A Processor Farm Example in Manifold

I. Herman and F. Arbab
Department of Interactive Systems
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
e-mail: ivan@cwi.nl, farhad@cwi.nl

Management of the communications among a set of concurrent processes arises in many applications and is a central concern in parallel computing. In this paper we introduce MANIFOLD: a *coordination* language whose sole purpose is to describe and manage complex interconnections among independent, concurrent processes. In the underlying paradigm of this language the primary concern is not with *what* functionality the individual processes in a parallel system provide. Instead, the emphasis is on *how* these processes are inter-connected and how their interaction patterns change during the execution life of the system. We also describe an approach to define an operational semantics for MANIFOLD. As an example application, we present the skeleton of a computing farm model described in MANIFOLD. The complete MANIFOLD program for this example is presented elsewhere.

1. INTRODUCTION

Specification and management of the communications among a set of concurrent processes is at the core of many problems of interest to a number of contemporary research trends. Communication issues also come up in virtually every other type of computing, and have influenced the design (or at least, a few constructs) of most programming languages. However, not much effort has been spent on conceptual models and languages whose sole prime focus of attention is on the coordination of interactions among processes.

This paper gives a brief overview of the MANIFOLD language and and example of its application to describe a processor farm. MANIFOLD is a parallel programming language where processes called *manifolds* use an event-driven control mechanism to coordinate the communications among other processes (manifolds as well as external). As such, like LINDA [1, 2], it is primarily a coordination language. However, there is no resemblance between LINDA and MANIFOLD, nor is there any similarity between the underlying models of these two languages. Inter-process communication in MANIFOLD is through broadcast of events and a dynamic data-flow network, built out of *streams* carrying *units* of data.

The rest of this paper is organized as follows. In §2, a brief description of the **MANIFOLD** model and language is presented. In §3, the expressive power of **MANIFOLD** is demonstrated through an example related to high performance computing.

2. THE **MANIFOLD** LANGUAGE

In this section we give a brief and informal overview of the **MANIFOLD** language.

The sole purpose of the **MANIFOLD** language is to describe and manage complex communications and interconnections among independent, concurrent processes. A detailed description of the syntax and the semantics of the **MANIFOLD** language and its underlying model is given elsewhere [3]. Other reports and articles contain more examples of the use of the **MANIFOLD** language [4, 5, 6, 7].

The basic components in the **MANIFOLD** model of computation are *processes*, *events*, *ports*, and *streams*. A process is a *black box* with well defined ports of connection through which it exchanges *units* of information with the other processes in its environment. The internal operation of some of these black boxes are indeed written in the **MANIFOLD** language, which makes it possible to open them up, and describe their internal behavior using the **MANIFOLD** model. These processes are called *manifolds.* Other processes may in reality be pieces of hardware, programs written in other programming languages, or human beings. These processes are called *atomic processes* in **MANIFOLD**. In fact, an atomic process is any processing element whose external behavior is all that one is interested in observing at a given level of abstraction. In general, a process in **MANIFOLD** does not, and need not, know the identity of the processes with which it exchanges information. Figure 1 shows an abstract representation of a **MANIFOLD** process.

Ports are regulated openings at the boundaries of processes through which they exchange units of information. The **MANIFOLD** language allows assigning special filters to ports for screening and rebundling of the units of information exchanged through them. These filters are defined in a language of extended regular expressions over bit string. Any unit received by a port that does not match its regular expression is automatically diverted to the `error` port of its manifold and raises a `badunit` event (see later sections for the details of events and their handling in **MANIFOLD**). The regular expressions of ports are an effective means for "type checking" and can be used to assure that the units received by a manifold are "meaningful".

Interconnections between the ports of processes are made with *streams.* A stream represents a flow of a sequence of units between two ports. Conceptually, the capacity of a stream is infinite. Streams are dynamically constructed between ports of the processes that are to exchange some information. Adding or removing streams does not directly affect the status of a running process. The constructor of a stream (which is a manifold) need not be the sender nor the receiver of the information to be exchanged: any third party manifold process can define a connection between the ports of a producer process and a consumer process. Furthermore, stream definitions in **MANIFOLD** are generally additive.

