
<<< done >>>
... compiling A'UM (/login/olive/aum/@sample/counter/test.counter) ...
<<< compiled to KL1-C (/login/olive/aum/@sample/counter/@test.kl1) >>>
<<< compiled to KL1-B (/login/olive/aum/@sample/counter/@test.asm) >>>
<<< done >>>
```

The last message "`<<< done >>>`" means that the compilation and loading of each class has been successfully done.

If there is no file for the specified file name, the following error message

```
??? Illegal File Name ???
```

will be displayed. Fix the file name and try again.

Generated Files

For a given $\mathcal{A}'\mathcal{UM}$ source file, the $\mathcal{A}'\mathcal{UM}$ compiler produces a set of files for each of the classes contained in the source file. Each file has a name starting with `@` and followed by their corresponding class name and some extention representing the kind of the file as follows:

`@ClassName.kl1`	KL1-C text file
`@ClassName.cls`	class information file
`@ClassName.asm`	KL1-B text file
`@ClassName.sav`	KL1-B binary file

For example, the above `counter.aum` file contains a definition of class `counter`, so files named `@counter.kl1`, `@counter.cls`, `@class.asm` and `@counter` are created.

2.5.2 Loading

To execute those $\mathcal{A}'\mathcal{UM}$ classes which were already compiled in the past, just loading should be done as follows:

For example, in the above, the counter program has been compiled. For the second time or later you test it, move to the directory, `@sample`, in the same way and just load it as follows:

:- **load_class([counter, test])**. *return*

2.5.3 Execution

$\mathcal{A}'\mathcal{UM}$ programs can be executed via the $\mathcal{A}'\mathcal{UM}$ program executer by issueing an $\mathcal{A}'\mathcal{UM}$ command, `start`. The $\mathcal{A}'\mathcal{UM}$ program executer, class `top`, is invoked by the `start` command, creates an instance of class `test` and then sends a message `test` to the instance. To execute any $\mathcal{A}'\mathcal{UM}$ program, class `test` with a method for message `test` must be prepared for that.

Let us execute the counter program which were compiled and loaded in the above.

1. Set trace switches of the two classes.

   :- **trace_class([counter, test])**. *return*

2. Invoke the $\mathcal{A}'\mathcal{UM}$ program executer.

   :- **start**. *return*

   Now the method `test` of the class `test` is being traced.

3. There is no debugger prepared for $\mathcal{A}'\mathcal{UM}$ yet. Move to the PDSS-CONSOLE window and see there what will happen at the Kl1 level.

Appendix A

Language Specification

A.1 Syntax

The whole syntax of language $\mathcal{A}'\mathcal{UM}$ is shown below:

Table A.1: Syntax

$<$ *ClassDefinition* $>$ ::=
    class $<$ *ClassName* $>$ "."
        [$<$ *SuperclassDefinition* $>$ "."]
        [$<$ *InletSlotDefinition* $>$ "."]
        [$<$ *OutletSlotDefinition* $>$ "."]
        { $<$ *Method* $>$ "." }
    end "."

$<$ *SuperclassDefinition* $>$ ::= **super** $<$ *SuperClassName* $>$ { ","$<$ *SuperClassName* $>$ }
$<$ *InletSlotDefinition* $>$ ::= **in** $<$ *SlotName* $>$ { ","$<$ *SlotName* $>$ }
$<$ *OutletSlotDefinition* $>$ ::= **out** $<$ *SlotName* $>$ { ","$<$ *SlotName* $>$ }

$<$ *ClassName* $>$ ::= $<$ *Name* $>$
$<$ *SuperClassName* $>$ ::= $<$ *ClassName* $>$
$<$ *SlotName* $>$ ::= $<$ *SlotName* $>$

$<$ *Method* $>$ ::= $<$ *Event* $>$ "|" $<$ *Actions* $>$ { ","$<$ *Actions* $>$ }

$<$ *Event* $>$ ::= $<$ *Receive* $>$ |$<$ *IsClosed* $>$
$<$ *Receive* $>$ ::= ":" $<$ *MessagePattern* $>$ "=" $<$ *UnorderedOutlet* $>$
$<$ *IsClosed* $>$ ::= "::"

$<$ *Action* $>$ ::= $<$ *NilExpression* $>$

```
< NilExpression > ::= < StreamClosing > |< VolatileObjectCreation >
< StreamClosing > ::= < OutletExpression > "::"

< OutletExpression > ::=
    < Outlet > |< PrimitiveObjectCreation > |
    < MessageSending > |< StreamMerging > |< OutletMacroExpresion >

< Outlet > ::= < UnorderedOutlet > |< OrderedOutlet >
< UnorderedOutlet > ::= < JointName >
< OrderedOutlet > ::= < JointName > "$" < Number >
< MessageSending > ::= < OutletExpression > ":" < MessaagePattern >
< StreamMerging > ::= < OutletExpression > "=" < InletExpression >

