Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

F. Arbab, I. Herman

MANIFOLD: A language for specification of inter-process communication

Computer Science/Department of Interactive Systems Report CS-R9065 November

MANIFOLD: A Language for Specification of Inter-Process Communication

Farhad Arbab and Ivan Herman

Interactive Systems
Centre for Mathematics and Computer Science
Kruislaan 413
1098 SJ Amsterdam
The Netherlands
Telephone: +31 20 5924058
Fax: +31 20 5924199
Telex: 12571 mactr nl
Email: farhad@cwi.nl and ivan@cwi.nl

ABSTRACT

Management of the communications among a set of concurrent processes arises in many applications and is a central concern in parallel computing. In this paper, we introduce a language whose sole purpose is to describe and manage complex interconnections among independent, concurrent processes. In the underlying paradigm of this language, MANIFOLD, the primary concern is not with *what* functionality the individual processes in a parallel system provide. Instead, the emphasis is on *how* these processes are interconnected and how their interaction patterns change during the execution life of the system.

It is interesting that the conceptual model behind the MANIFOLD language immediately leads to a very simple, but non-conventional model of computation. Contrary to most other models, *computation* in MANIFOLD is built out of *communications*. As such it advocates a view point reminiscent of the connectionist view: that all (conventional) computation can be expressed as interactions.

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.

1. Introduction

Specification and management of the communications among a set of concurrent processes is at the core of many problems of interest to a number of contemporary research trends. Although communications issues come up in virtually every type of computing, and have influenced the design (or at least, a few constructs) of most programming languages, not much effort has been spent on conceptual models and languages whose sole prime focus of attention is on process interaction. Notable exceptions include the theory of neural networks, and to some extent, the concept of dataflow programming and the theory of Communicating Sequential Processes.

In this paper, we introduce MANIFOLD: a language whose sole purpose is to describe and manage complex interconnections among independent, concurrent processes. A detailed description of the MANIFOLD model and the syntax and semantics of the MANIFOLD language is of course beyond the scope of this paper. The specification of the MANIFOLD model and system is given elsewhere[1]. We summarize only enough of the description of the MANIFOLD model to give an impression of its potentials. To give a flavor of the MANIFOLD language and show how it is used in parallel computing, in this paper we explain the implementation of a parallel bucket sort algorithm in MANIFOLD. More examples of the use of the MANIFOLD language are given elsewhere[2]. Only enough of the syntax and semantics of the language is discussed here to make the critical parts of the bucket sort example program understandable.

It is interesting that the conceptual model behind the MANIFOLD language immediately leads to a very simple, but non-conventional model of computation. The MANIFOLD model is conceptually as powerful as conventional models, e.g., the Turing Machine. However, contrary to most other models, *computation* in MANIFOLD is built out of *communications*.

The rest of this paper is organized as follows. Section 2 presents some of the motivation behind the design of MANIFOLD. In Section 3, we inspire an intuitive feeling for what MANIFOLD programming is like by comparing and contrasting it with a number of different styles of programming. Section 4 contains a summary of the key concepts in the MANIFOLD model. Section 5 explains the critical part of a complete MANIFOLD program which implements a parallel bucket sort algorithm. The complete MANIFOLD program itself appears in Appendix A. A flavor of the syntax and the semantics of the MANIFOLD language can be skimmed from the explanation of the piece of code presented in Section 5. Section 6 mentions some other application areas where the MANIFOLD style of programming seems to be a promising approach. Section 7 contrasts MANIFOLD with a few other systems with similar features or concerns. Finally, Section 8 contains a few concluding remarks about MANIFOLD.

Report CS-R9065
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

2. Motivation

One of the fundamental problems in parallel programming is coordination and control of the communications among the sequential fragments that comprise a parallel program. Programming of parallel systems is often considerably more difficult than (what intuitively seems to be) necessary. It is widely acknowledged that a major obstacle to a more widespread use of massive parallelism is the lack of a coherent model of how parallel systems must be organized and programmed. To complicate the situation, there is an important pragmatic concern with significant theoretical consequences on models of computation for parallel systems. Many user communities are unwilling and/or cannot afford to ignore their previous investment in existing algorithms and ''off-the-shelf'' software and migrate to a new and bare environment. This implies that a suitable model for parallel systems must be *open* in the sense that it can accommodate components that have been developed with little or no regards for their inclusion in an environment where they must interact and cooperate with other modules.

