The Manifold Coordination Language

To Cor Baayen, at the occasion of his retirement

F. Arbab
I. Herman
CWI
Email: farhad@cwi.nl, ivan@cwi.nl

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

As an example of the application of MANIFOLD, we present a series of small manifold programs which describe the skeletons of some adaptive recursive algorithms that are of particular interest in computer graphics. Our concern in this paper is to show the expressibility of MANIFOLD and its usefulness in practice. Issues regarding performance and optimization are beyond the scope of this paper.

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. The theory of neural networks and the connectionist view of computation emphasize the significance of the concept of *management of connections* versus the local computation abilities of each node. The concept of dataflow programming has a certain resemblance with connectionism, albeit, it is closer to the discrete world of conventional programming than neural networks. Theoretical work on concurrency, e.g., CCS [1] and CSP [2, 3], is primarily concerned with the semantics of communications and interactions of concurrent sequential processes. Communication issues also come up in virtually every other type of computing, and have influenced the design (or at least,

a few constructs) of most programming languages. However, not much effort has been spent on conceptual models and languages whose sole prime focus of attention is on the coordination of interactions among processes.

In their recent paper [4], Gelernter and Carriero elaborate the distinction between *computational models and languages* versus *coordination models and languages*. They correctly observe that relatively little serious attention has been paid in the past to the latter, and that "ensembles" of asynchronous processes (many of which are off-the-shelf programs) running on parallel and distributed platforms will soon become predominant.

MANIFOLD is a language whose sole purpose is to manage complex interconnections among independent, concurrent processes. As such, like LINDA [5, 6], it is primarily a coordination language. However, there is no resemblance between LINDA and MANIFOLD, nor is there any similarity between the underlying models of these two languages. The details of the MANIFOLD model and the syntax and semantics of the MANIFOLD language are, of course, beyond the scope of this paper and are described in a separate document [7]. In this paper, we give an overview of the MANIFOLD language and its implementation and present the skeleton of some recursive algorithms which are of particular interest in computer graphics. Also, an application of the language in the field of scientific visualization is presented. We summarize only enough of the description of the MANIFOLD model and language here, to make the examples and the significant implementation issues presented in this paper understandable.

The rest of this paper is organized as follows. In §2 the main motivations behind the MANIFOLD language and its underlying computing model are discussed. In §3 a more detailed description of the language is presented. In §4 we mention some of the application areas where MANIFOLD can prove to be a useful tool. In §5, we present the skeleton of a few adaptive recursive algorithms taken from the field of computer graphics. The purpose of these examples is to illustrate the use of some of the features of the MANIFOLD language and to demonstrate the general applicability of MANIFOLD concepts. The analysis of these programs gives us a good opportunity to show the descriptive power of MANIFOLD. In §6, we discuss some of the similarities and major differences between MANIFOLD and certain related systems and models for parallel computing. In §7 we mention some of the extensions and enhancements we plan to make to the MANIFOLD system in the future. Finally, §8 concludes this paper.

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. This is the case for such concurrent programming languages like Ada [8], Concurrent C [9, 10], Concurrent C++ [11], Occam [12] and many others (the interested reader may consult, e.g., the survey of Bal et al. [13] for more details on these languages).

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 extremely 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.

Contrary to languages that try to hide as much of the "chaos of parallelism" as possible behind a facade of sequential programming, MANIFOLD is based on the idea that allowing programmers to see and feel this parallelism is actually beneficial. It is a formidable intellectual experience to realize that if one frees oneself from the confines of the sequential paradigm and accepts that logical processes are "cheap" (that is, they are fast to activate and to communicate with), then a number of practical problems and applications can be described and solved incomparably more easily and more elegantly. In other words, there often *is* a pay-off in using parallel or distributed programming, even if higher speeds are not (necessarily) achieved. Just as a practical example, the basic approach of using multi-processing is very clearly one of the reasons for the un-

deniable technical superiority of the NeWS windowing system over X Windows [14]; also, almost all the applications listed in §4 fall in this category.

The assumption of having cheap logical processes is not only in line with the direction of future hardware development, it is also compatible with the current trend in the evolution of contemporary software systems. The increasingly more frequent use of so-called "light-weight" processes within conventional operating systems[1] is a clear indication (see, for example, the Brown University Thread Package [15], the so-called μSystem [16], or even the way some of the above cited languages, e.g., AT&T's Concurrent C, are implemented). More recent operating system designs offer light-weight processes in their kernels (e.g., OSF/1, based on the Mach system [17, 18] of Carnegie Mellon, or SunOS [19]).

Separating 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 allows delaying decisions about the interconnection patterns of these modules, which may be changed subject to a different set of concerns. This idea is one of the main motivations behind the development of the MANIFOLD system.

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 real 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 the 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.

3 The Manifold Language

In this section we give a brief and informal overview of the MANIFOLD language. The sole purpose of the MANIFOLD language is to describe and manage

[1]Some authors prefer the term "pseudo-parallelism" for such or similar forms of parallelism, again, see Bal et al [13].