< InletExpression > ::=
    < Inlet > |< StreamAppending > |< InletMacroExpression >

< Inlet > ::= "^" < JointName >
< StreamAppending > ::= < InletExpression > "\" < InletExpression >

< VolatileObjectCreation > ::=
    < ImmutableVolatileObjectCreation > |< MutableVolatileObjectCreation >
< ImmutableVolatileObjectCreation > ::=
    < Interface > "?" < ImmutableVolatileClassDefinition >
< MutableVolatileObjectCreation > ::=
    < Interface > "=>" < MutableVolatileClassDefinition >

< Interface > ::= < InletExpression > |< OutletExpression >

< ImmutableVolatileClassDefinition > ::=
    "(" [ < SuperclassDefinition > ";" ]
        < Method > { ";"< Method > } ")"
< MutableVolatileClassDefinition > ::=
    "(" [ < SuperclassDefinition > ";" ]
        [ < InletSlotDefinition > ";" ]
        [ < OutletSlotDefinition > ";" ]
        < Method > { ";"< Method > } ")"
```

Table A.2: Lexicon

```
< Lexicon > ::=
    < Name > |< Number > |< CharacterString > |< Variable > |< Delimiter >

< Name > ::= < NormalName > |< SpecialName > |< QuotedName >
< NormalName > ::= < LowercaseLetter > { < TrailingLetter > }
< SpecialName > ::= < SpecialCharacter > { < SpecialCharacter > }
< QuotedName > ::= “'” { < NameStringCharacter > } “'”

< Number > ::=
    < Digit > { < Digit > }

< CharacterString > ::=
    “"” { < StringCharacter > } “"”

< Variable > ::=
    < UppercaseLetter > { < TrailingLetter > }
< TrailingLetter > ::= < Letter > |< Digit > |“_”


< StringCharacter > ::= < NameStringCharacter > |“"” − “""”
< NameStringCharacter > ::= < Character > − “'”

< Character > ::= < SpecialCharacter > |< Delimiter > |< Letter > |< Digit >

< SpecialCharacter > ::=
    “!” |“#” |“$” |“&” |“*” |“+” |“-” |“.” |“/” |“:” |“<” |“=”
    “>” |“?” |“@” |“\” |“~” |“_” |“|” |“ ”

< Delimiter > ::=
    “"” |“%” |“'” |“(” |“)” |“,” |“;” |“[” |“]” |“‘” |“{” |“}”

< Digit > ::−
    “0” |“1” |“2” |“3” |“4” |“5” |“6” |“7” |“8” |“9”

< Letter > ::= < UppercaseLetter > |< LowercaseLetter >

< UppercaseLetter > ::=
    “A” |“B” |“C” |“D” |“E” |“F” |“G” |“H” |“I” |“J”
    |“K” |“L” |“M” |“N” |“O” |“P” |“Q” |“R” |“S” |“T”
    |“U” |“V” |“W” |“X” |“Y” |“Z”