Many approaches to parallel programming are based on the same computation models as sequential programming, with added on features to deal with communications and control. There is an inherent contradiction in such approaches which shows up in the form of complex semantics for these added on features. The fundamental assumption in sequential programming is that there is only one active entity, *the* processor, and the executing program is in control of this entity, and thus in charge of the application environment. In parallel programming, there are many active entities and a sequential fragment in a parallel application cannot, in general, make the convenient assumption that it can rely on its incrementally updated model of its environment.

To reconcile the ''disorderly'' dynamism of its environment with the orderly progression of a sequential fragment ''quite a lot of things'' need to happen at the explicit points in a sequential fragment when it uses one of the constructs to interact with its environment. Hiding all that needs to happen at such points in a few communication constructs within an essentially sequential language, makes their semantics complex. Inter-mixing the neat consecutive progression of a sequential fragment, focused on a specific function, with updating of its model of its environment and explicit communications with other such fragments, makes the dynamic behavior of the components of a parallel application program written in such languages difficult to understand. This may be tolerable in applications that involve only small scale parallelism, but becomes an extremely difficult problem with massive parallelism.

Separating the communication issues from the functionality of the component modules in a parallel system makes them more independent of their context, and thus more reusable. It also makes it possible to delay decisions about the interconnection patterns of these modules, which may be changed subject to a different set of concerns.

There are even stronger reasons in distributed programming for delaying the decision about the interconnections and the communication patterns of modules. Some of the basic problems with the parallelism in parallel computing become more acute in distributed computing, due to the distribution of the application modules over loosely coupled processors, perhaps running under quite different environments in geographically different locations. The implied communications delays and heterogeneity of the computational environment encompassing an application, become more significant concerns than in other types of parallel programming. This mandates, among other things, more flexibility, reusability, and robustness of modules with fewer hard-wired assumptions about their environment.

The tangible payoffs reaped from separating the communications aspect of a multi process application from the functionality of its individual processes include clarity, efficiency, and reusability of modules and the communications specifications. This separation makes the communications control of the cooperating processes in an application more explicit, clear, and understandable at a higher level of abstraction. It also encourages individual processes to make less severe assumptions about their environment. The same communications control component can be used with various processes that perform functions *similar* to each other from a very high level of abstraction. Likewise, the same processes can be used with quite different communications control components. This helps modularity, efficiency, and reusability.

3. 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. A comparison of MANIFOLD and some such models and systems is made in §7. 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 that end, we mention dataflow programming, shell scripts, and event driven programming in this section.

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 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 of course a single (dynamically changing) process interconnection 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.

4. 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

™UNIX is a trademark of AT&T Bell Laboratories process in MANIFOLD does not, and need not, know the identity of the processes with which it exchanges information. Figure 1 shows an abstract representation of a MANIFOLD process.

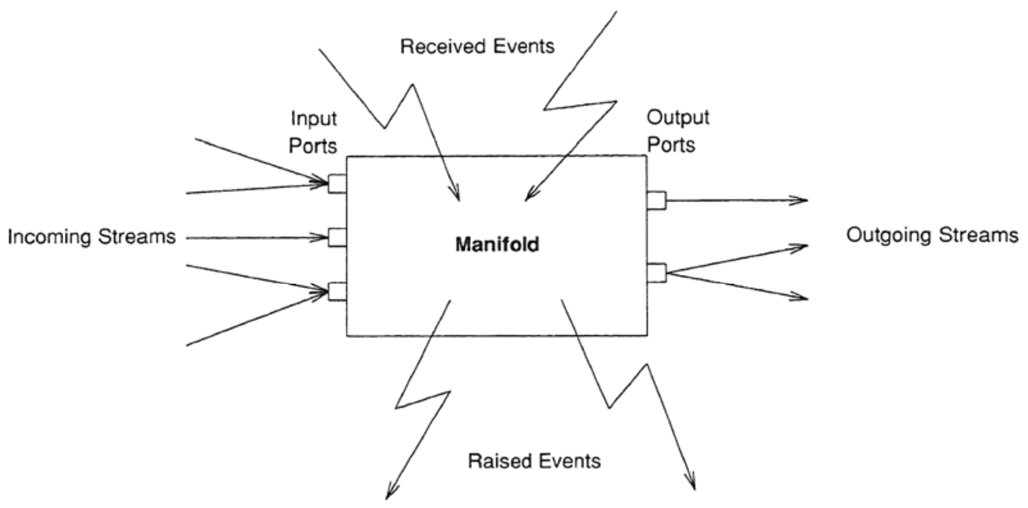

Figure 1 - The model of a process in MANIFOLD

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.

