CWI

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

F. Arbab, I. Herman

Examples in Manifold

Computer Science/Department of Interactive Systems Report CS-R9066 November

Examples in MANIFOLD

Farhad Arbab, Ivan Herman

Interactive Systems
Centre for Mathematics and Computer Science
Kruislaan 413, 1098 SJ Amsterdam
The Netherlands
Telephone: +31 20 5924056, +31 20 5924164
Email: farhad@cwi.nl, ivan@cwi.nl

ABSTRACT

This document gives an insight into the use of the MANIFOLD system by presenting a few short examples. The overall description and the more formal syntax and semantics of MANIFOLD are given in separate documents.

1987 CR Categories: C.1.2, C.1.3, C.2.m, D.1.3, D.3.2, F.1.2, I.1.3.
1985 Mathematical Subject Classification: 68N99, 68Q10.
Keywords and Phrases: parallel computing, MIMD, models of computation.

Table of Contents

1. Introduction 1
2. What is it Like? 1
3. The **MANIFOLD** Model of Computation 2
4. Examples 3
    4.1. Simple copy 3
    4.2. Action Synchronization 5
    4.3. Counter 5
    4.4. Control structures 7
    4.5. The repeater 9
    4.6. A Memory Cell 10
    4.7. A Resource Management Example 11
        4.7.1. The Static Case 11
        4.7.2. The Dynamic Case 17
    4.8. Bucket Sorting 18
5. Other Applications 23
6. Conclusion 23
7. Acknowledgments 24

1. Introduction

The examples given in the present report are meant to be a ''didactic'' help for using MANIFOLD. Some of the examples have a practical importance as well; as such, they may become part of the set of builtin processes in a MANIFOLD implementation. This is notably the case for the manifolds **pass** and **pass1** (both described in §4.1), **count** and **count1** (both described in §4.3), and the two versions of the **if** manner (see §4.4).

No detailed description of the MANIFOLD syntax and semantics is given in the present document, apart from some general outline of the underlying model of computation. The reader should refer to[1] for a more detailed description of the model and the general concerns leading to the specification of MANIFOLD and to the more complete MANIFOLD description[2], for the detailed presentation of the MANIFOLD semantics and syntax.

2. What is it Like?

The Webster's dictionary defines the term *manifold* as an adjective to mean:

> 1. having many forms, parts, etc. 2. of many sorts 3. being such in many ways 4. operating several parts of one kind.

It also defines *manifold* as a noun to mean:

> a pipe with several outlets, as for conducting cylinder exhaust from an engine.

MANIFOLD can be viewed from several different perspectives, each revealing similarities with the features and concerns of a different set of models and systems. However, it is useful to establish a few approximate reference points to inspire an intuitive feeling for what MANIFOLD is all about before encountering the details.

To the extent that the primary focus in MANIFOLD is the connections among processes, not the processes themselves, it is a *conductor* that orchestrates the interactions among a set of cooperating concurrent processes, without interfering with their internal operations. As such, MANIFOLD programming is vaguely reminiscent of writing shell scripts in a system like UNIX™. Similar to a shell script, the concurrency and interconnection issues are completely outside of the processes. However, the possibilities for defining and dynamically changing the interconnections among processes in MANIFOLD go much beyond what is offered in such simple shell scripts.

Orchestration of the interactions among a set of processes in MANIFOLD is done in an entity with multiple inlets and outlets, called a *manifold.* As the conductor of such interactions, a manifold has a number of *states*, each specifying a specific connection pattern. Connection patterns define links between the input and output ports of various processes, called *streams*, through which the information produced by one process is made available for consumption to another.

A manifold goes through state transitions as a result of observing in its environment the occurrences

™UNIX is a trademark of AT&T Bell Laboratories

Report CS-R9066
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

of *events* in which it is interested. State transitions cause dismantling of the interconnections set up in pre-transition states, and establish the ones defined in the post–transition states. As such, events are the principal control mechanism in MANIFOLD, which makes it an event driven programming system.

The streams among processes in MANIFOLD form a network of links for the flow of information that is reminiscent of dataflow networks. However, there are several major differences between MANIFOLD and dataflow programming. In MANIFOLD the connection patterns among processes change dynamically. Furthermore, processes are created and deleted dynamically as well. This by itself makes the connections graph of a MANIFOLD program, which is the combined effect of all its manifolds, very dynamic. However, there is more. The manifestation of a single manifold is also a single (dynamically changing) process inter–connection graph. Since manifolds too are processes, the combined graph of a MANIFOLD program is indeed not a simple graph, but a hyper–graph, where each node in itself is a dynamically changing graph of connections among processes.

Although conceptually, the dominant control mechanism in MANIFOLD is event driven, the dataflow type data driven style of control through streams is at least equally as important. A manifold can internally raise an event for itself, causing a state transition. This can be, for instance, due to the arrival of a unit of information in the pre–transition state through a certain stream, and may also depend on the contents of this information. Thus, there is a smooth transition between the two mechanisms of control in MANIFOLD. The coexistence of event driven and data driven control gives MANIFOLD a unique flavor.

3. The MANIFOLD Model of Computation

The basic components in the MANIFOLD model of computation are processes, events, ports, and streams. A process is a *black box* with well defined ports of connection through which it exchanges *units* of information with the other processes in its environment. The internal operation of some of these black boxes are indeed written in the MANIFOLD language, which makes it possible to open them up, and describe their internal behavior using the MANIFOLD model. These processes are called manifolds. In general, a process in MANIFOLD does not, and need not, know the identity of the processes with which it exchanges information.