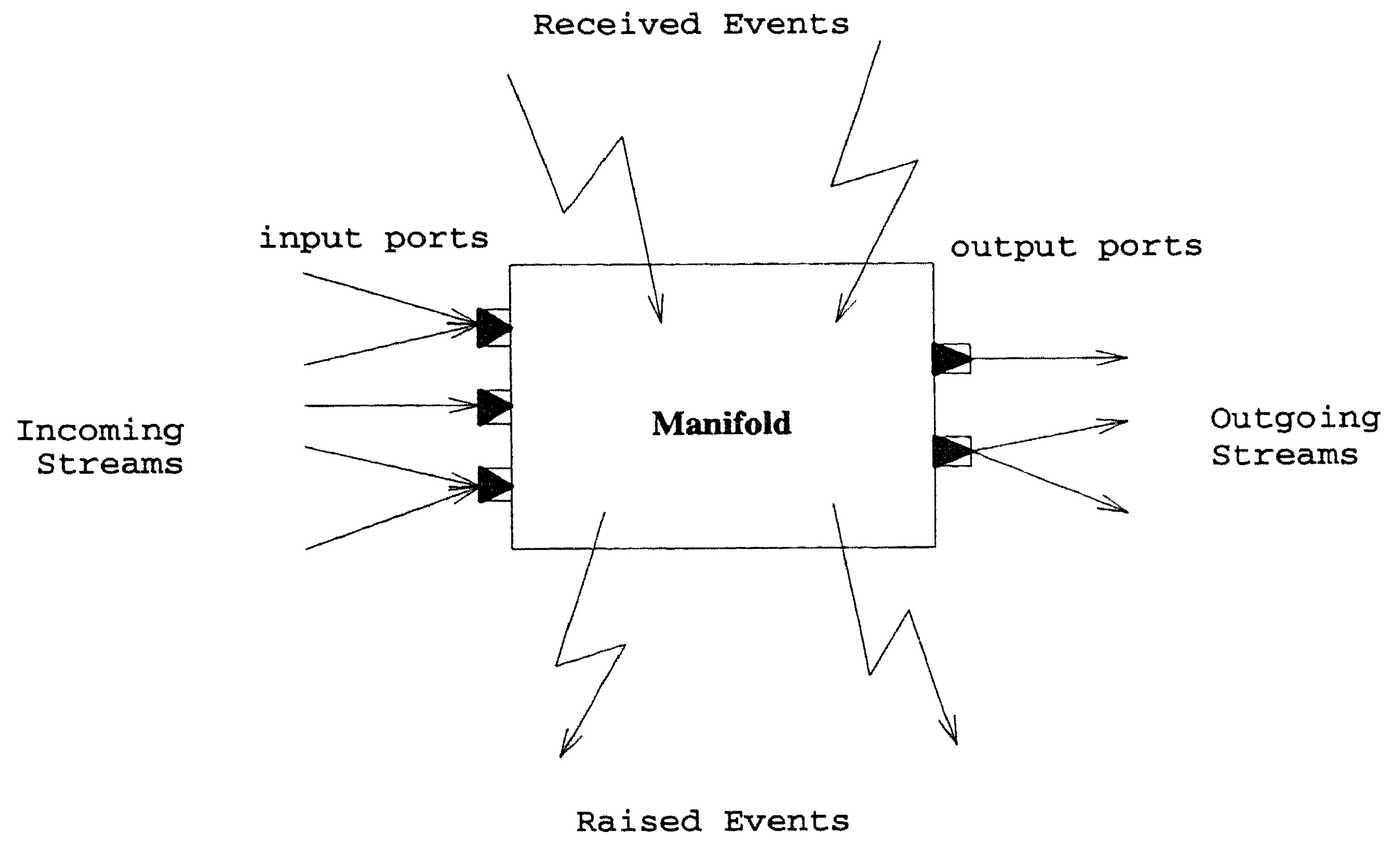

FIGURE 1. The model of a process in Manifold.

complex communications and interconnections among independent, concurrent processes. As stated earlier, a detailed description of the syntax and the semantics of the MANIFOLD language and its underlying model is given elsewhere [7]. Other reports contain more examples of the use of the MANIFOLD language [20, 21, 22, 23].

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

Ports are regulated openings at the boundaries of processes through which they exchange units of information. The MANIFOLD language allows assigning special filters to ports for screening and rebundling of the units of information exchanged through them. These filters are defined in a language of extended regular expressions. Any unit received by a port that does not match its regular

expression is automatically diverted to the `error` port of its manifold and raises a `badunit` event (see later sections for the details of events and their handling in MANIFOLD). The regular expressions of ports are an effective means for "type checking" and can be used to assure that the units received by a manifold are "meaningful."

Interconnections between the ports of processes are made with *streams*. A stream represents a flow of a sequence of units between two ports. Conceptually, the capacity of a stream is infinite. Streams are dynamically constructed between ports of the processes that are to exchange some information. Adding or removing streams does not directly affect the status of a running process. The constructor of a stream (which is a manifold) need not be the sender nor the receiver of the information to be exchanged: any third party manifold process can define a connection between the ports of a producer process and a consumer process. Furthermore, stream definitions in MANIFOLD are generally additive. Thus a port can simultaneously be connected to many different ports through different streams (see for example the network in Figure 2). The flows of units of information in streams are automatically replicated and merged at outgoing and incoming port junctions, as necessary. The units of information exchanged through ports and streams, are *passive* pieces of information that are produced and consumed at the two ends of a stream with their relative order preserved. The consumption and production of units via ports by a process is analogous to read and write operations in conventional programming languages. The word "passive" is meant to suggest the similarity between units and the data exchanged through such conventional I/O operations.

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

Once an event is raised by a source, it generally continues with its processing, while the event occurrence propagates through the environment independently. Event occurrences 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. Communication of processes through events is thus inherently asynchronous in MANIFOLD.

Each manifold defines a set of events and their sources whose occurrences it is interested to observe; they are called the *observable* set of events and sources,

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

Each state in a manifold defines a pattern of connections among the ports of some processes. The corresponding streams implementing these connections are created as soon as a manifold makes a state transition (caused by an event) to a new state, and are deleted as soon as it makes a transition from this state to another one. This is discussed in more detail in §3.2.

3.1 Manifold Definition

A manifold definition consists of a *header*, *public declarations*, and a *body*. The header of a manifold definition contains its name and the list of its formal parameters. The public declarations of a manifold are the statements that define its links to its environment. It gives the types of its formal parameters and the names of events and ports through which it communicates with other processes. A manifold body primarily consists of a number of *event handler blocks*, representing its different execution-time states. The body of a manifold may also contain additional declarative statements, defining *private* entities. For an example of a very simple manifold, see Listing 1 which shows the **MANIFOLD** source code for a simple program.[2] More complete manifold programs are also presented, e.g., in §5. Declarative statements may also appear outside of all manifold definitions, typically at the beginning of a source file. These declarations define global entities which are accessible to all manifolds in the same file, provided that they do not redefine them in their own scopes.

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

Each event handler block describes a set of actions in the form of a *group* construct. The actions specified in a group are executed in some non-deterministic order. Usually, these actions lead to setting up *pipelines* between various ports of different processes. A *group* is a comma-separated list of members enclosed in a pair of parentheses. In the degenerate case of a singleton group (which contains only one member) the parentheses may be deleted. Members of a group are either primitive actions, pipelines, or groups. The setting up of pipelines

[2] In this and other **MANIFOLD** program listings in this paper, the characters "//" denote the beginning of a comment which continues up to the end of the line. Keywords are typeset in bold.

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

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

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

    // Event handler for the event "e1"; several pipelines are
    //   set up (see Figure 2):
    e1: (B → input,C → A,A → B,output → A,B → C,input → output).