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.

The remainder of this section contains more detailed definitions of the basic concepts of the MANIFOLD model.

4.1. Processes

A *process* is an independent, autonomous, active entity that executes a procedure. A process has its own private processor and memory. Independence means that a process is not necessarily aware of the number and the nature of other processes that are simultaneously active in its environment. The environment of a process contains the set of other processes that directly or indirectly influence the behavior of the process or its performance.

Autonomous means that conceptually, no process exerts direct control on any other process. The only way to influence a process is through its input and output streams and the events to which it is sensitive. For example, once a process is activated, it cannot be ''forced'' to terminate by other processes, including its activator. However, it can be ''asked'' to terminate, by placing appropriate symbols in its input streams, or by raising an appropriate event. Similarly, there is no guarantee that a process will indeed read from its input streams, write to its output streams, react to some arbitrary event, or stay alive for any length of time.

The above model of communication is powerful enough to support all forms of interprocess communication. Therefore, in principle, there is no need for other forms of communication among processes. In practice, however, it may be desirable to allow other forms of inter-process communication, e.g., for convenience. For example, processes may need to communicate and influence each other through other means for purposes such as resource management, job control, side effects (e.g., files), interaction with the real world, etc., and may use mechanisms such as message passing, shared memory, etc. While the MANIFOLD model does not preclude such communications, *it assumes that all communication of interest with a process takes place through its input and output streams and via events.*

There are two kinds of processes: *atomic processes* and *manifolds*. An atomic process is similar to a black box whose internal structure and behavior are unknown. The set of atomic processes is application dependent, and thus, is neither predefined nor fixed. Examples of atomic processes include processes written in some programming language other than MANIFOLD, a hardware device, and a person interacting with a program.

A manifold is a process whose behavior and structure are described in the MANIFOLD language by a *manifold definition*. Manifolds ''orchestrate'' the communication and interaction among processes (atomic processes and other manifolds alike), and provide a dynamic means of control over a multiprocessing environment. The processor that runs a manifold is called the *manifold processor*.

4.2. Streams

A *stream* is a sequence of bits, grouped into (variable length) *units*. A stream represents a reliable, directed flow of information in time. Reliable means that the bits placed into a stream are guaranteed to flow through without loss, error, or duplication, with their order preserved. It does not, however, imply timing constraints. Directed means that there are always two identifiable ends in a stream, a *source* and a *sink*.

The size and the contents of the units that flow through streams are defined by their sources. Although units are meaningful inside streams, they imply no corresponding boundaries, types, tags, or interpretation on their contents at their sinks. Unit boundaries are used in streams to preserve the integrity of their information contents, and for synchronization purposes.

Conceptually, a stream in MANIFOLD has an unspecified capacity that is used as a FIFO queue, enabling asynchronous production and consumption of units by its source and sink. Streams in MANIFOLD are dynamically constructed and dismantled.

4.3. Ports

The connection between streams and processes is through ports. A *port* is a regulated opening at the boundary of a process, through which the information produced (consumed) by the process is placed into (picked up from) a stream. Regulated means that the information can flow in only one direction through a port: it either flows into or out of the process.

While streams are independent entities outside of processes, ports are properties of processes and are defined and owned by them. Information placed into one of its output ports by a process, flows out of the port only when it is connected to a stream. This ensures that no information is lost if a process writes to one of its output ports while it is not connected to any stream.

4.4. Events

An event is an asynchronous, non-decomposable message, broadcast by a process to its environment. Broadcasting such a message is called *raising the event*. Events are identified by their names, and can also be distinguished based on their sources (except, perhaps, when they are raised by atomic processes).

Although conceptually, an event is broadcast when it is raised, only a subset of the processes in its environment can pick up the broadcast and react to it. A process that picks up an occurrence of an event is called an *observer* (of the event and of its source). To pick up a broadcast event, a manifold must be in a state wherein the source of the event is *visible* to the manifold. In general, a manifold reacts only to a subset of the events it observes. These are the ones for which it has an event handler. The other observed events are ignored. Reacting to an event always causes a change of state in a (receiver) manifold.

Different occurrences of the same event from the same source may override each other before some of their observers get a chance to observe them. The overridden event occurrences are thus lost to those observers. This means that some event occurrences may be lost to some observers, but not to others, depending on the speed with which they *sample* their environment. Occurrences of events from different sources do *not* override each other. Occurrences of different events from the same source do not override each other, either.