< LowercaseLetter > ::=
    “a” |“b” |“c” |“d” |“e” |“f” |“g” |“h” |“i” |“j”
    |“k” |“l” |“m” |“n” |“o” |“p” |“q” |“r” |“s” |“t”
    |“u” |“v” |“w” |“x” |“y” |“z”
```

A.2 Operator Precedence

The lexical analysis on $\mathcal{A}'\mathcal{UM}$ programs is based on an operator precedence grammar. The precedences and types of each operator are shown below:

Table A.3: Operator Precedence

operator	*(precedence, type)*
->	(1050, xfx), (1050, xf)
-|	(1050, xfx), (1050, xf)
|	(1010, xfx)
,	(1000, xfy)
?	(1000, xfx), (500, xf)
=	(700, yfx)
\	(700, yfx)
==	(600, yfx)
\=	(600, yfx)
<	(600, yfx)
>	(600, yfx)
<-	(700, xfx)
+	(500, fx), (500, yfx)
-	(500, fx), (500, yfx)
not	(500, yfx)
/\	(500, yfx)
\/	(500, yfx)
	(400, yfx)
/	(400, yfx)
>>	(400, yfx)
<<	(400, yfx)
mod	(300, yfx)
:	(200, fx)
::	(210, xf)
##	(100,fx)
#	(100, fx)
^	(100, fx)
!	(100, fx)
$	(90, fx), (90, xfx)
‘	(10, fx)

A.3 Primitive Classes

Table A.4 shows primitive classes provided in $\mathcal{XAS}$.

Table A.4: Primitive Classes

class	*image*	*acceptable messages*
atom	a	`symbol(String)`
boolean	`'true`	`not(Negated)`
		`and(^Y, LogicalProduct)`
		`or(^Y, LogicalSum)`
		`xor(^Y, LogicaXsum)`
integer	1	`plus(Itself)`
		`minus(Complement)`
		`add(^Y, Sum)`
		`sub(^Y, Dif)`
		`mul(^Y, Product)`
		`div(^Y, Quotient)`
		`mod(^Y, Residue)`
		`shtl(^Y, LeftShifted)`
		`shtr(^Y, RightShifted)`
		`lt(^Y, TorF)`
		`nlt(^Y, TorF)`
		`gt(^Y, TorF)`
		`ngt(^Y, TorF)`
		`and(^Y, BitwiseProduct)`
		`or(^Y, BitwiseSum)`
		`xor(^Y, BitwiseXsum)`
string	`''hi''`	`size(Size)`
		`element_size(ElementSize)`
		`make_atom(Atom)`
		`element(^Position, Element)`
		`set_element(^Position, Element)`
		`make_symbol(Atom)`
class	`##stack`	`new(Obj)`
common		`eq(^Y, TorF)`
		`ne(^Y, TorF)`

A.4 Built-in Classes

Table A.4 shows built-in classes provided in $\mathcal{XAS}$.

Table A.5: Built-in Classes

class	*image*	*acceptable messages*

list	`[a|b]`	`car(Car)`
		`cdr(Cdr)`
		`set_car(^Car)`
		`set_cdr(^Cdr)`

vector	`{a,b}`	`size(Size)`
		`element(^Position, Element)`
		`set_element(^Position, ^Element)`

window	`i_am(^W)`	`show`
		`hide`
		`clear`
		`beep`
		`prompt(OldPrompt, ^NewPrompt)`
		`flush`

file	`i_am(^F)`	`open(^FileName, Mode)`
		`open(^FileName, Mode, Status)`

I/O common		`put(^Anything)`
		`putc(^Char)`
		`putl(^String)`
		`putb(^String)`
		`get(Anything)`
		`getc(Integer)`
		`getl(String)`
		`getb(String)`
		`nl`
		`tab(^N)`

A.5 Arithmetical/Logical Operation Macros

Table A.6 shows arithmentical and logical operation macros provided in $\mathcal{XAS}$.

Table A.6: Arithmetical and logical operation macros

macro expression	*mode*	*result*	*expansion*
`+ X`	$< Out >$	`Itself`	`X:plus(^Itself)::`
`- X`	$< Out >$	`Complement`	`X:minus(^Complement)::`
`X + Y`	$< Out >$	`Sum`	`X:add(Y, ^Sum)::`
`X - Y`	$< Out >$	`Difference`	`X:sub(Y, ^Difference)::`
`X * Y`	$< Out >$	`Product`	`X:mul(Y, ^Product)::`
`X / Y`	$< Out >$	`Quotient`	`X:div(Y, ^Quotient)::`
`X mod Y`	$< Out >$	`Residue`	`X:mod(Y, ^Residue)::`
`X >> Y`	$< Out >$	`Shifted`	`X:shtr(Y, ^Shifted)::`
`X << Y`	$< Out >$	`Shifted`	`X:shtl(Y, ^Shifted)::`
`not X`	$< Out >$	`TorF`	`X:not(^TorF)::`
`X /\ Y`	$< Out >$	`Product`	`X:and(Y, ^Product)::`
`X \/ Y`	$< Out >$	`Sum`	`X:or(Y, ^Sum)::`
`X xor Y`	$< Out >$	`Xsum`	`X:xor(Y, ^Xsum)::`
`[X|Y]`	$< Out >$	`List`	`##list:new(^L)::,` `L:set_car(X):set_cdr(Y)= ^List::`
`car X`	$< Out >$	`Car`	`X:car(X, ^Car)::`
`cdr X`	$< Out >$	`Cdr`	`X:cdr(X, ^Cdr)::`
`{X,Y}`	$< Out >$	`Vector`	`##vector:new(^V)::,` `V:set_element(1, X)` `:set_element(2, Y)= ^Vector::`
`X == Y`	$< Out >$	`TorF`	`X:eq(Y, ^TorF)::`
`X \= Y`	$< Out >$	`TorF`	`X:neq(Y, ^TorF)::`
`X < Y`	$< Out >$	`TorF`	`X:lt(Y, ^TorF)::`
`X > Y`	$< Out >$	`TorF`	`X:gt(Y, ^TorF)::`
`class_of X`	$< Out >$	`Class`	`X:class(^Class)::`
`X ?`	$< In >$	`^Who`	`X:who_are_you(Who)::`

Appendix B

Implementation Specification

B.1 KL1 Object Code

The $\mathcal{A}'\mathcal{UM}$ compiler generates a KL1-C program (`*.kl1`) from an $\mathcal{A}'\mathcal{UM}$ class definition. The generated KL1-C program is compiled to a KL1-B assemble program (`*.asm`) by the KL1 compiler embedded in PDSS.

B.1.1 Code Generation

For an $\mathcal{A}'\mathcal{UM}$ class definition, a KL1-C program is generated according to the following rules:

Classes:

- External classes: Each external class is represented as a KL1 module named "`@`" followed by the class name. The module name is the internal class name of the external class.

  For example, a module `@sift` is created for the class `sift`.
- Volatile classes: Each volatile class is represented as a part of the module of its external class, and is given an internal class name which consists of the module name of the external class and the occurrence level.

  For example, the mutable volatile class appearing in the class `sift` is given an internal class name `@sift_1@m1`.

Objects: Each object is represented as a sequence of tail recursive goals which has a predicate name of the internal class name and five arguments as follows:

```
'@sift'(Interface, SlotInfo, Global, CreaterAndScope, ClassInfo)
```

External objects and volatile objects are of the same form.

Primitive Objects: Each primitive object is implemented as an instance object of the primitive class in the same way as the above general objects are.

Basic Stream Operations: Each stream is represented as a list structure and stream operations are implemented as unification of list structures as follows:

`send(X, m, ^Y)`	Body unification `X = [m|Y]`
`close(X)`	Body unification `X = []`
`connect(X, ^Y)`	Body unification `X = Y`
`receive(^X, m, Y)`	Guard unification `X = [m|Y]`
`is_closed(^X)`	Guard unification `X = []`

Joints:

- Merge joints: Each intermediate merge joint which is on the way to an object, such as `merge(^X, ^Y, W)`, is implemented as construction of a vector consisting of the two incoming streams, `W = {X, Y}`.

  At the entrance of the interface stream of each object, a KL1 built-in predicate `merge(W, Z)` is inserted. The `merge(W, Z)` predicate is a process which interprets each incoming vector and generates a stream.
- Append joints: Each append joint, such as `append(X, Y, Z)`, invokes an $\mathcal{A}'\mathcal{UM}$ built-in predicate, `'$blt':append(X, Y, Z)`

Messages:

- Each message is given a *message tag* which represents whether the message is an atomic message or a compound message.

  Tags `i`, `a`, `b`, `s`, `l` and `v` denotes an integer message, atom message, boolean message, string message and vector message. Tag `c` denotes a compound message.
- Each argument of a compound message is given an *argument tag*, either `i`, `o` or `m`, to represent whether the argument is an inlet, an outlet or a primitive object.

Class Inheritance: The fifth argument of an object contains information of class inheritance.

B.1.2 Examples of KL1-C Generated Code

For those sample programs given in Example 1 and Example 2 of Chapter 1, the $\mathcal{A}'\mathcal{UM}$ compiler generates the following KL1-C code.

```
:- module '@counter' .
:- public '@counter'/1 ',' '@counter'/5 .

'@counter'(A) :- true |
        '$OBJECT_x':new($('@counter','@counter',['@counter'],
        ['@counter!n'],[[c(show({o})),c(set({i})),a(down),a(up)]],[n]),A) .

'@counter'([a(up)|H],B,C,$(D,E),F) :- true |
        G=[c('$get_outlet_slot'(m(o)),{a('@counter!n'),M})|L] ,
        '@integer':new(i(1),{N,C,O}) ,
        M=[c(add(i(o)),{N,P})|Q] ,
        L=[c('$set_outlet_slot'(m(i)),{a('@counter!n'),Q})|R] ,
        R=[c('$set_outlet_slot'(m(i)),{a('@counter!n'),P})|H] ,
        '$OBJECT':descend(G,B,O,$(D,E),F) .

'@counter'([a(down)|H],B,C,$(D,E),F) :- true |
        G=[c('$get_outlet_slot'(m(o)),{a('@counter!n'),M})|L] ,
        '@integer':new(i(1),{N,C,O}) ,
        M=[c(sub(i(o)),{N,P})|Q] ,
        L=[c('$set_outlet_slot'(m(i)),{a('@counter!n'),Q})|R] ,
        R=[c('$set_outlet_slot'(m(i)),{a('@counter!n'),P})|H] ,
        '$OBJECT':descend(G,B,O,$(D,E),F) .

'@counter'([c(set({i}),{L})|H],B,C,$(D,E),F) :- true |
        G=[c('$set_outlet_slot'(m(i)),{a('@counter!n'),L})|H] ,
        '$OBJECT':descend(G,B,C,$(D,E),F) .

'@counter'([c(show({o}),{L})|H],B,C,$(D,E),F) :- true |
        G=[c('$get_outlet_slot'(m(o)),{a('@counter!n'),L})|H] ,
        '$OBJECT':descend(G,B,C,$(D,E),F) .
```

Figure B.1: KL1-C code generated for class counter

```
:- module '@test' .
:- public '@test'/1 ',' '@test'/5 .

