1992

I. Herman, F. Arbab

More examples in Manifold

Computer Science/Department of Interactive Systems Report CS-R9214 May

CWI is het Centrum voor Wiskunde en Informatica van de Stichting Mathematisch Centrum
CWI is the Centre for Mathematics and Computer Science of the Mathematical Centre Foundation

CWI is the research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a non-profit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands organization for scientific research (NWO).

Copyright © Stichting Mathematisch Centrum, Amsterdam

More Examples in Manifold

I. Herman, F. Arbab
CWI
Department of Interactive Systems
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
Tel.: +31-20-592.4163, +31-20-592.4058
Fax.: +31-20-592.4199
Email: ivan@cwi.nl, farhad@cwi.nl

ABSTRACT

This document gives an additional insight into the use of the MANIFOLD system by presenting a set of non-trivial examples of programming using the MANIFOLD language. The development of these examples has been inspired by some general algorithmic patterns arising in the field of computer graphics and the use of computing farms.

The document presupposes that the reader is familiar with the syntax and the semantics of MANIFOLD.

1980 Math. Subject Classification: 68N15, 68Q05, 68U30
1987 CR Categories: C.1.2, C.1.3, C.2.m, D.1.3, D.3.2, F.1.2, I.1.3
Keywords and Phrases: Parallel computing, MIMD, Models of computation, coordination languages, computing and processing farms.

1 Introduction

The development of the MANIFOLD system ([1, 2, 3, 4]), 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 ([5]), 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" ([2, 6]). Working on practical examples has been a continuing activity in conjunction with the effective implementation work ([7, 8]). The most complex example of a MANIFOLD application which has been published up to now is the formalization of the GKS[1] input model (see [7]). It is worthwhile to note that for computer graphics experts, who are also involved with concurrency issues, the GKS input model has become a kind of test case for the applicability of a given concurrency model for the purposes of interaction in computer graphics (see [9, 10, 11, 12]). In this respect, our approach to the GKS model has already contributed to an ongoing research activity in computer graphics. In view of these, it is 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 ([3, 4, 13]). The present report is the outcome of our latest activities in this respect.

As a first result of our recent activities in this direction, this report presents two highly non-trivial examples of MANIFOLD programming. In §3 we describe a class of adaptive recursive algorithms. These algorithms are in widespread use in computer graphics as well as in other fields of computer science; the MANIFOLD language has proven to be very appropriate for describing this class of algorithms.

[1]Graphical Kernel System — An ISO Graphics Standard

Report CS-R9214
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

In §4 we deal with what is usually called a "computing farm", which has become extremely important lately in relation to high performance computing. This example is undeniably the largest and most complex program developed in MANIFOLD as of today.

2 An Assessment of the Manifold Model

One of our reasons for developing more complex programs in MANIFOLD is to test our own ideas concerning the MANIFOLD model and language against real practical problems. It was therefore very encouraging for us to see that the computing *model* advocated by MANIFOLD has proven to be extremely well adapted for our problems. We did not really face any significant difficulties in describing our problems using MANIFOLD. In fact, the main problem was with us rather than on the side of the MANIFOLD model: in spite of the fact that we developed the model ourselves, our own training and overwhelmingly "sequentially oriented" programming experience made it sometimes difficult for us to find the best possible solution at once. Programming using a "coordination language" like MANIFOLD does need an alternative way of thinking and hence an appropriate training. We feel, however, that it pays off to go through such and intellectual exercise which, by the way, is very challenging and exciting itself.

As far as the MANIFOLD *language* is concerned we did find some clumsiness here and there which lead us to some slight modifications or, to be more precise, to the definition of some new constructs to be added to the language as described in [1]. In the remaining part of this section, we describe these new constructs to make our examples fully understandable. We must stress, however, that the language extensions defined here are not really fundamental; they just make programming in MANIFOLD easier or more effective. In fact, looking at these extensions more closely, one may realize that all the examples described later can be described in the original version of the language at the expense of some clumsiness or loss of run-time efficiency.

2.1 Inline Manners

The introduction of inline manners is barely more than a convenient shorthand: it makes the program much more readable. It also highlights the real potentials of using manners; one tends to use them much more frequently than with the "traditional" syntax.

Inline manners resemble the block structure of C++ (not to be mixed with blocks in MANIFOLD!). This means that a MANIFOLD block may now look like:

```
label:
    {
        blocks
    }.
```

Semantically, this is equivalent to having the blocks within the "{" and "}" signs as a separate manner, which is then called from this block. In other words, inline manners are exactly like manners, except that their body appears within the outer scope (and hence they are not reusable). This also means that declarations, dereference statements, etc. may appear within an inline manner; their meaning is identical to the situation when they appear within a "real" manner.

Note that the use of inline manners make the circular search obsolete. We can therefore impose a rule which says that within a given scope, i.e., within a manifold or a (traditional or inline) manner there may be at most one block for a given label.

It is somewhat awkward to use the keyword `start` and `return` within inline manners. We use therefore the keywords `begin` and `end` as synonyms for `start` and `return` respectively.

2.2 Permanent Groups

It is sometimes necessary in a MANIFOLD program to have certain connections persist over state transitions. The only clean way of achieving this in the present version of the MANIFOLD language is to define a separate manifold process to set up the required pipeline(s) and activate this process when necessary. As long as this process is active, the corresponding streams will be alive. (Note that repeating the pipelines

in each block of the manifold is *not* an equivalent solution: state transitions result in `connected` and `disconnected` events which can alter the behavior of the processes appearing in the pipelines.)

In order to improve efficiency, the concept of permanent pipelines is introduced. One can now add a declaration of the form:

```
nobreak groups.
```

The meaning of this construction is as follows: the pipelines listed in the declaration are set up *at initialization time* when entering the manifold or the manner. The pipelines may break, following their usual breaking rules (i.e., when one of their components is deactivated) but the manifold processor does *not* break the connection on state transition, *as long as it is within the same scope*. This means that the pipelines will be broken when the processor leaves the current (traditional or inline) manner.

The pipelines in a `nobreak` declaration may contain symbols which are to be dereferenced; the manifold processor makes sure that all symbols are dereferenced *before* they are used to build up connections properly.

2.3 Port Identification

In the present version of the MANIFOLD language, if a process reference is passed either as an argument or is dereferenced, there is no way for the receiving process or manner to use any of its ports other than `input`, `output` and `error` (these ports are always available by default). This is sometimes a disagreeable restriction; we therefore extend the syntax to allow a more elaborate access to ports.

Much like the signature of a process, one can now use what we might call the *port signature* of a process. This means that the declaration of the process takes the form of:

```
process P[port1,port2|port3,port4](signature).
```

or, equivalently

```
process P[port1,port2|port3,port4]() deref some_port.
```

The meaning of this declaration is to say that not only `P` is a process (with possibly a given signature) but it also must have at least two `in` ports named `port1` and `port2` and at least two `out` ports called `port3` and `port4`. The MANIFOLD runtime checks the validity of these assertions (just as for signatures). As a result, one can now refer to the ports `P.port1`, `P.port3` etc. Of course, just as in the case of process signatures, the port signature may be missing, in which case we fall back to the "traditional" case.

It must be emphasized that the declaration does *not* say that `port1`, `port2`, `port3` and `port4` are the only ports of process `P`; it merely states that the process does have ports with these symbols (it may have more).

3 Adaptive Recursive Algorithms in Manifold

In this section, a well-known class of algorithms in the field of computer graphics and image processing is described using the MANIFOLD formalism. It is *not* the purpose of this section to analyze these methods from a strictly algorithmic point of view, nor do we intend to devise new versions of already existing algorithms. We simply intend to show the descriptive power of MANIFOLD using well-established algorithms.

It is beyond the scope of this paper to give all the specific details of each algorithm. The interested reader can consult one of the standard textbooks on computer graphics and/or image processing (e.g., [14] for computer graphics and [15] for image processing) or refer to the literature given in the references (e.g., [16, 17, 18, 19, 20] or others).

3.1 Warnock's Algorithm

One of the very well known problems in computer graphics is what is usually referred to as Hidden Surface Removal. The problem is as follows. When a three-dimensional scene, usually modeled using a large number of planar polygons in space, is visualized on a screen, all of its polygons must be projected

```
TestAndColor() atomic.
DivideArea()
port out firth_area.
port out second_area.
port out third_area.
port out fourth_area.
atomic.

export Warnock()
{
    event   subdivide, wait_to_die, done, finish.
    process test_and_color    is TestAndColor.
    process divide_area       is DivideArea.
    process v                 is variable.
    process n                 is variable.
    start:
        (activate v(),activate n()
         activate test_and_color(),
         input -> (-> test_and_color,-> v),
         void
        ).
    subdivide:
        {
            begin:
                (activate divide_area,
                 v -> divide_area,
                 divide_area.first_area  -> Warnock(),
                 divide_area.second_area -> Warnock(),
                 divide_area.third_area  -> Warnock(),
                 divide_area.fourth_area -> Warnock(),
                 n = 4
                );
                do wait_to_die.
            terminate:
                save.
        }.
    wait_to_die:
        void.
    terminate:
        n = n - 1;
        if( n = 0, do finish, do wait_to_die ).
    done:
        do finish.
    finish:
        deactivate parent.
}
```

Listing 1: Manifold Program for Warnock's Algorithm.

onto a plane (i.e., the plane of the display screen) from a given viewpoint. Mathematically, this projection is well understood, but there is an additional problem to solve: those polygons, or parts of polygons, that are occluded by another one, as seen from the selected viewpoint, must be eliminated. The removal of these (sub-)polygons is what is called the removal of hidden surfaces.

There are several well-known and widely applied solutions to this problem. One of the earliest is Warnock's algorithm which is described in detail in the literature, e.g., in [14]. A short description of this algorithm is as follows.

