Synchronous Operations as First-class Values

J.H. Reppy†

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Synchronous message passing via channels is an interprocess communication (IPC) mechanism found in several concurrent languages, such as **CSP**, **occam**, and **Amber**. Such languages provide a powerful selective I/O operation, which plays a vital role in managing communication with multiple processes. Because the channel IPC mechanism is "operation-oriented," only procedural abstraction techniques can be used in structuring the communication/synchronization aspects of a system. This has the unfortunate effect of restricting the use of selective I/O, which in turn limits the communication structure. We propose a new, "value-oriented" approach to channel-based synchronization. We make synchronous operations first-class values, called *events*, in much the same way that functions are first-class values in functional programming languages. Our approach allows the use of data abstraction techniques for structuring IPC. We have incorporated events into **PML**, a concurrent functional programming language, and have implemented run-time support for them as part of the **Pegasus** system.

I. Introduction

A common mechanism of interprocess communication (IPC) and synchronization in concurrent programming languages is synchronous message passing via channels (*channel IPC*). First exhibited in **CSP**[Hoare78], this mechanism has also been used in **occam**[INMOS83], **Amber**[Cardelli86] and the **Pegasus** system[RG86]. A channel IPC mechanism typically provides three synchronous operations: *send*, to send a message on a channel; *accept*, to read a message from a channel; and *select*, to allow non-deterministic communication on multiple channels.

These channel IPC operations are synchronous; for example, a process sending a message on a channel will block until another process executes a *matching*[1] accept operation on the same channel (and vice versa). The select operation allows a process to synchronize on the matching of one of a group of send and accept operations. If two, or more, of the group are immediately matchable, then one is selected non-deterministically, otherwise the first operation matched is selected. Associated with each operation is an optional guard, which must be true if the operation is to be matched, and an action that is executed if the operation is selected.

The select operation provides tremendous flexibility in allowing processes to manage communication with multiple processes. Unfortunately, there is no language structure to aid in the management of complex communication with a single process. One can use procedural abstraction to structure complex protocols, but the resulting abstraction cannot be used with select. In this paper, we present a channel IPC mechanism that uses

† Consultant. Author's current address: Department of Computer Science, Cornell University, Ithaca, NY 14853.

1. This use of the term *matching* should not be confused in any way with pattern matching.

© 1988 ACM 0-89791-269-1/88/0006/0250 $1.50

Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation
Atlanta, Georgia, June 22–24, 1988

first-class values, called *events*, to represent the synchronous operations. This "value-oriented" approach provides a uniform and extensible framework for programming synchronization.

We discuss the problems with the "operation-oriented" channel IPC mechanisms in more detail in the next section. Following this we present our "value-oriented" approach with synchronous operations as first-class values. We have incorporated our approach into **PML** (the system programming language of **Pegasus**[RG86]); in §IV we give an overview of **PML**, and in §V we give several examples of the use of events. Finally we briefly describe the implementation of events in **Pegasus**.

II. Channel IPC

Before we introduce our synchronization mechanism, we want to discuss the problems with channel IPC that motivate our design. For purposes of this section we will consider an "operation-oriented" channel IPC mechanism with three operations: send, accept and select.

We have been working on a system called **Pegasus**, that serves as a foundation for interactive applications[RG86]. A major aspect of this system is the **Pegasus Meta-language (PML)**, a concurrent functional programming language derived from **ML**[Milner85][MacQ85], with concurrency features derived from **Amber**. At an early stage of developing **Pegasus** we built experimental systems (written in C++) structured around channel IPC[RG86]. Our experience with these systems exposed several problems with the channel IPC mechanism as a programming notation. We describe these problems below and give realistic examples of where they can arise.

II.1 Building abstractions

One of the most important techniques of programming is building new abstractions from existing ones. The two most common abstraction mechanisms provided by modern languages are procedural and data abstraction. Because channel IPC is supported by *operations*, only procedural abstraction can be used in constructing new IPC abstractions.

As an illustration, consider *remote procedure call* (RPC). This useful abstraction can be implemented using channel IPC. The **PML** function

```
fun remote_F (x) == (send(x, arg_ch); accept (res_ch))
```

implements the client's side of such an abstraction[2]. Unfortunately, since the IPC protocol is hidden in a function, we are unable to use it in select.