The interconnections between the ports of processes are made with streams. A stream represents a flow of a sequence of units between two ports. Streams are constructed and removed dynamically between ports of the processes that are to exchange some information. The constructor of a stream need not be the sender or the receiver of the information to be exchanged: any third party manifold process can define a connection between the ports of a producer process and a consumer process. Furthermore, stream definitions in MANIFOLD are generally additive. Thus a port can simultaneously be connected to many different ports through different streams. The flows of units of information in streams are automatically replicated and merged at outgoing and incoming port junctions, as necessary. The units of information exchanged through ports and streams, are *passive* pieces of information that are synchronously produced and synchronously consumed at the two ends of a stream, with their relative order preserved.

Orthogonal to the stream mechanism, there is an event mechanism for information exchange in MANIFOLD. Contrary to units in streams, events are *active* pieces of information that are broadcast by their sources in the environment. In principle, *any* process in the environment can pick up such a broadcast event. In practice, usually only a few processes pick up occurrences of each event, because only they are *tuned in* to their sources. Occurrences of the same event from the same source can override each other from the point of view of some observer processes, depending on the difference between their *sampling rate* and the occurrence rate of the event. Otherwise, event occurrences are never ''lost'' in MANIFOLD.

Events are generally raised synchronously by their sources and dissipate through the environment. They are active pieces of information in the sense that in general, they are observed asynchronously and once picked up, they preemptively cause a change of state in the observer. Events are the primary control mechanism in MANIFOLD.

Each state in a manifold defines a pattern of connections among the ports of some processes. The corresponding streams implementing these connections are created as soon as a manifold makes a state transition (caused by an event) to a new state, and are deleted as soon as it makes a transition from this state to another one. In general, the set of sources whose events are honored by an observer manifold, as well as the set of specific events which are honored, are both state dependent.

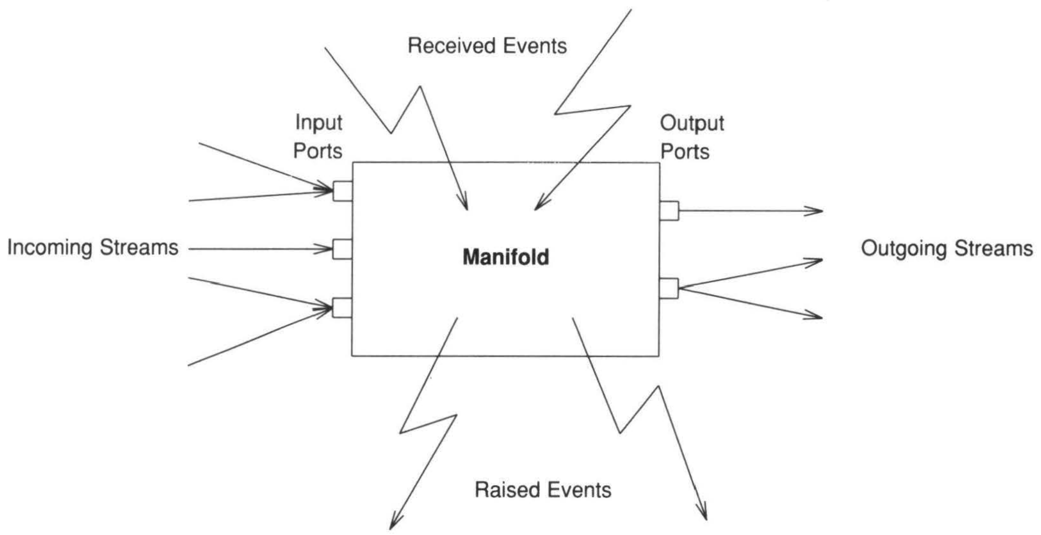

Figure 1 - A Process in MANIFOLD

4. Examples

4.1. Simple copy

One of the simplest manifolds is one that copies exactly one unit of its input to its output. We call this manifold **pass1**, and its definition is as follows.

```
pass1()
{
        start:
                getunit(input) → output.
}
```

Note that we have no explicit buffer declaration for this manifold's input and output ports. The default regular expression assigned to the standard input and standard output ports of a manifold is an expression which delivers the units it receives intact (e.g., the expression "\<\>"). Thus, the above manifold works independently of the unit sizes and makes no modifications to its input units.

Once activated, this manifold sets up a pipeline between the **getunit(input)** pseudo process and the manifold's standard output port. The **getunit(input)** action waits, if necessary, for a unit to arrive at the input port of the manifold, places it on the output stream and halts. The completion event causes the pipeline to breakup. Since there are no other actions, the manifold terminates.

A slightly more sophisticated manifold is one that copies all of its input units to its output stream. Below are two versions of such a manifold, which we call **pass**, with identical behavior.

```
pass()
{
          start:
                    getunit(input) → output;
                    do start.
}
```

The above manifold consists of a simple loop. **getunit(input)** waits for one input unit on the input port, and copies it as its output. ‘‘ **getunit(input) → output**’’ places the unit into the standard output port of the manifold. Once this pipeline terminates, the handler is entered again.

A simpler version of this manifold is shown below. However, the above form is more useful as the basis for several other manifolds that perform other tasks in addition to the simple copying.

```
pass()
{
          start:
                    input → output.
}
```

The above manifold is clearly simpler than the previous one, its specification is more concise. Its behavior is also very similar. Specifically, note that no actual transfer from the input buffer to the output buffer of the manifold takes place, before a complete unit is available in the input port.