This algorithm is based on a recursive area-subdivision of the computer screen. At each stage in the recursive subdivision process, the projection of each polygon has one of four relationships to the area of interest (which is, at the beginning, the full screen of the display):

1. *surrounding polygons* completely contain the area of interest;
2. *intersecting polygons* intersect the area;
3. *contained polygons* are completely inside the area;
4. *disjoint polygons* are completely outside the area.

Based on these tests, there are certain cases where the exact color(s) for rendering the area of interest can be determined very easily. Obvious cases include when all polygons are disjoint from the area (and hence the background color can be used), when there is only one polygon which either intersects the area or is contained in it, or when there is one and only one polygon which completely surrounds the area. There are also some less obvious but still easily decidable cases which the original version of the algorithm takes into account.

There are, however, cases where there is no easy way to color the area. In these cases, Warnock's algorithm subdivides the area into four equal sub-areas to simplify the problem and then the same method is applied recursively for each of the four sub-areas. The recursion stops when the dimension of the sub-area has reached the size of one pixel on the screen; some additional calculations are then done to determine the color of this single pixel.

3.1.1 A Manifold Program for Warnock's Algorithms

Before commenting further on the algorithm, let us see how its skeleton can be described using MANIFOLD. The complete listing of the program appears as Listing 1.

The program uses of two atomic processes which implement its truly algorithm specific and numerically oriented details. These atomic processes are "imported", which means that they are external to the present MANIFOLD source file and will be made available at link-time. `TestAndColor` is supposed to receive the description of an area on its standard input (as far as MANIFOLD is concerned, this description is just an abstract unit to be forwarded; we refer to it as "area handle" in what follows). It then performs the test on all polygons in the scene, following the scheme described in the previous section. The result of this step is either:

- the area can be filled without ambiguities, in which case `TestAndColor` raises the event `done`, fills the area with the calculated color(s) and terminates; or
- the area cannot be filled without ambiguities, in which case `TestAndColor` raises the event `subdivide` and terminates.

The atomic process `DivideArea` receives an area handle on its standard input; it has, apart from the standard ports, four publicly declared output ports, onto which it places the four area handles after it performs a subdivision. Once these units are produced, `DivideArea` terminates.

It is the manifold process `Warnock` that embodies the skeleton of Warnock's algorithm. It is important to understand the details of this program to gain a real insight into the descriptive power of MANIFOLD; this is why a more detailed description of this process is given in what follows.

In the declaration part of `Warnock`, two instances of the atomic processes described above are declared. This means that the manifold `Warnock` now has a reference for these processes and can, therefore, involve them into several parallel pipelines, if necessary. The additional two declarations concern two "utility"

processes (part of the standard environment of the MANIFOLD system) which are able to store some units and, if the type of the units permit, to perform some elementary arithmetic on them.

The start state of Warnock activates the two variable processes and the local instance of TestAndColor. A pipeline is then set up, which involves a group as well. This pipeline describes the following relationships:

- a unit (i.e., an area handle) arriving on the input of Warnock is redirected to the local instance of TestAndColor, and
- a *copy* of the same unit is "stored" in the variable v.

The use of the special process void in the same group ensures that the manifold is suspended in this block and must receive an external event to change its state (the process void never terminates and, therefore, the group of the state will never terminate; it can only be preempted). According to our specifications, these external events may be either subdivide or done, depending on the result of the test performed on the local area. (Note that although many instances of TestAndColor may be active and raise the events subdivide and/or done, the only instance of TestAndColor visible to an instance of Warnock is its locally declared one. This is why the other events raised by other instances cause no confusion.)

The state labeled subdivide is obviously the essential part of the manifold Warnock. The corresponding block contains, in fact, two statements, joined by the connective ";", which can be though of as a delimiter for sequential execution. In the first statement, the local instance of the atomic process DivideArea is activated and, also, four *independent* instances of the manifold Warnock are implicitly created and activated (using a process specification name in a statement, instead of declaring an instance in the declaration section, means the implicit creation and activation of an instance of that process). The pipelines defined in the group are fairly straight-forward:

- the content of the variable v is transferred to the area divider, and
- the four handles for the generated sub-areas are forwarded, respectively, to the four (recursive) instances of Warnock [2].

This series of pipelines are the ones which realize the recursive step.

The rest of the manifold Warnock makes sure that the processes are terminated properly. A separate variable (n) is used to store the (constant) value of 4. The top-level instance of Warnock waits for all of its "children" to deactivate before it deactivates itself. This is done by the combination of the states labeled wait_to_die and terminate. The basic idea is that each instance of the Warnock manifold sends a deactivation request to its parent before its own deactivation (see the state labeled end). This deactivation request is turned by the MANIFOLD system into a system event called terminate on the receiver's side; the particularity of this event is that it can always be caught in a manifold, irrespective of the visibility of its originator. This is exactly what the manifold Warnock does: it catches the event and checks against its counter to see if all of its children processes are deactivated before it terminates itself. The if statement used for this purpose is, in fact, a manner, with the obvious meaning and is part of the "standard" MANIFOLD environment.

Note that subdivide is, in fact, a label for an inline manner, which contains, apart from the subdivision procedure described above, a save action for the event terminate. The reason is to avoid a race condition which can happen in the block for subdivide. Indeed, it is perfectly possible that divide_area is still busy calculating, e.g., the fourth sub-area while the Warnock instance for, say, the first sub-area already terminates. Obviously, Warnock must *not* (yet) change state but it must not ignore the event either (otherwise a non-termination will occur). By using an inline manner we make sure that the event is neither lost nor prematurely preempts the state subdivide.

If no subdivision is necessary, Warnock makes a state transition to the block labeled done, which does an immediate state transition again. This, finally, leads to the termination of the manifold. Strictly speaking,

[2]The use of the term *recursive* is perhaps somewhat misleading here. Contrary to its common connotations in other programming languages, there is no implied "wait for return or death of your child" process in MANIFOLD. This means that a parent process can terminate (and have its resources deallocated) as soon as it spins off its (recursively created) children, if there is no functional requirement for it to wait for their results.

it is not necessary to have a separate intermediary state in this case (a block may have multiple labels). However, when our example is extended further in the next sections, having a separate state will prove to be beneficial.

3.2 Analysis of the Program

Warnock's algorithm is an example of the *image space algorithms* in computer graphics. These algorithms are primarily concerned with images and compute the attributes of each pixel on the screen. Resolution of the relationships among objects in a scene becomes a secondary concern. On the other hand, *object space algorithms* are concerned with the properties of and relationships among the objects in a scene and compute an image only after these relationships are determined. Warnock's algorithm is not very much in use today. Indeed, if the hidden surface removal is to be performed in image space, availability of powerful hardware makes other methods (primarily, the so called Z-buffer method) more attractive. Whether or not this preference will persist in the future is a matter of debate and its details are far beyond the scope of this paper.

Nevertheless, Warnock's algorithms is still of interest, because it is a very simple example of a general principle which seems to be extremely popular both in computer graphics and in image processing. This principle is what we might call *recursive subdivision*. The idea is the extremely simple, albeit very powerful, concept of divide and concur: if a problem cannot be solved at a given level, the underlying model is somehow divided and the same algorithm is used recursively on the results of the division. If the subdivision of the problem is chosen appropriately, the problem becomes more easily solvable for each of the results of the subdivision. Interestingly, with a properly chosen subdivision scheme, such algorithms are sometimes readily adaptable for parallel hardware.

Although, obviously, the principle of recursive subdivision is not restricted to computer graphics, its popularity within the computer graphics community seems to be related to the special nature of the field. Indeed, the geometric nature of the underlying problems often gives very clear clues for how to perform the subdivisions and how to control its recursion in an optimal way. Thus, the application of recursive subdivision is very natural in working with synthetic or digital images. Apart from Warnock's algorithm for removal of hidden surfaces, similar or more elaborate approaches can be used in calculating and/or displaying spline curves or surfaces [17], perform calculations on CSG[3] objects using quadtrees [16], digital filtering of images, global histogramming of digital images [21], parallelizing such time consuming rendering procedures as ray tracing [19] especially on CSG objects, performing the calculations necessary to visualize volumes [22], for NC tool path generation [23], etc.

What is the role of MANIFOLD in this respect? Looking at the program on Listing 1, it is clear that MANIFOLD has a real expressive power in describing the skeleton of a recursive subdivision algorithm. Note that the atomic processes used by the program are defined in a fairly abstract way; any atomic process, abiding to these specifications, can be "plugged in" the same MANIFOLD program to serve a different application. Although most of the algorithms listed above require a more sophisticated version of the algorithm (and we will elaborate on these improvements in the following sections), we believe the listing commented in detail in §3.1.1 makes the essential point: that using MANIFOLD it is possible to describe in a very concise and declarative form, the primary communication skeleton of a certain class of systems or algorithms without bothering with their computational details.

These examples also reveal another general and more important characteristic: most of the algorithms cited above were, originally, *not* meant for parallel hardware. Instead, the recursive subdivision approach made the problems at hand just (more) easily solvable and manageable; it was the expressive power of "parallelism" and not performance gains per se, that was important here. It is almost a "by-product" that some of these algorithms are good candidates for true parallelism. We use the term "some" because it is no even certain that all these algorithms run much more efficiently on a true, massively parallel hardware, than on a conventional sequential machine. There may be a trade-off between the obvious gains of parallelism and other considerations (e.g., bulk data access).