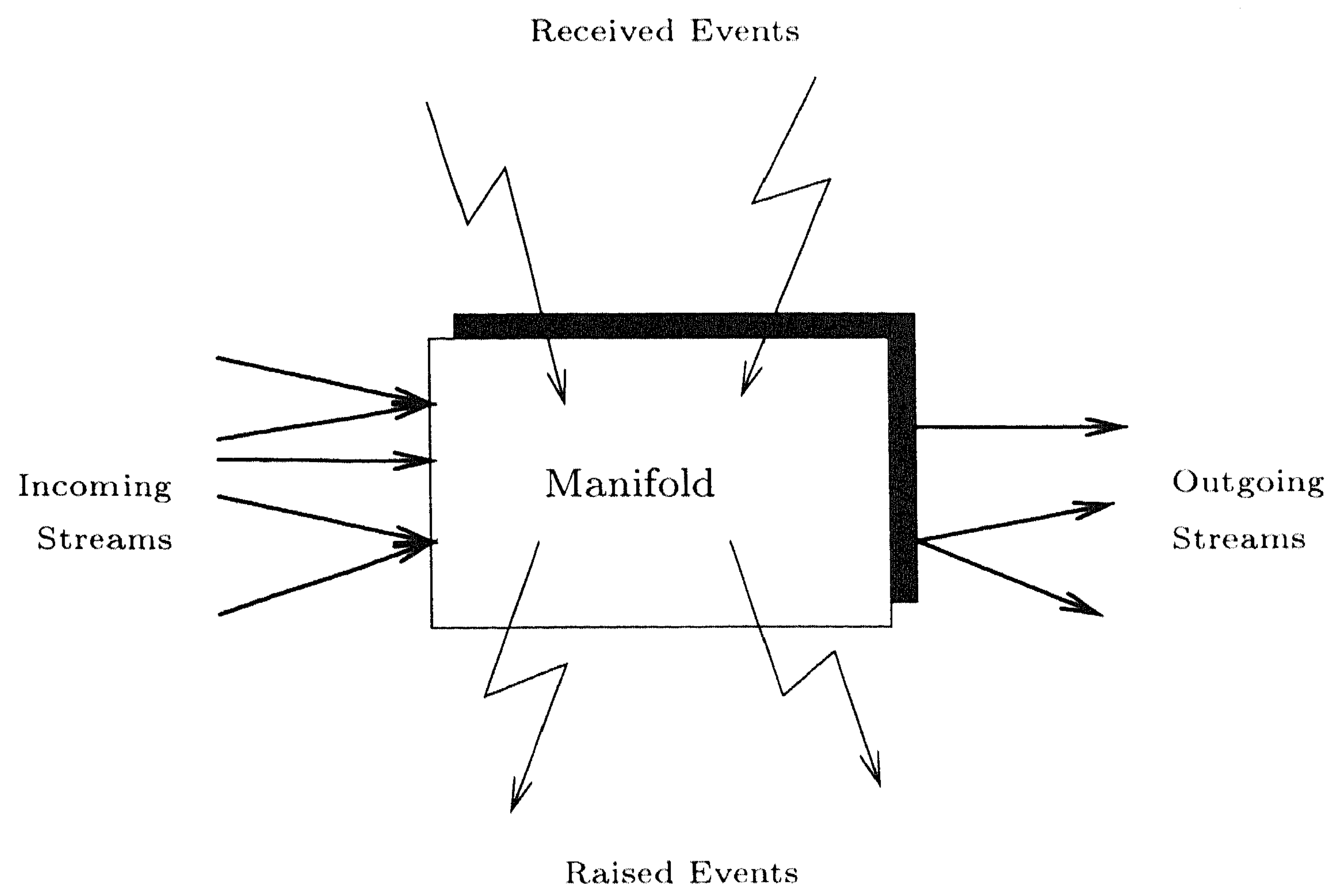

FIGURE 1. The model of a process in **MANIFOLD**

Thus a port can simultaneously be connected to many different ports through different streams (see for example the network in Figure 2). The flows of units of information in streams are automatically replicated and merged at outgoing and incoming port junctions, as necessary. The units of information exchanged through ports and streams are produced and consumed at the two ends of a stream with their relative order preserved. The consumption and production of units via ports by a process is analogous to read and write operations in conventional programming languages. Depending on its attribute, when a stream is removed, its queue of unconsumed units can either remain at its consumer port for future processing, or it can be flushed.

Independent of the stream mechanism, there is an event mechanism for information exchange in **MANIFOLD**. Contrary to units in streams, events are *atomic* pieces of information that are *broadcast* by their sources in their environment. In principle, *any* process in an environment can pick up a broadcast event. In practice, usually only a few processes pick up occurrences of each event, because only they are "tuned in" to their sources. Occurrences of the same event from the same source can override each other from the point of view of some observer processes, depending on the difference between the speed of the source and the reaction time of an observer. This provides an automatic *sampling* mechanism for observer processes to pick up information from their environment which is particularly useful in situations where a potentially significant mismatch between the speeds of a producer and a consumer is possible. Events are the primary control mechanism in **MANIFOLD**.

Once an event is raised, its source generally continues with its processing, while the event occurrence propagates through the environment independently. Event occurrences are observed asynchronously and once picked up, they preemptively cause a change of state in the observer. Communication of processes through events is thus inherently asynchronous in **MANIFOLD**.

Each manifold defines a set of events and their sources whose occurrences it is interested to observe; they are called the *observable* set of events and sources, respectively. It is only the occurrences of observable events from observable sources that are picked up by a manifold. Once an event occurrence is picked up by an observer manifold, it may or may not cause an immediate reaction by the observer. In general, each state in a manifold defines the set of events (and their sources) that are to cause an immediate reaction by the manifold while it is in that state. This set is called the *preemption set* of a manifold state and is a subset of the observable events set of the manifold. Occurrences of all other observable events are *saved* so that they may be dealt with later, in an appropriate state.

Each state in a manifold defines a pattern of connections among the ports of some processes. The corresponding streams implementing these connections are created as soon as a manifold makes a state transition (caused by an event) to a new state, and are deleted as soon as it makes a transition from this state to another one. This is discussed in more detail in §2.2.

2.1. MANIFOLD *Definition*

A manifold definition consists of a *header*, *public declarations*, and a *body*. The header of a manifold definition contains its name and the list of its formal parameters. The public declarations of a manifold are the statements that define its links to its environment. It gives the types of its formal parameters and the names of events and ports through which it communicates with other processes. A manifold body primarily consists of a number of *event handler blocks*, representing its different execution-time states. The body of a manifold may also contain additional declarative statements, defining *private* entities. For an example of a very simple manifold, see Listing 1 which shows the **MANIFOLD** source code for a simple program. The skeleton of a more complete manifold program is presented in §3 and other, more complex examples have been published elsewhere (e.g., [6, 8].) Declarative statements may also appear outside of all manifold definitions, typically at the beginning of a source file. These declarations define global entities which are accessible to all manifolds in the same file, provided that they do not redefine them in their own scopes.

Conceptually, each activated instance of a manifold definition – a *manifold* for short – is an independent process with its own virtual processor. A manifold processor is capable of performing a limited set of actions. This includes a set of *primitive actions*, plus the primary action of setting up *pipelines*.

Each event handler block describes a set of actions in the form of a *group* construct. The actions specified in a group are executed in some non-deterministic order. Usually, these actions lead to setting up *pipelines* between various ports of different processes. A *group* is a comma-separated list of members enclosed

```
// This is the header (there are no arguments):
example()
// These are the public declarations:
// Two ports are visible from the outside of the
//    manifold ‘‘example’’: one is an input port
//    and the other is an output one.
// In fact, these ports are the default ones.
port in  input.
port out output.
{
     // The body of the manifold begins here.
     //
     // private declarations:
     //    three process instances are defined:
     process A is A_type.
     process B is B_type.
     process C is C_type.

     // First block (activated when ‘‘example’’
     //    becomes active)
     // The processes described above are
     //    activated on their turn
     //    in a ‘‘group’’ construct:
     start: (activate A,activate B,activate C)
             ; do begin.

     // A direct transfer to this block has been
     //    given from ‘‘start’’.
     // Three pipelines in a group are set up:
     begin: (A -> B,output -> C,input -> output).

     // Event handler for the event ‘‘e1’’;
     //    several pipelines are
     //    set up
     e1: (B      -> input,
          C      -> A,
          A      -> B,
          output -> A,
          B      -> C,
          input  -> output).

     // Event handler for the event ‘‘e2’’;
     //    a single pipeline is set up
     e2: C -> B.
}
```