An observed event may cause a change of state in a manifold, or it may decide to ignore the event. The change of state in a manifold may affect its sensitivity and reaction to future events. In each new state, a manifold begins to *react* to the observed event that caused the change of state. An observed event may *preempt* a manifold's attempt to react to a previously observed event (from the same or a different source).

5. A Parallel Bucket Sort Example

In this section we introduce some of the key concepts of the MANIFOLD system by presenting a MANIFOLD program that implements a parallel bucket sort algorithm. The complete MANIFOLD program appears in Appendix A. However, only the critical parts of the program are explained in this section.

Our parallel sort algorithm is similar to the one presented by Suhler et al.[12], 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 Merger manifold merges its two incoming streams of ordered units into a single output stream of ordered units. We do not discuss the details of the Merger manifold here, because explaining more details of the syntax of the language is beyond the scope of this paper. (The *manners* that appear in the Merger manifold, for example, are dynamically nested subroutine calls.)

Observe that the behavior of the Sort_def manifold is recursive. The terminal case for this recursion is when the number of incoming units is less than the bucket size. In this case the corresponding instance of Sort_def simply connects the output of its atomic sorter to its own output (see Figure 2). In other cases, it splits the incoming units between an instance of the atomic sorter and another instance of itself (see Figure

3).

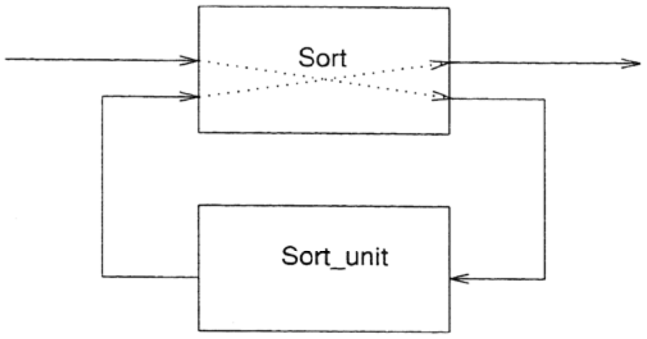

Figure 2 - Terminal Case for the Recursion

We use the core of the program, the Sort_def manifold, as a reference to explain how a manifold works, and clarify its syntax. This portion of the program appears below. A // symbol marks the start of a comment that extends to the end of the line.

The header of this manifold defines its name, Sort_def, and its parameter, limit. In its *public* declarations section following its header, the input and output ports of the manifold are defined. The declarations for input and output are indeed redundant, because they are the same as the default definitions for all manifolds. In addition to the ports, sort_full is also declared here as an event exchanged between this manifold and its environment.

The body of the manifold consists of the lines enclosed between the symbols { and }. In this case, it starts with some *private* declarations, all of which happen to define instances of various processes. For example, the line process Sort is Sort_def. defines Sort as an instance of the manifold Sort_def, and process Count is count1. defines Count to be an instance of the library manifold count1.

The bulk of the body of a manifold consists of a number of blocks, each labeled with a list of events. In this case it so happens that each block has only one label. The event start is raised automatically when an instance of a manifold is activated. The block labeled start is thus the first block that is entered in every manifold instance. In the case, the Sort_def manifold activates an instance of an atomic sorter process and an instance of a special manifold which is used to count the number of incoming units. It then sets up a pipeline connecting its own standard input to the standard input of its atomic sorter, with the counter in between. Thus, all incoming units will be directed to the atomic sorter. The manifold processor of this instance of the Sort_def manifold is now waiting for the expiration of the pipeline it just set up.

The counter manifold basically passes all of its input units on to its standard output, up to the point in time when it has passed limit number of units. It then halts. This event is observed by the instance of the Sort_def manifold that activated this counter instance, and it causes a state transition in death.Count. Reacting to this event, the processor of the Sort_def instance leaves the start block, dismantling the pipeline set up there, and finds the proper handler block for death.Count. This happens to be the last block in the Sort_def manifold.

Since the pipeline set up in start is now broken, no more units will flow to Sort_units. Instead, a Merge and a new instance of Sort_def are activated and, then, a number of parallel pipelines are set up in the block labeled death.Count. This block consists of the two activate actions mentioned above, followed by a construct called a *group*. A group is a comma-separated list of pipelines enclosed in a pair of parentheses, and represents parallel operation of its component pipelines. This is the situation depicted in Figure 3. From now on, all incoming units flow to the recursively activated instance of Sort_def.