4.2. Action Synchronization

The connective ";" and groups are the basic mechanisms in MANIFOLD for synchronization. The group construct causes the manifold processor to deal with all elements in the group concurrently. The ";", on the other hand, waits for the termination of the configuration on its left hand side, before proceeding with the constructs on its right hand side.

However, it is sometimes necessary to synchronize actions with the flow of units in a pipeline. For example, suppose that we want event "E" to be raised as soon as the first unit passes from "A" to "B" in the pipeline "A → B". The construct "(A → B, **raise** E)" does not work, because "E" may be raised just after the stream between "A" and "B" is set up, which may be too early. The construct "A → B ; **raise** E" does not work either, because "E" can be raised only after the pipeline between "A" and "B" has expired, perhaps after many units have passed through the stream.

The manifold " **perform**(action)", below, can be used to synchronize "action" with the passing of the first unit through a stream. Using this manifold, the construct "A → **perform** (**raise** E) → B" does the job.

```
perform(action)
process             action.
{
          event     act.
          event     copy.

          start:
                    guard(input, act);
                    idle.

          act:
                    activate action;
                    action;
                    do copy.

          copy:
                    input → output.
}
```

4.3. Counter

The following manifold copies its input units to its output port without any change. It also counts the number of units that pass through and raises the specified "X" event once the given "limit" is reached. A typical use of the manifold is of the form " **count**(limit,E)", which means that an event "E" will be raised when reaching "limit".

```
count(limit, X)
process         limit.
event           X.
{
        process n       is      variable.
        event           check.
        event           now.
        event           copy.

        start:
                activate limit;
                activate n;
                n=limit;
                do check.

        check:
                if(n<1, do now, do copy).

        now:
                raise X;
                do copy.

        copy:
                getunit(input) → output;
                n=n-1;
                do check.
}
```

Note that because parameters are *not* variables as in typical programming languages, we cannot use ''limit'' for counting in this manifold. Therefore, we must define a **variable** (see also §4.5), ''n'', and set its initial value to the value of ''limit'' to do our counting[†]. Note also that the infix use of =, -, and < is just syntactic sugar for (implicit) activation of appropriate processes to which the corresponding parameters are passed.

On activation of an instance of **count**, the event ''start'' is automatically raised, and thus its corresponding handler becomes active. Once the process of assignment is complete, the handler for the event ''check'' is activated. This handler simply activates an instance of a manner: ''if(n<1, **do** now, **do** copy)'' (see §4.4 for the detailed description of this manner). The processor now enters the manner and behaves accordingly. The net effect of the manner ''if'' is to raise the event ''now'' if ''n'' is less than 1, or the event ''copy'' otherwise.

In either case, the corresponding handlers for these events cannot be found in the ''if'' manner itself. The processor then searches the dynamic chain of manner activations to locate the proper blocks. In this case, the blocks are found in the calling manifold. The manner is thus left, and the processor enters the appropriate block of the manifold.

The handler for ''now'' raises the specified event ''X'', and then activates the handler for ''copy''.

The handler for ''copy'' sends out one input unit and decrements the counter ''n'' by one. The **getunit(input)** primitive action waits for the arrival of a unit in the input port, and delivers it as its output. This output is placed in the standard output port of the manifold. Upon termination of this process, the counter ''n'' is decremented, and then the manifold activates the handler for ''check'' again.

[†]A variable is simply a built–in manifold that keeps the input units it receives and copies them to its output port. It is similar to the **repeater** described in §4.5).

Note that this version of **count** keeps raising ''X'' for every input unit it receives once the *limit* has been reached. Replacing the final '' **do** check'' in the handler for ''copy'' by a '' **do** copy'' will raise ''X'' only once but will continue to copy its input to the output. Yet a third version of this manifold, below, counts and copies up to ''limit'' number of units and dies.

```
count1(limit, X)
process             limit.
event               X.
{
        process n           is          variable.
        event               copy.
        event               now.

        start:
                activate limit;
                activate n;
                n=limit;
                do copy.

        copy:
                getunit(input) → output;
                n=n-1;
                if(n<1, do now, do copy).

        now:
                raise X;
                halt.
}
```

4.4. Control structures

The typical higher level control structures can be built out of the primitives as manifolds or manners. For instance, following are two versions of an **if** manner.

```
manner if(B, T, E)
action              B.
action              T,E.
{
        event               then.
        event               else.

        start:
                B →     ( → trigger(true, then),
                          → trigger(false, else),
                          → check_bool(Berror) ).
        then:
                T.

        else:
                E.
}
```

```
manner if(B, T)
action          B.
action          T.
{
        event           then.
        event           else.

        start:
                B →     ( → trigger(true, then),
                          → trigger(false, else),
                          → check_bool(Berror) ).

        then:
                T.

        else:
                return.
}
```

Both versions of **if** assume that ``B'' is a pipeline which, when constructed, will produce a boolean result[†]. To check the validity of this assumption, we use the process ``check_bool'' which we assume produces the proper error messages and raises the event ``Berror'' if its input is not a boolean value. Because there is no handler for ``Berror'' in the **if** manner itself, if raised, it will percolate up the dynamic chain of manner activations until either a handler is found, or the calling manifold environment is reached.

Note that it is not essential for the validity check of the result of ``B'' to be done before it is supplied to the triggers. We take advantage of this and avoid the delay of the validity check for the cases that the result is indeed a boolean, by supplying the result of ``B'' to the triggers and the checker simultaneously, using the group construct.

Exactly one of the three events ``then'', ``else'', or ``Berror'' will be raised in this block. This, in turn, will activate its corresponding block. In the **if-then** version, the ``else'' event simply causes the processor to leave the manner and return to its calling environment.