Nevertheless, MANIFOLD is useful for expressing the communications and control structure of these algorithms, even if the actual implementation of a MANIFOLD system may run only on a conventional single-processor computer supporting simulated parallelism only (as in the case of our first experimental

[3]Constructive Solid Geometry

implementation based on Concurrent C++, see [3, 4]). This seems to be a clear case of a more general principle: it may be extremely beneficial to use mental models which use concurrency, communication, and coordination, as natural paradigms to grasp the essence of a problem and/or of an algorithm. Concurrency need not be considered a "necessary curse," as perceived by a large number of practitioners. On the contrary, it is often very helpful in conceptual simplification of the problem at hand. In their recent paper on coordination languages ([24]), Gelernter and Carriero, the authors of LINDA, stress that:

> ... in principle you can use the *same* coordination language that you rely on for parallel applications programming when you develop distributed systems. You can use the same model in building ... a file system.

We agree both with this statement, and with their implied position that the same language can also be used to describe systems and problems at large, that will not necessarily end up running in a parallel or distributed environment. We believe that as a coordination language, MANIFOLD is useful towards these ends.

3.3 Improvements to the Program

In this section we present enhancements to the MANIFOLD program described in §3.1 and evolve a better framework for expressing different version of the adaptive recursive algorithms mentioned above. The improvement to the program is done in two steps. First, the restriction of a fixed number of subdivisions is relaxed. Second, we allow the possibility of backward control in the recursive processes; i.e., allow a parent to wait for and use the results produced by its children.

3.3.1 Variable Number of Subdivisions

The program in §3.1 has an obvious restriction that may make it inappropriate for general use in other applications. This program has a "hardwired" subdivision feature: each area must be subdivided into exactly four sub-areas. Although this is natural in the case of Warnock's algorithm, and it is trivial to change the number four, imposing any fixed number by itself is a constraint that hinders more general usability of this program for other applications. In particular, a more general class of recursive subdivision algorithms use an adaptive subdivision scheme wherein the number of subdivisions at each level of recursion, as well as the subdivision boundaries, may depend on the data and thus cannot be predetermined.

In this section, we present an improvement to the MANIFOLD program of §3.1 that allows the number of subdivisions to be determined dynamically at each level. To put our revised MANIFOLD program in the right perspective, we remark that a later version of Warnock's algorithm, called the Weiler-Atherton algorithm (see [14]), subdivides the screen along polygon boundaries, rather than along the two mid-lines of the screen. Clearly, the Weiler-Atherton algorithm requires a variable number of subdivisions.

The revised MANIFOLD program now consists of two parts: the one in Listing 2 and the one in Listing 3. The first part is, in fact, a somewhat simplified version of the program in Listing 1. We have changed the specification of the `DivideArea` process: what we require now is that when `DivideArea` receives an area handle, it produces a series of area handles (one for each sub-area) on its standard output and then terminates.

The recursive step is now hidden into a separate manifold process, called `Distribute`. This program appears in Listing 3 and will be explained later. As far as the manifold `Warnock`[4] is concerned, `Distribute` receives the area handles for this level's sub-areas on its standard input and, somehow, takes care of the recursion. A separate pipeline is set up in the block labeled `subdivide` to send these handles to a local instance of `Distribute`. Note that now it is `Distribute` that is responsible for proper termination; consequently, the counter `n` has disappeared from `Warnock`.

As a commentary on MANIFOLD programming, note the difference between the two pipelines:

```
v -> divide_area, divide_area -> distribute
```

[4]By now "`Warnock`" is a misnomer for this program and "`Weiler_Atherton`" is probably a better name. However, we prefer to keep the name "`Warnock`" to preserve the similarity with the previous MANIFOLD program, for pedagogical reasons.

```
TestAndColor() atomic.
DivideArea()   atomic.
Distribute()   import.

Warnock()
{
    event   subdivide, done, finish.
    process test_and_color is TestAndColor.
    process v              is variable.
    process divide_area    is DivideArea.
    process distribute     is Distribute.

    start:
        (activate v,
         activate test_and_color,
         input -> (-> test_and_color,-> v),
         void
        ).
    subdivide:
        (activate divide_area,
         activate distribute,
         v -> divide_area,
         divide_area -> distribute
        );
        do finish.
    done:
        do finish.
    finish:
        deactivate parent.
}
```

Listing 2: Program with variable area subdivision; part I.

that appear as separate group members in the state `subdivide`, and the somewhat similar single pipeline:

```
v -> divide_area -> distribute
```

that may be mistaken as their equivalent. While the two alternatives work the same as long as the flow of units are concerned, they indeed behave quite differently on termination. In MANIFOLD, a pipeline breaks up as soon as any one of its processes terminates or raises a special event `break`. In case of our single pipeline, this can happen as soon as the process `v` has delivered its value, which can result in the breakup of the connection between `divide_area` and `distribute`, if they are all in the same pipeline. Having them in two separate pipelines in a group, as in the state `subdivide` in Listing 2, ensures that such premature breakups will not happen. (In MANIFOLD, a group terminates when all of its members are broken up.)

A number of constructs used in the original Warnock program (Listing 1) now appear in `Distribute` (see Listing 3). Using the counter `n` to count the number of activated child processes, as well as handling of their deactivations, are exactly the same as before. The primary difference is, of course, in the handling of a variable number of incoming units.

The `Distribute` manifold uses the built-in pseudo-process[5] `getunit` which acts as follows:

- it is suspended on a port of the 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);

[5]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.

```
Distribute()
{
    event   wait_for_death, main_cycle.
    port        in internal.
    process n   is variable.

    start:
        (activate n, n = 0); do main_cycle.
    main_cycle:
        {
            begin:
                getunit(input) -> internal; do next_area.
            next_area:
                (n = n + 1,getunit(internal) -> Warnock);
                do begin.
            terminate:
                save.
        }.
    disconnected_i.input:
    wait_for_death:
        void.
    terminate:
        n = n - 1;
        if( n = 0, do finish, do wait_for_death ).
    finish: .
}
```

Listing 3: Program with variable area subdivision; part II.

- if there is no unit available for delivery on the port *and* there is no external process connected to that port, `getunit` is not only suspended, but it also raises the `disconnected_i` event (with the selected port as the source of the event).

The `Distribute` manifold takes advantage of these features of `getunit`. In the inline manner labeled `main_cycle` (which contains, except for activation of the counter, the effective starting action of `Distribute`), a pipeline is set up using `getunit` with its output connected to another (externally non-visible) port of `Distribute`. The role of this pipeline is twofold:

1. When a unit arrives (actually, an area handle from the `DivideArea` process, although `Distribute` does not know the origin of the unit), it is picked and put into the `internal` port. Next, an internal state transition is made which results in the activation of a new instance of `Warnock`.
2. When there is no unit in the buffer of the `input` port of `Distribute`, *and* this port is no longer connected to any other port (which means that the connecting `DivideArea` process has terminated), `getunit` raises a `disconnected_i` event (which results in the preemption of the current state).

The rest is relatively clear: the unit stored in the `internal` port is picked by another instance of `getunit`, which passes it to an (implicitly activated) instance of `Warnock`, and the manifold returns to its waiting state in `main_cycle`.

It may not be immediately obvious why we use a separate state (`next_area`) to activate a new instance of `Warnock`. Indeed, merging the two states `main_cycle` and `next_area` is possible and also alleviates the need for the port `internal`, since we can use the pipeline

```
getunit(input) -> Warnock
```

in the block labeled `main_cycle`. However, the advantage of having two separate states instead of one is that we avoid an unnecessary activation of yet another instance of `Warnock` in each recursion. Using two distinct states, we can be sure that `Warnock` is activated if and only if there *is* another area handle in the `internal` port of `Distribute`.

```
TestAndColor() atomic.
DivideArea()   atomic.
Merge()        import.
Distribute()   import.

Warnock()
{
    event   subdivide, done, finish.
    process test_and_color  is TestAndColor.
    process v               is variable.
    process divide_area     is DivideArea.
    process distribute      is Distribute.
    port in  internal.

    start:
        (activate v,
         activate test_and_color,
         input -> (-> test_and_color -> ,-> v) -> internal,
         void
        ).
    subdivide:
        (activate divide_area,
         activate distribute,
         v           -> divide_area,
         divide_area -> distribute,
         distribute  -> output
        );
        do end.
    done:
        getunit(internal) -> output;
        do finish.
    finish:
        deactivate parent}.
}
```

Listing 4: Program with return values I.

3.3.2 Handling Return Values

The algorithms that can use the MANIFOLD programs in §3.1.1 and §3.3.1 are constrained by another limitation. Once the recursive branches of the algorithm start off, they do not communicate with their parents any more (or, to be precise, they have no communication expressed by the MANIFOLD program). This is fine (indeed, desirable) with the original Warnock's algorithm: the sub-areas of a screen can be filled independently of one another, and a parent has no reason to stay alive and take up resources once its children are started. However, this is obviously inappropriate in a number of other applications.

Once again, a slight improvement on Warnock's algorithm serves as a good motivating example. In §3.1.1 we assumed that the recursion stops when the size of an area reaches the size of a pixel. Strictly speaking, this assumption is true, but it results in aliasing problems (i.e., the appearance of "staircase" polygon edges and unpleasant color transitions). One of the anti-aliasing methods which can be easily used with Warnock's algorithm requires the recursion to go on at least one more step, to the level of sub-pixels. The color properties computed at sub-pixel levels are then returned to the pixel level routines, which in turn average them out to calculate the color of their pixels.

To use MANIFOLD for such an algorithm implies that (at least between the pixel and sub-pixel levels) each recursive branch must compute and return a value to its parent, and each parent must wait for the returned result of all of its children before it can complete its function and terminate. In this section, we modify our MANIFOLD programs to accommodate returned values.