The incompatibility of select and IPC abstractions is a significant problem in real systems; for example, consider a process that provides a *virtual terminal* interface between a window and a client. At any time, the process could receive input from either the client or the window manager; thus it must use a select. This means that the window manager's interface cannot be structured using an RPC-style interface, the IPC protocols must be open-coded to take advantage of select. This results in programs that are hard to maintain and debug.

A related problem is that of controlling access to channels. As noted above, if a process has access to a channel, then it can both read and write on that channel. Sometimes, however, it is desirable to limit access to a channel to read-only (or write-only). A simple example is a channel providing input from the outside world (such as from the mouse). Writing to such a channel makes no sense, but, again, procedural abstraction is the only way we can control access[3], and that costs us the use of select.

2. We describe **PML** in §IV. In this example, `remote_F` is being defined as a function, of one argument `x`, that first sends `x` on `arg_ch`, and then waits for a reply on `res_ch`. The reply is returned as the result of `remote_F`.

3. Obviously, run-time checks could be made for system channels such as the mouse, but in keeping with the statically typed flavor of **PML** we want these checks to be done at compile time.

II.2 Dynamic IPC configuration

A common structuring technique for concurrent systems is the use of server processes. Since the number of clients of a service can vary dynamically, the logical IPC structure also varies dynamically. Unfortunately, the select operation is statically structured. The guards allow select to be dynamically restricted, but there is no way to dynamically expand it. There are ways around this problem, but these tend to reduce efficiency and clarity.

II.3 Other synchronous operations

In addition to channel I/O operations, there are other ways a process may synchronize. Two common examples are waiting for another process to terminate, and delaying for some specified amount of real-time. In fact, some forms of the select operation incorporate real-time delay in a *timeout* feature. Unfortunately, in general, other synchronous operations are not part of the channel IPC framework, and thus cannot be included in select operations. While this is a minor point, we feel that the uniform treatment of synchronous operations is an important language design goal.

III. The Event Type

The problems raised in the previous section could be addressed by enriching the IPC mechanism, for example, by adding RPC functionality as a feature. This approach has the disadvantages of complicating the implementation and language definition, and of lost flexibility. Instead, our approach is to introduce an extensible view of synchronization that allows new synchronization abstractions to be built using the full power and flexibility of channel I/O.

The key to our solution is to decouple the act of synchronization from the description of synchronous operations. We introduce *event values*, which represent synchronous operations, such as waiting for a message on a channel. Events are first-class values and can be embedded in data-structures, passed as arguments and returned as results. By using this "value-oriented" approach to synchronization, we allow the full power of data abstraction techniques to be used in structuring IPC. Furthermore, our solution provides a uniform treatment of synchronization.

To synchronize on the event *ev*, a process uses the **sync** expression

sync *ev*

The result of this expression is the result value from *matching* the event *ev* (we discuss this more formally in §III.3). For example, if *ev* represents waiting for a message on a channel, then the result is the received message. Drawing an analogy with functions, **sync** is to event values as application is to function values.

Event values have the parameterized abstract type `Event[α]`, where the type variable α is the result type of matching the event (typing rules are given in §III.4). The internal structure of an event value is hidden, just as the internal implementation of a function value is hidden.

In the remainder of this section we describe event values and operations in detail, followed by the semantics of synchronization.

III.1 Base event values

The simplest event values are the *base* event values, which represent the actual synchronous operations. The two functions `transmit` and `receive` are used to build event values representing channel I/O operations. These are polymorphic functions with the types

```
transmit : (α×Channel[α]) → Event[Void]
receive : Channel[α] → Event[α]
```

where `Void` is a type with a single value. For example, the expression

```
transmit("hello world" , ch)
```

returns an event value, with type `Event[Void]`, that represents the operation of sending the string `"hello world"` on the channel `ch`. The expression

```
sync transmit ("hello world" , ch)
```

sends the message, and returns the void value.

Our framework accommodates other synchronous operations as well. The functions

```
delay : Integer → Event[Void]
wait  : ProcessId → Event[Void]
```

can be used to build events for real-time delay and synchronizing with process termination.

There are two distinguished event values, that are useful for boundary conditions:

```
anyevent : Event[Void]
noevent  : Event[α]
```

The value `noevent` is never matched, thus evaluating "**sync** `noevent`" causes the executing process to suspend forever. The value `anyevent`, on the other hand, is always matched immediately. So the expression "**sync** `anyevent`" is equivalent to the void constant value.

III.2 Composite event values

The power of the event mechanism is the ability to build new composite event values. There are two flavors of composite events: *choice events*, which provide the functionality of select; and *handler events*, which provide a way to filter the match results of events.

