Palmer and Naish [1991] describe NUA-Prolog, an and-parallel implementation of Prolog based on the BAM, otherwise similar to Andorra-I, but without a pre-processor guaranteeing consistency with sequential Prolog behaviour.

Bahgat [1991] describes an abstract machine for Pandora, an extension of PAR-LOG based on the BAM, an extension of the JAM with a choice-point stack for sharing. The model does not maintain the order of goals, which are not in linked lists. Instead, programmer intervention is required for the choice of a goal for a don't know nondeterministic step.

CHAPTER 7

# PORTS FOR OBJECTS

Can object-oriented programming be realised effectively in AKL? This chapter is an introduction to the concept of *ports*. It is argued that ports are highly useful for object-oriented programming in a concurrent logic programming language. It is also illustrated by examples how to use ports to emulate other communication schemes for concurrent programming.

## 7.1 INTRODUCTION

We regard *objects* for concurrent logic programming languages as processes, as first proposed by Shapiro and Takeuchi [1983], and later extended and refined in systems such as Vulcan [Kahn et al 1987], A'UM [Yoshida and Chikayama 1988], and Polka [Davison 1989] (Figure 7.1).

Some of these systems are embedded languages that only make restricted use of the underlying language. We are interested in full-strength combinations of the underlying language with an expressive concurrent object-oriented extension, with all the problems and opportunities this entails.

In this chapter we introduce *ports*, an alternative to streams, as communication support for object-oriented programming in concurrent constraint logic programming languages. From a pragmatic point of view, ports provide efficient many-to-one communication, object identity, means for garbage collection of objects, and opportunities for optimised compilation techniques for concurrent objects. It will also provide us with means for mixing freely objects and other data structures provided in concurrent logic programming languages. From a semantic point of view, ports preserve the monotonicity of the constraint store, which is a crucial property of all concurrent constraint languages.

In the next few sections we present some of the background of our work. First we examine some requirements on object-oriented systems. Then we discuss the notion of a communication medium, and review a number of proposals that do not meet our requirements.

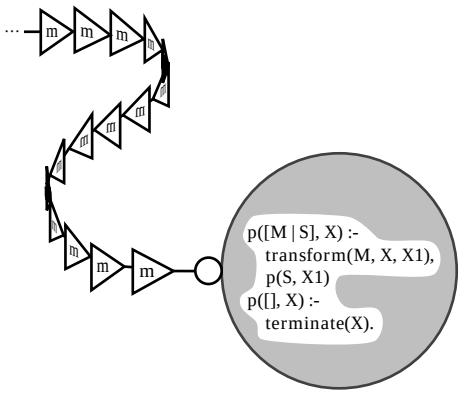


Figure 7.1. Objects as stream-consuming recursive processes

In the following sections, we introduce ports as a solution to these problems, one which also adds entirely new possibilities, such as a simple approach to optimised compilation of objects. We will also show that the Exclusive-read, Exclusive-write PRAM model of parallel computation can be realised quite faithfully in terms of space and time complexity using ports. This indirectly demonstrates that arbitrary parallel algorithms can be expressed quite efficiently. Finally, a number of examples illustrate how the functionality of many other languages is captured by ports.

## 7.2 REQUIREMENTS

Our starting point is a number of requirements on object-oriented languages, and we will let these guide our work. In so doing, we here only consider requirements on the object-based functionality, including requirements on the interaction between the host language and the concurrent object-oriented extension. Other aspects of object-oriented languages, such as inheritance, can be handled in many different ways (e.g., [Goldberg and Shapiro 1992]).

Since our goal is the integration of concurrent object and logic programming, we conform to the tradition of languages such as C++, where the object-oriented aspects are added to the underlying language and, in particular, allow objects and other data structures to be mingled freely. Thus, in a logic programming language, it should be possible to have objects *embedded* in a term data structure (Figure 7.2, next page). Since terms can be shared freely, this will allow concurrent objects to share, for instance, an array of concurrent objects.

