MANIFOLD version 1.0 programming: programs and problems

C.L. Blom

Computer Science/Department of Interactive Systems

Report CS-R9334 June 1993

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications. SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

P.O. Box 4079, 1009 AB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

MANIFOLD Version 1.0 Programming: Programs and Problems

Kees Blom

CWI, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract

MANIFOLD is a new programming language designed to control multiple concurrent activities by managing the communications among a number active communicating entities. The present initial version of the MANIFOLD language is a minimal language attempting solely to implement a conceptual model of asynchronously communicating processes, called the MANIFOLD model. Such a model can be useful in an implementation environment where processes and inter-process communication are fast and amply available (e.g. massive parallelism).

This document contains a tutorial introduction to the first experimental Version of the MANIFOLD language (using listings of simple working example programs), followed by four larger working programs. The detailed operation of each of these latter four programs is described with the help of illustrations, and a number of language features and problems are explained. We conclude, that the MANIFOLD conceptual model and language are both implementable and actually work, but also that in their initial first experimental versions, they have some imperfections which must be taken into account in the design and implementation of the next version of the MANIFOLD system.

AMS Subject Classification (1991): 68N99, 68Q10
Keywords & Phrases: parallel computing, coordination languages, parallel programming languages

1 Introduction.

MANIFOLD is a new programming language designed to control multiple concurrent activities by managing the communications among a number active communicating entities.

The communications between these entities imply transfer of information from one entity to another. The actual communication links may form arbitrary complex networks and change dynamically in a controlled fashion.

The basic entities are called *processes* which in fact may stand for anything that can be regarded as *active*, such as a computer running, a machine working, a person playing. In practice, these processes are cooperating programs running on various computers; MANIFOLD is instrumental in coordinating these processes.

The present initial version of the MANIFOLD language is a minimal language attempting solely to implement a conceptual model of communicating processes, called the MANIFOLD model, which is explained in detail in the language specification document[1], and from which much of the material in the present document is cited. Such a model can be useful in an implementation environment where processes and inter-process communication are fast and amply available (e.g. massive parallelism).

The present implementation of MANIFOLD consists of a compiler and a run-time system. The latter is written in Concurrent C++ [4] and is described in [3].

This document started as a Programmers Guide for the first version of the MANIFOLD language; however in gaining experience with the first experimental version of this language some lacunae and deficiencies were detected in both the conceptual model, the language and the implementation. Therefore the character of the document has changed into an account of experiments carried out with MANIFOLD Version 1.0, preceded by an informal introduction to the language.

In Section 2 we introduce the MANIFOLD language informally using simple working examples.

Report CS-R9334
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

In Section 3 we present some larger working programs and discuss some features and problems.

In the concluding Section, we discuss some of the lessons learned and make some recommendations for a next version of MANIFOLD.

2 An informal tour through the MANIFOLD programming language.

2.1 Basic concepts and characteristics.

The basic components of the MANIFOLD programming language are *processes*, *streams*, *events* and *ports*. The latter are associated with processes and define communication points, which processes may use to exchange information with each other trough streams.

There are two kinds of processes, *manifold processes* and *atomic processes*.

A *manifold process* is the execution of a set of actions specified in the MANIFOLD programming language.

An *atomic process* is any active entity which is *not* specified in the MANIFOLD programming language (e.g. a FORTRAN or C program) but for which a well-defined interface with the MANIFOLD programming language exists.

According the MANIFOLD model, a manifold process is always in one of several predefined *states*, and may change from one state to another whenever the process detects the occurrences of certain *events*. Each state has a label, defining the event(s) which trigger the execution of the actions contained in that state. These actions include activation and deactivation of processes, and the creation of connections between *output ports* and *input ports* of these processes. *Ports* are passages at the bounds of MANIFOLD processes, which control the flow of *units* of information.

This is probably best clarified by a simple familiar example.

```
/*
 * hello1.m - first Manifold program
 */
main
{
start:    "Hello, programmer !\n" -> sysoutput.
}
```

Figure 1: hello1.m: a simple example.

In Figure 1 `sysoutput` is a builtin name (for which a full specification exists outside the current source file) which happens to be an atomic process that has been built into the MANIFOLD run-time system. Its purpose is to wait for a connection to be established on its port named `input`, then forward everything it receives onto the standard output channel of the operating system where the MANIFOLD run-time system is installed (e.g. a terminal or a window) until all connections on its input port have been broken, whereupon it halts.

Next the predefined name `main` is declared followed by a list of state specifications (called *blocks*) between { and }. There is only 1 block, labeled `start` which contains only one action, namely " $\rightarrow$ ".

This action creates connections, in this case a connection between the string "`Hello programmer !\n`" and an instance of the aforementioned atomic process `sysoutput`, and waits for the connection(s) to be broken.

The command:

```
Manifold> manif -o hello1 hello1.m
```

compiles the MANIFOLD program contained in the file `hello1.m` and produces an executable file `hello1`. Next, the command:

```
Manifold> hello1
```

initializes the MANIFOLD runtime system, after which this system activates the manifold main, which must always be defined by the MANIFOLD programmer. As part of the initialization protocol, the event start is raised by the runtime system in main, consequently the block labeled with start becomes active. Entering this state results in the implicit activation of an instance of the atomic process sysoutput and the creation of a connection between the string constant (which is a process by itself) and the port input of the atomic process sysoutput, whereupon the main process waits until this connection is broken. As soon as the connection has been established one unit (the string) will be sent over this connection, whereupon the process representing the string halts, by which the connection will be broken. The sysoutput process

```
/*
 * hello2.m - some primitive actions in Manifold
 */
main
{
    process sysoutput_proc is sysoutput.

start.self:
    activate sysoutput_proc;                          // activate
    "Hello, programmer !\n"-> sysoutput_proc.input; // connect
    sysoutput_proc.                                   // await death of
                                                      // 'sysoutput_proc'
death.sysoutput_proc:
    halt.                                             // terminate
}
```

Figure 2: hello2.m: actions made explicit.

instance takes the unit from its input port and send it to the outside world as specified above. Detecting that all connections to its input port have been broken, sysoutput consequently halts.

In the meantime, the process main also halts, since the connection it had established has been broken, and no subsequent actions are specified.

The MANIFOLD runtime system, discovering that all manifold processes and atomic processes have halted, finally, also halts.

There are a couple of notable points in the description of the workings of this program.

Firstly, in MANIFOLD transfer of control is managed by raising events. In main, the block is entered since it has been labeled with the event start and this event has been raised by the runtime system.

Secondly, what a manifold process ultimately does is: activating and deactivating atomic processes, and creating and breaking up connections between the ports of these atomic processes. Therefore MANIFOLD can be regarded more like a *coordination language*, rather than a conventional *computational* programming language. This implies, that for many rather simple programming tasks other languages may be more appropriate than MANIFOLD, whereas MANIFOLD comes into play when a number of heterogeneous active entities must be put together to work on a common problem. This will be illustrated in the sequel.

For now, in order to become familiar with MANIFOLD concepts and language, initially a number of simple examples will be treated in detail to show the properties of MANIFOLD as clearly as possible.

2.2 Processes.

To illustrate how in MANIFOLD processes can be created, we rewrite the program in Figure 1 using more primitive constructs of the MANIFOLD language as shown in Figure 2, making every single step in the description above explicit. In main, there are now two blocks, of which the second will be entered as soon as the predefined event death has been raised by sysoutput_proc. The death event is actually raised by the MANIFOLD runtime system as part of the termination protocol when a process, such as

```
/*
 * hello3.m - the 'group' construct in Manifold
 */
main
{
    process sysoutput_proc is sysoutput.

start.self:
    (activate sysoutput_proc,                          // activate
     "Hello, programmer !\n" -> sysoutput_proc.input, // connect
      sysoutput_proc                                   // await death of
    ).                                                 // 'sysoutput_proc'

death.sysoutput_proc:
    halt.                                              // terminate
}
```

Figure 3: hello3.m: actions in a group.

sysoutput_proc, halts. The action halt in this block halts the process main. In the first block, there are now three distinguished actions, separated by ";" symbols. The ";" is an operator meaning sequential execution of its operands. Thus the connection is guaranteed to be made after the process activation has been completed. However, this is not necessary.

In Figure 3, all actions in the first block are *group*ed together between "(" and ")" and separated from each other by comma's. This implies that all of these actions will be executed in some non-deterministic order, possibly in parallel.

This construct, grouping together a number of actions between "(" and ")" separated by commas, is called the *group* construct and is a fundamental construct in MANIFOLD.

2.3 Events.

```
/*
 * event1.m - internal events causing state transitions.
 */
main
{
event state1, state2.

start.self:     do state1.
state1.self:    "This is state1.\n" -> sysoutput; do state2.
state2.self:    "This is state2.\n" -> sysoutput; halt.
}
```

Figure 4: event1.m: internal events.

As mentioned before, a MANIFOLD program consists of a number of labeled states and the transitions between the various states are controlled by *events*. The label of each state consists of a list of events (and optional sources) upon detection of which the state will be executed. Event occurrences are generated by executing one of two primitive actions, do and raise, and by the MANIFOLD run-time system when some predefined condition is met (e.g. start).

The do action can be used to switch from one state to another *inside* a manifold; as illustrated in Figure 4, where main goes trough several separate states before halting; do has no effect outside its

executing manifold.

Note that all user-defined events must be declared.

Switching to another state is done as follows:

- an event is picked non-deterministically from the incoming event list
- a circular search trough all available states is performed, starting at the current state downwards
- when a match is found with a block label then the state corresponding with the block is entered
- otherwise the next event is taken from the incoming event list until that list is empty, whereupon a manifold enters the `halt` state and halts (unless the current state contains an active connection).

```
/*
 * event2.m - external events causing state transitions.
 */
event event1, event2, wait.

main
{
process slave   is Slave.

start:  activate slave; do wait.

event1.slave:
        "main has received event1 from slave.\n"->sysoutput;
        raise event2; do wait.

wait:   slave.
}

Slave
{
start.self:
        raise event1; do wait.
event1: "slave has received event1.\n"->sysoutput;
        raise event2; do wait.
event2: "slave has received event2.\n"->sysoutput;
        raise event1; do wait.

wait:   parent.
}
```

Figure 5: event2.m: external events.

The action `raise` can be used to inform other processes *outside* a manifold of certain conditions; it has no effect inside its executing manifold. This is illustrated in Figure 5, where `main` activates another process `slave`, declared as an instance of the manifold `Slave`. The latter starts by raising `event1`, then goes into a `wait` state; meanwhile `main` picks up this event from process `slave`, raises `event2` and goes also into a `wait` state. At the same time the process `slave` finds `event2` to be raised, for which it has a label in a handler block, so it leaves its wait state, and executes the corresponding block, outputs a message and raises the `event1` again, etc. etc. in an eternal loop.