Choice events are constructed from one or more events by the **choose** expression, which has the form

choose { ev_1 | ··· | ev_n }

For example, the expression

```
sync choose { receive ch_a | receive ch_b }
```

will synchronize on receiving a message from either `ch_a` or `ch_b`. If both base events can be immediately matched, then one is selected non-deterministically. A `noevent` value in a **choose** is effectively ignored, since it cannot be matched. Thus, `noevent` can be used as `nil`, and **choose** as `cons` to dynamically build "lists" of events (see §V.2).

The other way of building composite events is the **handle** expression. This has the form

handle *ev* **with** *f*

where *ev* is an event valued expression and *f* is a function valued expression, called the *handler*. The handler is a filter that is applied to the match result of *ev*. For example, the following expression creates an input event that filters the input from the boolean channel `ch`[4].

```
handle receive ch with fn { true => "yes" | false => "no" }
```

To the users of this event, the channel looks like a string valued channel. A more useful application of handler events is to package IPC protocols (see §V.4).

III.3 Synchronization

While the semantics of synchronizing on base event values is self-evident (e.g., a `transmit` on a channel matches a `receive` on the same channel), synchronization on complex event values must be defined. For any event value *ev* there exists a semantically equivalent canonical event value $\hat{ev}$ with the form

choose { **handle** bev_1 **with** f_1 | ··· | **handle** bev_n **with** f_n }

where the bev_i are base event values. An event value *ev* is transformed into its canonical form $\hat{ev}$ by application of the semantics preserving rewrite rules in figure 1. By defining the synchronization semantics of $\hat{ev}$, we will have defined the semantics of *ev*.

4. The handler function in this example maps `true` to `"yes"` and `false` to `"no"`.

$ev \Rightarrow$ **choose** { ev }

$ev \Rightarrow$ **handle** ev **with fn** { x => x }

choose { ev_1 | $\cdots$ | ev_{j-1} | **choose** { ev'_1 | $\cdots$ | ev'_m } | ev_{j+1} | $\cdots$ | ev_n }
$\Rightarrow$ **choose** { ev_1 | $\cdots$ | ev_{j-1} | ev'_1 | $\cdots$ | ev'_m | ev_{j+1} | $\cdots$ | ev_n }

handle choose { ev_1 | $\cdots$ | ev_n } **with** f
$\Rightarrow$ **choose** { **handle** ev_1 **with** f | $\cdots$ | **handle** ev_n **with** f }

handle handle ev **with** f **with** f'
$\Rightarrow$ **handle** ev **with** $f' \circ f$

Figure 1. Event value rewrite rules

When a process evaluates the expression "**sync** $\hat{ev}$," it blocks until one of the bev_i can be matched. If more than one bev_i is matchable, then one is chosen non-deterministically, but satisfying the requirement that if there are matchable base events other than `anyevent`, then one of them will be selected[5]. If two of the bev_i match each other (e.g., a transmit and a receive on the same channel), then a run-time error occurs and an exception is raised. Once a base event, bev_k, has been chosen and matched, the handler f_k is applied to the match result. The result of this application is the result of the **sync** expression.

IV. PML Overview

We have incorporated events into the **Pegasus** system language **PML**. In this section we give a brief introduction **PML**, including the typing rules for events and some extended event notation. While space does not permit a comprehensive introduction to **PML**, we will provide enough information to allow readers to follow the examples in §V. Those familiar with standard ML[Milner85][MacQ85] should find little difficulty in reading **PML**.

IV.1 *Sequential PML*

PML is a statically typed polymorphic language with functions as first-class values. For example, the expression "**fn** { x => x }" is the polymorphic identity function and has the type $\alpha \rightarrow \alpha$. Discussion of ML-style polymorphic type systems can be found in [DM82] and [Cardelli85]. The declaration "**fun** `I(x) == x`" binds `I` to the identity function; the compiler will infer `I`'s type. Since `I` is polymorphic we can apply it to anything; for example the value of "`(I 1)`" is `1`, and the value of "`I(1, true)`" is the pair `(1, true)`. Notice that function application is denoted by juxtaposition. The value `()` is the null-tuple and has the type `Void`. Multiple bindings in a declaration are possible, for example "**val** `x == 1 && y == 2`."