    // Event handler for the event "e2"; a single pipeline
    //   is set up (see Figure 3):
    e2: C → B.
}
```

Listing 1. An example for a manifold process.

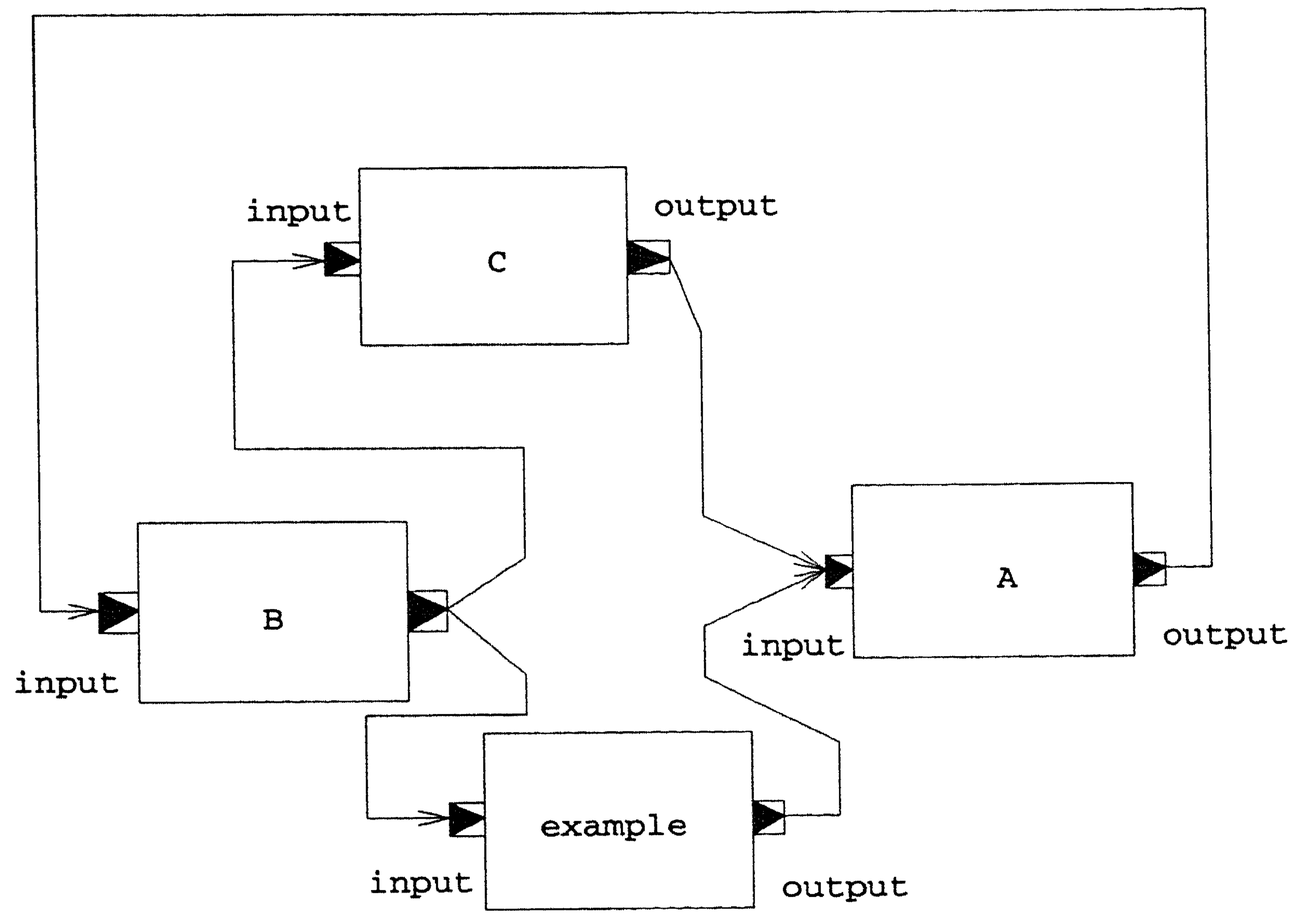

FIGURE 2. Connections set up by the manifold `example` on event `e1`.

within a group is simultaneous and atomic. No units flow through any of the streams inside a group before all of its pipelines are set up. Once set up, all pipelines in a group operate in parallel with each other.

A *primitive action* is typically *activating* or *deactivating* a process, *raising* an event, or a *do* action which causes a transition to another handler block without an event occurrence from outside. A *pipeline* is an expression defining a tandem of streams, represented as a sequence of one or more groups, processes, or ports, separated by right arrows. It defines a set of simultaneous connections among the ports of the specified groups and processes. If the initial (final) name in such a sequence is omitted, the initial (final) connection is made to the current input (output) port. Inside a group, the current input and output ports are the input and output ports of the group. Elsewhere, the current input and output ports are `input` and `output`, i.e., the executing manifold's standard input and output ports. As an example, Figure 2 shows the connections set up by the manifold process `example` on Listing 1, while it is in the handling block for the event `e1` (for the details of event handling see §3.2). Figure 3 shows the connections set up in the handling block for the event `e2`.

In its degenerate form, a pipeline consists of the name of a single port or process. Defining no useful connections, this degenerate form is nevertheless sometimes useful in event handler blocks because it has the effect of defining the named port or process as an observable source of events and a member of the preemption set of its containing block (see §3.4).

An event handler block may also describe sequential execution of a series of

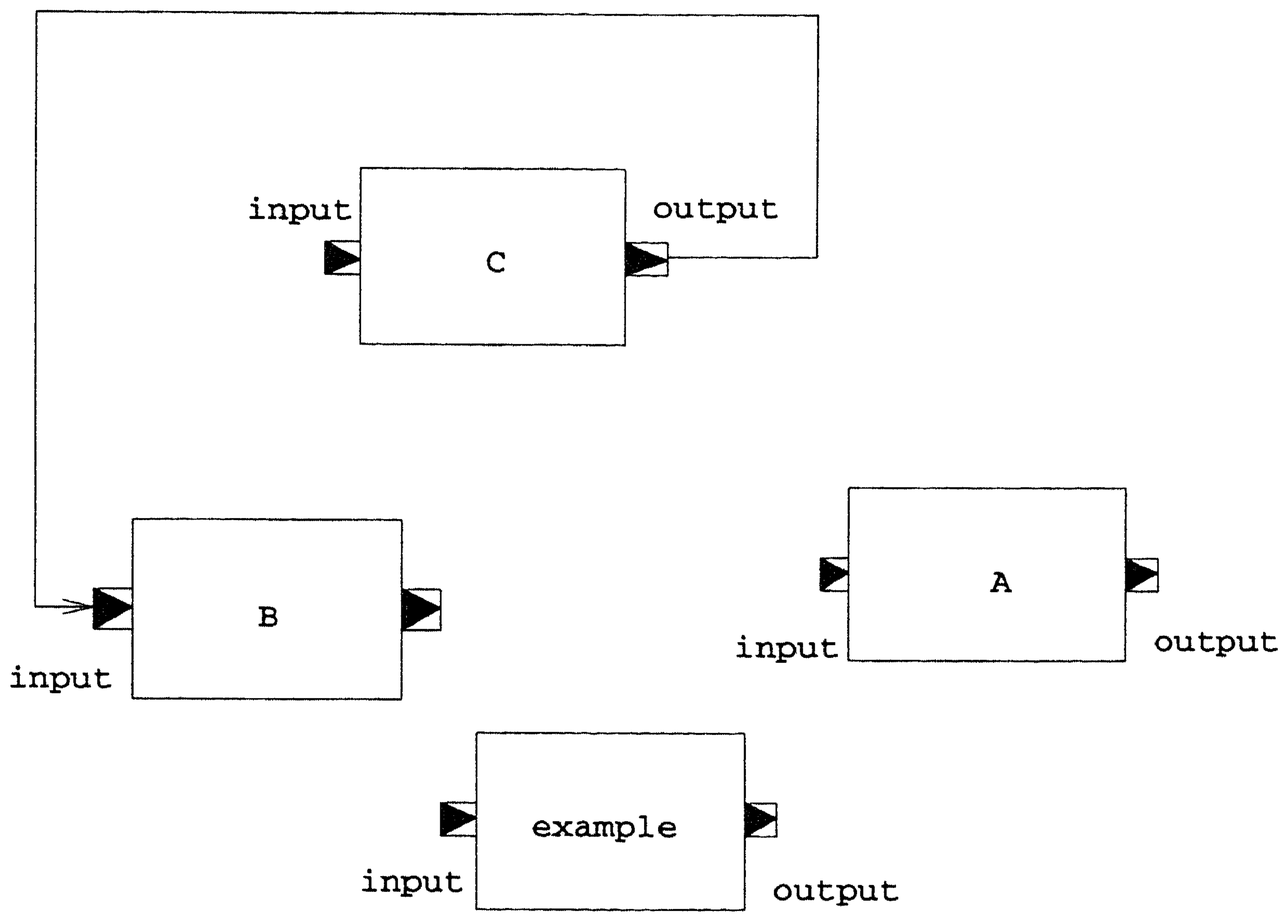

FIGURE 3. Connections set up by the manifold `example` on event `e2`.

(sets of) actions, by specifying a list of pipelines and groups, separated by the semicolon (;) operator[3]. In reaction to a recognized event, a manifold processor finds its appropriate event handler block and executes the list of sequential sets of actions specified therein. Once the manifold processor is through with the sequence in its current block, it terminates.

3.2 Event Handling

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

[3]In fact, the semicolon operator is only an infix *manner call* (see §3.5) rather than an independent concept in **MANIFOLD**. However, for our purposes, we can assume it to be the equivalent of the sequential composition operator of a language like Pascal.

and `event_source`, respectively.

The manifold processor finds the appropriate handler block for an observed event e raised by the source s, by performing a circular search in the list of block labels of the manifold. The list of block labels contains the labels of all blocks in a manifold in the sequential order of their appearance. The circular search starts with the labels of the current block in the list, scans to the end of the list, continues from the top of the list, and ends with the labels of the block preceding the current block in the list.

The manifold processor in a given manifold is sensitive to (i.e., interested in) only those events for which the manifold has a handler. All other events are to be ignored. Thus, events that do not match any label in this search do not affect the manifold in any way (however, see §3.5 for the case of called manners). Similarly, if the appropriate block found for an event is the keyword `ignore`, the observed event is ignored. Normally, events handled by the current block are also ignored.

The concept of an event in MANIFOLD is different than the concepts with the same name in most other systems, notably simulation languages, or CSP [2, 3]. Occurrence of an event in MANIFOLD is analogous to a flag that is raised by its source (process or port), *irrespective* of any communication links among processes. The source of an event continues immediately after it raises its flag, independent of any potential observers. This raised flag can potentially be seen by any process in the environment of its source. Indeed, it can be seen by any process to which the source of the event is *visible*. However, there are no guarantees that a raised flag will be observed by anyone, or that if observed, it will make the observer react immediately.

3.3 Event Handling Blocks

An event handling block consists of a comma-separated list of one or more block labels followed by a colon (:) and a single body. The body of an event handling block is either a group member (i.e., an action, a pipeline, or a group), or a single manner call(see §3.5). If the body of a block is a pipeline, and it starts (ends) with a $\rightarrow$, the port name `input` (respectively, `output`) is prepended (appended) to the pipeline.

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

The most specific form of a block label is a dotted pair $e.s$, designating event e from the source (port or process) s. The wild-card character * can be replaced for either e, or s, or both, in a block label. The form e is a short-hand for $e.*$ and captures event e coming from any source. The form $*.s$ captures any event from source s. Finally, the least specific block label is $*.*$ (or *, for short) which captures any event coming from any source.

3.4 Visibility of Event Sources

Every process instance or port defined or used anywhere in a manner (see §3.5) or manifold is an *observable* source of events for that manner or manifold. This simply means that occurrences of events raised by such sources (only) will be picked up by the executing manifold processor, provided that there is a handling block for them. The set of all events from observable sources that match any of the block labels in a manner or manifold is the set of observable events for that manner or manifold. The set of observable events of an executing manifold instance may expand and shrink dynamically due to manner calls and terminations (see §3.5). Depending on the state of a manifold processor (i.e., its current block), occurrences of observable events cause one of two possible actions: preemption of the current block, or saving of the event occurrence.

In each block, a manifold processor can react to only those events that are in the *preemption set* of that block. The MANIFOLD language defines the preemption set of a block to contain only those observable events whose sources appear in that block. This means that, while the manifold processor is in a block, except for the manifold itself, no process or port other than the ones named in that block can be the source of events to which it reacts immediately. There are other rules for the visibility of parameters and the operands of certain primitive actions. It is also possible to define certain processes as permanent sources of events that are visible in all blocks. A manifold can always internally raise an event that is visible only to itself via the do primitive action.

Once the manifold processor enters a block, it is immune to any of the events handled by that block, except if the event is raised by a do action in the block itself. This temporary immunity remains in effect until the manifold processor leaves the block. Other observable event occurrences that are not in the preemption set of the current block are saved.

3.5 Manners

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

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

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

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

The distinction between manners and manifolds is similar to the distinction between procedures and tasks (or processes) in other distributed programming languages. The term *manner* is indicative of the fact that by its invocation, a manifold processor changes its own context in such a way as to behave in a different manner in response to events.

Manner invocations are dynamically nested. References to all non-local names in a manner are left unresolved until its invocation time. Such references are resolved by following the dynamic chain of manner invocations in a last-in-first-out order, terminating with the environment of the manifold to which the executing processor belongs.

Upon invocation of a manner, the set of observable events of the executing manifold instance expands to the union of its previous value and the set of observable events of the invoked manner. The new members thus added to this set, if any, are deleted from the set upon termination of the invoked manner.