LISTING 1. Central Part of the Farm Node

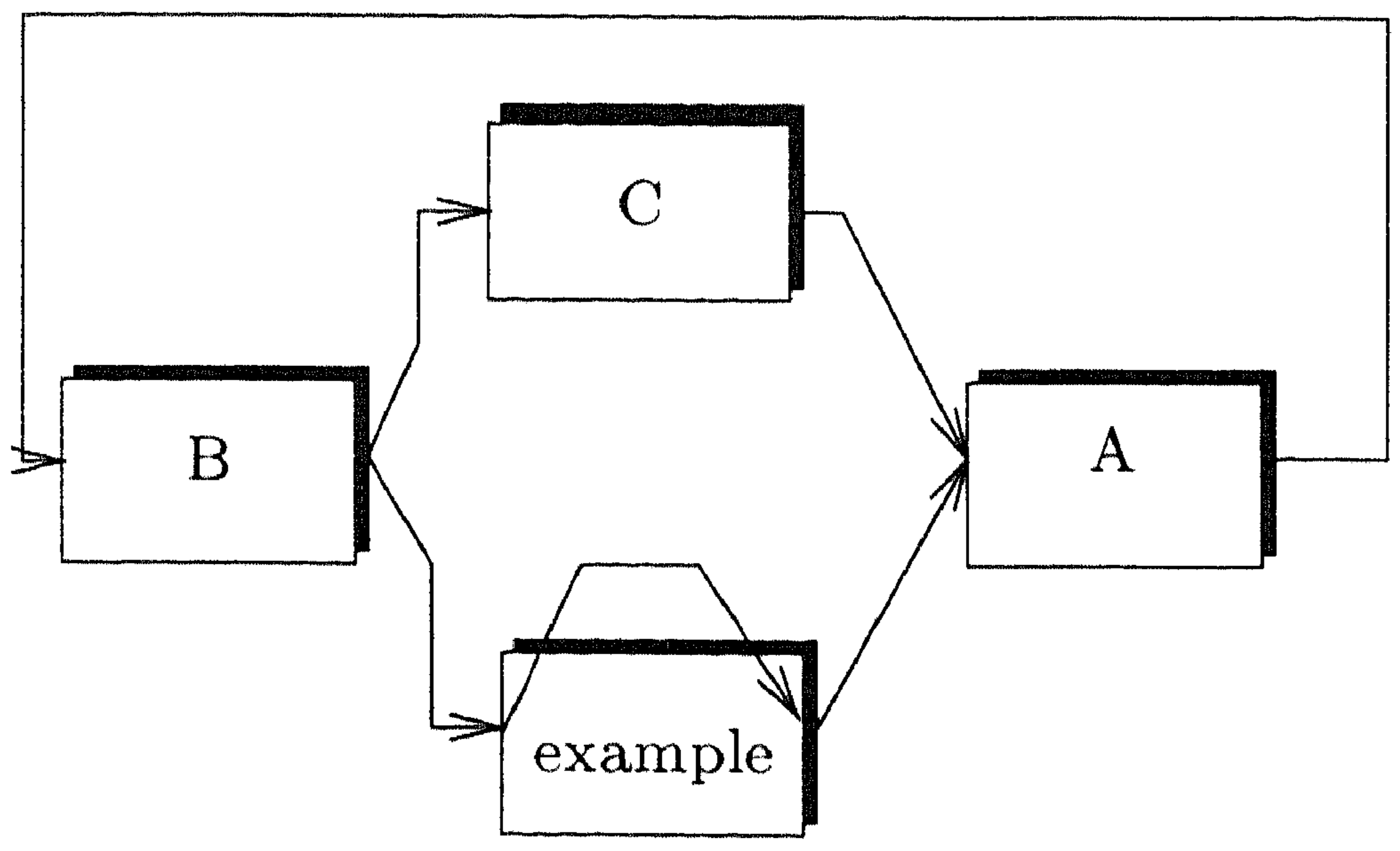

FIGURE 2. Connections set up by the manifold `example` on event `e1`.