PML contains a powerful concrete type declaration mechanism coupled with pattern matching. For example, the standard polymorphic list type and list append are defined by:

```
type List[α] == oneof { nil | cons(α×List[α]) };
fun append (nil, lst) == lst
|   append (lst, nil) == lst
|   append (cons(h, t), lst) == cons(h, append(t, lst))
```

The type declaration introduces `nil`, a polymorphic constant, and `cons`, a polymorphic *constructor*. The body of `append`, delimited by braces, is called a *match pattern*. A match pattern consists of a list of *pattern-expression* pairs separated by the "|" symbol. Another example is the polymorphic function

```
fun isNull nil == true | isNull * == false
```

5. This property of `anyevent` can be used to poll channels for input.

which tests for the empty list; note that "`*`" is the *wildcard* pattern. Match patterns are a generalization of the "case" construct found in many programming languages and are used in a number of PML constructs.

PML supports error handling by providing an *exception* mechanism. Exception identifiers are treated as special constructors of the predefined type "`Exn`." An expression of the form "**raise** *ex*" *raises* the exception denoted by the expression *ex*. The expression "*e* **except** `{ ZeroDiv => 0 }`" will return the value of "*e*," unless the exception `ZeroDiv` is raised, in which case it will return zero.

PML is an applicative style language and most values are immutable, but there are mutable values called *references*. References are created by the `ref` constructor, and updated by assignment. The unary operator "`!`" is used for dereferencing. As an example, the expression

```
let val x == ref 1 in x := !x+1; !x ni
```

yields the result 2.

IV.2 Concurrency features

The concurrent features of PML were originally derived from **Amber**. New processes are created using expressions of the form

process *e*

which creates a process to evaluate the expression *e*. As with functions, the process has a *closure* (i.e., the free variables of *e*). Since it is undesirable for processes to share side-effects, the values in a process' closure are copied rather than shared[6].

Processes communicate via typed channels. New channels are created by declarations of the form

chan ch_1 **of** τ_1 && $\cdots$ && ch_n **of** τ_n

Each declared channel ch_i has the type `Channel`$[\tau_1]$. In order to protect the PML type system the declared types of channels are restricted to mono-types[7].

IV.3 Typing rules

Events fit quite nicely into the PML type system. We use the standard technique of inference rules (see [Cardelli85]) for presenting the event operations' typing rules. The types of the base event constructor functions were given in I.1, the rules for composite events and **sync** are given in figure 2.

$$\frac{\mathbf{Env} \vdash ev \ : \ \texttt{Event}[\tau]}{\mathbf{Env} \vdash (\ \mathbf{sync}\ ev\ ) \ : \ \tau}$$

$$\frac{\begin{array}{c}\mathbf{Env} \vdash ev_1 \ : \ \texttt{Event}[\tau] \\ \cdots \\ \mathbf{Env} \vdash ev_n \ : \ \texttt{Event}[\tau]\end{array}}{\mathbf{Env} \vdash (\ \mathbf{choose}\ \{\ ev_1 \mid \cdots \mid ev_n\ \}\ ) \ : \ \texttt{Event}[\tau]}$$

$$\frac{\begin{array}{c}\mathbf{Env} \vdash ev \ : \ \texttt{Event}[\sigma] \\ \mathbf{Env} \vdash f \ : \ \sigma \rightarrow \tau\end{array}}{\mathbf{Env} \vdash (\mathbf{handle}\ ev\ \mathbf{with}\ f) \ : \ \texttt{Event}[\tau]}$$

Figure 2. Type inference rules

6. In a distributed implementation copying is obviously required, but for shared-memory or single CPU implementations the compiler can use type information to avoid copying of immutable values.

7. We can weaken this restriction using *weak* type variables, as is done with reference types in ML, but it is beyond the scope of this paper.

Event values are abstract; they cannot be decomposed. A user of an event value knows only its match result type.

IV.4 Extending the event notation

Certain forms are fairly common, so **PML** provides shorthand for them. The conventional channel I/O operations are defined by

```
fun send (x, ch) == sync transmit (x, ch);
fun accept ch == sync receive ch
```

Since the handler of a **handle** expression is often a function expression (i.e., "**fn** *mp*," where *mp* is a match pattern), we allow the short-hand "**handle** *ev* **with** *mp*" for the expression "**handle** *ev* **with fn** *mp*".

A **select** expression of guarded events is also provided[8]. The general form is

```
select
{ when $b_1$ => $ev_1$
| when $b_2$ => $ev_2$
. . .
| when $b_n$ => $ev_n$
| otherwise => exp
}
```

where the b_i are boolean valued expressions, and the ev_i are event valued expressions. The "**when** b_i =>" parts, called guards, are optional. Leaving a guard out is equivalent to the guard "**when** true =>." The **otherwise** clause is also optional; leaving it out is equivalent to the clause "**otherwise** => **raise** SelectFail."