Note also, that in MANIFOLD, ``overloading'' of manifold and manner names is possible, using the number of input parameters to disambiguate the references. Consequently, the programmer may freely use both versions of the **if** manner, with two and three parameters, respectively. The MANIFOLD system will use the proper manner, depending on the number of actual parameters supplied.

[†]A ``boolean'' is, in fact, just a special unit, which may have the values **true** and **false**.

4.5. The repeater

A repeater basically reads its input units and copies them to its output. In addition, it goes on repeating the last input on its output port as long as no new input arrives. The definition of the repeater is as follows.

```
repeater()
{
        process         I        is        pass.
        event           flush.
        event           get.

        start:
                activate I;
                activate (I → void);
                do get.

        get:
                (I → output, I → I,
                 (getunit( input ) → I; guard( input, flush ))).

        flush:
                I → pass1() → void;
                do get.

        end:
                deactivate( I );
                halt.
}
```

When starting the repeater, the manifold ''I'' (an instance of **pass**) is activated. The output of this manifold is permanently directed to **void**; this is to avoid the situation where unnecessary units would remain on the output port of ''I'' if this port is not connected to other port. Note that the statement '' **activate** (I → **void**);'' results in the activation of an internal manifold which would set up the pipeline proper; in other words, even if the repeater leaves the block labeled by ''start'', this internal manifold and, consequently, the pipeline set up by it, will remain alive.

The block labeled ''get'' sets up a pipeline shown on Figure 2: basically, a loop is defined where ''I'' feeds its output back onto its own input port. When entering this block there is nothing in the ports of ''I''. The only effect of the first two elements in the group is that the loop pipeline is set up. When, however, the **getunit** succeeds, the loop starts, sends the input unit into its output port.

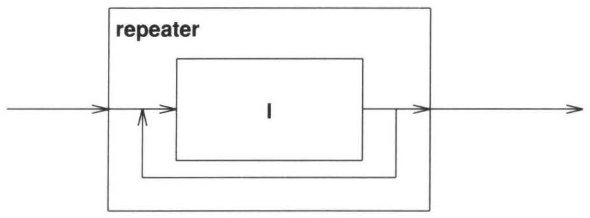

Figure 2 - Repeater Structure

The arrival of the next input unit will raise (via the **guard**) the event ''flush''. The only purpose of this block is (as its name suggests) to flush all previous instances of the previous input unit. This is done by breaking all pipelines; the output of ''I'' is directed exclusively to **void** and the (only) remaining unit on the input port of ''I'' is consumed via a **pass1** (remember that **pass1** dies when it has read one unit).

Note that the manifold **variable**, which is one of the built–in manifolds, is functionally equivalent to a **repeater** (although, of course, it is optimized for speed and efficiency by the run–time environment).

4.6. A Memory Cell

A memory cell is used to internally store a unit; in contrast to a **repeater**, it dispatches the unit only if it is explicitly requested to. It does not send, therefore, the value continually onto its output port.

```
memory_cell()
port    in        assign.
{
        process        R        is        repeater.
        event          wait.
        event          read.
        event          write.

        start:
                  activate( R );
                  ( guard( assign,read ),guard( input,write ) );
                  idle.

        read:
                  getunit(assign) → R;
                  guard( assign,read );
                  idle.

        write:
                  getunit(input);
                  R → pass1() → output;
                  guard( input,read );
                  idle.
}
```

Writing into the ''assign'' port of the manifold will result in the event ''read''; the value on ''assign'' will therefore be stored in the repeater ''R''. A new guard has to be started to react to the next ''assign'' write. When writing into the standard input of the manifold the event ''write'' will be activated; in the corresponding block one instance of the previously stored value will be output (note the use of a **pass1** manifold to secure that only one unit will be transmitted).

Having non–flushing queues is important here; indeed, with flushing queues, some units can be lost to the **getunit** action in the ''read'' block.

4.7. A Resource Management Example

The example presented below was motivated by a practical problem encountered in the course of a previous project, described in detail in ten Hagen et al[8]. Briefly, the goal of this project was to build a highly parallel graphics engine using dataflow techniques. What follows is a simplified description of the problems involved.

Conceptually, the graphics engine consisted of a number of independent processes. Each of these processes can be relatively complex by themselves, but this complexity is of no interest for the moment: they are considered to be **atomic** processes in MANIFOLD. For this example we assume that there are two generic types of such processes: transformation and drawer.

A transformation process encapsulates a number of complex calculations which are usual in computer graphics: matrix–vector multiplication on a list if points, projective division, eventual clipping etc. The peculiarity of the hardware described by ten Hagen et al[8] is that the machine may contain several instances of the very same generic process, setting up parallel transformation engines which can be run independently of one another. The result of a transformation is put into a global area, for example in shared memory. The transformation processes can therefore be viewed as special resources.

There is only one drawer process (usually a piece of hardware) but the access protocol to access this process involves resource management problems again. Data (which include geometric data as well as their attributes) are put in special queues for consumption by the drawer; as the time needed to fetch data from the shared memory area might be too long compared to the speed of the drawer, there may be several queues, fed in parallel to the actual processing of the drawer. As the hardware in use in the project was aimed at 3D operation using a Z–buffer, the exact order in which the graphics primitives are processed is irrelevant.

In case of ten Hagen et al[8], proper management of the resources involved was not a trivial task. The speed of the machine critically depends on the utilization of its transformation processors and of its drawer buffers. The necessary resource management procedures were written in dataflow assembly, which was a long and tedious task. We show, in what follows, that this resource management problem can be solved very simply using MANIFOLD.