Note that the second handler block of `Slave` which is labeled `event1` is never entered although it is raised by `Slave` but not by anyone else and `raise` has no effect inside that manifold.

Note further, that each manifold in this example has an additional handler block labeled `wait` (which is just a user-defined name) to which a jump is made after handling the current block in order be able to receive the next occurrence of the event which was mentioned in the label of the current block. This is done, since once the manifold enters a block, it is immune to any of the events handled by that block.

Note finally, that each `wait` block contains a list of so-called *preemptable event sources*, which may "kick you out" 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 and the *permanent* event sources (see § 3.2.3). Observable events are, loosely speaking, events for which the manifold has a handler block.

2.4 Ports and Streams.

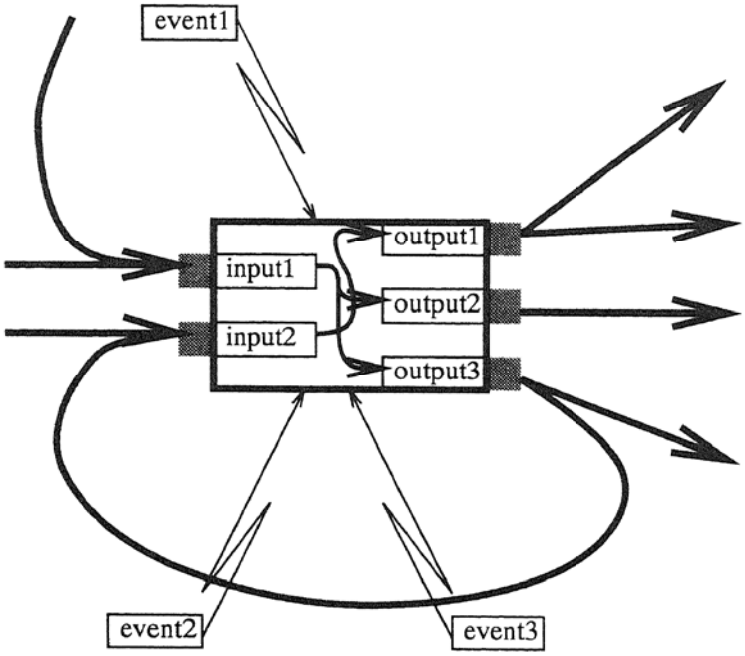


Figure 6: Schematic representation of a MANIFOLD.

Recall, that the basic components in the MANIFOLD programming model are *processes*, *events*, *ports* and *streams*. A *process* is a *black box* with a number of well defined *ports* on which *connections* can be made with the ports of other processes so that these processes can communicate with each other by exchanging *units* of information. These connections are called *streams*.

The management of these streams, how and when they are created and destroyed, is controlled by raising and receiving *events*, which is a *control* mechanism between processes independent of the existence of any stream, as illustrated in Figure 6.

Manifolds may setup streams between their own ports, between their ports and the ports of other manifolds, or between the ports of other manifolds, possibly in parallel, while at the same time other manifolds can make connections with their own ports !

In MANIFOLD ports are one-way passages and there are consequently two types of ports: `in` and `out` for receiving and sending information respectively. Each process has three predefined ports, named *input* (type: `in`), *output* and *error* (both type: `out`).

In Figure 7, besides `main` there are two other manifolds defined: `manifold_A` and `manifold_B`. They each have a `port in` and a `port out` declaration in their *public* declaration section (after the name of the manifold, but before the opening bracket). As it stands, the port declaration in `manifold_B` could have been omitted, since these are provided by default.

```
/*
 * port1.m - port declaration and stream connections
 */
main
{
process a       is manifold_A.
process b       is manifold_B.

start:  (activate a, activate b,
        "Unit\n"->a.i, a.o->b.input, b.output->a.i,
                                    b.output->sysoutput).
}

manifold_A
port in  i.
port out o.
{
start:  i->o.
}

manifold_B
port in  input.
port out output.
{
start:  input->output.
}
```

Figure 7: port1.m: port declaration and stream specification .

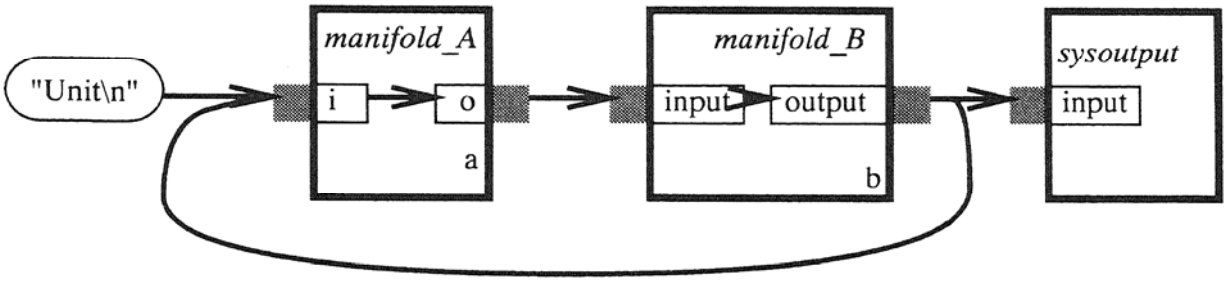


Figure 8: Stream connections between ports of processes in MANIFOLD.

At start-up time, both manifolds setup a *stream* between their ports for incoming and outgoing units using the *pipe* operator (right-arrow '→'), as illustrated in Figure 8. The left-hand side of a '→' is called the *source*, the right-hand side the *sink* of a pipe. Effectively this means, that both manifolds copy all units trough their declared `port in` to their declared `port out`.

These manifolds are used in `main`, where instances of both manifolds are declared (in the *private* declaration of `main`). These instances are the *processes* `a` and `b`, which are both `activated` after receiving the `start` event, while at the same time (i.e. atomically, within a group) connections are made between the *ports out* of `a` and `b` *and* the *ports in* of `b` and `a` respectively. Furthermore, a unit (the string "Unit\n") is sent to the incoming port `i` of `a`. Inside `a` units are moved from `i` to `o`. In `main` there is a stream from port `o` of `a` (`a.o`) to port `input` of `b` (`b.input`). Inside `b` units are moved from `input` to `output`. Finally in `main` there is a stream from `b.output` to `a.i`, so that all units are moved around. This is verified by creating a stream from `b.output` to `sysoutput` (implicit process activation) so that each unit passing through port `output` of `b` is also copied into the stream to `sysoutput` (multiple connections on an outgoing port).

Consequently this program prints the initial string "Unit\n" repeatedly on the screen in an endless loop.

Note that *inside* a manifold its *incoming* ports (e.g. `i` in `manifold_A`) appear as a *source* on the *left*-hand side of a '→' while the "other side of" these same ports, when used by other manifolds, appear as a *sink* on the *right*-hand side of a '→' (e.g. `a.i` in `main`).

Note, further, that manifolds may be used before defined.

```
/*
 * port2.m - short-hand notation for default ports; pipeline.
 */
main
{
process a       is manifold_A.
process b       is manifold_B.

start:  (activate a, activate b,
        "Unit\n"->a.i, a.o->b->a.i,
                        b->sysoutput).
}

manifold_A
port in  i.
port out o.
{
start:  i->o.
}

manifold_B
{
start:  ->.
}
```

Figure 9: port2.m: short-hand notations and pipeline construct.

Figure 9 is effectively the same program using system supplied defaults, omitting 'input' and 'output' where possible.

Note the use of the *pipeline* construct `'a.o->b->a.i'` in `main`, which is almost the same as the group construct `'(a.o->b.input, b.output->a.i)'`. The difference between the two is, that should one of the connections (streams) in the group construct break, the other remains intact, whereas when any connection in the pipeline construct breaks, the whole pipeline is dismantled and all other connections

are also broken up.

Finally it should be noted, that in MANIFOLD these constructs may be recursively combined to form arbitrary complex networks of communicating processes, which may (partly) change at any time one of the managing manifolds reacts on an incoming event.

```
/*
 * port3.m - switching connections
 */
event passed.

main
{
process a       is manifold_A.
process b       is manifold_B.

start:  (activate a, activate b, do passed).
passed.self,
passed.a: ("Unit1 "->a, a->b->sysoutput).
passed.b: ("Unit2 "->b, b->a->sysoutput).
}

manifold_A
{       event state1, state2.
start:  do state1.
state1: getunit(input)->;
        "passed a "->;    do state2.
state2: getunit(input)->; getunit(input)->;
        "passed a.\n"->;  raise passed; do state1.
}

manifold_B
{       event state1, state2.
start:  do state2.
state1: getunit(input)->;
        "passed b "->;    do state2.
state2: getunit(input)->; getunit(input)->;
        "passed b.\n"->;  raise passed; do state1.
}
```

Figure 10: port3.m: event driven pipeline switching.

This is illustrated in Figure 10, a somewhat more complex example and yet not a perfect one! The idea is to illustrate the passing of units trough pipelines by printing them, and the switching between pipelines by reacting on observed events.

In this example, `manifold_A` is always in one of its two states, `state1` (initially) or `state2`.

In `state1` a single unit is moved from port `input` to port `output` using the `getunit` primitive action. Thereafter, the string "passed a" is moved to `output`. Then `state2` is entered.

In `state2` two single units are moved from port `input` to port `output`, the string "passed a.\n" is moved to `output`, the event `passed` is raised and `state1` is reentered.

The definition of `manifold_B` is identical, except that it is initially in `state2`.

These manifolds are used in `main` as processes `a` and `b`. In the next state after `start` when the processes are activated, one unit, the string "Unit1" is copied to `a` and a pipeline from `a` via `b` to `sysoutput` is constructed. Consequently `a` in `state1` consumes one unit and produces two. These units are taken by `b` (initially being in `state2`) and moved to its `output` whereupon `b` also adds a unit (the string "passed

b.\n") and raises the global event passed. In the meantime the pipeline constructed in main moves the all units from the output port of b to the i port of a with a copy of each to sysoutput.

Both a and b switch their states, while in parallel (but not necessarily simultaneously) main receives the event passed from b, breaking up the pipeline and switching to another state in which a different pipeline is constructed. The execution of this MANIFOLD program results in the output:

```
Unit1 passed a passed b.
Unit2 passed b passed a.
Unit1 passed a passed b.
Unit2 passed b passed a.
Unit1 passed a passed b.
Unit2 passed b passed a.
Unit1 passed a passed b.
...
```

If you try this example you might notice that some lines get interspersed due to the fact that the multiple implicit activations of sysoutput result in multiple processes concurrently outputting on the same screen, without proper synchronization. This will be dealt with in § 3.2.

Note further, that in this example manifold_A and manifold_B do logically almost the same thing. It is thus desirable to combine them into one parameterized manifold declaration. This is done in 2.5.