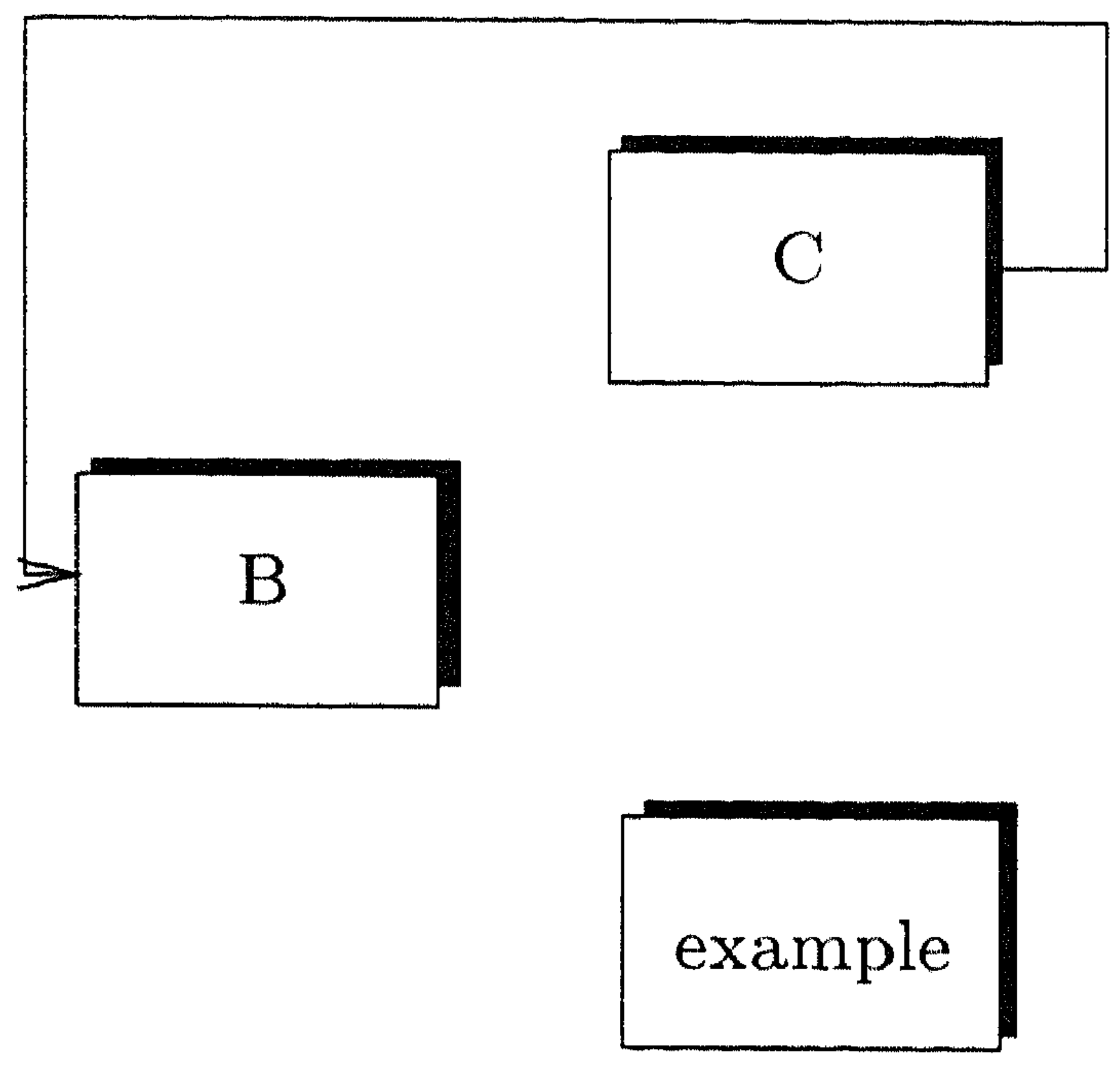

FIGURE 3. Connections set up by the manifold `example` on event `e2`

in a pair of parentheses. In the degenerate case of a singleton group (which contains only one member) the parentheses may be deleted. Members of a group are either primitive actions, pipelines, or groups. The setting up of pipelines within a group is simultaneous and atomic. No units flow through any of the streams inside a group before all of its pipelines are set up. Once set up, all pipelines in a group operate in parallel with each other.

A *primitive action* is typically *activating* or *deactivating* a process, *raising* an event, or a *do* action which causes a transition to another handler block without an event occurrence from outside. A *pipeline* is an expression defining a tandem of streams, represented as a sequence of one or more groups, processes, or ports, separated by right arrows. It defines a set of simultaneous connections among the ports of the specified groups and processes. If the initial (final) name in such a sequence is omitted, the initial (final) connection is made to the current input (output) port. Inside a group, the current input and output ports are the input and output ports of the group. Elsewhere, the current input and output ports are `input` and `output`, i.e., the executing manifold's standard input and output ports. As an example, Figure 2 shows the connections set up by the manifold process `example` on Listing 1, while it is in the handling block for the event `e1` (for the details of event handling see §2.2). Figure 3 shows the connections set up in the handling block for the event `e2`.

In its degenerate form, a pipeline consists of the name of a single port or process. Defining no useful connections, this degenerate form is nevertheless sometimes useful in event handler blocks because it has the effect of defining the named port or process as an observable source of events and a member of the preemption set of its containing block (see §2.4).

An event handler block may also describe sequential execution of a series of (sets of) actions, by specifying a list of pipelines and groups, separated by the semicolon (`;`) operator. In reaction to a recognized event, a manifold processor finds its appropriate event handler block and executes the list of sequential sets of actions specified therein. Once the manifold processor is through with the sequence in its current block, it terminates.

2.2. Event handling

Event handling in **MANIFOLD** refers to a preemptive change of state in a manifold that observes an event of interest. This is done by its manifold processor which locates a proper event handler for the observed event occurrence. An event handler is a labeled block of actions in a manifold. In addition to the event handling blocks explicitly defined in a manifold, a number of default handlers are also included by the **MANIFOLD** compiler in all manifolds to deal with a set of predefined system events. The manifold processor makes a transition to an appropriate block (which is determined by its current state, the observed event and its source), and starts executing the actions specified in that block. The block is said to *capture* the observed event (occurrence). The name of the event that causes a transfer to a handling block, and the name of its source, are available in each block through the pseudonyms `event_name` and `event_source`, respectively.

The manifold processor in a given manifold is sensitive to (i.e., interested in) only those events for which the manifold has a handler. All other events are to be ignored. Thus, events that do not match any label in this search do not affect the manifold in any way (however, see §2.5 for the case of called manners). Similarly, if the appropriate block found for an event is the keyword `ignore`, the observed event is ignored. Normally, events handled by the current block are also ignored.

The concept of an event in **MANIFOLD** is different than the concepts with the same name in most other systems, notably simulation languages [9], and CSP [10, 11]. Occurrence of an event in **MANIFOLD** is analogous to a flag that is raised by its source (process or port), *irrespective* of any communication links among processes. The source of an event continues immediately after it raises its flag, independent of any potential observers. This raised flag can potentially be seen by any process in the environment of its source. Indeed, it can be seen by any process to which the source of the event is *visible*. However, there are no guarantees that a raised flag will be observed by anyone, or that if observed, it will make the observer react immediately.

2.3. Event handling blocks

An event handling block consists of a comma-separated list of one or more block labels followed by a colon (:) and a single body. The body of an event handling block is either a group member (i.e., an action, a pipeline, or a group), or a single manner call (see §2.5).

Event handler block labels are patterns designating the set of events captured by their blocks. Blocks can have multiple labels and the same label may appear more than once marking different blocks. Block labels are filters for the events that a manifold will react to. The filtering is done based on the event names and their sources. Event sources in **MANIFOLD** are either ports or processes.

The most specific form of a block label is a dotted pair $e.s$, designating event e from the source (port or process) s. The form e is a short-hand which captures event e coming from any source.