'@test'(A) :- true |
        '$OBJECT_x':new($('@test','@test',['@test'],[],
        [[c(nop({o})),c(testA(o(o))),c(testM(o(o))),a(test)]],[n]),A) .

'@test'([a(test)|H],B,C,$(D,E),F) :- true |
        M=[c(testM(o(o)),{N,O})|P] ,
        P=[c(testA(o(o)),{S,T})|U] ,
        U=[c(nop({o}),{X})|H] ,
        N=[c(who_are_you({i}),{BD})] ,
        O=[c(who_are_you({i}),{BI})] ,
        S=[c(who_are_you({i}),{BN})] ,
        T=[c(who_are_you({i}),{BS})] ,
        '$blt':'@append'(BN,BS,CG) ,
        '$blt':'@append'(BI,CG,CF) ,
        '$blt':'@append'(BD,CF,X) ,
        '$OBJECT':descend(M,B,C,$(D,E),F) .
'@test'([c(testM(o(o)),{L,N})|H],B,C,$(D,E),F) :- true |
        '@class':new(m(a(counter)),{P,C,Q}) ,
        P=[c(new({o}),{R})] ,
        '@integer':new(i(5),{T,Q,U}) ,
        R=[c(set({i}),{T})|V] ,
        Y=[a(up)|Z] ,
        Z=[a(up)|BA] ,
        BA=[c(show({o}),{L})] ,
        BC=[a(down)|BD] ,
        BD=[a(down)|BE] ,
        BE=[c(show({o}),{N})] ,
        merge_in(BC,Y,V) ,
        '$OBJECT':descend(H,B,U,$(D,E),F) .
'@test'([c(testA(o(o)),{L,N})|H],B,C,$(D,E),F) :- true |
        '@class':new(m(a(counter)),{P,C,Q}) ,
        P=[c(new({o}),{R})] ,
        '@integer':new(i(5),{T,Q,U}) ,
        R=[c(set({i}),{T})|V] ,
        Y=[a(up)|Z] ,
        Z=[a(up)|BA] ,
        BA=[c(show({o}),{L})] ,
        BC=[a(down)|BD] ,
        BD=[a(down)|BE] ,
        BE=[c(show({o}),{N})] ,
        '$blt':'@append'(Y,BC,V) ,
        '$OBJECT':descend(H,B,U,$(D,E),F) .
'@test'([c(nop({o}),{L})|H],B,C,$(D,E),F) :- true |
        '$$sink':new({L,N,[]}) ,
        merge_in(N,I,C) ,
        '$OBJECT':descend(H,B,I,$(D,E),F) .
```

Figure B.2: KL1-C code generated for class test for the counter

```
:- module '@prime' .
:- public '@prime'/1 ',' '@prime'/5 ','
          '@prime_2@i1'/1 ',' '@prime_2@i1'/5 .

'@prime'(A) :- true |
        '$OBJECT_x':new($('@prime','@prime',['@prime'],[],
        [[c(generate(i(i,i))),c(primes(i(i)))]],[n]),A) .
'@prime'([c(primes(i(i)),{L,M})|H],B,C,$(D,E),F) :- true |
        '@integer':new(i(3),{N,C,O}) ,
        S=[c(generate(i(i,i)),{T,L,V})|H] ,
        '@class':new(m(a(sift)),{BA,O,BB}) ,
        BA=[c(new({o}),{BC})] ,
        '@integer':new(i(2),{BG,BB,BH}) ,
        M=[c(n({i}),{BG})|BI] ,
        BI=[c(n({i}),{BJ})|BK] ,
        BC=[c(do(i(o,i)),{BE,V,BK})] ,
        merge_in(BE,T,BU) ,
        merge_in(BJ,BU,N) ,
        '$OBJECT':descend(S,B,BH,$(D,E),F) .
'@prime'([c(generate(i(i,i)),{L,M,N})|H],B,C,$(D,E),F) :- true |
        '@integer':new(i(2),{P,C,Q}) ,
        L=[c(add(i(o)),{P,R})] ,
        U=[c(lt(i(o)),{V,W})] ,
        W=[c(who_are_you({i}),{Y})] ,
        BC=[c('$get_inlet_var'(m(i)),{a('Max'),M})|BI] ,
        BI=[c('$get_moutlet_var'(m(o)),{a('Max'),V})|BJ] ,
        BJ=[c('$get_inlet_var'(m(i)),{a('Ns'),N})|BN] ,
        BN=[c('$get_inlet_var'(m(i)),{a('NewX'),T})] ,
        '$$scope':new({"@prime_2",['Max','Ns','NewX'],[ [],[],[]]},{K,BD}) ,
        merge_in(BD,I,BB) ,
        merge_in(T,U,R) ,
        '@prime_2@i1'({Y,$(Q,BB),$(G,H),$(K,BC)}) ,
        '$OBJECT':descend(G,B,I,$(D,E),F) .