Roughly, the meaning of the **select** expression is to synchronize on a choice of those ev_i's with true guards. If no guard is true, then the **otherwise** clause is evaluated and there is no synchronization. Formally, the general form, given above, is semantically equivalent to

```
let
val $B_1$ == $b_1$ && $EV_1$ == if $B_1$ then $ev_1$ else noevent fi;
val $B_2$ == $b_2$ && $EV_2$ == if $B_2$ then choose { $EV_1$ | $ev_2$ } else $EV_1$ fi;
. . .
val $B_n$ == $b_n$ && $EV_n$ == if $B_n$ then choose { $EV_{n-1}$ | $ev_n$ } else $EV_{n-1}$ fi
in
   if $B_1$ or ··· or $B_n$ then
      sync $EV_n$
   else
      exp
   fi
ni
```

V. Using Events

To get a real feel for the power of events, one needs to see some concrete examples. In this section we give some detailed examples of the use of events to solve concurrent programming problems. In particular, we show how events can be used in ways that are not possible using the "operation-oriented" channel IPC. Most of these examples are simplified versions of techniques we are using in the **Pegasus** system.

8. We will use "**select**" to refer to this form and continue to use "select" to refer to selective I/O.

V.1 Controlling channel access

One of the problems with channel IPC raised in §II.1 was controlling access to channels. Using events, we can create abstract unidirectional views of channels. For example, the functions

```
fun mkReadView(ch) == receive ch
&& mkWriteView(ch) == fn { x => transmit(x, ch) }
```

provide a way to construct such views. Notice that `mkWriteView` returns an event valued function. The values produced by these functions can be used with select; which would not be the case if we had used `send` and `accept`.

V.2 Dynamic event lists

Server processes sometimes need to manage dynamically varying lists of channels. The following function builds an event to monitor a (non-empty) list of channels. Synchronizing on the event returns the message received and the index of the source channel.

```
fun ReadChList(chlst) == let
   fun RCL ( nil, *) == noevent
   |   RCL ( cons(ch, rest), i) == choose
         { handle receive ch with { msg => ( msg, i) }
         | RCL ( rest, i+1)
         }
   in RCL (chlst, 0) ni
```

Notice that the recursive function `RCL` binds the channel indices in the handlers.

V.3 Fault-tolerant channel I/O

Another example of building new IPC abstractions is "fault-tolerant" channel I/O. Consider a server process that allocates a dedicated channel for each client and maintains a dynamic event structure of receive events on these channels, as in the previous section[9]. If a client terminates without informing the server, problems can result. One solution is to define a new "fault-tolerant" input event, for channel `ch` dedicated to process `p`, that raises an exception if `p` is dead.

```
choose
{ receive ch
| handle wait p with { () => raise (DeadProc p) }
}
```

V.4 Supporting remote procedure call

As we mentioned above, the RPC paradigm is very useful in the construction of concurrent systems. Using events we can implement an RPC protocol quite easily. The client is provided with an event valued function `remote_F` that hides the details of the protocol.

```
fun remote_F(x) == handle transmit (x, arg_ch) with { accept(res_ch) }
```

Unlike the version of `remote_F` in §II.1, this can be used with select. As far as the client is concerned, `remote_F` is exactly like any base event function. To "call" `F` with *arg*, the client evaluates

```
sync remote_F(arg)
```

On the server side there is a corresponding event value that implements the server's half of the protocol.

```
val req_F == handle receive arg_ch with { x => send(F x, res_ch) }
```

If the server provides more than one service, it can use **`select`** to monitor requests.

9. The Pegasus window manager uses this strategy.

We can extend this protocol to pass exceptions, raised when the server services F, to the client. We wrap the result messages in a concrete type

```
    type ResultMsg == oneof { RES of ArgType | EX of Exn }
```

and use the following protocol.

```
    fun remote_F(x) == handle transmit (x, arg_ch) with
      { () => case accept res_ch of
          { RES ( y ) => y
          | EX ( ex ) => raise ex
          }
      };
    val req_F == handle receive arg_ch with
          { send (RES(F x) except { ex => EX(ex) }, res_ch) }
```

V.5 Putting it all together

We have shown above that events provide a general mechanism for structuring a number of different concurrent programming problems. In this section we demonstrate the flexibility of our approach, by showing how the various features of the previous examples can be combined into a more complex protocol.