In the first of the following two sections, we consider a simple case of this problem where the number of resources is predefined and fixed. This directly corresponds to the actual problem solved in ten Hagen et al[8]. The drawback of this first solution is that the number of resources is ''hardwired'' in the presented programs. Next, we consider a generalization of this resource management problem where the number of available resources is simply passed as an execution time value.

4.7.1. The Static Case

In this case we assume that the number of transformation processes and the number of drawer buffers are fixed. For the sake of simplicity we consider two transformation processes and two drawer buffers.

Following is the MANIFOLD program for this case. Note that the atomic processes (listed at the beginning) have only a very limited amount of information about the environment they work in; it might have been simpler to e.g. pipe the buffer processes directly to the drawer process and avoid the use of the manifolds ''Buff_Full'' and ''Draw_Start''. However, this would involve more knowledge on the part of these processes about the whole system environment.

```
//--------------------------------------------------------------------
//
// There are three types of atomic processes:
//   TR   is a transformation pipeline;
//        I/O ports:
//          input:              reference to geometric primitive to transform
//          output:             reference to transformed geometric primitive
//        Raised events:
//          trans_free:         ready to accept a new geometric primitive
//   BF   is a hardware buffer handler;
//        I/O ports:
//          input:              reference to the geometric primitive to store
//        Raised events:
//          buff_empty:         ready to accept a new geometric primitive
//          buff_full:  geometric primitive stored
//   DR   is a drawer;
//        I/O ports:
//          input:              buffer identifier of next primitive to draw
//        Raised events:
//          draw_free:          ready to draw
//
//--------------------------------------------------------------------
TR()
 port              in       input.
 port              out      output.
atomic.
pragma TR external "transformation"

BF()
 port              in input.
atomic.
pragma BF external "hardware_buffer"

atomic DR()
 port              in       input.
atomic.
pragma DR external "drawing_engine"
```

```
//
//
// Main manifold: it has to activate all necessary processes
//
//
process TR1              is      TR.
process TR2              is      TR.
process BF1              is      BF.
process BF2              is      BF.
process TC               is      Trans_Control.
process FDB              is      Fill_Draw_Buffer.
process DS               is      Draw_Start.
permanent TR1, TR2, BF1, BF2.

Main_manifold()
{
        process  DR      is      Drawer.
        event            setup.

        // Activation of all processes and manifold. Note that
        // the processes which are targets of activate actions are
        // *not* visible to the block and therefore cannot prematurely
        // terminate the block's execution.
        start:
                (activate TC, activate FDB, activate DS(DR),
                 activate Trans_Queue(TR1, TC), activate Trans_Queue(TR2, TC),
                 activate Buff_Empty_Queue(BF1, FDB),
                 activate Buff_Empty_Queue(BF2, FDB),
                 activate Buff_Full(BF1, DS), activate Buff_Full(BF1, DS),
                 activate BF1, activate BF2, activate DR);
                do setup.
        // A permanent pipeline is set up between TRi and FDB.
        // The standard input of this manifold is permanently connected
        // to TC.  It is assumed to carry the graphics primitives.
        setup:
                (TR1 → FDB, TR2 → FDB, → TC).
}
```

```
//-----------------------------------------------------------------
//
// Transf_Control
//    I/O ports:
//      input:          reference for the next geometric primitive
//      tr_buffer:      transformation process name
//
// tr_buffer is filled by Trans_Queue
//
// The manifold starts a transformation process with a new geometric
// primitive (if this latter is available and the transformation process
// is free to work)
//
Trans_Control()
port        in input.
port        in tr_buffer.
{
            start:
                        getunit(input) → $getunit(tr_buffer); do start.
}

//
// Trans_Queue
//    Caught events:
//      trans_free:     meaning that a transformation process is free to work
//
// The manifold reacts on the events by propagating the process name
// to the TC manifold (in fact, it makes use of the port
// tr_buffer of this latter manifold to queue up events). Note that events
// are queued, that is no "trans_free" events are lost!
//
Trans_Queue(T, TC)
process                 T,TC.
event                   trans_free.
{
            permanent T.

            start:
                        activate T;
                        idle.

            trans_free:
                        event_source → TC.tr_buffer; do start.
}
```

```
//------------------------------------------------------------------//
//
// Fill_Draw_Buffer
//    I/O ports:
//      input:                    reference for the next geometric primitive
//      bf_buffer:                buffer process name
//
// bf_buffer is filled by Buff_Empty_Queue
//
// This manifold passes the output of a transformation
// process to a buffer processes; the passed message is
// the data to be filled into the buffer and describes (somehow)
// an input primitive after it has been transformed by a transformer.
// The appropriate buffer identifier is passed via the bf-buffer input
// port.
//
Fill_Draw_Buffer()
port      in       input.
port      in       bf_buffer.
{
          start:
                   getunit(input) → $getunit(bf_buffer) ; do start.
}

//
// Buff_Empty_Queue
//    Caught events:
//      buff_free:    meaning that a transformation process is free to work
//
// This manifold reacts on the events by propagating the process name
// to the Fill_Draw_Buffer manifold (in fact, it makes use of the port
// bf_buffer of this latter manifold to queue up events). Note that events
// are queued, that is no "buff_free" events are lost!
//
Buff_Empty_Queue(B, FDB)
process            B,FDB.
event              buff_free.
{
          permanent B.

          start:
                   idle.

          buff_free:
                   event_source → FDB.bf_buffer; do start.
}
```

```
//--------------------------------------------------------------------//
//
// Buff_Full
//   Caught events:
//     buff_full:        buffer is full, ready to draw
//
// If a buffer is full, this event is propagated to DS
// Note that events are queued, that is no "buff_full" events are lost!
//
Buff_Full(BF, DS)
process              BF,DS.
event                buff_full.
{
        permanent BF.

        start:
                idle.

        buff_full:
                event_Source → DS; do start.
}
//
// Draw_Start
//   I/O ports:
//     input:                  internal name of buffer process
//   Caught events:
//     draw_free:    the drawer process is ready to draw
//
// In fact, Draw_Start reacts to two events: draw_free and buff_full
// (via the manifold Buff_Full) to draw the next primitive
//
Draw_Start( X )
process              X.
port                 in        input.
event                draw_free.
{
        permanent X.

        start:
                idle.

        draw_free:
                $getunit(input) → X; do start.
}
```