Listings 4 and 5 show the new version of our MANIFOLD program; they correspond to the Listings 2

```
Distribute()
{
    event   wait_for_death, output_arrived, finish, main_cycle.
    port            in internal.
    process n       is variable.
    process merge   is Merger.
    nobreak merge.output -> self.output.

    start:
        (activate n,
         guard(self.output,output_arrived),
         n = 0
        );
        do main_cycle.
    main_cycle:
        {
            begin:
                getunit(input) -> internal; do next_area.
            next_area:
                (n = n + 1,getunit(internal) -> Permanent(Warnock,merge));
                do main_cycle.
            terminate:
                save.
        }.
    disconnected_i.input:
        (activate merge, do wait_for_death ).
    wait_for_death:
        void.
    terminate:
        n = n - 1;
        if( n = 0, do finish, do wait_for_death ).
    output_arrived:
        save.
    finish:
        {
            begin:
                void.
            output_arrived: .
        }.
}
```

Listing 5: Program with return values II.

and 3, respectively. As in the previous section, we only highlight the differences between the old and the new versions in this section.

The specification of the atomic process `TestAndColor` is now slightly different. Representing the "bottom" of the recursion, this atomic process is also required to return a value to be forwarded to the upper level (e.g., the color value, in the anti-aliasing example). Additionally, a new process, called **`Merge`**, is defined: this process receives "values" on its standard input port and "merges" them into one value delivered on its output port (in our anti-aliasing example, this process calculates the average of color values it receives)[6]. What `Merge` does is to read an unknown number of units from its standard input, compute their "merged" result (e.g., their average), write it out to its standard output, and terminate. It detects the equivalent of an end-of-file on its standard input (if it is in fact an atomic process), or reacts

[6]Note that in Listing 4, the declaration of `Merge` does *not* specify whether it is an atomic process or yet another manifold. It simply states that its declaration is contained in a separate MANIFOLD source file, and will be available at link time.

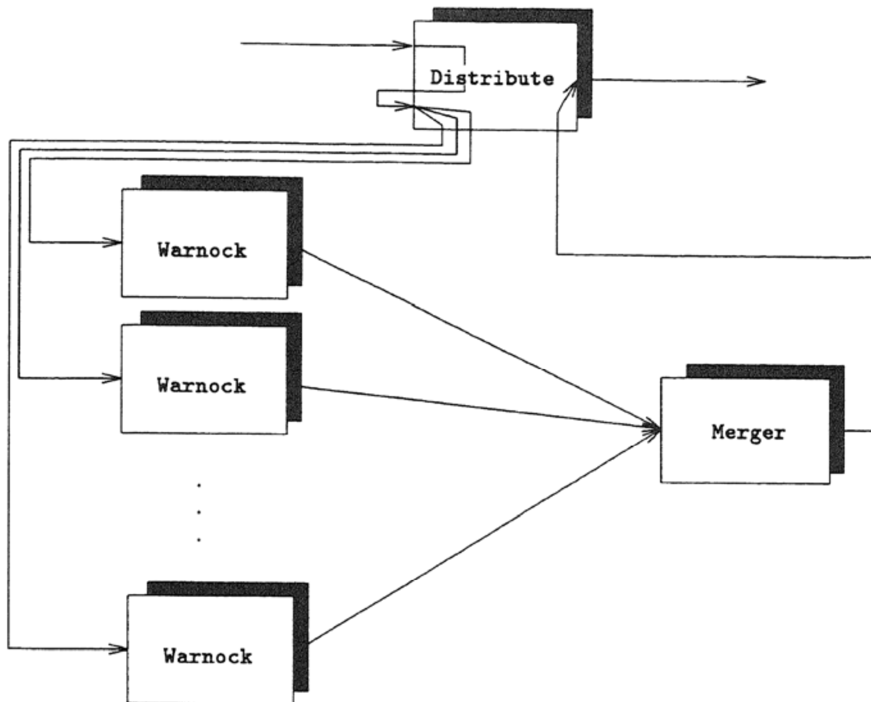

Figure 1: A pictorial representation of the manifold `Distribute`.

to a `disconnected_i` event (if it is another manifold), to realize that it has received all input units it is expected to process.

With these definitions in mind, the differences between the new and the old version of `Warnock` are not too difficult to understand. In the `start` block, the pipeline contains an additional item, which stores the output of `test_and_color` in a local port. Also, the new version of `Distribute` is expected to have an output, too, which is redirected to the output port of `Warnock`. Finally, the state labeled `done` is no longer only a state transition; it first reads the value produced previously by the bottom of the recursion and transfers it to the output port. Apart from these differences, the new version of the `Warnock` manifold has an identical behavior to the previous one.

The new version of `Distribute` uses a small manifold called `Permanent`, which might be part of the standard MANIFOLD environment. This manifold is as follows:

```
Permanent(middle,second)
process middle.
process second.
{
    start:
        input -> middle -> second.
}
```

The meaning of this manifold is clear: it sets up a pipeline which remains unbroken as long as members of the pipeline are active. Remember that, according to the specification of MANIFOLD, if a manifold leaves a state, all pipelines set up in that state are broken before leaving. The use of the `Permanent` manifold is to avoid this breakup.

`Distribute` now sets up a slightly more complicated network of connections. Figure 1 is a pictorial representation of these connections. At the initialization of `Distribute`, a permanent connection (using the `nobreak` declaration) is set up from the output port of `merge` (an instance of `Merge`) to the output port of the running instance of `Distribute`. Note that this is a perfectly legitimate setup: ports of a process

instance (e.g., `merge`) can be connected in pipelines even before the process is activated. Additionally, another pseudo-process, `guard`, is activated. The role of this pseudo-process is to raise an event (named in its argument) if a unit appears on its designated port.

The pipelines set up in the state `next_area` are slightly different: the connection between each new instance of `Warnock` and `merge` is set up using `Permanent` described above, to prevent its breakup in case of a state transition.

The two events `disconnected_i.input` and `wait_for_death` are now handled by two distinct states. The state labeled `wait_for_death` is the same as before: it is used to wait to receive the right number of `terminate` events before dying. The new state for `disconnected_i.input` activates `merge` and then makes a transition to `wait_for_death`.

There is a subtlety about `merge` that needs more explanation here. Our specification of `Merge` states that it receives an unknown number of input units, and detects the equivalent of an end-of-file to know they have been exhausted. Thus, we must make sure that at least all connections between `merge` and its suppliers are established before it is activated. This is why we connect all instances of `Warnock` to `merge` before arriving at `disconnected_i.input` where we activate it.

Before terminating, `Distribute` must not only wait for all of its local instances of `Warnock` to terminate, but it must also make sure that the output value of `merge` has actually arrived and is transferred out of its output port. This is done by the event `output_arrived` which is raised by `guard`. Note the use of a separate inline manner for the effective termination: occurrences of the event `output_arrived` must be saved, and they must not affect this manifold as long as the "finish" state is not reached. This is what is achieved by using the separate inline manner.

4 The Computing Farm Model

As it became more feasible to use a large amount of, possibly loosely coupled, very powerful processors in parallel to solve highly computing intensive applications, the notion of *computing farms* has 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[7]. 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, which means that a computing farm can be considered as a large-scale MIMD and very much 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 usable. In [19], 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 report, 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 adapted for such applications.

4.1 The Abstract Model

The original and simplest computing farm model, (as described, e.g., in [19]) 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 communicates them to the farms as required, and also collects the results as they are produced by the network.

The farm processor consists of two distinct parts, namely an application-specific part and a wrapper. Figure 2 gives an overview of the original structure.

[7]The term "processor farm" is also in use, in, e.g., [19]. We have chosen to use the terminology "computing farm" instead, to refer to a more general situation when not only "processors" are used.

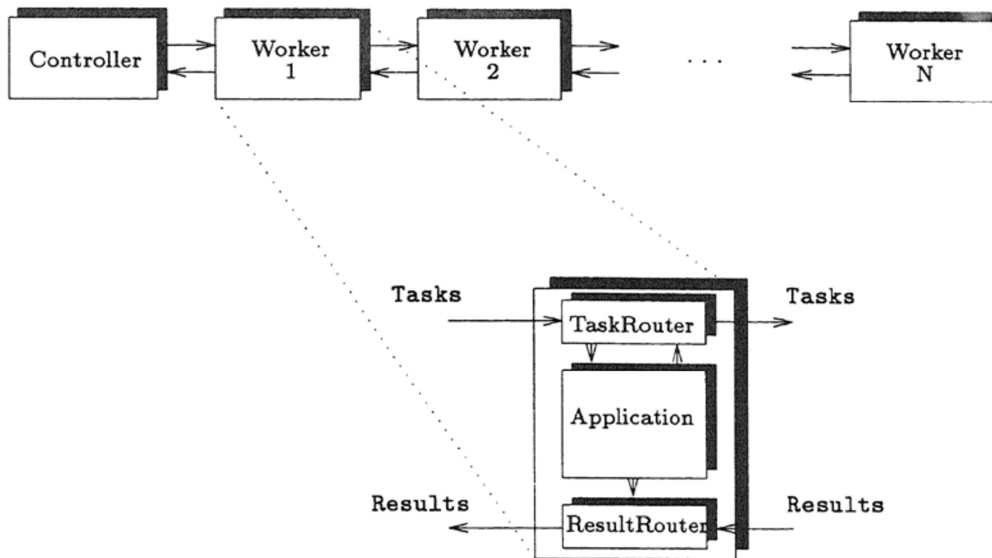

Figure 2: The Computing Farm Model

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. 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 (clearly inspired by the transputer and OCCAM world, which, in turn, have their roots in CSP, see [25, 26]) 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 as it is in MANIFOLD, it is also true that MANIFOLD gives the possibility to enhance a computing farm by adding some dynamism to it, namely:

- to allow a node to die, i.e., its encapsulated application program may abort, and
- to allow the controller to dynamically add a new node to the list.

Clearly, the addition of these new features increases the usability of the model for environments where the nodes are not necessary as reliable as they are supposed to be in a transputer network. A set of workstations hanging off a local area network obviously represents such a less reliable environment.