Note that a slight modification of Sort_def in this block can improve the performance of the sort algorithm by simplifying the function of the Merge. Sort_def can use the first incoming unit as a ''pivot'' and

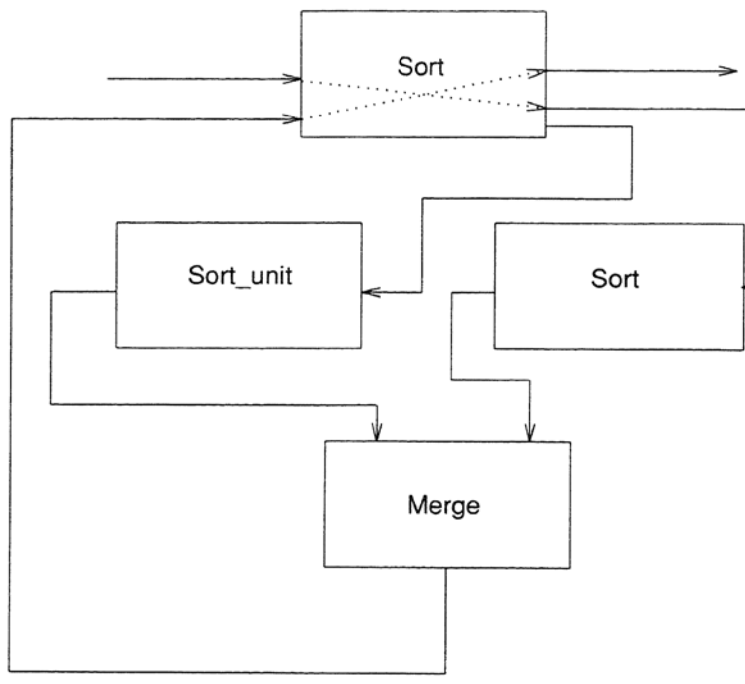

Figure 3 - Recursive branch

send the first limit number of units that are smaller than this pivot to the Sort, and the rest to its recursive incarnation.

```
//
// 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.
{
          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.

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

          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 ).
}
```

6. 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[11, 15] 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 Artificial Intelligence applications, *open* systems, such as Computer Integrated Manufacturing applications, and the complex control components of systems such as Intelligent Computer Aided Design.

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[10] 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[6] 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[9, 14]. MANIFOLD may offer an appropriate paradigm for expressing the dynamic behavior of such complex inter-connection networks.

7. Related Work

The general concerns which led to the design of MANIFOLD are not new. The CODE system[3, 4] provides a means to define dependency graphs on sequential programs. The programs can be written in a general purpose programming language like Fortran or Ada. The translator of the CODE system translates dependency graph specifications into the underlying parallel computation structures. In case of Ada, for example, these are the language constructs for rendezvous. In case of languages like Fortran or C, some suitable language extensions are necessary. Just as in traditional dataflow models, the dependency graph in the CODE system is static.

The MANIFOLD streams that interconnect individual processes into a network of cooperating concurrent active agents are somewhat similar to links in dataflow networks. However, there are several important differences between MANIFOLD and dataflow systems. First, dataflow systems are usually fine-grained (see for example Veen[16] or Herath et. al[7] for an overview of the traditional dataflow models). The MANIFOLD model, on the other hand, is essentially oblivious to the granularity level of the parallelism, although the MANIFOLD system is mainly intended for coarser-grained parallelism than in the case of traditional dataflow. Thus, in contrast to most dataflow systems where each node in the network performs roughly the equivalent of an assembly level instruction, the computational power of a node in a MANIFOLD network is

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

much higher: it is the equivalent of an arbitrary process. In this respect, there is a stronger resemblance between MANIFOLD and such more advanced dataflow environments like the so called Task Level Dataflow Language of Suhler et al[12].

Second, the dataflow like control through the flow of information in the network of streams is not the only control mechanism in MANIFOLD. Orthogonal to the mechanism of streams, MANIFOLD is an event driven paradigm. State transitions caused by a manifold's observing occurrences of events in its environment, dynamically change the network of a running program. This seems to provide a very useful complement to the dataflow like control mechanism inherent in MANIFOLD streams.

Third, dataflow programs usually have no means of reorganizing their network at run time. Conceptually, the abstract dataflow machine is fed with a given network once at the initialization time, prior to the program execution. This network must then represent the connections graph of the program throughout its execution life. This lack of dynamism together with the fine granularity of the parallelism cause serious problems when dataflow is used in realistic applications. As an example, one of the authors of this paper participated in one of the very rare practical projects where dataflow programming was used in a computer graphics application[13]. This experience shows that the time required for effective programming of the dataflow hardware (almost 1 year in this case) was not commensurate with the rather simple functionality of the implemented graphics algorithms.