4.7.2. The Dynamic Case

As a more complicated case we assume that the number of transformation processes can vary depending on the local hardware configuration; also that there exists a means (e.g. by inspecting the available memory area and/or number of processors) to determine this number at start up time. The manifolds presented above should be extended so that a variable number of transformation processes may be used.

The extension of the example is very simple. Indeed, the only place in the program where the number of the transformation processes play a role are the lines:

```
activate Trans_Queue(TR1, TC), activate Trans_Queue(TR2, TC),
```

appearing in the manifold Main_manifold(). If, however, this line is replaced by something like:

```
cycle:
        activate( Trans_Queue( TR,TC ) );
        n = n + 1;
        if( n == number_of_transformations, do go_on, do cycle ).
```

then, a separate instance of ''Trans_Queue'' with its own corresponding ''TR'' are created in the given cycle. All other manifolds (including ''Trans_Queue'') remain unchanged.

A further possible generalization of the example is to allow a variable number of buffers as well. The way to do that is identical to what has been done for a variable number of transformations.

4.8. Bucket Sorting

In this section we present a MANIFOLD program that implements a parallel bucket sort algorithm. Our parallel sort algorithm is similar to the one presented by Suhler et al.[7], for a dynamic dataflow environment. The two algorithms, however, are not identical.

The essence of the algorithm is as follows. There exists an atomic process (perhaps a piece of hardware) that performs an efficient sorting of a number of input units, provided that this number is below a fixed threshold, *b*. For example, if *b* is 2, all that this atomic process has to do is a simple compare to decide the proper order of its two input units. The aim is therefore to start off as many instances of this atomic process as possible, passing up to *b* units of the incoming stream to each, and then merge the sorted output streams of the parallel sort processes into the final sorted output stream.

The core of the solution is a manifold called ''Sort_def''. This manifold receives all the units on its input and produces the sorted units on its output. It counts the number of incoming units and forwards the first bucket of units to an instance of the atomic sorting process. The size of a bucket, *b*, is the value of the variable limit. In case the original input contains more than one bucket–full of units, ''Sort_def'' directs the output of the atomic sorter to a so called ''Merger'' manifold. The ''Sort_def'' manifold then activates a new instance of itself and directs the rest of its incoming units to this new instance. The output of the new instance of ''Sort_def'' is directed to the same ''Merger''. Finally, the output of the ''Merger'' is connected to the output of the former instance of ''Sort_def''. The behavior of the manifold is therefore recursive. The ''bottom'' of the recursivity is when the number of incoming units is smaller than ''limit'' in which case it just redirects the output of the atomic sorter process to its own output (see also Figure 3.). In other cases, it splits the incoming units between the atomic process and another instance of itself (see Figure 4.).

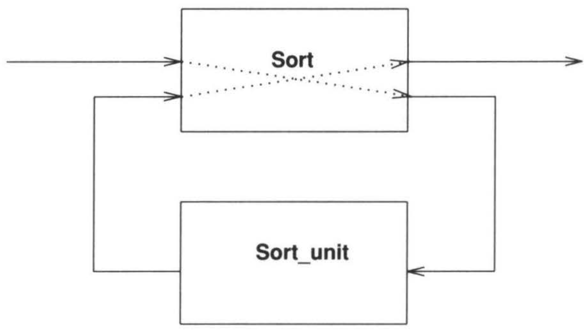

Figure 3 - Non recursive branch

The merger manifold merges the two incoming streams of sorted units into a single sorted output stream. It simply switches between its incoming lists based on the comparison of their next two units. To make it more general to use, comparison is done by another atomic process. Using these two atomic processes, the manifolds themselves can perform the logical process of sorting without interpreting the units themselves: this is left to the atomic processes.

The merger manifold uses a separate manner to handle the two cases. This manner (''next_element'') essentially performs the switch between the two incoming ports if necessary. If one of the two ports becomes empty (in other words, a **disconnected** event occurs), the manner sets up a direct pipeline between the other input and the output port which results in a fast copy of all remaining units. Note that we have made use of the fact that input ports are, by default, non–flushing; in other words, breaking the pipeline does not mean that the units in the pipeline get lost.

Note that a slight modification of ''Sort_def'' can improve its performance by modifying the function of ''Merge''. Indeed, ''Sort_def'' can use the first incoming unit as a ''pivot'' and send the first ''limit'' number of units that are smaller than this pivot to the ''Sort'', and the rest to its recursive incarnation.