In this section, the MANIFOLD program for the computing farm is explained in more details. The complete listing of this MANIFOLD program appears in Appendix A. The overall structure of the program is the following.

- 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 again 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

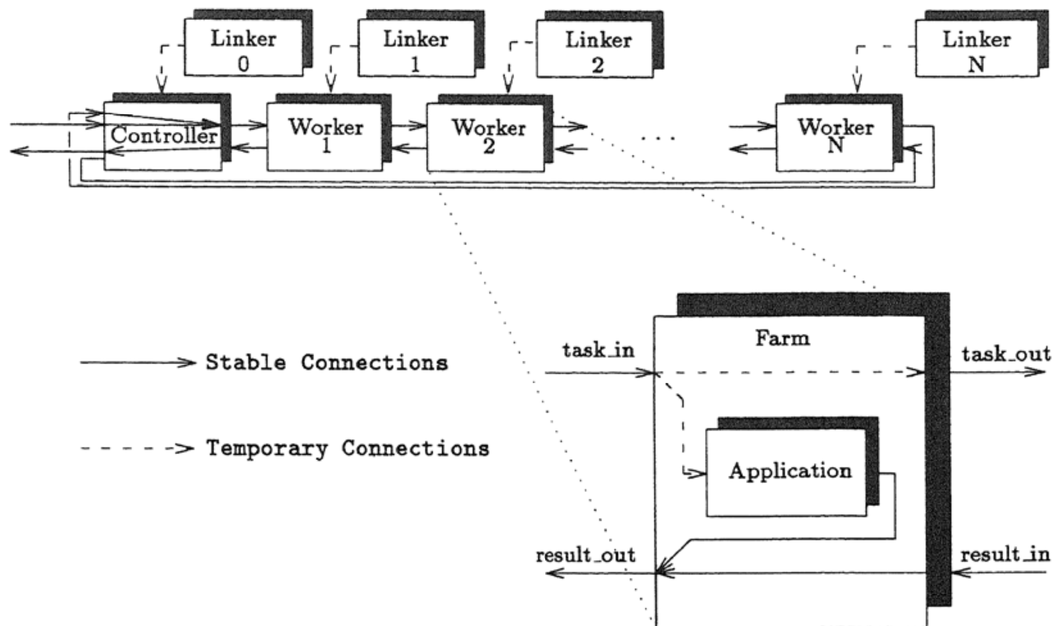

Figure 3: Computing Farm in MANIFOLD

"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 given in Figure 3. Note that there are some major differences between this version and the model presented in Figure 2:

- 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, whereas this becomes difficult in a CSP-oriented approach (which is the underlying model behind the original description).
- The farm nodes are linked into a ring. This is necessary to allow dynamic reconfiguration; more about that later.
- The connections between the adjacent nodes are "active", in the sense that they are controlled by yet another process. This is again necessary to allow dynamic reconfiguration.

In the following sections we first describe the "basic" behavior of all manifolds involved, i.e., their actions towards securing a constant information flow within the computing farm. In a separate section on deactivation (§4.6) the remaining parts of these manifolds will be described. The reason is that the deactivation of a node requires coordination and synchronization among a farm node and both its left and right linkers; it is better to describe this deactivation looking at all these manifolds jointly instead of describing their behaviors separately.

4.2 The Farm Nodes

A `Farm` node's basic functionalities are described in a (nested) inline manner labelled `normal_operation` (see page 30 or Listing 7). However, to get there, the manifold must go through several internal state transitions (see page 29 or Listing 6).

First, the application program (i.e., `Appl`) must be activated as well as an internal variable which stores a boolean constant describing the current state of the application. This latter is necessary for the correct deactivation of the manifold (see §4.6). Another (boolean) variable (`appl_requires`) is used to store whether or not the application has a pending request for work. Finally, yet another local variable,

```
Farm(n)
process  n.
port in  input "'procref'"
port in  task_in.
port out task_out.
port in  result_in.
port out result_out.
event    asks_for_work.
{
    process Appl           is Application[task_in|task_out]().
    process appl_state     is variable.
    process appl_requires  is variable.
    process m              is variable.
    port                   in internal.
    event                  goto_new_linker, new_linker, input_arrived.
    permanent              Appl, parent.

    start:
        (activate Appl(),
         activate appl_state(), true->appl_state,
         activate appl_requires(), false->appl_requires,
         activate m(), m = 0,
        );
        (guard(input,goto_new_linker),void).
    goto_new_linker:
        do new_linker.
    new_linker:
        {
            process   linker deref self.input.
            event     normal_operation.
            permanent linker.
            begin:
                guard(input,goto_new_linker);
                if(appl_state, do normal_operation, do end_of_all).
    .
    .
    .
```

Listing 6: Startup of the Farm Node

called m, is activated; this variable serves as a counter to count the task descriptions "cached" for the application program[8].

The previous set of operation ends with an idle state coupled with the activation of the pseudo-process guard set on the standard input of the manifold. The intention is to wait for the right linker to transmit its own process identifier to the farm node; this is necessary to make this linker visible as the source of events, which control the deactivation protocol (see §4.6). This process identifier is dereferenced within a subsequent inline manner.

The reason why a guard is used (instead of, e.g., a direct dereferencing at this point) is that the right linker may change at run-time if an adjacent farm nodes deactivates itself (see the section on deactivation, §4.6) in which case the value of the process reference linker must be renewed. In other words, a general mechanism must be used here to allow the manifold Farm to dereference this process reference again at a later stage. The effective dereferencing of the process reference is performed in another inline manner

[8]Note that in the original description in §4, this "cache" was only one level deep, whereas we permit an arbitrary depth here, controlled by the variable process size. This latter is an input argument to the manifold.

labelled new_linker.

At the beginning of this manner, it must be decided whether the application process is active or not. While this is obviously superfluous the first time, in view of what has been said before, the inline manner might be re-entered again in a later stage. On the other hand, it is also possible that the local farm node has already been entered its own deactivation procedure (see §4.6), that is the application process is no longer active (and hence the value of the variable appl_state may be false).

4.2.1 The Main Control Part

After all these transitions, the manifold is now in the inline manner labelled normal_operation (page 30 or Listing 7). This is the manner which controls the task descriptions' correct transmission 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 should read a task description on its task_in port (to be absolutely precise, it should be suspended on this port until the task description arrives). It is also supposed to produce its result unit on its output port called result_out.

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

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

which 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 (private) port of the farm node called internal is used as a cache. Finally, the (boolean) variable appl_requires notifies the manifold that the application has requested a new task description but the farm node was not able to provide one (i.e., the application is pending).

The farm node, within this manner, 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; it is the sub-manner labelled input_arrived which takes care of them. This manner consists of an if-elseif-else type construct which does the following:

- if the application is pending, the unit is read and forwarded to it at once; else
- if the internal cache (i.e., the internal port) is full, the unit is directly transferred to the next farm node; else
- the unit is put into the cache and the counter is incremented.

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

Symmetrically, if the application requires a new task description (see the sub-manner labelled by the corresponding event), the farm node looks at the number of "cached" units; if there is none, the boolean variable is set to "true", otherwise a unit is read and transferred to the application (and the counter is decremented).

The MANIFOLD code describing the state transitions is quite straightforward. Care should be taken at each step not to loose incoming events which would influence the value of the counter. It is to avoid such discrepancies that the two major actions are enclosed within inline manners. The only role of these manners is to allow a save for all interesting events. Note that these events are also saved within the "bottom" scope as well (i.e., on the manifold level), to safeguard the events even if a new linker appears (see page 31 in the Appendix).

The reason why an internal port is used (instead of simply piling up the units directly on the port Appl.task_in) is to perform deactivation properly. Indeed, if the application program deactivates itself, the still unused task descriptions must be made available to the next node. In order to do this properly, the farm node retains these units as long as possible. More about this later.

```
normal_operation:
    {
        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
                    ),
                    if( m > = size,
                        getunit(task_in) -> task_out,
                        (getunit(task_in) -> self.internal,
                         m = m + 1
                        )
                    )
                );
                (guard(task_in,input_arrived),do wait).
                input_arrived.task_in:
                    save.
                goto_new_linker.input: *.Appl:
                    save.
           }.

        asks_for_work.Appl:
           {
                begin:
                   if( m = 0,
                       true -> appl_requires,
                       (getunit(internal) -> Appl.task_in,
                        m = m - 1
                       )
                   );
                   do wait.

                input_arrived.task_in:
                    save.
                goto_new_linker.input: *.Appl:
                    save.
           }.
    }.
```

Listing 7: Central Part of the Farm Node

4.3 The Linker Process

The "normal" behavior of the manifold Link (see page 32 or Listing 8) is to set up the necessary streams between its left and right farm nodes. This is done in the begin block of an internal inline manner:

```
begin:
  &self             -> left.input;
  (left.task_out    -> right.task_in,
   right.result_out -> left.result_in,
   guard(input,go_to_startup)
  ).
```

The block includes a pipeline which transfers a single process reference unit. As said before (see §4.2), the farm node dereferences this unit in order to receive events coming from this link node. Setting up of this pipeline will have the side-effect of raising locally an event in the farm node, via the guard installed by this node (see §4.2). This is how a farm node is notified of a possible change in its linker node.

To get to this point, though, the manifold must dereference the process references it has to work on. At first glance, this seems to be an over-complication; one may think that using input arguments instead of dereferencing would solve the problem more properly. However, as a result of deactivation (see §4.6), the processes this manifold has to manage might change at run-time. To cope with this sort of dynamism, process references must be sent via ports and they must be explicitly dereferenced by `Link`. The mechanism is very similar to the one described for farm nodes; a `guard` is used to generate an event to notify the manifold of the arrival of a new set of process references.