A manner invocation can either terminate normally or it can be preempted. Normal termination of a manner invocation occurs when a `return` primitive action is executed inside the manner. This returns the control back to the calling environment right after the manner call (this is analogous to returning from a subroutine call in conventional programming languages). Preemption occurs when a handling block for a recognized event occurrence cannot be found inside the actual manner body. This initiates a search through the dynamic chain of activations similar to the case of resolving references to non-local names, to find a handler for this event. If no such handler is found, the event occurrence is ignored. If a suitable handler is found, the control returns to its enclosing environment and all manner invocations in between are abandoned.

Manners are simply declarative "subroutines" that allow encapsulation and reuse of event handlers. The search through the dynamic chain of manner calls is the same as dynamic binding of handlers in calling environments, with event occurrences picked up in a called manner. Preemption is nothing but cleanly structured returns by all manner invocations up to the environment of a proper handler.

In principle, dynamic binding can be replaced by the use of (appropriately typed) parameters. Our preference for dynamic binding in manners is motivated by pragmatic considerations. Suppose a piece of information (e.g., how to handle a particular event, or where to return to) must be passed from a calling environment A, to a called environment B, through a number of intermediaries; i.e., B is *not* called directly by A, but rather, A calls some other "subroutine" which calls another one, which calls yet another one, ..., which eventually calls

B. Passing this information from A to B using parameters means that all intermediaries must know about it and explicitly pass it along, although it has no functional significance for them. Dynamic binding alleviates the need for this explicit passing of irrelevant information and makes the intermediary routines more general, less susceptible to change, and more reusable.

3.6 Scope Rules

The scope of a name is the syntactic context wherein that name is known as to denote the same entity. The scope of the names of atomic process specifications, manner definitions, and manifold definitions contained in a source file is the entire source file. The scope of the names defined in the private declarative section (inside the body) of a manifold or manner is the manifold or the manner itself. The scope of the names defined in the declarative statements outside of any manifold or manner definition, is the entire source file.

Ports of a manifold or atomic process are accessible to any process that knows its name and the name of its ports. Ports of a process, together with the events defined in its public declaration section, provide the communication links of a process with other processes running in its environment.

Except in manners, non-local names (i.e., used but not defined in a context), are statically bound to the entities with the same name in their enclosing contexts. It is a compile-time error if such a non-local name remains unresolved. The binding of non-local names (i.e., used but not defined) in manners is dynamic: these names are bound upon activation of a manner to the entities with the same name in the environment of its caller. The chain of manner activations leading to the present activation are traversed all the way up to the environment of a manifold instance, in search of appropriate targets for this binding. Names that remain unresolved at this point are bound to appropriate benign defaults (e.g., `void` described in §5.1.1).

MANIFOLD supports separate compilation. This is a very effective mechanism for modularization of large applications. In principle, all names defined and used in a source file are strictly local to that file. Names (of events, manners, manifolds, or atomic processes) that are used in different source files and must indeed designate the same entity at execution time, must be explicitly declared as such using `extern`, `import`, and `export` constructs (see [7]).

4 Applications

The MANIFOLD language has already been used to describe some simple examples, like a parallel bucket sort algorithm, a simplified version of a (graphics) resource management and the like. The interested reader is referred to the reports published elsewhere [20, 21]. These examples were primarily meant to test the MANIFOLD concepts themselves. In this section we mention some of the possible application areas for MANIFOLD in large-scale and non-trivial parallel systems.

MANIFOLD is an effective tool for describing interactions of autonomous active agents that communicate in an environment through address-less messages 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[4]. 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 User Interface Management Systems (see, e.g., [24]) has shown that concurrency, event driven control mechanisms, and general interconnection networks are all necessary for effective graphical 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. As an example, it has recently been used to successfully reformulate the GKS[5] input model [25]; this work is regarded as a starting point in the development of new concepts for highly flexible, reconfigurable graphics systems suitable for parallel environments.

Separating the specification of the dynamically changing communication patterns among a set of concurrent 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. For example, 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 [26] were already a first step in this direction, although in the 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

---

[4]In fact, given the previous experiences of the authors, the problems arising in user-interface techniques provided some of the basic motivation to start this project in the first place.

[5]Graphical Kernel System is the ISO Standard for Computer Graphics.

Center [27], the commercially available AVS Visualization Package of Stardent Computer Ltd. [28], the IRIS Explorer system [29] and others, work on the basis of inter-connecting a whole set of different software/hardware components in a more sophisticated communication network. The successes of these packages, and mainly the general ideas behind them, point toward a more general development trend which leads to reconsideration of the software architecture used for graphics packages in general.

For the emerging new technologies and application areas that are expected to result in a tremendous growth in computer graphics in the nineties, a new software base is necessary to accommodate demands for high performance special hardware, dedicated application systems, distributed and parallel computing, scientific visualization, object-oriented methods and multi-media, to name just a few. Some of the major technical concerns in the specification and the development of new graphics systems is *extensibility* and *reconfigurability*. To ensure these features it is feasible to envisage a highly parallel architecture which is based on the concept of cooperating, specialized agents with well defined but reconfigurable communication patterns. An "orchestrator" like MANIFOLD can prove to be quite valuable in such applications.

5 Adaptive Recursive Algorithms in Manifold

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

It is beyond the scope of this paper to give all the specific details of each algorithm. The interested reader can consult one of the standard textbooks on computer graphics and/or image processing (e.g., [30] for computer graphics and [31] for image processing) or refer to the literature given in the references (e.g., [32, 33, 34, 35, 36] or others).

5.1 Warnock's Algorithm

One of the very well known problems in computer graphics is what is usually referred to as Hidden Surface Removal. The problem is as follows. When a three-dimensional scene, usually modeled using a large number of planar polygons in space, is visualized on a screen, all of its polygons must be projected onto a plane (i.e., the plane of the display screen) from a given viewpoint. Mathematically, this projection is well understood, but there is an additional problem to solve: those polygons, or parts of polygons, that are occluded by another one, as seen from the selected viewpoint, must be eliminated. The removal of these (sub-)polygons is what is called the removal of hidden surfaces.

There are several well-known and widely applied solutions to this problem. One of the earliest is Warnock's algorithm which is described in detail in the