2.4. Visibility of event sources

Every process instance or port defined or used anywhere in a manner (see §2.5) or manifold is an *observable* source of events for that manner or manifold. This simply means that occurrences of events raised by such sources (only) will be picked up by the executing manifold processor, provided that there is a handling block for them. The set of all events from observable sources that match any of the block labels in a manner or manifold is the set of observable events for that manner or manifold. The set of observable events of an executing manifold instance may expand and shrink dynamically due to manner calls and terminations (see §2.5). Depending on the state of a manifold processor (i.e., its current block), occurrences of observable events cause one of two possible actions: preemption of the current block, or saving of the event occurrence.

In each block, a manifold processor can react to only those events that are in the *preemption set* of that block. The **MANIFOLD** language defines the preemption set of a block to contain only those observable events whose sources appear in that block. This means that, while the manifold processor is in a block, except for the manifold itself, no process or port other than the ones named in that block can be the source of events to which it reacts immediately. A manifold can always internally raise an event that is visible only to itself via the `do` primitive action.

2.5. Manners

The state of a manifold is defined in terms of the events it is sensitive to, its visible event sources, and the way in which it reacts to an observed event. The possible states of a manifold are defined in its blocks, which collectively define its behavior. It is often helpful to abstract and parameterize some specific behavior of a manifold in a subroutine-like module, so that it can be invoked in different places within the same or different manifolds. Such modules are called *manners* in **MANIFOLD**.

A *manner* is a construct that is syntactically and semantically very similar to a manifold. Syntactically, the differences between a manner definition and a manifold definition are:

1. The keyword `manner` appears in the header of a manner definition, before its name.
2. Manner definitions cannot have their own port definitions.

Semantically, there are two major differences between a manner and a manifold. First, manners have no ports of their own and therefore cannot be connected to streams. Second, a manner invocation never creates a new processor. A manifold activation always creates a new processor to "execute" the new instance of the manifold. To invoke a manner, however, the invoking processor itself "enters and executes" the manner.

3. AN EXAMPLE PROGRAM

The development of the **MANIFOLD** system ([3, 4, 6]), which started in spring 1990, has always been motivated by practical considerations. While developing a new computing model and a language which is describable and analyzable by theoretical means as well (see [12]), we were always influenced by practical issues, by implementability, and ease of use. It is not a coincidence that, at a very early stage of the development, we already tried to describe and formalize programs which had also a practical "touch" ([4]). Working on practical examples has been a continuing activity in conjunction with the effective implementation work ([8, 5]). One of the most complex example of a **MANIFOLD** application which has been published up to now is the formalization of the GKS[15] input model (see [8]). In view of these, it was a quite natural step for us to look at more complex examples, now that the first experimental implementation of the **MANIFOLD** system is also operational ([6, 13]). The example we present in what follows is the outcome of our latest efforts in this direction, which led to [7].

3.1. The computing farm model

In the last few years, it has become more feasible to use a large number of possibly loosely coupled, very powerful processors in parallel to solve highly computing intensive applications. The notion of *computing farms* has thus come to the fore. In the general model of a computing farm, it is immaterial whether its individual processors are processing elements connected via a hardware bus or other direct communication media (e.g., a transputer network and hardware) or full-blown workstations connected via a local area network. What is important is that different and sometimes complicated communication patterns are set up to solve a given application problem. The exact topology of communication depends on the application proper. It is also part of the underlying model, although rarely stated explicitly, that the individual processors are fairly autonomous.

This means that a computing farm can be considered to be a large-scale MIMD and very coarse-grained parallel system. In view of the practical importance of such computing environments, it is important to have a flexible means to describe and/or modify these various topologies. We feel that a language like **MANIFOLD** is particularly well suited for such tasks.

In this section, a simple topology, namely a ring, is described in **MANIFOLD**. It must be stressed that other alternative topologies are also feasible and useful. In [14], for example, Green gives a whole range of alternative forms in relations to, e.g., distributed ray tracing. The programming techniques presented in the following subsections are easily adaptable for other communication patterns as well. In this paper, the aim is *not* to give an exhaustive presentation of all possible computing farm models; instead, we intend to show that the expressive power of **MANIFOLD** is particularly well suited for such applications.

The specific model used in this example is taken from [14], and reflects a bias toward the transputers. This model is *not* the most appropriate model for a processor farm in **MANIFOLD**. Processor farms can be modeled more easily in **MANIFOLD**. However, we keep the original model of [14] in this example for "pedagogical" reasons.

3.1.1. The abstract model

The original and simplest computing farm model (as described, e.g., in [14]) is as follows. A farm consists of two types of processors: a single controller, and one or more farm processors. The controller generates new tasks and transfers them to the farm processors as required. It also collects the results as they are produced by the farm processors.

A farm processor consists of two distinct parts: an application-specific part and a wrapper. Figure 4 shows an overview of the original structure.

The router process for tasks on a farm node contains a buffer to store tasks so that a new task can be passed to the application with a minimal latency when it becomes idle (i.e., this buffer plays the role of a cache). On receipt of a new task, a router places it in its buffer if the buffer is not full. Otherwise it is passed on to the next farm node. When a task has been processed, its result is passed to the results router which returns it to the controller. The controller then sends a new task to the network to replace the one which was completed. The system is initialized by sending sufficient tasks to the network to fill the capacity of the farm nodes and their buffers.

This original model presupposes that the controller sets up a configuration once and lets the farm nodes do their job. While it is easy to describe this model in **MANIFOLD**, using **MANIFOLD** allows us to enhance a computing farm by making it dynamic. In **MANIFOLD**, we can allow a farm node to die and leave the farm. We can also let the controller dynamically add a new farm node. Clearly, the addition of these new features increases the usability of the model for environments where the nodes are not necessarily reliable.

In this section, the **MANIFOLD** program for the computing farm is explained in more details. The overall structure of the program is as follows.