The manifold `Link` will remain in the begin block of its internal inline manner as long as it is not preempted by events coming either from its left or its right farm node, or by an event generated by its guard, as described above. The farm node events are used to deactivate these nodes. The remainder of the manifold `Link` is exclusively devoted to the deactivation procedure, described in more detail in §4.6.

4.4 Start-up (the Controller)

The start-up of the whole computing farm is done by the manifold called `Controller` (see pages 33 and 34 or Listings 10 and 11). The keyword `export` means that this manifold can be accessed from outside the present MANIFOLD source file.

This manifold has all the input and output port specifications as the farm nodes (plus some more); this enables the system to set up a ring, rather than a linear list of nodes. Also, this means that, as far as the linker processes are concerned, there is no difference whatsoever between linking real farm nodes or the controller to the first farm node.

As shown in Figure 2, the last farm node sends the (unused) task descriptions back to the beginning by forwarding them to the `task_in` port of the `Controller`. The controller must then forward these task descriptions back into the ring. In this sense, the `task_in` port plays an active role. This is not the case for the port `result_out`; obviously, the controller should not re-send any result unit into the ring. This port is there for symmetry only, to allow the linker to set up the links properly.

The external world is accessed by the ports `external_task_in` and `external_result_out`. Strictly speaking, there is no need for the first one (the new task descriptions could just be sent onto the port `task_in`), but there *is* a need for the second one; therefore having both of these two ports is for the sake of symmetry.

What the controller has to do at first is to start up farm nodes and their corresponding linker processes in a cycle; this cycle is controlled by an input argument of the process `n`. The second argument, `size`, is just transferred to all individual farm nodes to control the size of their local caches. The essence of this generation is described in a separate manner, called `StartNextFarmNode` (see Listing 9).

The only non-obvious part of this manner is the following. Each new linker must have a reference to its left and right nodes as well as to its previous linker process. Consequently, the process references for the "previous" farm node and for the previous linker should be available when a new pair is generated. To achieve this, dereferencing is used: at step i, the process references of the (just generated) $\texttt{Farm}_i$ and $\texttt{Link}_i$ are put into a local port `internal`. Then, at step $i+1$, when $\texttt{Farm}_{i+1}$ and $\texttt{Link}_{i+1}$ are generated, this internal port is used to dereference the local process references for the manner. This procedure is

```
Link()
port in input "`procref'".
{
    permanent parent.
    event     startup,go_to_startup.
    start:
        (guard(input,go_to_startup),void).
    go_to_startup:
        do startup.
    startup:
        {
            process left[result_in|task_out]()  deref input.
            process right[result_in|task_out]() deref input.
            process previous_link               deref input.
            begin:
                &self            -> left.input;
               (left.task_out    -> right.task_in,
                right.result_out -> left.result_in,
                guard(input,go_to_startup)
               ).
            wish_to_die.right:
               {
                   nobreak right.result_out -> left.result_in.
                   begin:
                      &left->output; &previous_link->output; void.
                   wish_to_die.left:
                      ignore.
               }.
            wish_to_die.left:
               {
                   process new_left[result_in|task_out]()
                                       deref previous_link.output.
                   process over_previous_link()
                                       deref previous_link.output.
                   nobreak left.task_out -> right.task_in.
                   begin:
                      (guard(input,noevent),raise empty_buffers,left).
                   buffers_emptied.left:
                      (deactivate previous_link,
                       deactivate left);
                       <&new_left,&right,&over_previous_link> -> self.input
                      );
                      do start.
                   go_to_startup.input:
                      ignore.
               }.
           go_to_startup.input:
               do start.
        }.
    terminate.*: .
}
```

Listing 8: The Linker Process

```
manner StartNextFarmNode()
{
    process next_farm     is     Farm().
    process next_linker   is     LinkAnimals();
    process previous_farm[task_in,result_in|task_out,result_out]()
                                    deref internal.
    process previous_linker()       deref internal.
    start:
       (activate next_farm(size),activate next_linker(),
        <&next_farm,&next_linker>                   -> self.internal,
        <&previous_farm,&next_farm,&previous_linker> -> next_linker.input
       ).
}
```

Listing 9: Startup of a new Farm Node and its Associated Linker

```
export Controller(n,size)
process n,size.
port in  input "'procref'".
port in  task_in.
port out task_out.
port in  result_in.
port out result_out.
port in  external_task_in.
port out external_result_out.
{
    process m           is  variable().
    process first_farm  is  Farm().
    process first_link  is  Link().
    port    in  internal "'procref'".
    event                   generation_cycle,next_one,
                            end_of_generation.
    start:
        (activate m(),
         m=1,
         activate first_farm(size),activate first_link(),
         <&self,&first_farm,&self> -> first_link.input,
         <&first_farm,&first_link> -> self.internal
        );
        do generation_cycle.
    generation_cycle:
        if(m = n, do end_of_generation, do next_one).
    next_one:
        StartNextFarmNode(); m = m + 1; do generation_cycle.
```

Listing 10: Startup of the Controller

initialized from within the controller manifold: the very first farm node and the first linker are activated from within its start block and their references are put onto the port `internal` explicitly.

Note that the newly generated linker process is initialized by sending a triplet of process reference units to its input port (the matchfix operator `<...,...>` is a shorthand for an atomic process which "glues" the units into one). The regular expression assigned to the input port of the receiver (i.e., `'procref'`) will take care of "cutting" the unit into three separate process reference units.

```
end_of_generation:
    {
        event  main_block.
        nobreak ((external_task_in->,task_in->) -> task_out,
                 result_in -> external_result_out
                ).
        begin:
            LinkEndNode();
            &first_link -> self.internal;
            do main_block.
        main_block:
            void.
        start_new_one.*
            {
                process first_link() deref input.
                process first_farm[task_in,result_in|task_out,result_out]()
                                deref internal.
                process new_farm         is        Farm().
                process new_linker       is        Link();
                begin:
                    (activate   new_farm(size),activate new_linker(),
                     <&self,&new_farm,&self>             ->new_linker.input,
                     <&new_farm,&first_farm,&new_linker>->first_link.input,
                     &new_farm                           -> self.internal
                    );
                    do main_block.
                start_new_one.*:
                    save.
            }.
    }.
```

Listing 11: Central Part of the Controller

Coming back to the main body of the controller manifold, the generation of all the internal farm nodes is done in a cycle:

```
generation_cycle:
    if(m = n,do end_of_generation,do next_one).
next_one:
    StartNextFarmNode(); m = m + 1; do generation_cycle.
```

Each of these generation steps starts up $\texttt{Farm}_i$ and $\texttt{Link}_i$.

The heart of the controller manifold action is in the (inline) manner labelled `end_of_generation` (see page 34 or Listing 11). It sets up a permanent pipeline which connects the ring to the external world; it also completes the generation of the full ring, using a separate manner called `LinkEndNode` (a simplified version of `StartNextFarmNode`) which generates the last farm node and links it to the controller itself. The manifold then goes into the `main_block`, which simply puts the controller into idle an state (i.e., only transfer of units takes place). The only events which can possibly preempt the manifold from now on are those that signal either the activation of a new node in the ring (see §4.5), or the termination of the whole computing farm. Both of these actions are described in separate sections below.

4.5 New Activation

Once all nodes are set up, the only real action the controller process performs (besides termination) 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.

These actions are all performed in the inline manner labelled `start_new_one` within the controller.

In order to properly activate and initialize a new linker (which links the controller to a newly generated farm node) and to re-initialize the former first linker node, the controller needs the process references of the old first link as well as a reference to the old first farm node. The reference to the old first link is readily available. Indeed, remember that the first action a linker does is to put its own process reference to the input port of its "left" node which is, in this case, the controller itself. By dereferencing this port, the controller can access this process reference without problems.

In order to access the old first node, a method used before is re-used here. First of all, at the end of startup generation procedure, `Controller` puts the process reference of the very first farm node onto the internal port. This properly initializes the usual procedure: every time a new farm node is generated, the old farm node's reference is acquired by dereferencing this port. Also, the reference to the newly generated process is put back onto the internal port to make it available for the next round.

In view of these, the code to start up a new node is straightforward (see page 34 or Listing 11). The controller (re-)initializes the linker processes: the triplets containing references to the left and right farm nodes as well as to the previous linker process are put onto the input port of the respective linker processes. The guards installed on these ports will force the linkers to (re-)initialize themselves. Then, by performing an explicit state transition to outside the inline manner, the controller process is ready to re-enter the generation manner again (on request) and dereference the process references anew.

4.6 Deactivation

Deactivation of a farm node is the most complex operation in the computing farm. 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.

There are basically two actions which are performed when a farm node intends to deactivate:

- All units residing on the farm node must reenter the ring properly. This involves the possibly cached task descriptions and the not-yet-forwarded result units.
- The node must be deleted from the ring and new 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 (see §4.4), this latter has no permanent record of the references to the farm nodes and the linker nodes. Consequently, deactivation must be done locally and it involves the left and right linker of the farm node to be deactivated. In other words, there is no "central authority" to take care of deactivation[9].

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 $\texttt{Farm}_i$ and will preempt its normal operation by causing a transition to a special (inline) manner (see page 31 or Listing 12).

The first action in this manner is to change the value of the boolean variable `appl_active`. Note that this is done in (yet another) inline manner because setting this variable should not, by any means, be preempted (the events `connected_i.task_in` and `goto_new_linker.input` are the events that may arrive at this point, signalling a change in the topology of the ring; see later).

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

```
death.Appl:end_of_all:
    {
        event  raise_wish_to_die.
        begin:
           {
                begin:
                    false->appl_active.
                goto_new_linker.input: connected_i.task_in:
                     save.
           }; do raise_wish_to_die.
        raise_wish_to_die:
           (raise wish_to_die,linker).
        empty_buffers.linker:
            Empty_Buffer(internal,task_out);
            Empty_Buffer(result_in,result_out);
            Empty_buffer(task_in,task_out);
            (raise buffers_emptied,void).
        connected_i.task_in:
            do raise_wish_to_die.
        input_received.*:
            ignore.
    }.
```

Listing 12: Deactivation of a Farm Node

Once this is done, a special event is raised (wish_to_die) and then $\texttt{Farm}_i$ waits until the right linker gives the authorization to proceed.