2.5 Parameters.

```
/* port4.m -  parameterized switching connections */
event passed.
main
{
process a,b     is pass_units.
start:  (activate a("a"), activate b("b"), do passed).
passed.self,
passed.a: ("Unit1 "->a, a->b->sysoutput).
passed.b: ("Unit2 "->b, b->a->sysoutput).
}
pass_units(id_string)
        port in id_string.
{       event state1, state2.
start:  if (id_string == "a", do state1, do state2).
state1: getunit(input)->;
        "passed "->; id_string ->; ", "->;  do state2.
state2: getunit(input)->; getunit(input)->;
        getunit(input)->; getunit(input)->;
        "passed "->; id_string->; ".\n"->;
        raise passed; do state1.
}
```

Figure 11: port4.m: a parameterized manifold.

In Figure 11 the previous example has been rewritten so that the two manifolds are replaced by a single slightly more complex one with one parameter.

The types of formal parameters are declared in the public declaration section of the manifold definition and can be process, port in, port out, or event.

Process parameters can be (de)activated, appear in pipelines, appear as "source" in the label of a handler block and can be passed on as actual parameter for process or port formal parameters in activating other manifolds.

Port parameters can appear in pipelines (as source/sink for `port in`/`port out`, respectively) and can be used as port parameters in activating other manifolds.

Event parameters can appear in the primitive actions `do` and `raise` and as event parameters in activating other manifolds. They **cannot** be used in the label of a handler block.

Manifold names can be overloaded, thus the same manifold name can be used in multiple manifold definitions, provided that their formal parameter type lists differ.

Actual parameters should match their corresponding formal parameters, except that processes are allowed for port in/out formal parameters, where the process name is replaced by process.output or process.input respectively.

2.6 Manners.

```
/* port5.m - switching connections with a manner. */
event passed.
main
{       process a,b     is pass_units.

start:  (activate a("a"), activate b("b"), do passed).
passed.self,
passed.a: ("Unit1 "->a, a->b->sysoutput).
passed.b: ("Unit2 "->b, b->a->sysoutput).
}
// the keyword 'export' makes a module accesible from
// other files
        export
pass_units(id_string)
        port in id_string.
{       event state1, state2.
start:  if (id_string == "a", do state1, do state2).
state1: getunit(input)->;
        "passed "->; id_string ->; ", "->;  do state2.
state2: getunits (input, output, 4);
        "passed "->; id_string->; ".\n"->;
        raise passed; do state1.
}
manner getunits (source, sink, number)
        port in source. port out sink.  process number.
{       event loop, next, exit.
        process n is variable.
start:  activate n; n = number; do loop.
loop:   if (n <= 0, do exit);
        getunit(source)->sink; n=n-1; do next.
next:   do loop.
exit:   deactivate n; return.
}
```

Figure 12: port5.m: manners (subroutines) with parameters.

In the previous example the construct "if" was used, which in MANIFOLD is not a language primitive, but rather a built-in *manner*, which is a subroutine-like language construct in MANIFOLD. An example is in Figure 12, where the repeated use of `getunit` in `state2` has been replaced by one manner call to `getunits`, designed to move a parameterized number of units from one port to another.

Manners look very much like manifolds. They may have parameters, private processes and events, and labeled blocks with the same expressions and primitive actions as manifolds.

They do not have their own ports (other than parameters), but may have free access to their calling manifold's ports. Also there is no process associated with manners, each process may call any manner using the same syntax as for implicit activation. They cannot appear in pipeline or group constructs, however, since they do not have ports.

Manners have two additional formal parameter types: `manner` and `group`. The actual parameter must be: a possibly nested manner call or a possibly nested expression containing pipelines, groups, and primitive actions.

Manners differ from manifolds primarily by their behaviour: when in a manner, any label in that manner overrides the same label in the calling environment. Thus they enable the calling manifold to change the way (or the manner) in which it reacts to events from the outside world, hence the name.

```
/* port6.m - delayed switching connections. */
extern event passed.     // 'extern events' can be shared across files.

pass_units (id_string)  // to import a module defined in another file,
    port in id_string.  // its declaration must be repeated and the
import.                 // keyword 'import' replaces the body
                        // of the manifold.
main
{       process a,b     is pass_units.
start:  (activate a("a"), activate b("b"), do passed).
passed.self,
passed.a: delay (10); ("Unit1 "->a, a->b->sysoutput).
passed.b: delay (10); ("Unit2 "->b, b->a->sysoutput).
}
manner delay (number) port in number.
{       event   loop,next, exit.
        process n       is variable.
start:  activate n; n = number; do loop.
loop:   n = n - 1; if (n == 0, do exit); do next.
next:   do loop.
exit:   deactivate n.
}
```

Figure 13: port6.m: manner `delay`.

More often, however, they are used just like a subroutine: to isolate parts of code that group naturally together and/or occur often, so that a decomposition of code into more manageable smaller modules can be achieved.

When the Manifold processor enters a manner, the set of observable events increases by the events in the handler block of the manner and decreases when the manner is left.

When in a manner a preemptable event is caught for which the manner has no handler, but its calling environment (e.g. the manifold) has one, the manner execution is abandoned and the corresponding handler block in the calling environment is invoked.

Preemptable events are events whose sources appear in the current block, some system-defined events, and all events whose sources have been declared permanent by the MANIFOLD programmer.

Preemption of manners will be treated in more detail in § 3.2.3.

Some final notes about the example. First, the built-in manner `if` is used twice with a different parameter set. This is an example of overloading and the two `if`'s are really implemented as different manners. In both cases the first parameter is an expression involving logical operators e.g."==" and "<=". This will be explained in more detail in the next section. The second parameter is "`do` *event*". This is

an example of a primitive action as an actual parameter for a group type formal parameter.

Finally, if you try these examples, your output should become increasingly interspersed beyond the point where it is meaningful. In Figure 13 a simple *delay* has been inserted before setting up new connections; the delay itself is programmed as a manner containing an adjustable idle loop. This has the effect, that the process setting up a new instance of sysoutput is delayed sufficiently to allow the previous instance to finish. This method is not generally usable since the loop count causing the delay may need to be adjusted for each run; a sound method to accomplish synchronization using semaphores will be discussed in § 3.2.

```
/* port7.m - separate compilation of modules */

// 'extern events' can be shared across files
extern event passed.

// 'export' manifolds/manners can be 'import'ed in other files
export pass_units (id_string)
        port in id_string.
{       event state1, state2.
start:  if (id_string == "a", do state1, do state2).
state1: getunit(input)->;
        "passed "->; id_string ->; ", "->;  do state2.
state2: getunits (input, output, 4);
        "passed "->; id_string->; ".\n"->;
        raise passed; do state1.
}
        manner
getunits (source, sink, number)
        port in source. port out sink.  process number.
{
        event loop, next, exit.
        process n is variable.

start:  activate n; n = number; do loop.
loop:   if (n <= 0, do exit);
        getunit(source)->sink; n = n - 1; do next.
next:   do loop.
exit:   deactivate n; return.
}
```

Figure 14: port7.m: export modules to allow separate compilation.

The program in Figure 13 is not complete, the parts that have not been changed have been stored in another file as in Figure 14; the modules that are to be used in other files are prepended with the keyword export and the events that are to be shared between files are prepended with the keyword extern. These files are typically suited to build object libraries from. The header of the modules which are defined in other files must be declared in all files where they are used, with the keyword import as their bodies. These declarations typically go into include files corresponding to object files in libraries. The commands:

```
Manifold> manif -M3 port6.m
Manifold> manif -o port7 port7.m port6..o
```

will compile the MANIFOLD program contained in the file port6.m and halt after the third stage of the compilation process, leaving the object code in port6..o; then the program modules in port7.m will be compiled and linked with that object file to produce an executable port7.

2.7 Atomic Processes and Operators.

In MANIFOLD the real computation is done by *atomic processes*. In all examples so far, we silently used some of them, which are already built-in in the MANIFOLD language and compiler (e.g. `sysoutput`, `variable` and operators like "==" (equal), "-" (minus)).

```
sysoutput atomic.
pragma atomic internal sysoutput.

main
{
start:      "Hello programmer.\n" -> sysoutput.
}
```

Figure 15: atom1.m: atomic process wrapper declaration.

The MANIFOLD programmer can also define and activate his own atomic processes.

Not using the built-in mechanisms, the "Hello" program in Figure 1 can be rewritten as in Figure 15, containing the declarations for the necessary atomic process *wrapper*, which consists of an `atomic` declaration similar to an `import` declaration, and a `pragma` specification. The latter is not part of the MANIFOLD language proper, but gives the MANIFOLD compiler instructions on how to generate the correct code.

```
#include "ap_interface.h"

void sysoutput(mf, parent)
     void *mf, *parent;
{
  int port, size;
  char unittype;
  apEvent   event;
  apSource *source;

  while (1) {
    switch (ap_await_anything (mf, &port, &event, &source)) {
    case EVENT:
      if (ap_events_equal (&event, mTerminate)
       || ap_events_equal (&event, mDisconnected_i))
    ap_terminate (mf);
      break;
    case UNIT:
      unittype = ap_unit_type (mf, port);
      if (unittype == STRING)
    printf ("%s", ap_peek_unit_data (mf, port));
      ap_delete_unit (mf, port);
      break;
    }
  }
}
```

Figure 16: atom1.c: an atomic process in C.

For the MANIFOLD compiler, the `atomic` declaration really defines a complete module. It is to be regarded as a *manifold* containing the MANIFOLD part of the atomic process interface.

If the same atomic process is to be used in another file, the same `import` declaration is needed in that file as for manifolds. In the file that generates the wrapper manifold, the `export` keyword must be

inserted before the atomic declaration.

The guest-language dependent interface consists of a function library and an associated include file: ap_interface.h. An example of how you can write your own sysoutput atomic process in the language "C" (capable of only printing strings) is given in Figure 16; more complete information is available in the reference manual[6].

With this interface a number of built-in atomic processes have been realized, these are listed in Table 1 and are used in our examples.

Further, some arithmetic (+,-,*, /, % and mod) and logical (<, >, ==, <=, >= and !=) operators are recognized by the MANIFOLD compiler and transformed into the implicit activation of appropriate atomic processes. All these processes take two operands and (if operands each deliver one unit with matching types) produce one unit containing the result on their output ports and then terminate when their output ports become empty.