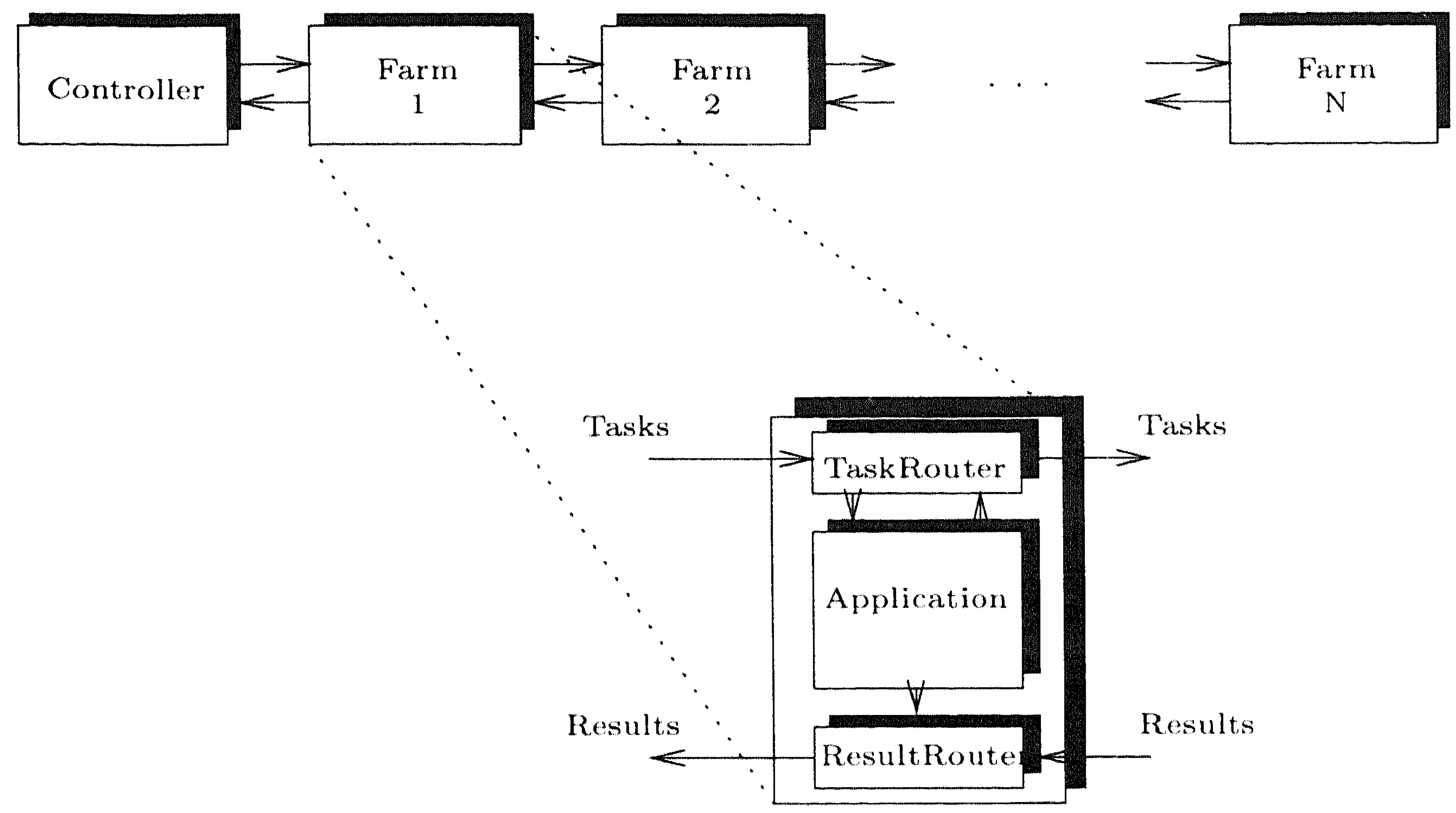

FIGURE 4. The Computing Farm Model

- Each application unit is an instance of an imported process.
- The application units are managed by a wrapper (called `Farm`), which is a manifold.
- A separate manifold process, called `Link`, controls the streams connecting two adjacent `Farms`.
- The controller process (which is a manifold) is responsible for setting up the starting configuration and for adding a new `Farm` node if requested. This process interfaces the whole processor farm to the external world.

By "controlling" the streams between adjacent farm nodes, we mean that the streams between $\texttt{Farm}_i$ and $\texttt{Farm}_{i+1}$ are set up and broken up by the manifold $\texttt{Link}_i$. For convenience, we refer to $\texttt{Farm}_i$ as the "left" node of the linker manifold $\texttt{Link}_i$. Likewise, the "right" node of $\texttt{Link}_i$ is $\texttt{Farm}_{i+1}$. We also say that $\texttt{Link}_i$ is the "right linker" of the node $\texttt{Farm}_i$.

An overview of our **MANIFOLD** program is shown in Figure 5. Note that there are some major differences between this version and the model presented in Figure 4:

- The wrapper is one process only (instead of two in the original model). The reason is that a manifold is able to set up and manage parallel information flows within one process).
- The farm nodes are linked into a ring. This is necessary to allow dynamic reconfiguration.