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

The role of the two linker nodes are quite asymmetric at this point. We can say that $\texttt{Link}_i$ "controls" the deactivation process, whereas $\texttt{Link}_{i-1}$ plays a somewhat more passive role. What $\texttt{Link}_i$ has to do is as follows:

1. it must control $\texttt{Farm}_i$ in emptying its buffers;
2. it must re-initialize itself to properly link $\texttt{Farm}_{i-1}$ and $\texttt{Farm}_{i+1}$;
3. it must deactivate both $\texttt{Link}_{i-1}$ and $\texttt{Farm}_i$.

In order to perform step 2, $\texttt{Link}_i$ must have a reference to $\texttt{Farm}_{i-1}$, which it does not have by default. $\texttt{Link}_{i-1}$ does have this reference (its "left" node); consequently, this information must be passed from $\texttt{Link}_{i-1}$ to $\texttt{Link}_i$. An obvious way to do that is to use the dereferencing mechanism again. $\texttt{Link}_i$ dereferences a process reference on a port of $\texttt{Link}_{i-1}$. To do that, $\texttt{Link}_i$ must have a reference to $\texttt{Link}_{i-1}$; this is why this reference is passed as part of the initializing triple to the manifold Link.

In view of these facts, the reaction of $\texttt{Link}_{i-1}$ and $\texttt{Link}_i$, respectively, is as follows. $\texttt{Link}_{i-1}$ re-sets an output connection for $\texttt{Farm}_i$ to allow this latter manifold to forward the still remaining results. It also puts two process references onto its output port, namely the reference for $\texttt{Farm}_{i-1}$ and the reference to $\texttt{Link}_{i-2}$. Finally, it goes into an idle state; essentially, it waits for its deactivation which will be performed by $\texttt{Link}_i$.

$\texttt{Link}_i$ must also set up an output connection for $\texttt{Farm}_i$, to allow this latter to forward its remaining task descriptions. It must dereference some local process references (originating from $\texttt{Link}_{i-1}$), to eventually re-initialize itself. Then, $\texttt{Link}_i$ raises an event (empty_buffers) which will be reacted to by the left farm

node (i.e., the one which wishes to deactivate). This farm node must then empty all its internal ports and signal that by raising another event, `buffers_emptied`.

At this point, $\texttt{Link}_i$ is free to deactivate all processes and re-initialize itself. Deactivation is simply done by using the `deactivate` primitive action. Re-initialization is done by putting new process references into its own input port and by an explicit state transition to another inline manner. This last state transition forces the linker to dereference the values of "left" and "right" again and sets up the new streams connecting $\texttt{Farm}_{i-1}$ to $\texttt{Farm}_{i+1}$.

It may seem disturbing that the possibility of external re-initialization of $\texttt{Link}_i$ is "switched off" at this point (by setting `noevent` as a guard event and by explicitly ignoring the possible arrival of the `go_to_startup` event, see page 32 or Listing 8). However, external re-initialization of a linker process is performed exclusively when the linker process happens to be the first linker, linking the controller process to the first farm node. Obviously, if this were the case, $\texttt{Link}_i$ could not have received a `wish_to_die` event from its left node. Consequently, it would not be in the state described above.

Finally, we come back to the behavior of $\texttt{Farm}_i$. Being in an idle state, it waits for the event `empty_buffers` to arrive (coming from $\texttt{Link}_i$). It then empties its internal port (i.e., the "cache"), its `task_in` port which may still contain some not-yet-forwarded task descriptions, and its `result_in` port which may also contain some units.

Note that the event `empty_buffers` is raised by $\texttt{Link}_i$ when both $\texttt{Link}_{i-1}$ and $\texttt{Link}_i$ have preempted their "normal" mode of operations. In other words, no incoming streams are connected to $\texttt{Farm}_i$ any more. Consequently, $\texttt{Farm}_i$ can simply rely on the `disconnected_i` event for emptying its ports.

Once its ports are empty, $\texttt{Farm}_i$ can be deactivated and so can $\texttt{Link}_{i-1}$. This is signalled by raising the event `buffers_emptied` which triggers the final deactivation as described above.

4.6.1 Concurrent Deactivation

There is one more issue which must be addressed here. It is theoretically possible that when $\texttt{Farm}_i$ is engaged in its deactivation (i.e., it raises the event `wish_to_die`) either $\texttt{Link}_{i-1}$ or $\texttt{Link}_i$ is already entered into a separate deactivation protocol triggered by either $\texttt{Farm}_{i-1}$ or $\texttt{Farm}_{i+1}$ respectively. We must therefore check to see whether the protocol described above arrives to its termination in these cases as well.

As the first case let us suppose that $\texttt{Farm}_{i-1}$ has also started a deactivation process and, consequently, $\texttt{Link}_{i-1}$ is in the state labelled `wish_to_die.left`. This results in the suspension of $\texttt{Link}_i$ (it will not receive the process references to be dereferenced, which means a suspension). Eventually, $\texttt{Link}_{i-1}$ will terminate its deactivation process and will therefore reconnect $\texttt{Farm}_{i-2}$ to $\texttt{Farm}_i$.

Setting up such a connection will induce the `connected_i.task_in` event in $\texttt{Farm}_i$. Consequently, what this latter must do is to *repeat* the event `wish_to_die`. This forces $\texttt{Link}_{i-1}$ to react on the deactivation of $\texttt{Farm}_i$ and proceed as described above. Note that $\texttt{Link}_i$ will not be affected by this repeated raising of the event: being in the state triggered by this very event, the second occurrence of this event is automatically ignored. The repetition of the event is secured by the block

```
connected_i.task_in:
    do raise_wish_to_die.
```

i.e., $\texttt{Farm}_i$ will go "back" to an earlier state.

Things get a bit more complicated if $\texttt{Link}_i$ is the one which is already engaged in a deactivation procedure (triggered by $\texttt{Farm}_{i+1}$). Indeed, $\texttt{Link}_i$ will be *deactivated* after a while and $\texttt{Link}_{i+1}$ will take its place. $\texttt{Farm}_i$ must then renew its process reference value for "linker", otherwise it will not be able to receive events coming from $\texttt{Link}_{i+1}$.

The re-initialization of $\texttt{Link}_{i+1}$ will result in its process reference to appear on the input port of $\texttt{Farm}_i$. Within this latter manifold, the guard on the input port is still active. The arrival of this new process reference will preempt $\texttt{Farm}_i$ which will go *out* of its current scope and reenter the inline manner again to dereference a new value for "linker". Furthermore, the local variable `appl_active` will produce the value of `false` at the very beginning of this inline manner. Consequently, $\texttt{Farm}_i$ will repeat the event `wish_to_die` again with the new value of the linker. Here again, $\texttt{Link}_{i-1}$ will not be affected by the occurrence of this event.

4.7 Termination

The termination of the whole computing farm is done by the deactivation of the `Controller` process. Deactivation of this process results in raising the system-event `terminate`. The corresponding block raises this event again, deactivates the locally activated variables and halts.

The net effect of raising the `terminate` event in the `Controller` is that all instances of `Farm` as well as of the manifold `Link` will be able to react on this event. Indeed, the `Controller` process is the `parent` of all these processes. Consequently, all events raised by `Controller` are potentially visible for all farm nodes and the linker processes. All that needs to be done is to declare `parent` to be `permanent` in all these manifolds.

The reaction on the `terminate` event is straightforward: after deactivating all locally activated processes (i.e., variables) the process should die. This is done in the corresponding blocks of `Farm` and `Link` (see the Appendix).

5 References

[1] F. Arbab, "Specification of Manifold," Tech. Rep. to appear, Centrum voor Wiskunde en Informatica, Amsterdam, 1992.

[2] F. Arbab and I. Herman, "Manifold: A language for specification of inter-process communication," in *Proceedings of the EurOpen Autumn Conference* (A. Finlay, ed.), (Budapest), pp. 127–144, September 1991.

[3] F. Arbab, I. Herman, and P. Spilling, "An overview of Manifold and its implementation," Tech. Rep. CS-R9142, Centrum voor Wiskunde en Informatica, Amsterdam, 1991.

[4] F. Arbab, I. Herman, and P. Spilling, "An overview of Manifold and its implementation," *Concurrency, Practice and Experience*, to appear.

[5] E. Rutten, F. Arbab, and I. Herman, "Formal specification of Manifold: a preliminary study," Tech. Rep. to appear, Centrum voor Wiskunde en Informatica, Amsterdam, 1992.

[6] F. Arbab and I. Herman, "Examples in Manifold," Tech. Rep. CS-R9066, Centrum voor Wiskunde en Informatica, Amsterdam, 1990.

[7] D. Soede, F. Arbab, I. Herman, and P. ten Hagen, "The GKS input model in manifold," *Computer Graphics Forum*, vol. 10, pp. 209–224, September 1991.

[8] F. Arbab, I. Herman, and P. Spilling, "Interaction management of a window manager in Manifold," in *Proceedings of the ICCI'92 Conference* (W. Koczkodaj, ed.), (Toronto), IEEE, June 1992.

[9] D. Duce, R. van Liere, and P. ten Hagen, "Components, frameworks and GKS input," in *Eurographics'89 Conference Proceedings* (W. Hansmann, F. Hopgood, and W. Straßer, eds.), (Amsterdam), pp. 87–106, North Holland, 1989.