2.8 Debugging.

A limited low-level debugging facility is available by specifying the "-d1" flag in the "manif" compile-and-link command. This results in logging the flow of control of the user program by the MANIFOLD run-time system on stderr (state transitions, manner calls, process switching and process termination). The debugging output log of the program in Figure 12 may look like:

```
1000007 : enter __main__block_169__
1000007 : leave __main__ block
1200008 : enter __pass_units_iblock_83__
1400009 : enter __pass_units_iblock_83__
1000007 : enter __main__block_170__
1200008 : continue __pass_units_i block
1400009 : continue __pass_units_i block
1200008 : leave __pass_units_i block
1400009 : leave __pass_units_i block
1200008 : enter __pass_units_iblock_84__
...
1400009 : enter __getunits_iopblock_128__
1400009 : continue __getunits_iop block
1400009 : jump out __getunits_iop block
1400009 : enter __getunits_iopblock_130__
...
1200008 : leave __getunits_iop block
1200008 : enter __getunits_iop _halt_block__
1200008 : leave __getunits_iop _halt_block__
```

The first column identifies each process, the next column a control action and an identifier referring to the current state; this identifier consists of a manifold/manner name and a block number; each labeled block in the original MANIFOLD program becomes a numbered block at run-time, in the same order. The control actions displayed during logging are:

- enter - control has been transferred to the displayed block
- leave - control will been transferred from the end of the displayed block to another block
- continue - a preemptable point ("';" and implicit activation) in the displayed block has been passed
- jump out - control will be transferred from the displayed block to another block by preemption, that is before the actions at the end of the current block have been executed

When logging is enabled, relative timing of processes may change, so the program behaviour may differ from the behaviour with logging disabled (compiled without "$-d1$" flag); consequently the interpretation of the logging output must be done with an appropriate pinch of salt.

pass	D	move all units from 'input' to 'output'
pass1	DT	move one unit from 'input' to 'output', then halt
count(limit,event)	D	as 'pass', but raises 'event' for each unit passed after 'limit' units have been passed silently
count1(limit,event)	DT	as 'count', except that it halts after raising 'event' once
CountThenPass(limit,event)	D	as 'count', except after raising 'event' it waits to be *reconnected* on its 'output' port (receives both 'disconnected_o' and 'connected_o'), then passes the remaining units silently
trigger(pattern,event)	DT	as pass, but it raises 'event' and halts as soon as a unit matches 'pattern'; that unit is not passed
PermConn(prod, cons)	P	set up the pipeline "*prod*− > *cons*", 'prod' and 'cons' are processes; this pipeline is permanent in the sense that it will not be broken until either this instance of 'PermConn', 'prod' or 'cons' halt.
PermConn(pi,po)	P	as 'PermConn' above, except that formal parameter types are: 'port in' and 'port out' for 'pi' and 'po' respectively
variable	P	whenever 'connected_i.input' is received, take one unit from 'input', store it and produce it on 'output' whenever 'connected_o.output' is received
read (filename,mode,size)	E	I/O interface processes.
write(filename)	D	'read' and 'sysinput' will produce units on 'output',
sysinput (mode,size)	E	the others (output processes) take units from 'input'.
sysoutput()	D	'mode' can be any valid unit type (BOOL, CHAR, INT,
syserror()	D	FLOAT, STRING, PROCESS_REF, PORT_REF,EVENT_REF)
perm_sysoutput()	P	'filename' must be a string representing a valid filename;
perm_syserror()	P	otherwise event 'bad_filename' will be raised.
perm_write(filename)	P	The output processes postpone halting until they have received 'disconnected_i.input'.
if(cond, then)	M	manner that executes 'then' if 'cond' delivers 'true' 'cond' can be 'process' or 'group', 'then' can be 'manner' or 'group'
if(cond, then, else)	M	as 'if' above, plus 'else' (same types as 'then') is executed when 'cond' delivers 'false'

Termination conditions:
D = Disconnect, halt after receiving 'disconnected_i.input'
E = EOF,halt after receiving EOF in the file and port 'output' is empty
M = Manner
P = Perpetual, halt after receiving 'terminate'
T = Temporary, halt after performing its function once

Table 1: MANIFOLD builtin process library.

3 Manifold Programs.

In this section some larger working MANIFOLD programs are presented, each with a short description of their operation.

3.1 Fibonacci series.

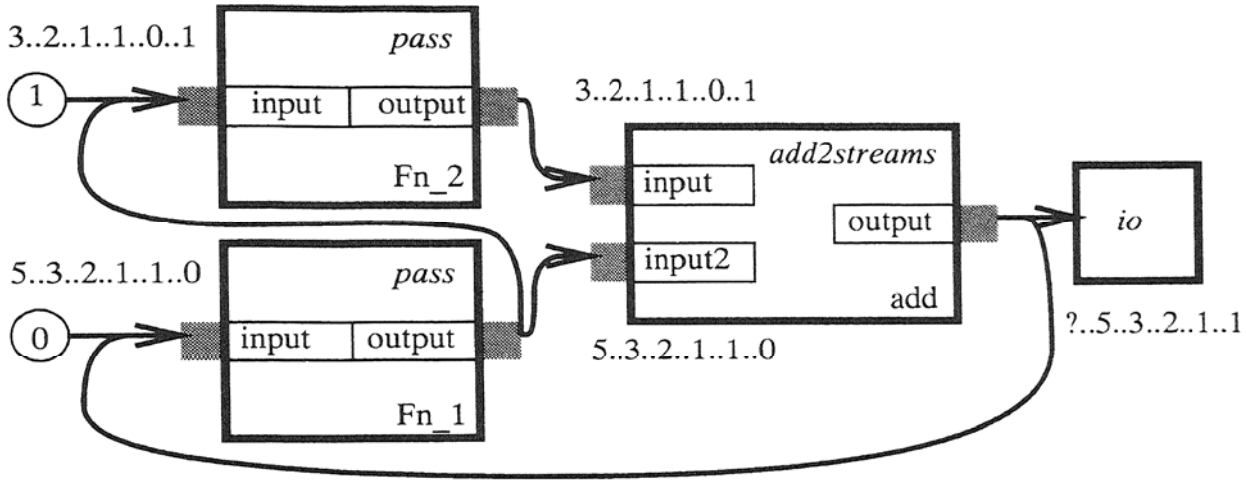


Figure 17: Network of processes to compute the Fibonacci series.
The series of inputs for the processes are shown up to the computation of f(5).

The first program is a classic academic example to show how to use a programming language: the computation of the Fibonacci series. This program is due to Eric Rutten[5].

3.1.1 Program description.

Figure 17 shows a network of processes aimed to produce the Fibonacci series at the input port of `io`. This series is defined as:

$$\begin{cases} f(0) = 1 \\ f(1) = 1 \\ f(n) = f(n-1) + f(n-2) \end{cases}$$

In the middle is of the figure a process named `add`, an instance of the manifold `add2streams`, which has two input ports (`input` and `input2`). This process repeatedly takes one unit from each of its input ports, adds them, and puts the result on its output port. This unit is copied and moved both to the implicitly activated manifold `io` (for outputting on the screen) and the process `Fn_1`, an instance of `pass`, which copies each unit from its `input` port to its `output` port. This unit is copied and moved both to process `Fn_2` and the `input2` port of `add`. The `add` process also needs the *previous* unit on `input` from `Fn_2`, to produce the *next* unit $F(n+1) = F(n) + F(n-1)$ in the Fibonacci series.

3.1.2 Program listing.

```
/*
 * fibo.m - Manifold 1.0 program to compute Fibonacci series,
 *          F(n) = F(n-1)+F(n-2) for n>=2, F(0) = 1, F(1) = 1.
 *          first unit on 'stdin' is # of iterations;
 *          on 'stdout' the index and the Fibonacci number are printed.
 *
 * Annotations used:
```

```
 * M - manner
 * P - permanent process    (does not die)
 * L - long living process (does not die immediately)
 * T - temporary process    (dies immediately after activation and/or connecting)
 */
#include <built_in.i>   // contains declarations for builtin (atomic) processes

/*      P
 * main - setup network of processes to compute and the Fibonacci series up to
 *        'limit' elements, which is the first unit on port 'input'.
 */
main
{
        process add     is add2streams. // P.
        process Fn_1,
                Fn_2    is pass.        // P.

/* read # of iterations from standard input */
start:  activate limit; activate Fn_1; activate Fn_2; activate add;

        syserror = "Please enter limit (max 47): ";
        limit    = sysinput(mINT, 32);
        syserror = "Limit = ";
        syserror = limit;
        syserror = "\n";
/*
 * 'Fn_1' lags behind 1 unit, 'Fn_2' lags behind 2 units, so that:
 *  on 'add.output' each unit F(n) = F(n-1)+F(n-2).
 */
        (
         0->Fn_1,
         1->Fn_2,
         Fn_1->add.input2,
         Fn_1->Fn_2->add->Fn_1->io
        ).
}
/*              P
 * add2streams - take pairwise all units from both port 'input' and 'input2',
 *                 and put the sum of each pair in port 'output'
 */
add2streams port in input2.
{
        process a,b     is variable.    // P.
        event   get, loop.

start:  activate a; activate b; do get.
get:    getunit(input) ->a;     // variables 'a' and 'b' needed, because
        getunit(input2)->b;     // 'getunit' cannot be an argument of '+'
        a+b->;
        do loop.
loop:   do get.
}
/*    P
 * io - print and count each unit arriving on port 'input';
```

```
 *      shutdown if 'limit' (global) is reached
 */
process limit   is variable.    // P.

io
{
        event   loop, next, exit.
        process sysout  is perm_sysoutput.
        process index   is variable.    // P.

start:  activate sysout;
        activate index;
        index = 0;
        do loop.

loop:   index   ->pass1 ->sysout;
        "\t"            ->sysout;
        getunit(input)  ->sysout;
        "\n"            ->sysout;
        index = index + 1;
        if (index >= limit, do exit);
        do next.
next:   do loop.

exit:   cancel.
}
```

3.1.3 Comments.

The program starts with activating the necessary processes, then asks the user for input and sends the user input (from `sysinput`) to the process `limit`, which is an instance of `variable`.

Here the use of an auxiliary process, the implicitly activated instance of `pass1` (which copies one unit from its `input` port to its `output` port and then halts) is needed because otherwise the program would hang (the pipeline ``` `sysinput(...)->limit` ``` would never break because neither of these two processes halt [1]).

This is an example of the *pipeline breakup* problem. Other problems when using pipelines with multiple $\rightarrow$' *s* may arise when one or more participating processes halt, because the resulting flow of units is then undetermined in general.

For example, if the pipeline containing the welcome message
`"Please enter limit:"->syserror`
would contain a `pass1`, e.g:
`"Please enter limit:"->pass1->syserror`
there is a great probability that this whole pipeline will be broken before the unit containing the string reaches `syserror` because a constant is defined on page 35 in [1] to behave like an implicitly activated process, putting one unit on its output port and terminating subsequently. As a result, nothing would be visible in most runs of the program.

Therefore multiple $\rightarrow$'s are only safe to use when all processes in the pipeline are *perpetual* (i.e. they never halt).