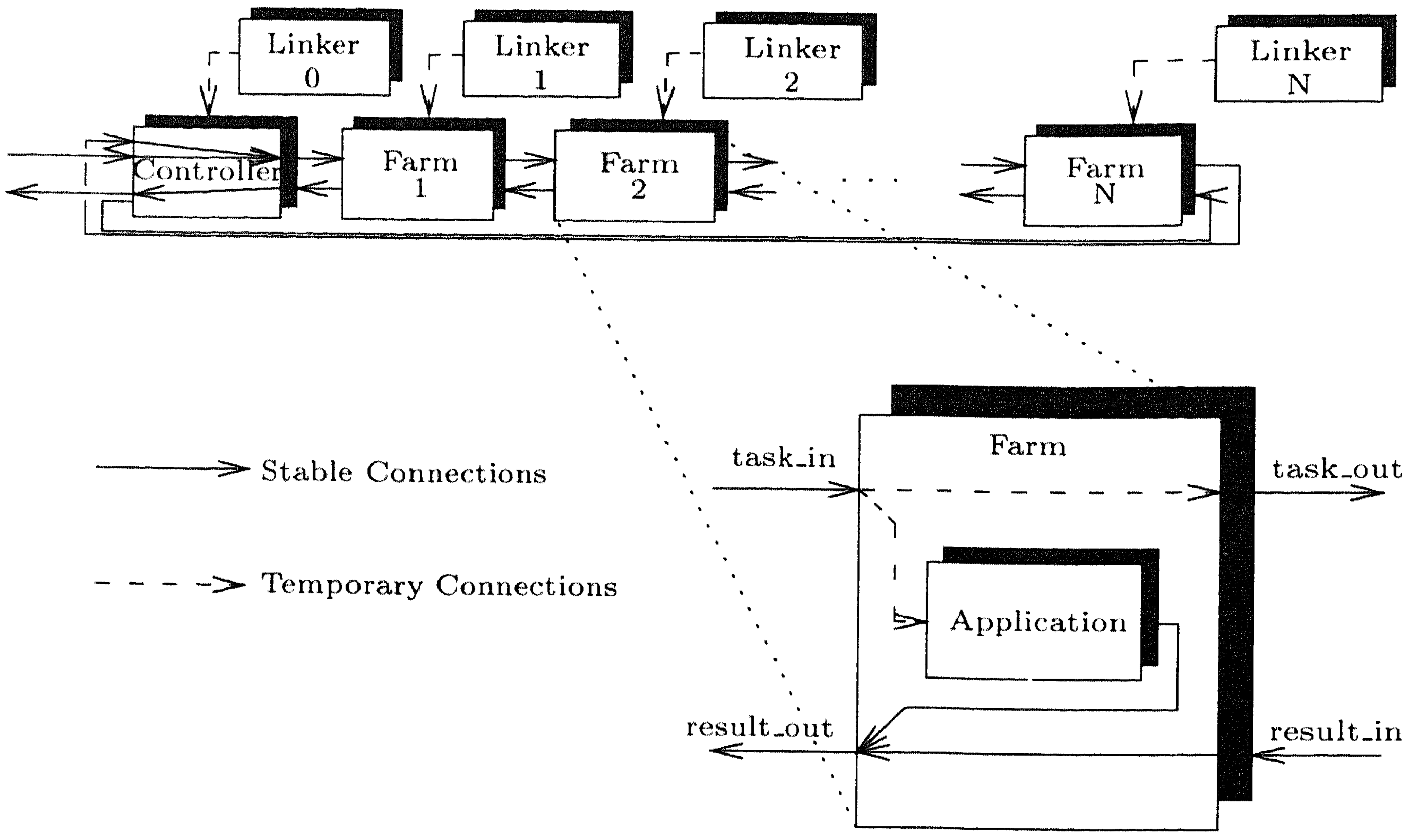

FIGURE 5. Computing Farm in **MANIFOLD**

- The connections between the adjacent nodes are "active" in the sense that they are controlled by yet another process. This is also necessary to allow dynamic reconfiguration.

The limits imposed by the size of this paper do not allow us to comment and/or to present the full **MANIFOLD** program. Instead, only a simplified skeleton is presented. In particular, we have omitted the use of an internal cache in the farm nodes. The state of the application is simply recorded in the farm node; if a new task description arrives, the state controls whether this description should be forwarded to the next node or to the application program. Also, proper handling of some race conditions on events requires more delicate considerations in the programs. We have omitted these. Finally, only a few of the manifolds involved are described in detail. The interested reader may find the annotated version of a complete implementation (including, e.g., internal caching to avoid latency) as well as the full listing of the program in a separate report [7].

The listings below use two additional pseudo-processes[1] which are both provided by the **MANIFOLD** language. The built-in pseudo-process `getunit` suspends on a port of its caller, as long as there is no unit available for delivery on the port. When a unit is or becomes available, this unit is sent out onto the output port of `getunit` and the pseudo-process terminates (i.e., the pipelines in which it is involved are broken). The pseudo-process `guard` behaves as a process *installed* on a port of its caller manifold. It raises an event (its argument) inside

---

[1] By *pseudo-process* we mean one of the primitive actions of **MANIFOLD** that behave like a real process in a pipeline, although they are not truly separate processes.

```
event   wait.
nobreak
    (result_in->,Appl.result_out->)
       ->result_out.
begin:
    (guard(task_in,input_arrived),do wait).
wait:
    void.
input_arrived.task_in:
    if( appl_requires,
        (getunit(task_in) -> Appl.task_in,
         false -> appl_requires
        ),
        getunit(task_in) -> task_out
    );
    do begin.
asks_for_work.Appl:
    true -> appl_requires; do wait.
```

LISTING 2. Central Part of the Farm Node

its caller manifold if a unit arrives in this port.

3.1.2. The farm nodes

The heart of a farm node is presented in Listing 2. This is indeed the portion of the code which controls the correct transmission of task descriptions either to the next farm node, or to the local application.

The assumptions on the behavior of the application are very simple. It is supposed to raise the event `asks_for_work` when it requires a new task description to work on and, afterwards, it must read a task description from its `task_in` port (to be absolutely precise, it must be suspended on this port until the task description arrives). It is also supposed to produce its result unit on its output port `result_out`.

A farm node sets up a permanent set of pipelines to transmit its results back to the controller, using the declaration:

```
nobreak (result_in->,Appl.result_out->)->result_out.
```

This installs a permanent stream from the node's own `result_in` to `result_out` and, also, a stream to forward the results from the application to `result_out`. In other words, this latter port merges the two streams to the outside world. The (boolean) variable `appl_requires` notifies the manifold that the application has requested a new task description.

The farm node, within this portion of the code, reacts on two events: either on the arrival of a new task description on its `task_in` port or on the event notifying that the application is ready to work on a new task. The former event is raised by a `guard` installed by the farm node, and the latter is raised by the application.

The task descriptions arrive on the port `task_in`, and the farm node either:

- reads the unit and forwards it to the application, if the application is pending, or
- transfers the unit directly to the next farm node.

In both cases, the guard must be re-installed.

If the application requires a new task description, this fact is stored in the internal boolean variable.

3.1.3. The linker process

The "normal" behavior of the manifold `Link` (see Listing 4) is simply to set up the necessary streams between its left and right farm nodes. This is done by:

```
(left.task_out->right.task_in,right.result_out->left.result_in)
```

The manifold `Link` remains in this block as long as it is not preempted by events coming either from its left or its right farm nodes. The farm node events are requests for their deactivation.

3.1.4. Start-up (the controller)

The `Controller` manifold is responsible for starting up the whole computing farm. Its behavior is relatively simple: it activates all `Farm` and `Link` processes in a cycle. This includes a `Link` process to link the `Controller` itself to the first farm node in the list.

Once all nodes are set up, the only real action the controller process performs is the addition of a new node to the ring. This is done on an external request, by the reception of an (external) event `start_new_one`.