'@prime_2@i1'(A) :- true |
        '$OBJECT_i':new($('@prime','@prime_2@i1',['@prime_2@i1'],[],
        [[b(false),b(true)]],[n]),A) .
'@prime_2@i1'([b(true)|G],B,C,$(D,E),F) :- true |
        L=[c(n({i}),{N})|P] ,
        D=[c(generate(i(i,i)),{U,V,P})|Y] ,
        E=[c('$get_moutlet_var'(m(o)),{a('Max'),V})|BF] ,
        BF=[c('$get_moutlet_var'(m(o)),{a('Ns'),L})|BI] ,
        BI=[c('$get_moutlet_var'(m(o)),{a('NewX'),BO})] ,
        merge_in(U,N,BO) ,
        '$OBJECT':terminate(G,B,C,$(Y,[]),F) .
'@prime_2@i1'([b(false)|G],B,C,$(D,E),F) :- true |
        E=[] ,
        '$OBJECT':terminate(G,B,C,$(D,[]),F) .
```

Figure B.3: KL1-C code generated for class `prime`

```
:- module '@sift' .
:- public '@sift'/1 ',' '@sift'/5 ','
          '@sift_1@m1'/1 ',' '@sift_1@m1'/5 ','
          '@sift_1@m1_2@i1'/1 ',' '@sift_1@m1_2@i1'/5 ','
          '@sift_1@m1_2@i1_2@i1'/1 ',' '@sift_1@m1_2@i1_2@i1'/5 .

'@sift'(A) :- true |
        '$OBJECT_x':new($('@sift','@sift',['@sift'],[],
        [[c(do(i(o,i)))]],[n]),A) .

'@sift'([c(do(i(o,i)),{L,M,O})|H],B,C,$(D,E),F) :- true |
        P=[c(initialize(i(i)),{L,O})|M] ,
        '@sift_1@m1'({P,$(C,W),$(G,H)}) ,
        '$OBJECT':descend(G,B,W,$(D,E),F) .

'@sift_1@m1'(A) :- true |
        '$OBJECT_m':new($('@sift','@sift_1@m1',['@sift_1@m1'],
        ['@sift_1@m1!me','@sift_1@m1!next','@sift_1@m1!to_next',
        '@sift_1@m1!primes'],[[c(n({i})),c(initialize(i(i)))]],[n]),A) .

'@sift_1@m1'([c(initialize(i(i)),{L,M})|J],B,C,$(D,E),F) :- true |
        G=[c('$set_outlet_slot'(m(i)),{a('@sift_1@m1!me'),L})|O] ,
        '@integer':new(i(0),{R,C,S}) ,
        O=[c('$set_outlet_slot'(m(i)),{a('@sift_1@m1!next'),R})|T] ,
        T=[c('$set_outlet_slot'(m(i)),{a('@sift_1@m1!primes'),M})|J] ,
        '$OBJECT':descend(G,B,S,$(D,E),F) .

'@sift_1@m1'([c(n({i}),{L})|J],B,C,$(D,E),F) :- true |
        G=[c('$get_outlet_slot'(m(o)),{a('@sift_1@m1!me'),O})|N] ,
        M=[c(mod(i(o)),{O,P})] ,
        '@integer':new(i(0),{R,C,S}) ,
        P=[c(eq(i(o)),{R,T})] ,
        T=[c(who_are_you({i}),{V})] ,
        Z=[c('$get_inlet_var'(m(i)),{a('X'),L})|BF] ,
        BF=[c('$get_moutlet_var'(m(o)),{a('X'),M})] ,
        '$$scope':new({"@sift_1@m1_2",['X'],[ []]},{K,BA}) ,
        merge_in(BA,H,Y) ,
        '@sift_1@m1_2@i1'({V,$(S,Y),$(N,J),$(K,Z)}) ,
        '$OBJECT':descend(G,B,H,$(D,E),F) .
```

Figure B.4: KL1-C code generated for class `sift` (to be continued)

```
'@sift_1@m1_2@i1'(A) :- true |
        '$OBJECT_i':new($('@sift','@sift_1@m1_2@i1',['@sift_1@m1_2@i1'],[],
        [[b(false),b(true)]],[n]),A) .

'@sift_1@m1_2@i1'([b(true)|G],B,C,$(D,E),F) :- true |
        E=[] ,
        '$OBJECT':terminate(G,B,C,$(D,[]),F) .