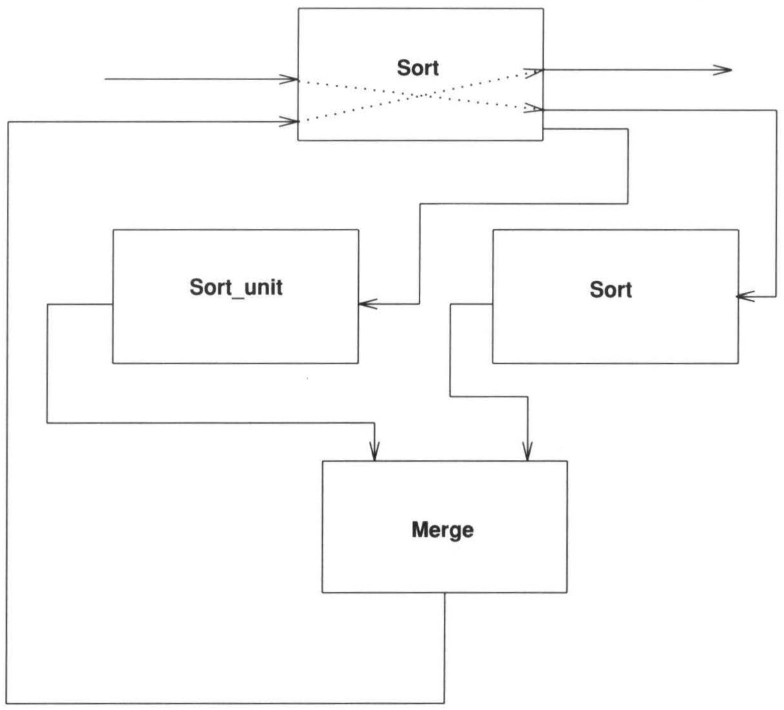

Figure 4 - Recursive branch

```
//
// Compare_units_def process:
//    I/O ports:
//          a:                   first unit to compare
//          b:                   second unit to compare
//          output:              boolean result, true iff a <= b
Compare_units_def()
 port    in        a.
 port    in        b.
 port    out       output.
atomic.
pragma Compare_units_def internal "compare"

//
// Sort_units_def process:
//    I/O ports:
//          input:               units to sort (up to "end of file", i.e. broken port)
//          output:              sorted units
Sort_units_def()
 port    in        input.
 port    out       output.
atomic.
pragma Sort_units_def internal "sort"
```

```
//---------------------------------------------------------------------//
manner next_element( smaller,smaller_data,larger,larger_data,
            dest_smaller,dest_larger,other_port )
port    in      smaller.
action          smaller_data.
port    in      larger.
action          larger_data.
port    in      dest_smaller,dest_larger.
action          other_port.
{
        event go_on.
        start:
                do go_on.
        go_on:
                smaller_data → pass1 → output;
                getunit( smaller ) → (→ dest_smaller, → smaller_data );
                larger_data → pass1 → dest_larger;
                if( getunit(result), do go_on, other_port ).

        disconnected.smaller:
                larger_data → pass1 → output;
                larger → output.

        disconnected.larger:
                do finish.
}
```

```
//
// Merge manifold:
//    I/O ports:
//          a:                  first list of units
//          b:                  second list of units
//          output:             sorted & merged units
//          result:             result of comparison
//
// uses a process "Compare" (of type "Compare_units_def")
// to compare two units; the latter returns a boolean unit
// on input port "result"
//
Merge_def()
port    in      a.
port    in      b.
port    in      result.
port    out     output.
{
        process         store_a         is      variable.
        process         store_b         is      variable.
        event           a_st_b.
        event           b_st_a.
        event           finish.
        process         Compare         is      Compare_units_def.
        permanent       Compare → result.

        start:  // activate registers and reads in the two first values
                activate Compare;
                ( getunit(a) → (→ Compare.a, → store_a ),
                  getunit(b) → (→ Compare.b, → store_b ) );
                if( getunit(result), do a_st_b, do b_st_a ).

//-----------------------------------------------------------------------//

    a_st_b: // a <= b
        next_element( a,store_a,b,store_b,Compare.a,Compare.b,do b_st_a).

    b_st_a: // a > b
        next_element( b,store_b,a,store_a,Compare.b,Compare.a,do a_st_b).

        finish:
                deactivate Compare.
}
```

```
//
// Effective Sorter
//    I/O ports:
//          input:                units to sort
//          output:               sorted units
//    Caught events:
//          sort_full:            the number of incoming events have reached "limit"
//
// Makes a recursive call to itself if  the number of the incoming units
// is more than the "limit"
// To count the incoming event, the manifold "count1" is used.
// Halts when all units are sorted and sent
//
Sort_def( limit )
port      in        input.
port      out       output.
event               sort_full.
{
          process             Sort                is        Sort_def.
          process             Sort_units          is        Sort_units_def.
          process             Merge               is        Merge_def.
          process             Count               is        count1.

          start:
                    activate Sort_units;
                    activate Count( limit );
                    input → Count → Sort_units;
                    idle.

//------------------------------------------------------------------------//

          disconnected.input:  // There are no more units than limit!
                    deactivate( Count );
                    Sort_units → output.

          death.Sort_units:
                    halt.

//------------------------------------------------------------------------//

          death.Count:
                    activate Merge;
                    activate Sort(limit);
                    ( Sort_units        → Merge.b,
                     Sort               → Merge.a,
                     input              → Sort,
                     Merge              → output ).
}
```

5. Other Applications

The possible application areas for MANIFOLD are numerous. It is an effective tool for describing interactions of autonomous active agents that communicate in an environment through message passing and global broadcast of events. For example, elaborate user interface design means planning the cooperation of different entities (the human operator being one of them) where the event–driven paradigm seems particularly useful. In our view, the central issue in a user interface is the design and implementation of the communication patterns among a set of modules. Some of these modules are generic (application independent) programs for acquisition and presentation of information expressed in forms appealing to humans. Others are, ideally, acquisition/presentation-independent modules that implement various functional components of a specific application. Previous experience with systems like DICE[6, 10] has shown that concurrency, event driven control mechanisms, and general interconnection networks[†] are all necessary for effective graphics user interface systems. MANIFOLD supports all of that and in addition, provides a level of dynamism that goes beyond many other user interface design tools.