Generation of a new node involves activation of a new farm node and adding it to the ring. The simplest way to do this is to add the new node at the head of the ring. For this purpose, the following steps must be taken:

- activate a new farm node;
- activate a new linker to link the controller to this new farm node; and
- re-initialize the linker process which used to link the controller to the (formerly) first node of the ring.

Due to space limitations, details of these steps are not presented here; the reader may consult [7] for the details.

```
event  wish_to_die.
death_start:
    (raise wish_to_die,linker).
empty_buffers.linker:
    Empty_Buffer(result_in,result_out);
    Empty_buffer(task_in,task_out);
    (raise buffers_emptied,void).
```

Listing 3. Deactivation of a Farm Node

3.1.5. Deactivation

Deactivation of a farm node is the most complex operation in the computing farm. We assume here that deactivation is the result of the death of an application instance. When this happens, the wrapper farm node also deactivates as a process and is deleted from the farm.

When a farm node intends to deactivate, the node must be deleted from the ring and new streams must be set up between the two adjacent farm nodes. Because the farm nodes and the linker processes are activated in a cycle by the controller manifold, this latter has no permanent record of the references to the farm nodes and the linker nodes. Consequently, deactivation of a farm node must be done locally and involves its left and right linker. In other words, there is no "central authority" to take care of deactivation[2].

If an application i is deactivated, it automatically raises (as all processes in a **MANIFOLD** system do) the **death** event. This event is caught by the wrapper manifold **Farm**$_i$ and preempts its normal operation by causing a transition to a special portion of the code (see Listing 3). In this block, a special event is raised (**wish_to_die**) and then **Farm**$_i$ waits until its right linker gives it the authorization to proceed.

The event **wish_to_die** will be caught by **Link**$_{i-1}$ and **Link**$_i$ (see Listing 4); indeed, these are the only two processes that have a process reference to **Farm**$_i$ (and are, consequently, sensitive to the events raised by this process). Because of their respective "position", one will see this event as coming from its "right" node (this is the case for **Link**$_{i-1}$) and the other one will see the event as coming from its "left". In both cases, the linker processes will preempt their normal state and change to a state specially defined for the death of one of the nodes. Note that this state transition disconnects **Farm**$_i$ from its adjacent farm nodes: the state transition in the respective linker nodes will break the corresponding streams. More details on the deactivation of farm nodes is explained in [7].

4. CONCLUSIONS

MANIFOLD uses the concepts of modern programming languages to describe and

---

[2] This "distributed" approach to deactivation is, as a matter of fact, much more in line with the computing model advocated by **MANIFOLD**.

```
start:
    (left.task_out    -> right.task_in,
    right.result_out -> left.result_in).
wish_to_die.right:
    (right.result_out -> left.result_in,
     void).
wish_to_die.left:
   {
        begin:
           (raise empty_buffers,
            left.task_out -> right.task_in).
        buffers_emptied.left:
           (deactivate previous_link,
            deactivate left);
           <re_init>
   }.
```

Listing 4. Details of a Linker Node

manage connections among a set of independent processes. The unique blend of event driven and data driven styles of programming, together with the dynamic connection graph of streams seem to provide a promising paradigm for parallel programming. The emphasis of **Manifold** is on orchestration of the interactions among a set of autonomous agents, each providing a well-defined segregated piece of functionality, into an integrated parallel system for accomplishing a larger task. The declarative nature of the **Manifold** language and the **Manifold** model's separation of communication and coordination from functionality, both significantly contribute to simplify programming of large, complex parallel systems.

In the **Manifold** model, each process is responsible to *protect* itself from its environment, if necessary. This shift of responsibility from the producer side to the consumer of information seems to be a crucial necessity in open systems, and contributes to reusability of modules in general. This model imposes only a "loose" connection between an individual process and its environment: the producer of a piece of information is not concerned with who its consumer is. In contrast to systems wherein most, if not all, information exchange takes place through targeted send operations within the producer processes, processes in **Manifold** are not "hard-wired" to other processes in their environment. The lack of such strong assumptions about their operating environment makes **Manifold** processes more reusable.

References

1. N. Carriero and D. Gelenter (1989). LINDA in context. *Communications of the ACM*, vol. 32, pp. 444–458.

2. W. Leler (1990). LINDA meets UNIX. *IEEE Computer*, vol. 23, pp. 43–54, February.
3. F. Arbab (1992). Specification of Manifold. *Tech. Rep. to appear*, CWI, Amsterdam.
4. F. Arbab and I. Herman (1991). Manifold: A language for specification of inter-process communication. *Proceedings of the EurOpen Autumn Conference* (A. Finlay, ed.), (Budapest), pp. 127–144.
5. F. Arbab, I. Herman and P. Spilling (1992). Interaction management of a window manager in Manifold. *Computing and Information ICCI'92* (W. Koczkodaj, P. Lauer and A. Toptsis, eds.), (Toronto), IEEE Press.
6. F. Arbab, I. Herman and P. Spilling. An overview of Manifold and its implementation. *Concurrency: Practice and Experience*, to appear.
7. I. Herman and F. Arbab (1992). More examples in examples in Manifold. *Tech. Rep. CS-R9214*, CWI, Amsterdam.
8. D. Soede, F. Arbab, I. Herman and P. ten Hagen (1991). The GKS input model in Manifold. *Computer Graphics Forum*, vol. 10, pp. 209–224, September.
9. O. Dahl, B. Myhrhang and K. Nygaard (1970). Simula 67 common base language. *Tech. Rep. S-22*, Norwegian Computing Center, Oslo.
10. C. Hoare (1978). Communicating sequential processes. *Communications of the ACM*, vol. 21.
11. C. Hoare (1985). *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science, New Jersey: Prentice-Hall.
12. E. Rutten, F. Arbab and I. Herman (1992). Formal specification of Manifold: a preliminary study. *Tech. Rep. CS-R9215*, CWI, Amsterdam.
13. I. Herman, F. Arbab and F. Burger (1992). The Manifold stack machine. *Tech. Rep. to appear*, CWI, Amsterdam.
14. S. Green (1991). *Parallel Processing for Computer Graphics.* Research Monographs in Parallel and Distributed Computing, London: Pitman.
15. ISO 7942 (1985). *The Graphical Kernel System* (GKS), International standard on computer graphics, ISO.