```
TestAndColor()  import.
DivideArea()
 port out first_area.
 port out second_area.
 port out third_area.
 port out fourth_area.
 import.

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

Listing 2. Manifold Program for Warnock's Algorithm.

literature, e.g., in [30]. A short description of this algorithm is as follows.

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

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

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

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

5.1.1 A Manifold Program for Warnock's Algorithms

Before commenting further on the algorithm, let us see how its skeleton can be described using MANIFOLD. The complete listing of the program appears as Listing 2.

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

- the area can be filled without ambiguities, in which case `TestAndColor` raises the event `done`, fills the area with the calculated color(s) and terminates; or

- the area cannot be filled without ambiguities, in which case `TestAndColor` raises the event `subdivide` and terminates.

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

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

In the declaration part of `Warnock`, two instances of the atomic processes described above are declared. This means that the manifold `Warnock` now has a reference for these processes and can, therefore, involve them into several parallel pipelines, if necessary. The additional two declarations concern two "utility" processes (part of the standard environment of the MANIFOLD system) which are able to store some units and, if the type of the units permit, to perform some elementary arithmetic on them.

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

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

The manifold is suspended in this block and must receive an external event to change its state. According to our specifications, these external events may be either `subdivide` or `done`, depending on the result of the test performed on the local area. (Note that although many instances of `TestAndColor` may be active and raise the events `subdivide` and/or `done`, the only instance of `TestAndColor` visible to an instance of `Warnock` is its locally declared one. This is why the other events raised by other instances cause no confusion.)

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

- the content of the variable `v` is transferred to the area divider, and

- the four handles for the generated sub-areas are forwarded, respectively, to the four (recursive) instances of `Warnock` [6].

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

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

Note that there are two blocks in `Warnock` with the same label `terminate`. The reason is to avoid a race condition which can happen in the block for `subdivide`. Indeed, it is perfectly possible that `divide_area` is still busy calculating, e.g., the fourth sub-area while the `Warnock` instance for, say, the first sub-area already terminates. Obviously, `Warnock` must *not* (yet) change state but it must not ignore the event either (otherwise a non-termination will occur). By putting a separate block for `terminate` with the statement `save` we make sure that the event is neither lost nor preempts the state `subdivide`.

If no subdivision is necessary, `Warnock` makes a state transition to the block labeled `done`, which does an immediate state transition again. This, finally, leads to the termination of the manifold. Strictly speaking, it is not necessary to have a separate intermediary state in this case (a block may have multiple labels). However, when our example is extended further in the next sections, having a separate state will prove to be beneficial.

5.2 Analysis of the Program

Warnock's algorithm is an example of the *image space algorithms* in computer graphics. These algorithms are primarily concerned with images and compute the attributes of each pixel on the screen. Resolution of the relationships among objects in a scene becomes a secondary concern. On the other hand, *object space algorithms* are concerned with the properties of and relationships among the objects in a scene and compute an image only after these relationships

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

are determined. Warnock's algorithm is not very much in use today. Indeed, if the hidden surface removal is to be performed in image space, availability of powerful hardware makes other methods (primarily, the so called Z-buffer method) more attractive. Whether or not this preference will persist in the future is a matter of debate and its details are far beyond the scope of this paper.

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

Although, obviously, the principle of recursive subdivision is not restricted to computer graphics, its popularity within the computer graphics community seems to be related to the special nature of the field. Indeed, the geometric nature of the underlying problems often gives very clear clues for how to perform the subdivisions and how to control its recursion in an optimal way. Thus, the application of recursive subdivision is very natural in working with synthetic or digital images. Apart from Warnock's algorithm for removal of hidden surfaces, similar or more elaborate approaches can be used in calculating and/or displaying spline curves or surfaces [33], perform calculations on CSG[7] objects using quadtrees [32], digital filtering of images, global histogramming of digital images [37], parallelizing such time consuming rendering procedures as ray tracing [35] especially on CSG objects, performing the calculations necessary to visualize volumes [38], etc.

What is the role of MANIFOLD in this respect? Looking at the program on Listing 2, it is clear that MANIFOLD has a real expressive power in describing the skeleton of a recursive subdivision algorithm. Note that the atomic processes used by the program are defined in a fairly abstract way; any atomic process, abiding to these specifications, can be "plugged in" the same MANIFOLD program to serve a different application. Although most of the algorithms listed above require a more sophisticated version of the algorithm (and we will elaborate on these improvements in the following sections), we believe the listing commented in detail in §5.1.1 makes the essential point: that using MANIFOLD it is possible to describe in a very concise and declarative form, the primary communication skeleton of a certain class of systems or algorithms without bothering with their computational details.

These examples also reveal another general and more important characteristic: most of the algorithms cited above were, originally, *not* meant for parallel

[7] Constructive Solid Geometry

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

Nevertheless, MANIFOLD is useful for expressing the communications and control structure of these algorithms, even if the actual implementation of a MANIFOLD system may run only on a conventional single-processor computer supporting simulated parallelism only (as in the case of our first experimental implementation based on Concurrent C++). This seems to be a clear case of a more general principle: it may be extremely beneficial to use mental models which use concurrency, communication, and coordination, as natural paradigms to grasp the essence of a problem and/or of an algorithm. Concurrency need not be considered a "necessary curse," as perceived by a large number of practitioners. On the contrary, it is often very helpful in conceptual simplification of the problem at hand. Gelernter and Carriero ([4]) stress that:

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

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

5.3 Improvements to the Program

In this section we present enhancements to the MANIFOLD program described in §5.1 and evolve a better framework for expressing different version of the adaptive recursive algorithms mentioned above. The improvement to the program is done in two steps. First, the restriction of a fixed number of subdivisions is relaxed. Second, we allow the possibility of backward control in the recursive processes; i.e., allow a parent to wait for and use the results produced by its children.