The previously mentioned TDFL model[12] changes the traditional dataflow model by adding the possibility to use high level sequential programs as computational nodes, and also a means for dynamic modification of the connections graph of a running program. However, the equivalent of the event driven control mechanism of MANIFOLD does not exist in TDFL. Furthermore, the programming language available for defining individual manifolds seems to be incomparably richer than the possibilities offered in TDFL.

Following a very different mental path, the authors of LINDA[5] were also clearly concerned with the reusability of existing software. LINDA uses a so called generative communication model, based on a *tuple space*. The tuple space of LINDA is a centrally managed space which contains all pieces of information that processes want to communicate. A process in LINDA is a black box. The tuple space exists outside of these black boxes which, effectively, do the real computing. LINDA processes can be written in any language. The semantics of the tuples is independent of the underlying programming language used. As such, LINDA supports reusability of existing software as components in a parallel system, much like MANIFOLD.

Instead of designing a separate language for defining processes, the authors of LINDA have chosen to provide language extensions for a number of different existing programming languages. This is necessary in LINDA because seemingly, its model of communication (i.e., its tuple space and the operations defined for it) is not sufficient by itself to express computation of a general nature. The LINDA language extensions on one hand place certain communication concerns inside of the "black box" processes. On the other hand, there is no way for a process in LINDA to influence other processes in its environment directly. Communication is restricted to the information contained in the tuples, synchronously and voluntarily placed in and picked from the tuple space. We believe a mechanism for direct influence (but not necessarily direct control), such as the event driven control in MANIFOLD, is desirable in parallel programming.

One of the best known paradigms for organizing a set of sequential processes into a parallel system is the Communicating Sequential Processes model formalized by Hoare[8]. CSP is a very general model which has been used as the foundation of many parallel systems. Sequential processes in CSP are abstract entities that can communicate with each other via pipes and events as well. CSP is a powerful model for describing concurrent systems. However, there is no way in CSP to dynamically change the communications patterns of a running parallel system, unless such changes are hard coded inside the communicating processes. In contrast, MANIFOLD clearly separates the functionality of a process from the concerns about its communication with its environment, and places the latter entirely outside of the process. It then completely takes over the responsibility for establishing and managing the interactions among processes in a parallel system.

8. 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.

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

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.

9. Acknowledgment

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.

Appendix A

```
//
// 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.
port        in larger.
{
            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.
{
          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.

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

          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 ).
}
```

References

1. Arbab, F., ''Specification of Manifold,'' Technical Report (in preparation), Centre for Mathematics and Computer Science (CWI), Amsterdam (1990).
2. Arbab, F. and Herman, I., ''Examples in Manifold,'' Technical Report (in preparation), Centre for Mathematics and Computer Science (CWI), Amsterdam (1990).
3. Browne, J. C., Azam, M., and Sobek, S., ''CODE: A Unified Approach to Parallel Programming,'' *IEEE Software* (July 1989).
4. Browne, J. C., Lee, T., and Werth, J., ''Experimental Evaluation of a Reusability–Oriented Parallel Programming Environment,'' *IEEE Transactions on Software Engineering* **16**, pp. 111-120 (1990).
5. Carriero, N. and Gelernter, D., ''Linda in Context,'' *Communication of the ACM* **32**, pp. 444-458 (1989).
6. Dyer, S., ''A Dataflow Toolkit for Visualization,'' *IEEE Computer Graphics & Application*, pp. 60-69 (July 1990).
7. Herath, J., Yamaguchi, Y., Saito, N., and Yuba, T., ''Dataflow Computing Models, Languages, and Machines for Intelligence Computations,'' *IEEE Transactions on Software Engineering* **14**, pp. 1805-1828 (1988).
8. Hoare, C. A. R., *Communicating Sequential Processes,* Prentice–Hall, New Jersey (1985).
9. Matsumoto, G., ''Neurons as Microcomputers,'' *Future Generations Computer Systems* **4**, pp. 39-51 (1988).
10. 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).
11. 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).
12. 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).
13. 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).
14. 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).
15. van Liere, R. and ten Hagen, P. J. W., ''Introduction to Dialogue Cells,'' Technical Report, Centre for Mathematics and Computer Science (CWI), No. CS-R8703, Amsterdam (1987).
16. Veen, A. H., ''Dataflow Machine Architecture,'' *ACM Computing Surveys* **18** (1986).