Higher-level object-oriented languages provide automatic *garbage collection* of objects that are no longer referenced (Figure 7.3, next page). Programmers do not have to think about when objects are no longer in use, nor do they have to deallocate them explicitly. The high-level nature of logic programming languages makes it desirable to provide garbage collection of objects, just as of other data structures.

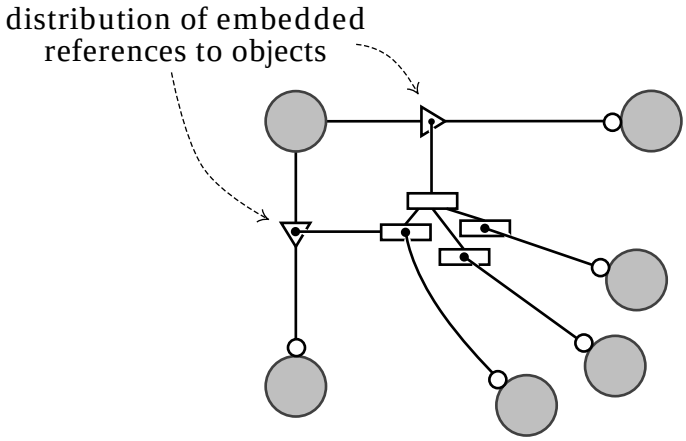


Figure 7.2. Objects embedded in terms

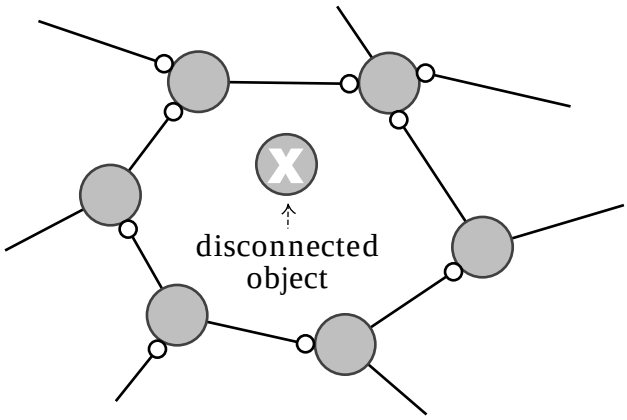


Figure 7.3. Garbage-collectable disconnected object

Note that, in the goal-directed view of logic programming, an object is a goal, which has to be proved; it cannot just be thrown away. However likely the assumption that a goal is provable without binding variables, it cannot be taken for granted. In a concurrent object-oriented setting, another dimension is added, in that an object may still be active, and affect its environment, although it is no longer referred to by other objects. Even if there are no incoming messages, an object may wish to perform some cleaning up before being discarded. Thus, garbage collection of objects in concurrent (logic) programs should involve notifying an object that it will no longer receive messages. It is up to the object to decide to terminate.

In addition, an implementation of objects should provide

- *light-weight* message sending and method invocation, the cost of which should preferably be similar to that of a procedure call,
- *compact* representation of objects, the size of which should be dominated by the representation of instance variables,
- *conservation* of memory, which means that a state transition should only involve modifying relevant instance variables—objects are reused.

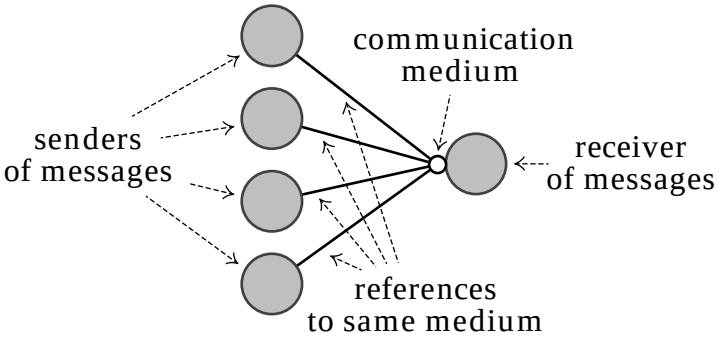


Figure 7.4. The communication medium

This chapter will address all but the last of the above requirements. The last requirement is generally solved in concurrent logic programming languages by providing a mechanism for detecting single references, and reusing the old instance variables.