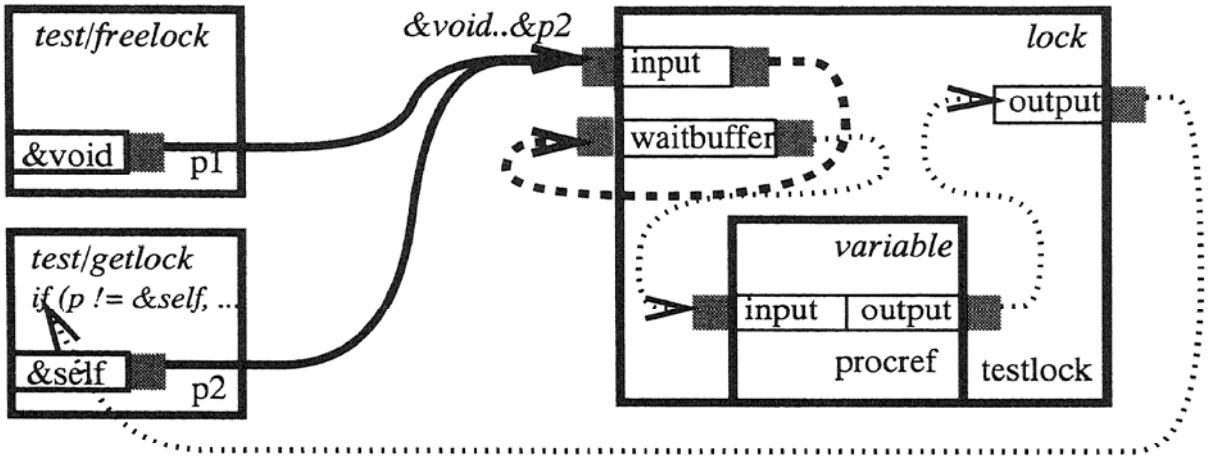
To move one unit between processes or ports it is generally safe to use the assignment operator "=".

3.2 Binary semaphores.

A binary semaphore is an entity `S` upon which two operations are defined: `P(S)` and `V(S)` defined (see, for example ref.[8]). When the operation `P(S)` completes, the process executing that operation has

exclusive access on the semaphore S until it executes the operation V(S), meaning that during this time the operation P(S) will not be completed for any other process. Such a process will be delayed until the process for which P(S) has been completed executes V(S). The operation P(S) is an *indivisible operation*, which means that if two or more processes execute P(S) simultaneously, the operation completes for one of them while the other processes will be delayed until the process that has been granted the lock, executes a V(S) .

3.2.1 Program description.



connections involved in requesting/releasing a semaphore

connections involved in processing a lock reqest

connections involved in releasing a lock and granting the next request waiting

Figure 18: Processes involved in semaphore implementation; process networks needed for both setting and releasing a lock are shown.

The operations on semaphores are implemented as two manners `getlock` and `freelock` for `P` and `V` respectively, each having one process argument. The actual argument when called must be a global process being an instance of manifold `lock` which is also defined in this file.

`getlock` activates the lock process (which is a no-op if it was already active) and sends a unit to it containing a unique reference to the process requesting the lock (&self). Next it repeatedly checks on the output of the lock process until the lock has been granted [1], then returns to the caller.

`freelock` releases the lock by sending a special unit to the lock process (containing a unique reference to predefined `void`).

The lock process itself works as follows. When lock requests units are available on its input port, each unit is taken, stored in the variable `tmp` and inspected to see if it is a `P` or a `V` request (a reference to `void` is `V`).

For a `P`, lock request, when the lock is available (`procref == &void`), the request is granted: the process reference of the requestor is put both in `procref` and on the output port. When the lock is not

[1] This is *busy waiting*. A more efficient implementation is possible by declaring a private event in `getlock`, sending a reference to that event to `lock`, and wait until `lock` raises that event. This method is used in § 3.3.

available, the process reference of the requestor is put on "the other side" of the input port `waitbuffer`, where it will remain queued up behind other request until all lock requests in front have been granted and released.

For `V`, release request, the next unit in `waitbuffer` will be granted as before; if `waitbuffer` is empty, the port event `disconnected_i.waitbuffer` will be raised by the system to indicate that fact and then `procref` is reinitialized to `&void`.

3.2.2 Program listing.

```
// sema.m - binary semaphore in Manifold 1.0
#include <built_in.i>
#define manifold
#define P(x) getlock(x)
#define V(x) freelock(x)

        export manner
getlock (p)
        process p.
{
        permanent p.
        event check, waitp, put.


start:  activate p;                     // init lock process
        do put.
put:    putunit(&self,p);               // set the lock, save
        do check.                       // lock events while
                                        // setting the lock
check:  if (p != &self, do waitp);      // check if the lock
        return.                         // was granted
waitp:  do check.                       // else wait for p
}
        export manner
freelock (p)
        process p.
{
start:  &void->p.                        // free the lock
}
        export manner
putunit (pi,po) // put one unit 'pi' on destination 'po'
        port in pi.
        port out po.
{
        process p1 is pass1.

start:  (activate p1, pi->p1, p1->po);
        deactivate p1.
}
        manner
grant
        process tmp, procref dynamic.
        event wait, give dynamic.
{
start:  putunit (tmp, procref);
```

```
        do give.
}
        manner
release
        process procref dynamic.
        event wait, give dynamic.
        port in waitbuffer dynamic.
{
start:  getunit (waitbuffer) -> procref;
        do give.
disconnected_i.waitbuffer:
        &void -> procref.
}
        manner
handle_unit
        process tmp, procref dynamic.
        port in waitbuffer dynamic.
{
start:  if (tmp == &void, release);
        if (procref == &void, grant);
        putunit (tmp, self.waitbuffer).
}
        export manifold // read and store process references from 'input',
lock                    // or references to 'void' to release the lock.
        port in waitbuffer.
{                       // on output port, produce ref. to process
                        // to which the lock has been granted.
        process procref, tmp is variable.
        event give, wait, getunits, nxtunit.

connected_o.output:
// if the lock is available, grant it,
// otherwise the consumer will hang until it is released
        if (procref == &void, &void->output);
        do wait.
connected_i.input: save.
getunits:
        getunit(input)->tmp;
        handle_unit;
        do nxtunit.
give:   putunit(procref,output);
        do getunits.
nxtunit:do getunits.

start:  activate procref;       &void->procref;
        activate tmp;           &void->tmp;
        do wait.
connected_i.input:      save.
connected_o.output:     save.
wait:   void.
connected_i.input:      do getunits.
}
/*
 * test program for getlock, freelock.
```

```
 */
process testlock is lock.

test (p) process p.
{
event next.
start:  P(testlock);
         syserror = p; syserror = "has lock.\n";
        V(testlock);
        do next.
next:   do start.
}
        manner
starttest (n) process n.
{
        event   exit.
        process nm1 is variable.        // 'n' minus 1
        process ntest is test.          // 'n'th instance of test
start:  activate nm1;
        nm1 = n - 1;
        activate ntest (n);
        if (nm1 == 0, do exit);
        starttest (nm1).
exit:   deactivate nm1.
}
main
{
        process    Nproc is variable.
start:  activate Nproc;
        syserror = "Please enter Nproc (# of parallel test processes): ";
        Nproc    = sysinput(mINT, 32);
        syserror = "Nproc = ";
        syserror =  Nproc;
        syserror = "\n";

        starttest (Nproc).
}
process printlock is lock.
        export manner
print(x) port in x.
{
start:  P (printlock);
        sysoutput = x;
        V (printlock).
}
/*
 * usage of getlock, freelock.
 */
        export manner
print(x1,x2) port in x1,x2.
{
start:  P (printlock);
        sysoutput = x1;
        sysoutput = x2;
```

```
        V (printlock).
}
        export manner
print(x1,x2,x3) port in x1,x2,x3.
{
start:  P (printlock);
        sysoutput = x1;
        sysoutput = x2;
        sysoutput = x3;
        V (printlock).
}
// cleanup printer lock process
        export manner
deactivate_print (mess) port in mess.
{
start:  print (mess);
        deactivate printlock.
}
        export manner
deactivate_print
{
start: deactivate_print ("Deactivating printlock...\n").
}
```

3.2.3 Comments.

This example shows various aspects of the usage of (input) ports. Since non-flushing streams have unlimited capacity used as FIFO queues ([1], page 6), the input port `waitbuffer` can be used as a storage buffer for pending lock requests.

The connections between various ports in this example are all transient, they are broken as soon as one unit has passed (e.g. manner `putunit`).

Multiple connections to and from each port may exist at any time (e.g. port `input` of `lock` in Figure 18); each time when the *first* connection is made to or from a port the event `connected_i` or `connected_o` is raised respectively ([1], page 37-38); similarly each time when the *last* connection to or from a port is broken (and buffers are empty) the event `disconnected_i` or `disconnected_o` is raised respectively.

The source of these events is `self`, and they are always *permanent* event sources (the MANIFOLD programmer may declare *permanent* event source using the `permanent` declarative, see [1], page 22).

Great care has to be taken in utilizing *permanent event sources* simultaneously with *manner calls*, which are used passim (e.g. assignment "=" and sequential operation ";" are manner calls). These manner calls may end prematurely when preempted by a permanent event source such as a port producing an event. For example, when this happens in the manner `print` in the above program after the call `P(printlock)` but before the call `V(printlock)`, this lock will never be released and on the next call `P(printlock)` deadlock will result.

This is an example of the *manner preemption* problem. The only method to avoid this, is that the originator of permanent event sources defines a `save` block for these events, placed in such a way that all manner calls used in the whole MANIFOLD program may return normally, i.e. not being preempted by an event from a permanent source. The place for a `save` block must be chosen such that when a circular search is initiated through all handler blocks to find a handler for that event, a handler containing the keyword `save` will be found.

This is shown in the above program, the manifold `lock` needs to utilize both `connected_i.input'` and `connected_o.output'` and has all handler blocks protected where manners are used by appropriate `save` handler blocks.

3.3 Dining Philosophers.

The dining philosophers problem is a classical example of resource sharing between a number of simultaneously active entities (see e.g. ref [8]). Five active entities ("philosophers") are sitting at a round table each equipped with one chopstick and wanting to eat; however to be able to eat each philosopher needs two chopsticks; after completing the meal (in finite time) each philosopher drops both chopsticks so that they become available for other neighboring users.

The problem is to devise and implement a scheme (algorithm) so that each of them will be able to eat, with equal probability in the long run. For example, when they all would start attempt to eat together at the same time, nobody gets two chopsticks and nobody would ever eat while waiting for their neighbor's chopstick to become available (deadlock); when they would release the left chopstick when the right chopstick is not available (so that a neighbor gets a change to eat), there exists the probability that when they all do this repeatedly at the same time, nobody ever gets two chopsticks (starvation).

This problem can be elegantly modelled and solved using binary semaphores as shown in ref [8]; here we model the problem using the communication mechanisms available in MANIFOLD after an original idea of F. Arbab.