5.3.1 Variable Number of Subdivisions

The program in §5.1 has an obvious restriction that may make it inappropriate for general use in other applications. This program has a "hardwired" subdivision feature: each area must be subdivided into exactly four sub-areas.

Although this is natural in the case of Warnock's algorithm, and it is trivial to change the number four, imposing any fixed number by itself is a constraint that hinders more general usability of this program for other applications. In particular, a more general class of recursive subdivision algorithms use an adaptive subdivision scheme wherein the number of subdivisions at each level of recursion, as well as the subdivision boundaries, may depend on the data and thus cannot be predetermined.

In this section, we present an improvement to the **MANIFOLD** program of §5.1 that allows the number of subdivisions to be determined dynamically at each level. To put our revised **MANIFOLD** program in the right perspective, we remark that a later version of Warnock's algorithm, called the Weiler-Atherton algorithm (see [30]), subdivides the screen along polygon boundaries, rather than along the two mid-lines of the screen. Clearly, the Weiler-Atherton algorithm requires a variable number of subdivisions.

The revised **MANIFOLD** program now consists of two parts: the one in Listing 3 and the one in Listing 4. The first part is, in fact, a somewhat simplified version of the program in Listing 2. We have changed the specification of the `DivideArea` process: what we require now is that when `DivideArea` receives an area handle, it produces a series of area handles (one for each sub-area) on its standard output and then terminates.

The recursive step is now hidden into a separate manifold process, called `Distribute`. This program appears in Listing 4 and will be explained later. As far as the manifold `Warnock`[8] is concerned, `Distribute` receives the area handles for this level's sub-areas on its standard input and, somehow, takes care of the recursion. A separate pipeline is set up in the block labeled `subdivide` to send these handles to a local instance of `Distribute`. Note that now it is `Distribute` that is responsible for proper termination; consequently, the counter `n` has disappeared from `Warnock`.

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

v → divide_area, divide_area → distribute

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

v → divide_area → distribute

that may be mistaken as their equivalent. While the two alternatives work the same as long as the flow of units are concerned, they indeed behave quite differently on termination. In **MANIFOLD**, a pipeline breaks up as soon as any one of its processes terminates or raises a special event `break`. In case of our single pipeline, this can happen as soon as the process `v` has delivered its value,

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

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

Warnock()
{
        process test_and_color   is TestAndColor.
        process v                is variable.
        process divide_area      is DivideArea.
        process distribute       is Distribute.


      start:
         ( activate v,
           activate test_and_color,
           input → (→ test_and_color,→ v),
         ).
      subdivide:
         ( activate divide_area,
           activate distribute,
           v → divide_area,
           divide_area → distribute
         );
           do end.
      done:
           do end.
      end:
           deactivate  parent.
}
```

Listing 3. Program with variable area subdivision; part I.

```
Distribute()
{
        port          in internal.
        process n     is variable.

      start:
          ( activate n, n = 0 );   do main_cycle.
      main_cycle:
          getunit(input) → internal;
          do next_area.
      next_area:
          (n = n + 1, getunit(internal) → Warnock);
          do main_cycle.
      terminate:
          save.
      disconnected.input: wait_for_death:
          void.
      terminate:
          n = n - 1;
          if( n == 0,   do end,   do wait_for_death ).
      end: .
}
```

Listing 4. Program with variable area subdivision; part II.

which can result in the breakup of the connection between `divide_area` and `distribute`, if they are all in the same pipeline. Having them in two separate pipelines in a group, as in the state `subdivide` in Listing 3, ensures that such premature breakups will not happen. (In MANIFOLD, a group terminates when all of its members are broken up.)

A number of constructs used in the original Warnock program (Listing 2) now appear in `Distribute` (see Listing 4). Using the counter `n` to count the number of activated child processes, as well as handling of their deactivations, are exactly the same as before. The primary difference is, of course, in the handling of a variable number of incoming units.

The `Distribute` manifold uses the built-in pseudo-process[9] `getunit` which acts as follows:

- it is suspended on a port of the caller, as long as there is no unit available for delivery on the port;
- when a unit is or becomes available, this unit is sent out onto the output port of `getunit` and the pseudo-process terminates (*i.e.*, the pipelines in

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

which it is involved are broken);

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

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

1. When a unit arrives (actually, an area handle from the `DivideArea` process, although `Distribute` does not know the origin of the unit), it is picked and put into the `internal` port. Next, an internal state transition is made which results in the activation of a new instance of `Warnock`.

2. When there is no unit in the buffer of the `input` port of `Distribute`, *and* this port is no longer connected to any other port (which means that the connecting `DivideArea` process has terminated), `getunit` raises a `disconnected` event (which results in the preemption of the current state).

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

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

`getunit`(input) → Warnock

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

5.3.2 Handling Return Values

The algorithms that can use the MANIFOLD programs in §5.1.1 and §5.3.1 are constrained by another limitation. Once the recursive branches of the algorithm start off, they do not communicate with their parents any more (or, to be precise, they have no communication expressed by the MANIFOLD program). This is fine (indeed, desirable) with the original Warnock's algorithm: the subareas of a screen can be filled independently of one another, and a parent has

```
Permanent(inp,outp)
 port  out inp.
 port  in  outp.
{
      start:
          inp → outp.
}


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

Listing 5. Programs to set up permanent pipelines.

no reason to stay alive and take up resources once its children are started. However, this is obviously inappropriate in a number of other applications.

Once again, a slight improvement on Warnock's algorithm serves as a good motivating example. In §5.1.1 we assumed that the recursion stops when the size of an area reaches the size of a pixel. Strictly speaking, this assumption is true, but it results in aliasing problems (*i.e.*, the appearance of "staircase" polygon edges and unpleasant color transitions). One of the anti-aliasing methods which can be easily used with Warnock's algorithm requires the recursion to go on at least one more step, to the level of sub-pixels. The color properties computed at sub-pixel levels are then returned to the pixel level routines, which in turn average them out to calculated the color of their pixels.

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

Listings 6 and 7 show the new version of our MANIFOLD program; they correspond to the Listings 3 and 4, respectively. As in the previous section, we only highlight the differences between the old and the new versions in this section.