Separating the specification of the dynamically changing communication patterns among the modules from the modules themselves seems to lead to better user interface architectures. A similar approach can also be useful in applications of real time computing where dynamic change of interconnection patterns (e.g., between measurement and monitoring devices and actuators) is crucial. Complex process–control systems, must orchestrate the cooperation of various programs, digital and/or analogue hardware, electronic sensors, human operators etc. Such interactions may be more easily expressed and managed in MANIFOLD.

Coordination of the interactions among a set of cooperating autonomous intelligent experts is also relevant in Distributed AI applications, *open* systems, such as Computer Integrated Manufacturing applications, and the complex control components of systems such as Intelligent CAD.

Recently, scientific visualization has raised similar issues as well. The problems here typically involve a combination of massive numerical calculations (sometimes performed on supercomputers) and very advanced graphics. Such functionality can best be achieved through a distributed approach, using segregated software and hardware tools. Tool sets like the Utah Raster Toolkit[5] are already a first step in this direction, although in case of this toolkit the individual processes can be connected in a pipeline fashion only. More recently, software systems like the apE system of the Ohio Supercomputer Center[3] work on the basis of inter–connecting a whole set of different software/hardware components in a more sophisticated communication network. An ''orchestrator'' like MANIFOLD can prove to be quite valuable in such applications.

Advances in neuroscience have shown that to properly model the nervous system requires massively parallel systems where, in contrast to conventional neural networks, each node in the system has the computational complexity of a microcomputer[4, 9]. MANIFOLD may offer an appropriate paradigm for expressing the dynamic behavior of such complex inter–connection networks.

6. Conclusion

The unique blend of event driven and data driven styles of programming, together with the dynamic connection hyper–graph of MANIFOLD seems to provide a promising paradigm for parallel programming. The emphasis of MANIFOLD is on orchestration of the interactions among a set of autonomous *expert* agents, each providing a well-defined segregated piece of functionality, into an integrated parallel system for accomplishing a larger task.

The MANIFOLD model of communication is conceptually powerful enough to express general purpose computing. Therefore, although the primary purpose of MANIFOLD is to manage communications, the same language also expresses computation in terms of communication. Thus, it is theoretically possible to replace *every* process in a MANIFOLD program by a manifold that expresses the same computation in terms of interactions among a set of finer–grained processes. This refinement can recursively be carried out all the way down to the level where each process expresses the functionality contained in a piece of hardware.

[†] In the case of DICE, this is actually a strict hierarchy, and has turned out to be one of its shortcomings in practice.

7. Acknowledgments

We wish to thank our colleagues at the Interactive Systems Department for their direct and indirect contributions to the work reported in this paper. In particular, Paul ten Hagen inspired the original concerns and the motivation for MANIFOLD by his earlier work on the Dialog Cells, and through our numerous ongoing discussions. Kees Blom helped to refine the formal syntax for the MANIFOLD language and is presently working on the MANIFOLD compiler. Per Spilling and Dirk Soede's exercises in MANIFOLD and their ongoing contributions to the project are also acknowledged and much appreciated.

References

1. Arbab, F. and Herman, I., ''MANIFOLD: A Language for Inter–Process Communication,'' Technical Report, Centrum voor Wiskunde en Informatica (CWI), Amsterdam (1990, to appear).
2. Arbab, F., ''Specification of MANIFOLD,'' Technical Report, Centrum voor Wiskunde en Informatica (CWI), Amsterdam (1990, to appear).
3. Dyer, S., ''A Dataflow Toolkit for Visualization,'' *IEEE Computer Graphics & Application*, pp. 60-69 (July 1990).
4. Matsumoto, G., ''Neurons as Microcomputers,'' *Future Generations Computer Systems* **4**, pp. 39-51 (1988).
5. Peterson, J.W., Bogart, R.G., and Thomas, S.W., ''The Utah Raster Toolkit,'' in *Proceedings of the Usenix Workshop on Graphics*, Monterey, California (November 1986).
6. Schouten, H.J. and ten Hagen, P.J.W., ''Dialogue Cell Resource Model and Basic Dialogue Cells,'' *Computer Graphics Forum* **7**, pp. 311-322 (1988).
7. Suhler, P.A., Bitwas, J., Korner, K.M., and Browne, J.C., ''TDFL: A Task–Level Dataflow Language,'' *Journal of Parallel and Distributed Computing* **9**, pp. 103-115 (1990).
8. ten Hagen, P.J.W., Herman, I., and de Vries, J.R.G., ''A Dataflow Graphics Workstation,'' *Computers and Graphics* **14**, pp. 83-93 (1990).
9. Thorpe, S.J., ''Image Processing by the Human Visual System,'' in *Advances in Computer Graphics VI*, ed. I. Herman and G. Garcia, EurographicSeminar Series, Springer Verlag, Berlin - Heidelberg - New York - Tokyo (1990).
10. van Liere, R. and ten Hagen, P.J.W., ''Introduction to Dialogue Cells,'' Technical Report, Centrum voor Wiskunde en Informatica (CWI), No. CS-R8703, Amsterdam (1987).