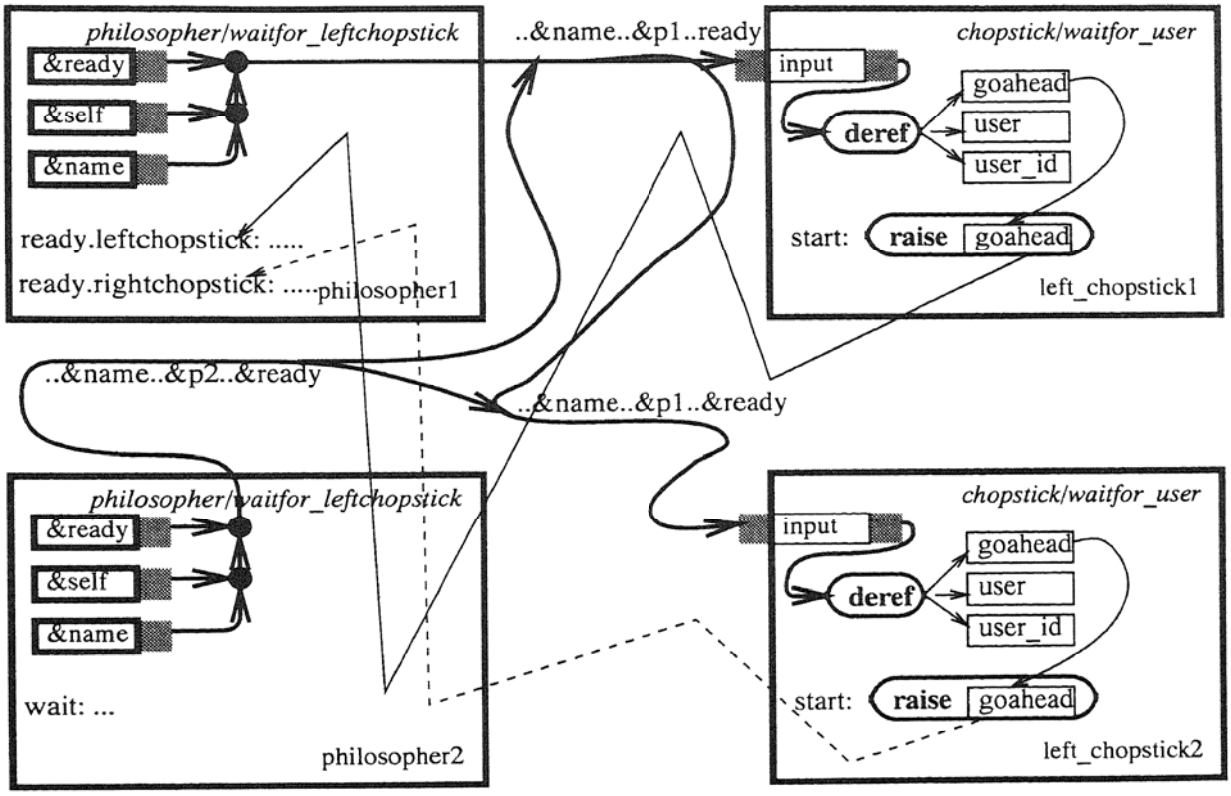


Figure 19: The Dining Philosophers Problem in MANIFOLD .
To the left, two philosophers competing for two chopsticks to the right. Philosopher 1 got both chopsticks, they send them events for which philosopher 2 has no handler. The stream of units is depicted as to be just after philosopher 1 requested both chopsticks, but prior to this moment the events shown have been raised by the chopsticks.

3.3.1 Program description.

Philosophers and chopsticks are implemented as communicating processes implemented in MANIFOLD. In Figure 19 the minimum configuration of two philosophers and two chopsticks is shown.

In the recursive manner create_philosopher (initially called by main) two new process instances are declared and activated, one for a philosopher and one for its right chopstick, while the left chopstick, belonging to its previously declared neighbor, is passed as a parameter. Only for the last philosopher a different scheme is used, its right chopstick is the globally declared process firstchopstick, which has been activated by main.

Each philosopher in its start block sends three units to both its right and left chopsticks under global semaphore protection. These units are references, denoted by a "&", to a local event ready, self (the philosopher process) and name (an integer only used for printing purposes), in that order. Thereafter, the philosopher goes into a wait state, to be taken out by events from any of the chopsticks he wants.

Now if two philosophers send this package of three units at the same time to the same chopstick, by the semaphore protection one package will be queued after the other.

The chopsticks, in the manner waitfor_user, take these units from their input ports, and dereference them to local names goahead, user, and user_id.

Having done this, the event goahead is raised. Since this was originally a local event, it can only be picked up by the philosopher from which it originated. The chopstick, in waitfor_user, waits there until an event from user breaks this group.

The philosopher awakes, find that the event ready had been fired from one of its chopsticks, goes into the corresponding state, and waits in a manner for the other chopstick to become available. When this happens, the statement "do ev" is executed, for which there is no handler in the manner. But there is one in the calling scope, eat. Thus the manner execution is preempted (returned from) and the state labeled eat in philosopher is entered. In this state the event done is raised, which is picked up by all processes which are susceptible for it (i.e. preemptable by it). These are both chopsticks, in the manner waitfor_user. Again, a "do" is executed, for which there is no handler, but there is one in the calling scope, chopstick. So the manner execution is preempted and the actions at the label next are executed, which is simply "do start". Therefore the process repeats it cycle calling the manner waitfor_user, in there dereferencing the triple, etc.

At the same time, the philosopher repeats its cycle by executing "do start", putting three reference units under semaphore protection on the input ports of both its chopsticks.

3.3.2 Program listing.

```
/* dinphi.m - dining philosophers problem in Manifold 1.0 */
#define NUMBER_OF_PHILOSOPHERS 5
#include <built_in.i>
#include "sema.i" // contains declarations for processes defined in "sema.m"

event done.
        manner
waitfor_user
        process n dynamic.
        event next dynamic.
{
        event goahead deref input.
        process user deref input.
        process user_id deref input.

start:  (raise goahead, user).

done.user:
        print("Philosopher ", user_id, "drops chopstick ", n, ".\n");
        do next.
}
chopstick (n) process n.
port in input.
```

```
{
        event   next.

start:  waitfor_user.
next:   do start.
}
        manner
waitfor_rightchopstick (ev)
        event ev.
        process name dynamic.
        process rightchopstick, rightchopstick_id, leftchopstick_id dynamic.
        event ready dynamic.
{
        event wait.

start:  print ("Philosopher ", name,
               "takes in left hand chopstick ", leftchopstick_id, ".\n");
        do wait.

ready.rightchopstick:
        save.
wait:   rightchopstick.
ready.rightchopstick:
        print("Philosopher ",  name,
              "takes in right hand chopstick ", rightchopstick_id, ".\n");
        do ev.

}
        manner
waitfor_leftchopstick (ev)
        event ev.
        process name dynamic.
        process leftchopstick, leftchopstick_id, rightchopstick_id dynamic.
        event ready dynamic.
{
        event wait.

start:  print ("Philosopher ", name,
               "takes in right hand chopstick ", rightchopstick_id, ".\n");
        do wait.
ready.leftchopstick:
        save.
wait:   leftchopstick.
ready.leftchopstick:
        print ("Philosopher ", name,
               "takes in left hand chopstick ", leftchopstick_id, ".\n");
        do ev.

}
process put3units is lock.

philosopher (name, leftchopstick, rightchopstick,
                        leftchopstick_id, rightchopstick_id)
        process name, leftchopstick, rightchopstick,
```

```
                        leftchopstick_id, rightchopstick_id.
{
        event ready, next, eat, restart.

ready.leftchopstick:
        waitfor_rightchopstick (eat).
ready.rightchopstick:
        waitfor_leftchopstick (eat).

eat:    print("Philosopher ", name, "is dining.\n");
        raise done;
        print("Philosopher ", name, "is thinking.\n");
        do start.
start:  P (put3units);
          &ready->(->leftchopstick, ->rightchopstick);
          &self ->(->leftchopstick, ->rightchopstick);
          &name ->(->leftchopstick, ->rightchopstick);
        V (put3units);
        do next.
ready:  save.
next:   (leftchopstick, rightchopstick).
}
        process
firstchopstick is chopstick.
        manner
create_philosopher (nparam, leftchopstick, leftchopstick_id)
                process nparam, leftchopstick, leftchopstick_id.
{
        process n is variable.
        process rightchopstick is chopstick.
        process this_philosopher is philosopher.
        event   last, quit.

start:  activate n;
        n = nparam;
        if (n < 1, do quit);
        if (n == 1, do last); // if commented out, sit at straight table
        activate rightchopstick (n);
        activate this_philosopher (n, leftchopstick, rightchopstick,
                                        leftchopstick_id, n);
        create_philosopher (n-1, rightchopstick, n).
last:   activate this_philosopher (n, leftchopstick, firstchopstick,
                                        leftchopstick_id, 1).
quit:   deactivate n.
}

main
{
start:
        activate firstchopstick (1);
        create_philosopher (NUMBER_OF_PHILOSOPHERS, firstchopstick, 1).
}
```

3.3.3 Comments.

In this example several new features of the MANIFOLD language are introduced:

- the reference operator "&", which can be used for event names, process names and local port names. This is a primitive action, producing one unit representing its operand that can be forwarded trough pipelines.
- the keyword "deref", which can be used as an attribute after event, process or port declarations in the private declaration section of a manifold or manner. The keyword "deref" must be followed by a port name (of type port in), a process name or a process.port (of type port out) construct. It is the programmers responsibility to guarantee that the type of the units coming out of the specified port at run-time match the specified declarative, otherwise the result is unpredictable and the manifold may hang. In the example this is achieved by placing semaphores around the reference producing actions. (Another method is packing multiple units into one, and unpacking them in the deref port using regular expressions, a topic which is not treated in this document).

  The result of the deref declarative is that the declared name can be used in the same way as the original object that was the operand of the matching reference action, except that dereferenced events cannot appear in the label section of handler blocks.
- distribution of a pipeline over a group: the reference units are passed to multiple destinations in one action using the *group* construct "(..,...,..)" and properly placed "->" operators (MANIFOLD specification [1], page 30-31).

Note that in the manner create_philosopher the first parameter nparam is copied into the active local process n (variable). This is not necessary when *actual* parameters are constant processes, variable processes, or other processes which are designed to repeat producing the unit on an output port each time they are connected on that port. It is necessary, however, if the *actual* parameter is an *expression* (e.g. n-1), since the processes to which *operators* like "-" (minus) map are designed to terminate after producing one unit (MANIFOLD specification [1], page 42).

In other words, in MANIFOLD, there is no call-by-value mechanism since the notion of a value, variable or data does not exist in the MANIFOLD language. The only notions are events, processes, ports and connections between ports. If in the example program the *formal* parameter nparam would have been used everywhere instead of the locally declared variable process n, the program would hang at the second if manner call in the second call of the manner create_philosopher (first recursive call).

3.4 Bucket sort.

The idea of the "bucket sort" example is to split an unsorted input stream into a number of unsorted "buckets" to be sorted in parallel by an equal number of sorters; then merge the output of all of them together to form one sorted output.