## 7.3 COMMUNICATION MEDIA

A *communication medium* is a data type that carries messages between processes acting as objects (Figure 7.4).

The medium is used as a handle to an object, and is regarded as the *object identifier* from a programmer's point of view.

In the concurrent constraint view that we take, the communication medium is managed (described and inspected) using constraints. All constraints are added to a shared *constraint store*. To *send* a message means to impose a constraint on the medium which allows a process to detect the presence of a message, and *receive* the message by inspecting its properties. A sender of messages is a *writer* of the medium. A receiver of messages is a *reader* of the medium. A message that is received once and for all is said to be *consumed*. A medium can be *closed* by imposing constraints that disallow additional messages. A receiver can detect that a medium is closed.

An important property of the constraint store in concurrent (constraint) logic programming languages is that it is monotone. Addition of constraints will produce a new constraint store that entails all the information in the previous one. This property is important because it implies that once a process is activated by the receipt of a message, this activation condition will continue to hold until the message is consumed, regardless of the actions of other processes.

The discussion above leads us to the following requirements on our communication medium.

- A solution should preserve the monotonicity of the constraint store.
- The number of operations required to send a message (make it visible to an end receiver) should be constant (for all practical purposes), independent of the number of senders. All senders should be given equal oppor-

tunity, according to a first come first served principle. We call this the *constant delay property*.

- When a part of the medium that holds a message has been consumed, it should be possible to deallocate or reuse the storage it occupied, by garbage collection or otherwise.
- To provide completely automatic garbage collection of objects, it should be possible to apply the closing operation automatically (when the medium is no longer in use).
- To enable sending multiple messages to embedded objects, it should be possible to send multiple messages to the same medium.

The last requirement seems odd in conventional object-oriented systems. It is however a problem in all single-assignment languages including the current (constraint) logic programming languages.

In the remainder of this section we discuss a number of communication media that have been proposed for concurrent (constraint) logic programming languages.

### *7.3.1 Streams*

The list is by far the most popular communication medium in concurrent logic programming. In this context lists are usually called *streams*. A background to streams and techniques for binary merging (see below) is given by Shapiro and Mierowsky [1984].

A message $m$ is sent on the stream $S$ by binding $S$ to a list pair, $S = [m \mid S']$. The next message is sent on the stream $S'$. A stream $S$ is closed by making it equal to the *empty list*, $S = []$. A receiver, which should be waiting for $S$ to become equal to either a list pair or the empty list, will then either successfully match $S$ against a list pair $[M \mid R]$, whereupon $M$ will be made equal to $m$ and $R$ to $S'$, in which case the next message can be received on $S'$, or match $S$ against [], in which case no further messages can be received. By the *end-of-stream* we mean a tail of the stream that is not yet known to be a list pair or an empty list.

To achieve the effect of several senders on the same stream, there are two basic techniques: (1) The stream is split into several streams, one for each sender, which are interleaved into one by a *merger* process. (2) The language provides some form of atomic “test-and-set” unification, which allows *multiple writers* to compete for the end-of-stream.

*Merging* is typically achieved either by a tree of binary mergers, or by a multiway merger. The binary-merge tree is built by splitting a stream as necessary (Figure 7.5).

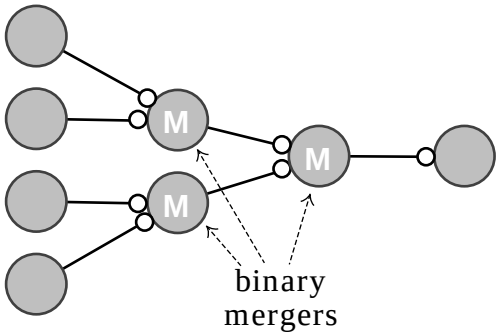


Figure 7.5. Binary merge network

Clearly, this technique does not have the constant delay property as the (best case) cost is O(log *m*) in the number of senders *m*. A *multiway merger* is a single merger process that allows input streams to be added and deleted dynamically (Figure 7.6).