The specification of the atomic process `TestAndColor` is now slightly different. Representing the "bottom" of the recursion, this atomic process is also required to return a value to be forwarded to the upper level (e.g., the color value, in the anti-aliasing example). Additionally, a new process, called `Merge`, is defined: this process receives "values" on its standard input port and

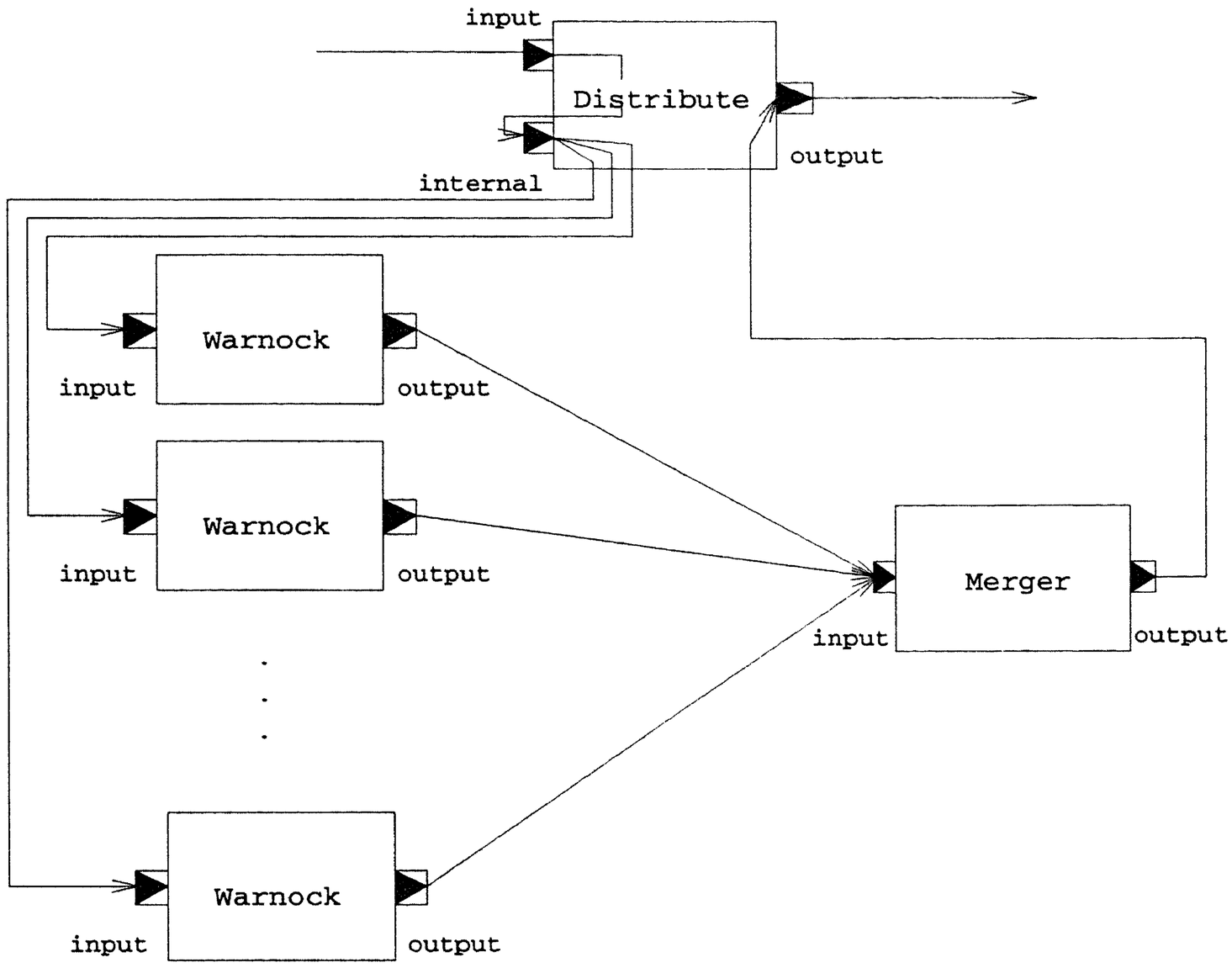

FIGURE 4. A pictorial representation of the manifold `Distribute`.

"merges" them into one value delivered on its output port (in our anti-aliasing example, this process calculates the average of color values it receives)[10]. What `Merge` does is to read an unknown number of units from its standard input, compute their "merged" result (e.g., their average), write it out to its standard output, and terminate. It detects the equivalent of an end-of-file on its standard input (if it is in fact an atomic process), or reacts to a `disconnected` event (if it is another manifold), to realize that it has received all input units it is expected to process.

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

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

transfers it to the output port. Apart from these differences, the new version of the `Warnock` manifold has an identical behavior to the previous one.

The new version of `Distribute` uses two small manifolds of Listing 5 which are usually part of the standard MANIFOLD environment. The meaning of these two manifolds is clear: they set up pipelines which remain unbroken as long as members of the pipeline are active. Remember that, according to the specification of MANIFOLD, if a manifold leaves a state, all pipelines set up in that state are broken before leaving. The use of the `Permanent` manifolds is to avoid this breakup.

`Distribute` now sets up a slightly more complicated network of connections. Figure 4 is a pictorial representation of these connections. In the startup state of `Distribute`, a permanent connection (using the first version of `Permanent` in Listing 5) is set up from the output port of `merge` (an instance of `Merge`) to the output port of the running instance of `Distribute`. Note that this is a perfectly legitimate setup: ports of a process instance (e.g., `merge`) can be connected in pipelines even before the process is activated. Additionally, another pseudo-process, `guard`, is activated. The role of this pseudo-process is to raise an event (named in its argument) if a unit appears on its designated port.

The pipelines set up in the state `next_area` are slightly different: the connection between each new instance of `Warnock` and `merge` is set up using `Permanent`, to prevent its breakup in case of a state transition. This is where the second version of `Permanent` is used (note that the different signatures of the two `Permanent` manifolds disambiguates the choice).

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

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

Before terminating, `Distribute` must not only wait for all of its local instances of `Warnock` to terminate, but it must also make sure that the output value of `merge` has actually arrived and is transferred out of its output port. This is done by the event `output_arrived` which is raised by `guard`. Note the use of the `save` action for this event; its role is the same as for the event `terminate`, as explained earlier.

6 Related Work

The general concerns which led to the design of MANIFOLD are not new. The CODE system [39, 40] 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 the case of Ada, for example, these are the language constructs for rendezvous. In the 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 [41] or Herath et. al [42] 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 much higher: it is the equivalent of an arbitrary process. In this respect, there is a stronger resemblance between MANIFOLD and such higher level dataflow environments like the so called Task Level Dataflow Language (TDFL) of Suhler et al. [43].

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 contains 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 only once at 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 [44]. This experience shows that the time required for the 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 [43] 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, 6] were also clearly concerned with coordination of communications and 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 intended to express computation of a general nature by itself. 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, voluntarily placed into and picked up 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 [2, 3] which served also as a basis for the development of the language Occam [12]. Clearly not a programming language by itself, 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 the behavior of concurrent systems. However, it lacks some useful properties for constructing real systems. For example, 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. The communications between a process and its environment are an integral part of its semantics in CSP. Occam inherits both of these characteristics from CSP. In contrast, MANIFOLD clearly separates the functionality of a process from the concerns about its communication with its environment, placing the latter entirely outside of the process itself. The responsibility for establishing and managing the interactions among processes in a parallel system is completely

taken over by manifolds. A manifold orchestrates the interactions among a set of processes (some of which may be other manifolds) without their knowledge.

Another significant difference between CSP (and Occam) and MANIFOLD is that all communication in CSP is synchronous, whereas everything (including events) in MANIFOLD are asynchronous. Furthermore, the data-flow-like means of communication and its associated control mechanisms are deemed especially important in MANIFOLD, for which it has first class support through special language constructs.

An important distinction between MANIFOLD and many other systems (e.g., Occam) is that they generally fix the number of processes, the topology of the communication network, and the potential connectivity of each individual process at compile time. MANIFOLD processes, on the other hand, do not know who they are connected to, can be created dynamically, and can be dynamically connected/disconnected to/from other processes while they are running.

An ISO standard for open systems interconnection is the language LOTOS (Language Of Temporal Ordering Specification)[45, 46, 47]. It is a formal description technique based on the temporal ordering of observable behavior of concurrent processes. The LOTOS language is based on a concurrency model of parallelism described by Milner, called CCS (see [1]). (CCS is similar in its flavor to CSP, although there are significant differences between them.) The atomic form of interaction in LOTOS is through events which, as in CSP, synchronize their participating processes. The behavior of a process in LOTOS is described in *behavior expressions* that are composed of simpler behaviors using sequential and choice operators. LOTOS includes many other language constructs, e.g., to support abstract data types. Nevertheless, its view of parallelism is essentially the same as CSP.