Since in general the sort is the hard part, parallelism is potentially advantageous here; splitting and merging are obviously potential bottlenecks.

This example originally appeared in ref. [2].

3.4.1 Program description.

The program consists of two manifolds and one atomic process which are instantiated (created and activated) recursively, called sort, merge and AtomicSort. Further, there is an atomic process read, and one instance the manifold CountCheck (Figure 20). The actual sorting is performed by the atomic process AtomicSort. In the example it sorts only two units but the idea is that it can take any number of units. In a loop, CountCheck passes units from its input port to its output port while counting them until the parameter limit has been reached. Then it raises the global event CountReached and waits for the event connected_o.output on its port output, whereupon it reenters its main loop. Moreover, all units are inspected to match a second parameter (target_var). When this happens, the event TargetFound is raised and the manifold attempts to terminate, but does not do so before all pending units (if any) on its

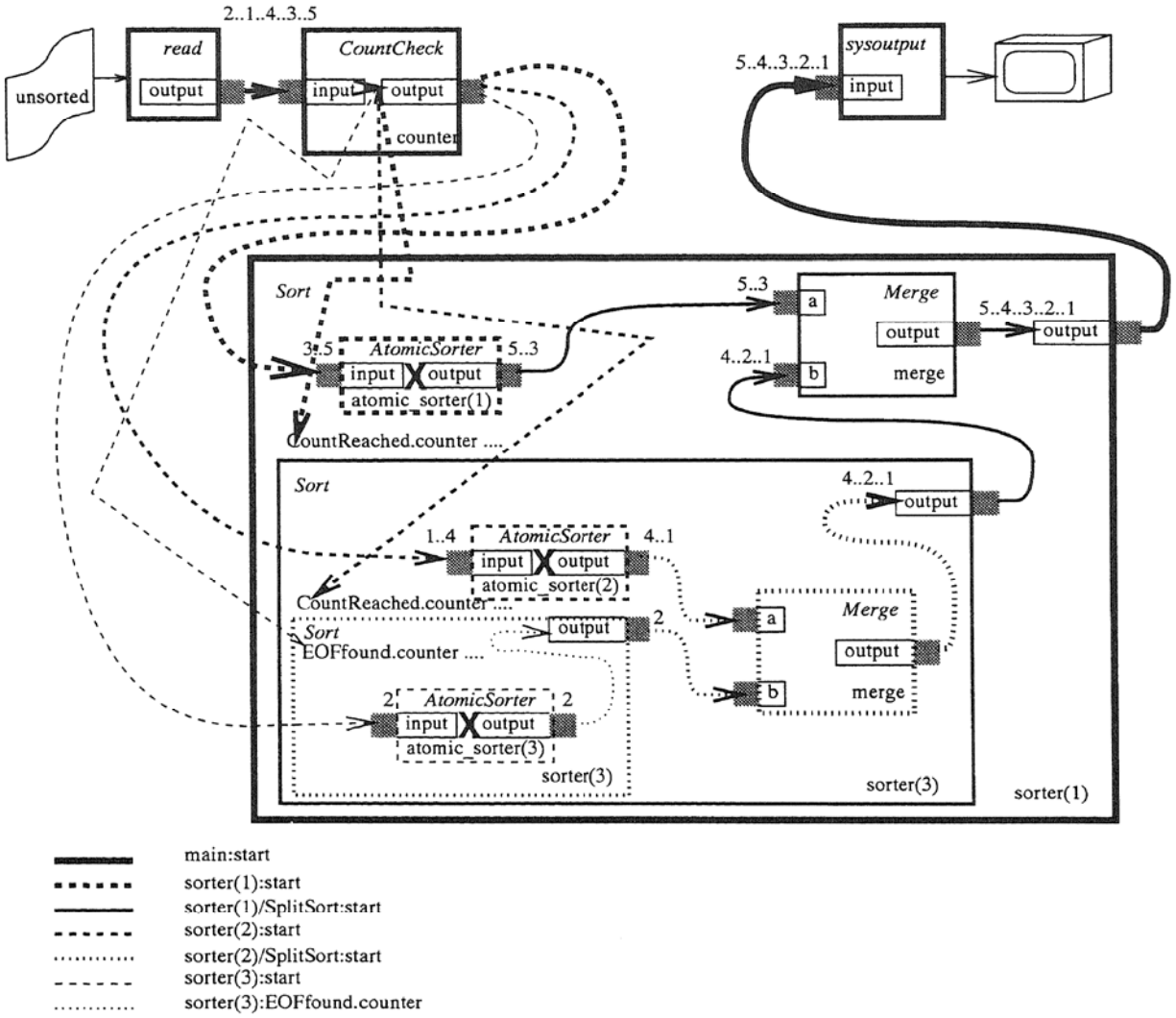


Figure 20: Bucket sort in MANIFOLD for 5 unsorted units (5,3,4,2,1).

output port have left the port (disconnected_o.output must be received after connected_o.output). This manifold is used to distribute the unsorted units in pairs (limit == 2) over a number of sorter processes. After receiving CountReached, each, except the last sorter process, instantiates another sorter, a merge process and an atomic_sorter process in the manner SplitSort. The last sorter connects the (sorted) output from its atomic_sorter to its own output port, which, in the recursive network of processes one level higher, is connected to one of merge's two input ports. The other input port of merge is connected to the output port of the next atomic_sorter and the output port of merge is connected to the output port of the next sorter, etc.

In main, the unsorted units are read from file and pipelined to counter followed by a special unit EOF on which counter checks; further the output of the topmost sorter is pipelined to sysoutput, so that the sorted units may be inspected and/or stored.

The code of the merge manifold is straightforward and its explanation is left as an exercise (except for the manner Getunit it uses, which is explicated below).

3.4.2 Program listing.

```
// sort.m - Bucket sort in  Manifold 1.0 . Input from file 'unsorted'.
//
//       CountCheck- copy units, count and hold them, and check each one.
//       SplitSort- recursively create and set up sorters and mergers.
//       Sort     - sort units coming from the global 'counter' process,
//                  putting the first N in an atomic sorter and calling
//                  SplitSort to take care of the rest
//       main     - read units, send them to 'counter', start up the sort.

#define EOF "EOF"
#define NUNITS 2

#include "built_in.i"
#include "sema.i"

AtomicSorter()                  atomic.
        pragma atomic internal AtomicSorter().

        manifold
Merge() port in a. port in b.   import. extern event MergeDone.

        manner
Getunit (pi, unt, label)
        port in pi. process unt. event label.   import.

        extern event
TargetFound, CountReached,PortEmpty.
/*********************************************************************/
// CountCheck - copy all units from port 'input' to port 'output'
//       counting until 'limit' is reached and checking them to see if
//       there is a unit matching 'target_var'.
//       In the former case, 'CountReached' is raised and the manifold
//       suspends copying units until it senses a 'new' connection on its
//       output port.
//       In the latter case (there is a unit matching 'target_var'):
//       - if there are units left in 'output', 'TargetFound' is raised
//         and the manifold waits untill its 'output' port has become
//         empty, then terminates;
```

```
//      - if ther are no unuts left in 'outport', 'PortEmpty' is raised
//        and the manifold terminates.
/*************************************************************************/

        manifold
CountCheck (limit, target_var)
        process limit, target_var.
{
        process n       is      variable.
        process unit    is      variable.
        process targetVar is    variable.
        process limitVar is     variable.
        process ok_to_die is    variable.
        process want_to_die is  variable.
        event   copy,  found, counted,
                loop, next, wait, compare, try_to_die, exit.

start:  activate   n;
        activate   unit;
        activate   targetVar;

limitVar;
        activate   ok_to_die;
        activate   want_to_die;
        limitVar  = limit;
        ok_to_die = false;
        want_to_die = false;
        n         = limitVar;
        targetVar = target_var;
        do wait.
copy:
        ok_to_die = false;
        n = limitVar;
        do loop.
loop:
        Getunit (input, unit, found);
        if (unit == targetVar, do found);
        unit->pass1->output;
        n = n-1;
        if (n<1, do counted, do next).
next:   do loop.

counted: raise CountReached;
        do wait.
found:  if (n == limitVar, raise PortEmpty, raise TargetFound);
        want_to_die = true;
        do try_to_die.

disconnected_i.input, // save for handling by Getunit
disconnected_o.output,
connected_o.output:
        save.
wait:   (input, output).
```

```
connected_o.output:
        do copy.
disconnected_o.output:
        ok_to_die = true;
        if (want_to_die == true, do try_to_die, do wait).
disconnected_o.output,
connected_o.output:
        save.

try_to_die:
        if (ok_to_die == false, do wait);
        if (want_to_die == false, do wait,
                do exit).
terminate, exit:
        deactivate n;
        deactivate unit;
        deactivate targetVar;
        deactivate limitVar;
        deactivate ok_to_die;
        deactivate want_to_die;
        halt.
}
        process
counter is CountCheck.

/*************************************************************************/
// SplitSort -  create and connect another sorter and merger.
/*************************************************************************/
        manner
SplitSort
        process atomic_sorter dynamic.
{
        process merge           is Merge.
        process next_sorter     is Sort.

start:  (
         activate next_sorter,
         activate merge,
         atomic_sorter->merge.a,
         next_sorter  ->merge.b, merge->output
        ).
MergeDone.merge:
        return.
}
/*************************************************************************/
// Sort - sort units originating from the global 'counter' process,
//       putting the first N in an atomic sorter and calling 'SplitSort'
//       to take care of the rest. When the input (from 'counter') has
//       dried up, direct the output of the atomic sorter, if any, on
//       port 'output'. If the call is recursive, 'output' will be
//       connected by 'SplitSort' to a merger process.
/*************************************************************************/
        manifold
Sort
```

```
{       event   wait_empty.
        process atomic_sorter   is AtomicSorter.

start:  (activate atomic_sorter,
         counter->atomic_sorter).

TargetFound.counter:
        atomic_sorter->output;
        do wait_empty.
disconnected_i.output:
        save.
wait_empty:
        output.

PortEmpty.counter,
disconnected_i.output:
        halt.
disconnected_i.output:
        save.

CountReached.counter:
        SplitSort;
        do wait_empty.
CountReached.counter,   // these events from 'counter' will be handled
PortEmpty.counter,      // in nested calls, must be ignored
TargetFound.counter,    // in this environment
terminate:
        ignore.
}
/*************************************************************************/
// main - read units from file 'unsorted', send them to 'counter', start up
//        the sort, connect its output to 'sysoutput' to print them,
/*************************************************************************/
        manifold
main()
{
        event   exit.
        process sorter          is Sort.

start:  read( "unsorted", mINT, 0 )->counter;
        EOF->counter;
        activate sorter;
        (
        activate counter (NUNITS, EOF),
        sorter->sysoutput).
death.sorter:
        do exit.
exit:   deactivate counter;             // cleanup
        halt.
}
// merge.m - merge part of sort program in Manifold 1.0
//       Getunit - "safe" version of 'getunit'
//       Merge   - merge two streams from port 'a' and port 'b' to 'output'.
```

```
#include <built_in.i>
#include "sema.i"

/*************************************************************************/
// Getunit - read unit from input port 'pi' into process
//       'unit' (is variable); jump to 'label' if port is empty.
// Prerequisites:
//       'unit' must be activated in the calling environment;
//       the caller must have 'disconnected_i.pi: save.'
/*************************************************************************/
        export manner
Getunit (pi, unit, label)
        port in pi. process unit. event label.
{
        event it, goon, discnctd.

start:  unit = &goon;
        do it.
disconnected_i.pi:
        save.
it:     getunit(pi)->unit;
        do goon.
disconnected_i.pi:
        do discnctd.
discnctd:                       // regenerate 'disconnected_i' to avoid
        if (unit != &goon,      // hang in 'getunit' on next call
                (&system->self.pi, getunit(pi)->void));
        do goon.
goon:   if (unit == &goon,   do label);
        if (unit == &system, do start);
        return.
}
        extern event
TargetFound, CountReached, PortEmpty.
        extern event
MergeDone.
/*************************************************************************/
// Merge - merges all incoming units from port 'a' and port 'b' to port
//       'output'. Raises 'MergeDone' when ready, then waits for
//       'disconnected_i.output' (output flushing).
/*************************************************************************/
        export  manifold
Merge   port in a.
        port in b.
{
        process aVar, bVar      is variable.
        event   compare, a_smaller, b_smaller, exit,
                a_loop, a_next, b_loop, b_get, bnext.

start:  activate aVar;
        activate bVar;
        Getunit(a,aVar,b_get);
        Getunit(b,bVar,a_loop);
        do compare.
```

```
compare:
        if (aVar < bVar, do a_smaller, do b_smaller).
a_smaller:
        output = aVar;
        Getunit(a,aVar,b_loop);
        do compare.
b_smaller:
        output = bVar;
        Getunit(b,bVar,a_loop);
        do compare.
disconnected_i.a,
disconnected_i.b:
        save.

b_loop: output = bVar;                    // move rest in b
        do b_get.
b_get:  Getunit(b,bVar,exit);
        do b_loop.

a_loop: output = aVar;                    // move rest in a
        Getunit(a,aVar,exit);
        do a_next.
a_next: do a_loop.
disconnected_o.output:
        save.
exit:   raise MergeDone;
        output.
disconnected_o.output:
        deactivate aVar;
        deactivate bVar;
        halt.
}
/*
 * atomic_sorter.c - Manifold 1.0 atomic process in 'C'
 *
 *       AtomicSorter - sort 2 units from 'input'
 *                      and put them on 'output'
 */

#include <ap_interface.h>

void AtomicSorter (mf, parent)
     void *mf, *parent;
{
  int           port     = ANY_PORT;
  apEvent       event;
  apSource*     source   = ANY_SOURCE;
  int           val[2]; /* value of unit. */
  int           n = 0;
  boolean       eoi = false;

  while (n < 2 && eoi == false )
    {
      switch (ap_await_anything( mf, &port, &event, &source)) {
```

```
      case UNIT :
        if (port == INPUT ) {
          if (ap_getint( mf, INPUT, val+n) !=   WRONG_TYPE)
            n++;
        }
        else ap_delete_unit ( mf, port );
        break;
      case EVENT :
        if ( ap_events_equal( &event, mTerminate) )
          ap_terminate ( mf );
        else if ( ap_events_equal( &event, mDisconnected_i))
          eoi = true;
        break;
        default :
          break;
      }
    }
  if (n == 0 )
    ap_terminate ( mf );
  else if ( n == 1 )
    ap_putint ( mf, OUTPUT, val[0] );
  else if ( val[0] <= val[1] )
    {
      ap_putint ( mf, OUTPUT, val[0] );
      ap_putint ( mf, OUTPUT, val[1] );
    } else
      {
        ap_putint ( mf, OUTPUT, val[1] );
        ap_putint ( mf, OUTPUT, val[0] );
      }
  ap_await_port_empty ( mf, OUTPUT );
  ap_terminate ( mf );
}

void AtomicSorter_cleanup(mf) void* mf; {}
```