'@sift_1@m1_2@i1'([b(false)|G],B,C,$(D,E),F) :- true |
        D=[c('$get_outlet_slot'(m(o)),{a('@sift_1@m1!next'),M})|L] ,
        '@integer':new(i(O),{N,C,O}) ,
        M=[c(eq(i(o)),{N,P})|Q] ,
        L=[c('$set_outlet_slot'(m(i)),{a('@sift_1@m1!next'),Q})|R] ,
        P=[c(who_are_you({i}),{U})] ,
        '@sift_1@m1_2@i1_2@i1'({U,$(O,X),$(R,W),$(E,[])}) ,
        '$OBJECT':terminate(G,B,X,$(W,[]),F) .

'@sift_1@m1_2@i1_2@i1'(A) :- true |
        '$OBJECT_i':new($('@sift','@sift_1@m1_2@i1_2@i1',
        ['@sift_1@m1_2@i1_2@i1'],[],[[b(false),b(true)]],[n]),A) .

'@sift_1@m1_2@i1_2@i1'([b(true)|G],B,C,$(D,E),F) :- true |
        D=[c('$set_outlet_slot'(m(i)),{a('@sift_1@m1!next'),U})|N] ,
        N=[c('$set_outlet_slot'(m(i)),{a('@sift_1@m1!to_next'),T})|S] ,
        '@class':new(m(a(sift)),{V,C,W}) ,
        V=[c(new({o}),{X})] ,
        S=[c('$get_outlet_slot'(m(o)),{a('@sift_1@m1!primes'),BB})|BA] ,
        BB=[c(n({i}),{BC})|BD] ,
        BA=[c('$set_outlet_slot'(m(i)),{a('@sift_1@m1!primes'),BF})|BE] ,
        X=[c(do(i(o,i)),{Z,T,BG})] ,
        E=[c('$get_moutlet_var'(m(o)),{a('X'),BQ})] ,
        merge_in(Z,U,BO) ,
        merge_in(BC,BO,BQ) ,
        merge_in(BF,BG,BD) ,
        '$OBJECT':terminate(G,B,W,$(BE,[]),F) .

'@sift_1@m1_2@i1_2@i1'([b(false)|G],B,C,$(D,E),F) :- true |
        D=[c('$get_outlet_slot'(m(o)),{a('@sift_1@m1!to_next'),M})|L] ,
        M=[c(n({i}),{N})|P] ,
        L=[c('$set_outlet_slot'(m(i)),{a('@sift_1@m1!to_next'),P})|Q] ,
        E=[c('$get_moutlet_var'(m(o)),{a('X'),N})] ,
        '$OBJECT':terminate(G,B,C,$(Q,[]),F) .
```

Figure B.5: KL1-C code generated for class sift

```
:- module '@test' .
:- public '@test'/1 ',' '@test'/5 ','
          '@test_1@m1'/1 ',' '@test_1@m1'/5 .

'@test'(A) :- true |
        '$OBJECT_x':new($('@test','@test',['@test'],[],
        [[c(nop({o})),a(test)]],[n]),A) .

'@test'([a(test)|H],B,C,$(D,E),F) :- true |
        '@class':new(m(a(prime)),{L,C,M}) ,
        L=[c(new({o}),{N})] ,
        '@integer':new(i(20),{P,M,Q}) ,
        N=[c(primes(i(i)),{P,R})] ,
        V=[c(nop({o}),{W})|X] ,
        BA=[c(initialize({i}),{W})|R] ,
        '@test_1@m1'({BA,$(Q,BG),$(X,H)}) ,
        '$OBJECT':descend(V,B,BG,$(D,E),F) .

'@test'([c(nop({o}),{L})|H],B,C,$(D,E),F) :- true |
        '$$sink':new({L,N,[]}) ,
        merge_in(N,I,C) ,
        '$OBJECT':descend(H,B,I,$(D,E),F) .

'@test_1@m1'(A) :- true |
        '$OBJECT_m':new($('@test','@test_1@m1',['@test_1@m1'],
        ['@test_1@m1!result'],[[c(n({i})),c(initialize({i}))]],[n]),A) .

'@test_1@m1'([c(initialize({i}),{L})|J],B,C,$(D,E),F) :- true |
        G=[c('$set_outlet_slot'(m(i)),{a('@test_1@m1!result'),L})|J] ,
        '$OBJECT':descend(G,B,C,$(D,E),F) .