[10] D. Duce, R. van Liere, and P. ten Hagen, "An approach to hierarchical input devices," *Computer Graphics Forum*, vol. 9, pp. 15–26, 1990.

[11] G. Faconti, M. Caneve, E. Salvatore, and N. Zani, "A LOTOS view of the input model of standard graphics systems," in *Formal Methods in Computer Graphics* (D. Duce and G. Faconti, eds.), Heidelberg: EurographicSeminar Series, Springer Verlag, 1992.

[12] G. Faconti, F. Paternò, and N. Zani, "Towards the improvement of the input model of graphics systems," tech. rep., CNR-CNUCE, Pisa, 1991.

[13] I. Herman, F. Arbab, and F. Burger, "The manifold stack machine," Tech. Rep. to appear, Centrum voor Wiskunde en Informatica, Amsterdam, 1992.

[14] J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics — Principles and Practice.* Reading: Adison–Wesley, 1990.

[15] R. Gonzalez and P. Wintz, *Digital Image Processing.* Reading, Massachusetts: Addison–Wesley, 1983.

[16] W. Bronsvoort, F. Jansen, and F. Post, "Design and display of solid models," in *Advances in Computer Graphics VI* (G. Garcia and I. Herman, eds.), Heidelberg: EurographicSeminar Series, Springer Verlag, 1991.

[17] R. Bartels, J. Beatty, and B. Barsky, *An Introduction to Splines for Use in Computer Graphics & Geometric Modelling.* Los Altos, California: Morgan Kaufmann Publishers, Inc., 1987.

[18] M. Cohen and J. Painter, "State of the art in image synthesis," in *Advances in Computer Graphics VI* (G. Garcia and I. Herman, eds.), Heidelberg: EurographicSeminar Series, Springer Verlag, 1991.

[19] S. Green, *Parallel Processing for Computer Graphics.* Research Monographs in Parallel and Distributed Computing, London: Pitman, 1991.

[20] F. Crow, "Parallel computing for graphics," in *Advances in Computer Graphics VI* (G. Garcia and I. Herman, eds.), Heidelberg: EurographicSeminar Series, Springer Verlag, 1991.

[21] H. Siegel, J. Armstrong, and D. Watson, "Mapping computer-vision related tasks onto reconfigurable parallel processing systems," *IEEE Computer*, vol. 25, pp. 54–64, February 1992.

[22] D. Laur and P. Hanrahan, "Hierarchical splatting: Progressive refinement algorithm for volume rendering," *Computer Graphics (SIGGRAPH'91)*, vol. 25, pp. 285–288, July 1991.

[23] A. Hansen and F. Arbab, "Fixed-axis tool positioning with built-in global inteference checking for NC path generation," *IEEE Journal of Robotics and Automation*, vol. 4, December 1988.

[24] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Communication of the ACM*, vol. 35, pp. 97–107, February 1992.

[25] C. Hoare, *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science, New Jersey: Prentice-Hall, 1985.

[26] INMOS Ltd., *OCCAM 2, Reference Manual.* Series in Computer Science, London — Sydney — Toronto — New Delhi — Tokyo: Prentice-Hall, 1988.

A The Full Code for the Processing Farm

```
Application()
port in  task_in.
port out result_out.
event    asks_for_work.
import.

event wish_to_die.
event empty_buffers.
event buffers_emptied.

//------------------------------------------------------------------------
Empty_Buffer(inp,outp)
port in  inp.
port out outp.
{
    event next_one.
    begin:
        getunit(inp) -> outp;
        do next_one.
    next_one:
        do begin.
    disconnected_i.inp:
        return.
}

Farm(n)
process  n.
port in  input "'procref'"
port in  task_in.
port out task_out.
port in  result_in.
port out result_out.
event    asks_for_work.
{
    process Appl           is Application[task_in|task_out]().
    process appl_state     is variable.
    process appl_requires  is variable.
    process m              is variable.
    port                   in internal.
    event                  goto_new_linker, new_linker, input_arrived.
    permanent              Appl, parent.

    start:
        (activate Appl(),
         activate appl_state(), true->appl_state,
         activate appl_requires(), false->appl_requires,
         activate m(), m = 0,
        );
        (guard(input,goto_new_linker),void).
    goto_new_linker:
        do new_linker.
```

```
new_linker:
    {
        process   linker deref self.input.
        event     normal_operation.
        permanent linker.
        begin:
            guard(input,goto_new_linker);
            if(appl_state, do normal_operation, do end_of_all).
        normal_operation:
            {
                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
                            ),
                            if( m > = size,
                                getunit(task_in) -> task_out,
                                (getunit(task_in) -> self.internal,
                                 m = m + 1
                                )
                            )
                        );
                        (guard(task_in,input_arrived),do wait).
                        input_arrived.task_in:
                            save.
                        goto_new_linker.input: *.Appl:
                            save.
                   }.

                asks_for_work.Appl:
                   {
                        begin:
                           if( m = 0,
                               true -> appl_requires,
                               (getunit(internal) -> Appl.task_in,
                                m = m - 1
                               )
                           );
                           do wait.

                        input_arrived.task_in:
                            save.
                        goto_new_linker.input: *.Appl:
                            save.
                   }.
            }.
```

```
            death.Appl:end_of_all:
                {
                    event  raise_wish_to_die.
                    begin:
                       {
                            begin:
                                false->appl_active.
                            goto_new_linker.input: connected_i.task_in:
                                 save.
                       }; do raise_wish_to_die.
                    raise_wish_to_die:
                       (raise wish_to_die,linker).
                    empty_buffers.linker:
                        Empty_Buffer(internal,task_out);
                        Empty_Buffer(result_in,result_out);
                        Empty_buffer(task_in,task_out);
                        (raise buffers_emptied,void).
                    connected_i.task_in:
                        do raise_wish_to_die.
                    input_received.*:
                        ignore.
                }.
        }.
    *.Appl: task_in.input_arrived:
        save.
    terminate.*:
        (deactivate Appl,
         deactivate appl_state,
         deactivate appl_requires,
         deactivate m
        ).
}
```

```
  ()
 t in input  pro ef'

    permanent parent.
    event     startup,go_to_startup.
    start:
        (guard(input,go_to_startup),void).
    go_to_startup:
        do startup.
    startup:
        {
            process left[result_in|task_out]()  deref input.
            process right[result_in|task_out]() deref input.
            process previous_link               deref input.
            begin:
                &self            -> left.input;
               (left.task_out    -> right.task_in,
                right.result_out -> left.result_in,
                guard(input,go_to_startup)
               ).
            wish_to_die.right:
               {
                   nobreak right.result_out -> left.result_in.
                   begin:
                      &left->output; &previous_link->output; void.
                   wish_to_die.left:
                      ignore.
               }.
            wish_to_die.left:
               {
                   process new_left[result_in|task_out]()
                                         deref previous_link.output.
                   process over_previous_link()
                                         deref previous_link.output.
                   nobreak left.task_out -> right.task_in.
                   begin:
                      (guard(input,noevent),raise empty_buffers,left).
                   buffers_emptied.left:
                      (deactivate previous_link,
                       deactivate left);
                       <&new_left,&right,&over_previous_link> -> self.input
                      );
                      do start.
                   go_to_startup.input:
                      ignore.
               }.
           go_to_startup.input:
               do start.
        }.
    terminate.*: .
}
```

```
extern event    start_new_one.  // This event is used to generate a new node

manner StartNextFarmNode()
{
    process next_farm     is     Farm().
    process next_linker   is     LinkAnimals();
    process previous_farm[task_in,result_in|task_out,result_out]()
                                    deref internal.
    process previous_linker()       deref internal.
    start:
       (activate next_farm(size),activate next_linker(),
        <&next_farm,&next_linker>                      -> self.internal,
        <&previous_farm,&next_farm,&previous_linker> -> next_linker.input
       ).
}

manner LinkEndNode()
{
    process next_linker   is      LinkAnimals();
    process previous_farm[task_in,result_in|task_out,result_out]()
                                    deref internal.
    process previous_linker()       deref internal.
    start:
       (activate next_farm(size),activate next_linker(),
        <&previous_farm,&self,&previous_linker> -> next_linker.input
       ).
}

export Controller(n,size)
process n,size.
port in  input "'procref'".
port in  task_in.
port out task_out.
port in  result_in.
port out result_out.
port in  external_task_in.
port out external_result_out.
{
    process m           is  variable().
    process first_farm  is  Farm().
    process first_link  is  Link().
    port    in  internal "'procref'".
    event                   generation_cycle,next_one,
                            end_of_generation.
    start:
        (activate m(),
         m=1,
         activate first_farm(size),activate first_link(),
         <&self,&first_farm,&self> -> first_link.input,
         <&first_farm,&first_link> -> self.internal
        );
        do generation_cycle.
    generation_cycle:
        if(m = n, do end_of_generation, do next_one).
    next_one:
        StartNextFarmNode(); m = m + 1; do generation_cycle.
```

```
    end_ [illegible] atio
        {
            event  main_block.
            nobreak ((external_task_in->,task_in->) -> task_out,
                     result_in -> external_result_out
                    ).
            begin:
                LinkEndNode();
                &first_link -> self.internal;
                do main_block.
            main_block:
                void.
            start_new_one.*
                {
                    process first_link() deref input.
                    process first_farm[task_in,result_in|task_out,result_out]()
                                    deref internal.
                    process new_farm        is         Farm().
                    process new_linker      is         Link();
                    begin:
                        (activate   new_farm(size),activate new_linker(),
                         <&self,&new_farm,&self>            ->new_linker.input,
                         <&new_farm,&first_farm,&new_linker>->first_link.input,
                         &new_farm                          -> self.internal
                        );
                        do main_block.
                    start_new_one.*:
                        save.
                }.
        }.
    start_new_one.*:
        save.
    terminate:
        (deactivate m,raise terminate).
}
```