3.4.3 Comments.

This program differs in some respects from the one which originally appeared in ref. [2], page 18-22.

- First of all, in the present program, a rather complicated manifold, `CountCheck`, is used to control the distribution of units over the `sorters` (by counting and holding the units), whereas in the original program the more straightforward manifold `count1` (ref. [2] page 7) is used in the `start` block of `sort` in the pipeline:

  ```
  input->Count->Sort_units;
  ```

  (ref [2] page 22).

  The original idea was, that here `Count`, an instance of `count1`, would die (terminate) after having counted the specified number of units and that this event, `death.Count`, would trigger the next step in the recursion. This would work if in the above pipeline the connection between input and `Count` would have been implemented as a *synchronous* stream, where each unit leaves its producer port indivisibly at the same time as it is consumed by its consumer port(s).

However, in MANIFOLD streams are *asynchronous* (ref [1], page 6) and many units may have left the local port `input`, as many as the run-time system can handle, waiting for consumption at `Count`'s `input` port, when its counter expires and the process `Count` dies, together with all these waiting units in its `input` port.

This is the *lost units* problem; it has been remedied in the present program by taking care that `CountCheck` does not terminate, but hold its units for the next consumer, until all units have been passed and the last connection on its `output` port has vanished.

- Next, the manifold `merge` uses a rather peculiar manner `Getunit` to get the units from its input ports instead of the primitive action `getunit`.

  This manner gets a unit from its first parameter `pi`, puts this unit in its second parameter `unit`, and jumps to the `label` in its third parameter when there are no more units in the specified input port and all connections on this port have been broken. The `disconnected_i` event is used to determine this: when a transfer is attempted on a disconnected input port using `getunit`, the event `disconnected_i` is raised (ref [1], page 28 and 38), which can be used to determine that this port has become empty. However, this same event will also be raised by the system when the number of connections at that port drops from one to zero. This may happen by some external cause, e.g. another process having made and broken a connection to that port.

  When this happens simultaneously with the `getunit` primitive action *while preemptable* by the `disconnected_i` event (to check for empty port) *and* a unit is available (so the port is not empty) the event is nevertheless raised (since the last connection has broken) which could rightly be interpreted to mean that this port has become empty (so there was no unit). In this case sometimes the last unit is lost non-deterministically.

  This is the *last unit* problem; it has been remedied by initializing the destination `unit` (second parameter of `Getunit`) with a value that nowhere else in the system can exist (`&goon`). After the `getunit` primitive action it is checked whether the `unit` still has this initial value. If so, it must be concluded that there was no unit available. This alone is not sufficient, however. If the last unit is being processed *and* the event `disconnected_i` is raised simultaneously by breaking the last connection, the next time the `getunit` primitive action is performed on this port, `disconnected_i` will not be raised any more, because this has already been done [2]. Since there are no more units, the program will hang on this `getunit` primitive action. In order to avoid this, the `disconnected_i` event will be forced in this case, by producing a dummy unit `&system` on this port and a `getunit` to get rid of the dummy so that on the next call, the desired event will be raised.

As mentioned in the introduction to the present example, it is potentially advantageous to do the hard work in parallel by distributing the `atomic_sorter` processes. Since the `CountCheck` and the Merge manifolds are potential bottlenecks, they should be rewritten as atomic processes. Further, the `atomic_sorter` processes should be rewritten to take a more substantial number of units and do an efficient sort on them.

Having done all that, setting up a real distributed application in MANIFOLD is now simple: for instance if it is desired to have all instances of `AtomicSorter` run in parallel on different processes, the only thing that needs to be added in the source file is the line:

```
pragma weight 10000 distribute AtomicSorter.
```

This instructs the MANIFOLD compiler to generate the code that will enable a primitive load balancer to distribute these processes over a run-time determined number of processors, the `weight` (by default 100) is the number used for balancing. The load balancer tries to maintain a equal sum of weights on each physical processor.

A more sophisticated example, distributed ray-tracing, where this has been exploited has been described in [7].

[2] Actually, this is an irreparable error in the implementation of `getunit`.

4 Conclusions.

A number of actually working programs have been made in MANIFOLD Version 1.0. In implementing and using this new language, a number of problems have been encountered, which probably could not have been detected otherwise.

The most interesting of these are: the *lost units* problem; the *manner preemption* problem and the *last unit* problem.

However, despite these imperfections, the MANIFOLD conceptual model based on event-driven asynchronously communicating processes has proven to be implementable and it works.

It is believed that by constraining the programmer to *asynchronous* communication, the points in a program where synchronization is mandatory will be better understood and the number of these points will be minimized.

Although such a program will be more difficult to design and implement with current programming methodologies, it will run faster than similar programs built on a synchronous language paradigm (as almost all conventional programming languages are) and it will be more easily adaptable to future hardware configuration changes (e.g. massive parallelism).

Therefore a new coherent version of the MANIFOLD language must be designed, implemented and tested taking into account as much as possible from what has been learned in the first experimental version. Furthermore the importance of the development of new programming methodologies must be stressed (e.g. visualization of MANIFOLD program execution, proof systems for correctness of MANIFOLD programs, object-oriented MANIFOLD on top of the base language to deal with more complex problems).

Acknowledgements.

I would like to thank Farhad Arbab for his close reading of the draft, Per Spilling for his help with LaTeX and the atomic processes, and above all Freek Burger for his continuous support with the simultaneous debugging of my programs, my compiler and the MANIFOLD run-time system.

References

[1] F. Arbab. Specification of manifold version 1.0. Technical Report CS–R9220, Centrum voor Wiskunde en Informatica, Amsterdam, 1992.

[2] F. Arbab and I. Herman. Examples in manifold. Technical Report CS–R9066, Centrum voor Wiskunde en Informatica, Amsterdam, 1990.

[3] F. Arbab, I. Herman, and P. Spilling. An overview of manifold and its implementation. Technical Report CS-R9142, Centrum voor Wiskunde en Informatica, Amsterdam, 1991.

[4] N. Gehani and W.D. Roome. *The Concurrent C Language.* Silicon Press, Summit, NJ, 1989.

[5] E.P.B.M. Rutten. Minifold: a kernel for a manifold-like coordination language. Technical Report CS–R9252, Centrum voor Wiskunde en Informatica, Amsterdam, 1992.

[6] P. Spilling and F. Burger. Manifold project: Manifold toolkit reference manual. unpublished manual, 1992.

[7] P. Splling and F. Arbab. manray - a replicated workers program in manifold. Technical Report to be published, Centrum voor Wiskunde en Informatica, Amsterdam, 1993.

[8] A.S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall International, Inc., Englewood Cliffs, New Jersey, 1992.