Say we want a server which provides an RPC interface to clients, and which needs to know which client is making the request. Furthermore, we want the server to be informed of client failures (as in §V.3) and we want to pass exceptions to the client (as in the previous example). The following two functions, respectively, build the client and server sides of the protocol, given the client process id and the argument and result channels.

```
    fun mk_remote_F (arg_ch, res_ch) == fn
      { x => handle transmit(x, arg_ch) with
        { () => case accept res_ch of
          { RES ( y ) => y
          | EX ( ex ) => raise ex
          }
        }
      }
    && mk_req_F_event  (client_pid, arg_ch, res_ch) == choose
      { wait client_pid => raise (DeadProc client_pid)
      | handle receive arg_ch with
          { send (RES(F x) except { ex => EX(ex) }, res_ch) }
      };
```

The server process can then use the technique of §V.2 to manage a dynamic list of clients.

VI. Implementation

We have implemented events on a single CPU machine as part of a complete reimplementation of **Pegasus**. The implementation consists of the event manager and a collection of other managers, such as the channel manager, for the various kinds of base events. The managers are monitors written in C++. Event values are represented by heap allocated tree structures, with the base events as leaves, and the choice and handler events as interior nodes.

A process synchronizes on an event *ev* by calling the event manager. The event manager parses *ev*, constructing a list of its base events, and a handler list for each base event. The handler lists correspond to the composition of handler functions in §III.3 (figure 1). The event manager then logs the base events with the appropriate base event managers and, assuming no base event is immediately matched, it suspends the process. The base event managers notify the event manager when logged base events are matched. The event manager then applies the matched base event's handlers to the match result, and returns the result to the process.

The order in which the base events of an event are logged affects the behavior of a system. For example, if a process repeatedly synchronizes on the value "**choose** { bev_1 | bev_2 }" and if we always log the base events in the order bev_1, bev_2, then if bev_1 is always immediately matchable, bev_2 will be *starved.* To avoid this problem, we log the base events in a pseudo-random order. This guarantees, with high probability, that if a base event bev_i of an event value ev is matchable and a process repeatedly synchronizes on ev, then bev_i will eventually be matched. In otherwords, `select` and `choose` are *fair*[10].

VII. Conclusions

We have presented a new approach to synchronization that uses first-class values, called event values, to define synchronous operations. The major benefit of this approach is that the full power of data abstraction can be used in structuring the IPC of concurrent programs. New abstractions, such as RPC-style communication, can be built and used without losing the flexibility and power of selective I/O. Our approach also has the advantage of providing a uniform framework for managing other synchronous actions, such as real-time delay and process termination. We have given a number of realistic examples of the use of events. Finally we sketched the implementation of events in the **Pegasus** system.

Acknowledgements

Tim Teitelbaum's comments greatly improved the organization of this paper. I would also like to acknowledge my **Pegasus** co-conspirators, Emden Gansner and Steve North, for their comments.

References

[AS83] Andrews, G.R., Schneider, F.B. "Concepts and Notations for Concurrent Programming," *Computing Surveys*, V. 15, Nr. 1, March, 1983, pp. 3-43.

[Cardelli85] Cardelli, L. "Basic polymorphic typechecking," *Polymorphism*, V. 2, Nr. 1, January 1985.

[Cardelli86] Cardelli, L. "Amber," *Combinators and Functional Programming Languages*, Lecture Notes in Computer Science 242, Springer-Verlag, 1986, pp.21-47.

[DM82] Damas, L., Milner, R. "Principal Type-Schemes for Functional Programs," *Conference Record of the 9th ACM Symposium on Principles of Programming Languages*, January 25-27, 1982, pp. 207-212.

[Hoare78] Hoare, C.A.R. "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8, August, 1978, pp. 666-677.

[INMOS83] INMOS Limited. *Occam Programming Manual*, Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[Milner85] Milner, R. "The Standard ML Core Language," *Polymorphism*, V. 2, Nr. 2, October 1985.

[MacQ85] MacQueen, D. "Modules for Standard ML," *Polymorphism*, V. 2, Nr. 2, October 1985.

[RG86] Reppy, J.H., Gansner, E.R. "A Foundation for Programming Environments," *Proceedings of the Second ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, December 9-11, 1986, pp. 218-227.

10. Formally this is *weak fairness*, although for practical purposes we can assume *strong fairness* (ie., that "*eventually*" is bounded).