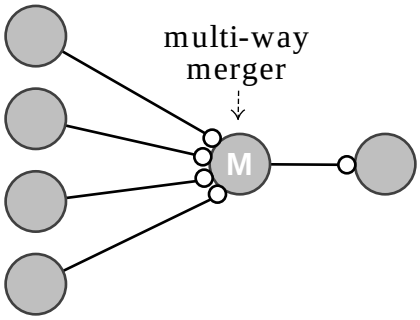


Figure 7.6. Multiway merger

A *constant delay multiway merger* cannot be expressed in most concurrent logic programming languages (AKL is an exception), but it is conceptually clean, and it is quite possible to provide one as a language primitive. We will assume that all multiway mergers have constant delay.

A number of disadvantages of merging follow.

- A merger process has to be created whenever there is the slightest possibility that several senders will send on the same stream. For many purposes this is not a problem. Once a multiway merger is created, adding and deleting input streams is fairly efficient. Yet, needing one feels like an overhead in an object-oriented context, where references to objects should be freely distributable.
- Explicit closing of all streams to all objects is necessary, since otherwise, either the program will eventually deadlock, or some objects will continue to be suspended, forever occupying storage.
- The merger process itself occupies storage, it also wastes storage when creating a new merged stream, and if it uses the standard mechanisms for suspension, it is likely to be (comparatively) inefficient.

- Messages have to be sent on the current end-of-stream variable, which is changed for every message sent. Assume that a process sends messages on streams which are embedded in a data structure, e.g., a tree containing objects. When a message has been sent on one stream, the new end-of-stream variable has to be "put back" into the tree. Usually this means building a new version of the tree, with the new end-of stream in place of the old, possibly reusing some unaffected parts of the old tree but necessarily allocating some new nodes. (However, if some form of single reference optimisation is employed by the language implementation new parts may reuse old storage: e.g., [Chikayama and Kimura 1987; Kahn and Saraswat 1990; Foster and Winsborough 1991]). Even worse, if this tree is to be shared with another process, where the possibility of the other process sending messages on the embedded streams cannot be excluded, two copies of the tree have to be created (allocating new nodes for at least one). All the streams have to be split in two (by merging), one for each copy.
- A serious problem is the *transparent message-distribution problem*. A message is usually a term $m(C_1, \ldots, C_n)$ where the $C_i$'s are message components. Suppose we want to implement a transparent message-distributor object, which when it receives a message, of any kind, will distribute it to a list of other objects. Without prior knowledge of the components of messages, the distributor object cannot introduce the merging required for stream components.

An advantage with merging is that it allows list pairs to be reused in the merger and deallocated by the receiver as soon as a message has been consumed. In some cases, in a system with fairly static object-structure, explicit closing of all streams as a means for controlled termination of objects can be considered an advantage. Another general advantage of all stream communicating systems is the implicit sequencing of messages from a source object to a destination. This simplifies synchronisation in many applications.

*Multiple writers* can only be expressed in some languages with atomic "test-and-set" unification. The drawbacks of multiple writers are summarised as follows:

- The cost for multiple writers is typically O($m$) per message, when there are $m$ senders, and is therefore even further from the constant delay property than merging.
- The delay is proportional to the number of messages that have been sent.
- An inactive sender may hold a reference to parts of the stream that have already been consumed by the intended receiver, making deallocation impossible.
- It is difficult to close a stream. Some termination detection technique, such as *short-circuiting* (see, e.g., [Shapiro 1986a]), has to be used. In practice, this outweighs the advantages of multiple writers.

An advantage of multiple writers is that no merger has to be created. Moreover, several messages can be sent on the same stream, and not only by having explicit access to the end-of-stream variable.

*Neither merging nor multiple writers* provides a general solution to the problem of automatic garbage collection of objects. There are only special cases, as exemplified by A'UM [Yoshida and Chikayama 1988].

### *7.3.2 Multiway Merging*

A constant delay merger written in AKL is presented in this section, as an indication of its expressiveness, and to illustrate a novel programming technique: bagof with feedback.