'@test_1@m1'([c(n({i}),{L})|J],B,C,$(D,E),F) :- true |
        G=[c('$get_outlet_slot'(m(o)),{a('@test_1@m1!result'),N})|M] ,
        L=[c(who_are_you({i}),{P})] ,
        M=[c('$set_outlet_slot'(m(i)),{a('@test_1@m1!result'),T})|J] ,
        '$blt':'@append'(P,T,N) ,
        '$OBJECT':descend(G,B,C,$(D,E),F) .
```

Figure B.6: Kl1-C code generated for class test for the prime number generator

Appendix C

Sample Programs

$\mathcal{XAS}$ is distributed with several sample programs to introduce the taste of $\mathcal{A'UM}$. The sample programs are stored in subdirectories under the diretory, `@sample`

subdirectory	*source program files*	`.asm` *files to be loaded*
`counter`	`test.counter` `counter.aum`	`@test.asm` `@counter.asm`

`stack`	`test.stack`
	`stack.aum`
	`element.aum`
	`bottom.aum`
		`@test.asm`
		`@stack.asm`
		`@element.asm`
		`@bottom.asm`

`reverse`	`test.reverse` `reverse.aum`	`@test.asm` `@reverse.asm`

`prime`	`test.prime`
	`prime.aum`
	`sift.aum`
		`@test.asm`
		`@prime.asm`
		`@sift.asm`

`dp`	`test.dp`
	`@dpmain.aum`
	`@dpmatch.aum`
	`@brownie.aum`
	`@cm.kl1` (KL1-C program)
		`@test.asm`
		`@dpmain.asm`
		`@dpmatch.asm`
		`@brownie.asm`
		`@cm.asm`

C.1 Counter

The counter program is the one which was given as Example 1 and shown in Figure 1.18.

1. Change the directory to `counter`

   :- **cd counter.** *return*

2. Load classes and execute.

   :- **load_class([test, counter]).** *return*
   :- **start.** *return*

Then four counter values, `Um`, `Dm`, `Ua` and `Da`, will be displayed.

C.2 Stack

The stack program is to simulate a stack object which receives messages: `push(^Data)`, `pop(Data)`, `read(Data)`.

1. Change the directory to `counter`

   :- **cd counter.** *return*

2. Load classes and execute.

   :- **load_class([test, counter]).** *return*
   :- **start.** *return*

3. For the prompt, "`Command ?= `", input either "push.*return*", "pop.*return*", "`read`.*return*", or "end.*return*".

   - For command **push**, another prompt "`Data ?= `" appears. Input any type of data. Then the top element of the stack is displayed.
   - For commands **pop** and **read**, the obtained data and the new top of the stack will be displayed.
   - The command end terminates the execution.

C.3 Tree Reverse

The tree-reverse program is to reverse a given binary-tree vector.

1. Change the current directory to `reverse`.

   :- **cd reverse.** *return*

2. Load classes and execute.

   :- **load_class([test, reverse]).** *return*
   :- **start.** *return*

3. A window will appear. For the prompt, "`Original Tree ?= `", input a binary-tree vector. Preceeded by a sequence of acknowledgements from the leaf elements contained in the vector, informing that they have just been reached, a generated reverse vector is displayed.

C.4 Prime Number Generator

The prime number generator program is the one which was given as Example 2 in Figure 1.23.

1. Change the current directory to `reverse`.

```
:- cd dp. return
```

2. Load classes and execute.

```
:- load_class( [test, prime, sift] ). return
:- start. return
```

3. A window will appear. For the prompt, "`Max ?= `", input a positive integer. Then a sequence of prime numbers less than the given maximum number will be displayed.

C.5 DP Matching

The DP-matching program is to analyze the homology of two given protein sequences. Each protein sequence is represented as a vector of characters each of which means one of the following twenty kinds of amino acid:

g	glycine	d	aspartic acid	k	lysine
a	alanine	n	asparagine	r	arginine
s	serine	e	glutamic acid	h	histidine
t	threonine	q	glutamine		
p	proline				
l	leucine	f	phenylalanine	c	cysteine
i	isoleucine	y	tyrosine		
m	methionine	w	tryptophan		
v	valine				

Homology analysis is done assuming the mutation of amino acid. One kind of amino acid may mutate to another kind. Mutability between twenty kinds of amino acid is given as a cost matrix. If the mutability between two kinds of amino acid is over some standard value, they are thought to be matched. Also insertion and deletion are assumed.

1. Change the current directory to dp.

```
:- cd dp. return
```

2. Load classes and execute.

```
:- load_class( [test, dpmain, dpmatch, brownie, cm] ). return
:- start. return
```

3. For the prompt, "`Sequence A ?= `", input a sequence such as `{v,e,d,q,k,l,t,s,k,c}`. Similarly, for the prompt, "`Sequence B ?= `", input another sequence such as `{v,e,n,k,l,t,r,p,k,c}`. In a minute or so, a homology matrix will appear, which shows how much the given two sequences are homologous.

Bibliography

[Yoshida88] K. Yoshida and T. Chikayama: *$\mathcal{A}'\mathcal{UM}$ – A Stream-Based Concurrent Object-Oriented Language –*, Proc. of the International Conference of Fifth Generation Computer Systems '88, ICOT, November 1988.

[PDSS89] *PDSS Manual (Version 2.51)*, ICOT, August 1989.