The way multiway mergers are normally used, a process requests a new input stream from the merger on one of the existing input streams, either by sending a special message which is caught by the merger, e.g., S = [new_stream($S_1$) | $S_2$], or by generalising streams to *stream trees* using an additional constructor, e.g., S = split($S_1$, $S_2$), which is the solution adopted here.

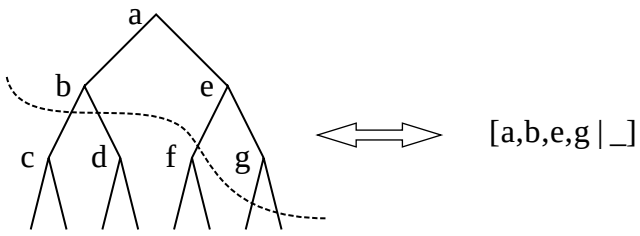


Figure 7.7. A stream tree with corresponding merged stream

In Figure 7.7, the relation between a (stylised) stream tree and a corresponding merged stream is shown. It is assumed that the merger has considered the messages up to the border drawn. Other messages (c, d, f) have been sent, i.e., are visible in the constraint store. It is easy to write a merger satisfying the first requirement using binary mergers, but the delay is then unbounded. The second requirement means that there should be a constant *k*, such that the number of computation steps required to detect the presence of and send the three known messages c, d, and f, should not exceed 3*k*.

To prepare the way for the complete solution, a simpler *ether* agent is shown first, which does not preserve the order of messages on input streams.

It is called with an input stream tree and an output stream. Ether uses unordered bagof to collect messages on the input streams (Figure 7.8).

```
ether(T, S) :=
        unordered_bagof(M, tree_member(M, T), S).
```

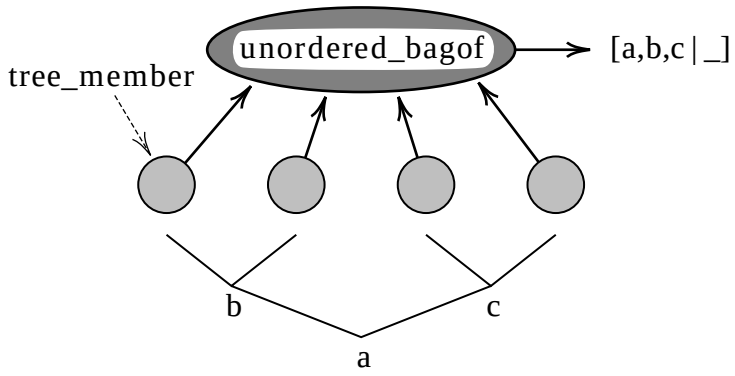


Figure 7.8. Ether merging

```
tree_member(M, split(S1, S2)) :-
    → ( true ? tree_member(M, S1)
      ; true ? tree_member(M, S2) ).
tree_member(M, [M1 | S]) :-
    → ( M = M1 ? true
      ; true ? tree_member(M, S) ).
tree_member([], Y) :-
    → fail.
```

The declarative reading of tree_member is that the message in its first argument is a member of the tree formed by split nodes and stream nodes in its second argument. The unordered bagof agent will collect these members to the output stream, when they become available along the fringe of this tree.

However, since the elements of an input stream will appear as different solutions, the unordered bagof may collect them in an arbitrary order, and thus their order is not preserved on the output stream. To remedy this, two processes are necessary, one bagof-based process to collect input streams, and one that guarantees sequentialisation of messages.

The improved merger consists of two co-operating agents: (1) a bagof agent that collects streams on which messages have been sent, and (2) a server that for each stream either removes a message (and feeds the rest to the generator in the bagof agent), splits the stream (and feeds the two new streams to the generator), or closes the stream (Figure 7.9).

```
merger(S0, S) :=
        server(B, A, S, 1),
        unordered_bagof(I, generator(I, [S0 | A]), B).
```

The generator process enters the streams as alternative solutions for the bagof agent. It constrains the streams to be known to have one of the possible forms, [], [_ | _], or split(_,_), before they are passed to the server.