As mentioned in §2, the complexity of using languages like Ada, Occam, and Concurrent C++ can become overwhelming in highly parallel applications that require dynamically changing communication patterns. The MANIFOLD environment offers an abstraction of the necessary communication facilities which can then be built on top of a distributed programming language like Concurrent C++, or Ada.

7 Directions for Further Work

More experience is needed with a fully operational MANIFOLD system to evaluate its potentials and the adequacy of its constructs in real, practical applications. Nevertheless, it is already clear that certain changes and extensions to the MANIFOLD language can have a positive impact on its use in large and complex systems. Several such improvements are currently in our list, of which we mention only a few major ones here.

For instance, the notion of *derived manifolds* may be a useful extension to the language. This concept leads to a hierarchy of manifold definitions with inheritance, analogous to the class hierarchies in object oriented languages. Language support for such syntactic conveniences seem to be quite useful in

large software developments.

An issue that we have encountered a few times in our examples is a need for *directed events*. Strictly speaking, the concept of event in the MANIFOLD model is, of course, contrary to the notion of *directed events*, because MANIFOLD events are broadcast and can be picked up by any process in the environment. We do not yet know how important the need for *directed events* is, because we have been able to do without them so far. Nevertheless, the effect of *directed events* can be supported at the language level in MANIFOLD by introducing proper constructs to explicitly control the observability of event sources and/or the preemption sets of manifolds. Observability and preemption sets are both defined implicitly in the current MANIFOLD language: they are derived by the compiler from the source code. Symmetric to the way in which a third party process can define streams between two other processes in the current MANIFOLD language, new language constructs can allow processes to define and modify observability and/or preemption sets.

8 Conclusions

This paper is an overview of the MANIFOLD system and sketches the highlights of its implementation. More experience is still necessary to thoroughly evaluate the practical usefulness of MANIFOLD. However, our experience so far indicates that MANIFOLD is well suited for describing complex systems of cooperating parallel processes.

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

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

The recursive algorithms as well as the example related to the IRIS Explorer system, described in MANIFOLD, are only small-scale albeit important practical examples for the usage of MANIFOLD. However, MANIFOLD can be used to implement more complex interactions, e.g., in a user interface toolkit, as well. For example, in a separate paper, [25], we describe an implementation of the GKS logical input device in MANIFOLD.

In our view, massive parallel systems and the current trend in computer technology toward *computing farms* open new horizons for large applications and present new challenges for software technology. Classical views of parallelism in programming languages that are based on extensions of the sequential programming paradigm are ill-suited to meet this challenge. We also believe that it is counter-productive to base programming paradigms for computing farms and massively parallel systems solely on strictly synchronous communication. Many of the ideas underlying the MANIFOLD system, if not the present MANIFOLD language itself, seem promising towards this goal.

Acknowledgment

We are thankful for the direct and indirect contributions of all members of the MANIFOLD group at CWI. In particular, Paul ten Hagen inspired some of the original concerns and motivation for MANIFOLD. Kees Blom helped to refine the formal syntax for the MANIFOLD language and produced its first compiler. Eric Rutten is developing the formal semantics of MANIFOLD. Dirk Soede's exercises in MANIFOLD, his ongoing work with Anco Smit on a visual interface to MANIFOLD, and especially Freek Burger's programming work on the MANIFOLD run-time system are also much appreciated.

Last, but not least, we thank the comments of our paper's anonymous referees. We took the liberty of paraphrasing some of the comments made by one referee in our revised conclusion. The same referee also encouraged us to change our original example of a window manager. Motivated by his suggestion, we worked out the present set of examples, which we believe show the concepts and relevance of MANIFOLD much better.

References

1. R. Milner, *Communication and Concurrency.* Prentice Hall International Series in Computer Science, New Jersey: Prentice Hall, 1989.
2. C. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, August 1978.
3. C. Hoare, *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science, New Jersey: Prentice-Hall, 1985.
4. D. Gelernter and N. Carriero, "Coordination languages and their significance," *Communication of the ACM*, vol. 35, pp. 97–107, February 1992.
5. N. Carriero and D. Gelenter, "LINDA in context," *Communications of the ACM*, vol. 32, pp. 444–458, 1989.

6. W. Leler, "LINDA meets UNIX," *IEEE Computer*, vol. 23, pp. 43–54, February 1990.
7. F. Arbab, "Specification of manifold," Tech. Rep. to appear, Centrum voor Wiskunde en Informatica, Amsterdam, 1992.
8. United States Department of Defense, *Reference Manual for the Ada Programming Language*, November 1980.
9. N. Gehani and W. Roome, "Concurrent C," *Software — Practice and Experience*, vol. 16, pp. 821–844, 1986.
10. N. Gehani and W. Roome, *The Concurrent C Programming Language*. Summit NJ: Silicon Press, 1989.
11. N. Gehani and W. Roome, "Concurrent C++: Concurrent programming with class(es)," *Software — Practice and Experience*, vol. 18, pp. 1157–1177, 1988.
12. INMOS Ltd., *OCCAM 2, Reference Manual*. Series in Computer Science, London — Sydney — Toronto — New Delhi — Tokyo: Prentice-Hall, 1988.
13. H. Bal, J. Steiner, and A. Tanenbaum, "Programming languages for distributed computing systems," *ACM Computing Surveys*, vol. 21, pp. 261–322, 1989.
14. W. Roberts, M. Slater, K. Drake, A. Simmins, A. Davidson, and P. Williams, "First impression of NeWS," *Computer Graphics Forum*, vol. 7, pp. 39–58, 1988.
15. T. Doeppner Jr., "A threads tutorial," Tech. Rep. CS-87-06, Brown University, 1988.
16. P. Buhr and R. Stroobosscher, "The μSystem: Providing light-weight concurrency on shared-memory multiprocessor computers running UNIX," *Software — Practice and Experience*, vol. 20, pp. 929–964, 1990.
17. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for UNIX development," in *Proceedings of the Summer Usenix Conference*, (Atlanta, GA), July 1986.
18. D. Black, "Scheduling support for concurrency and parallelism in the Mach operating system," *IEEE Computer*, vol. 23, pp. 35–43, May 1990.
19. SUN Microsystems, *SunOS Manuals, Lightweight Processes*, revision A ed., 1990.
20. F. Arbab and I. Herman, "Examples in Manifold," Tech. Rep. CS-R9066, Centrum voor Wiskunde en Informatica, Amsterdam, 1990.
21. F. Arbab and I. Herman, "Manifold: A language for specification of interprocess communication," in *Proceedings of the EurOpen Autumn Conference* (A. Finlay, ed.), (Budapest), pp. 127–144, September 1991.
22. F. Arbab, I. Herman, and P. Spilling, "Interaction management of a window manager in Manifold," in *Computing and Information ICCI'92* (W. Koczkodaj, P. Lauer, and A. Toptsis, eds.), (Toronto), IEEE Press, June 1992.
23. I. Herman and F. Arbab, "More examples in examples in Manifold," Tech. Rep. CS-R9214, Centrum voor Wiskunde en Informatica, Amsterdam, 1992.
24. H. Schouten and P. ten Hagen, "Dialogue cell resource model and basic

dialogue cells," *Computer Graphics Forum*, vol. 7, no. 3, pp. 311–322, 1988.
25. D. Soede, F. Arbab, I. Herman, and P. ten Hagen, "The GKS input model in manifold," *Computer Graphics Forum*, vol. 10, pp. 209–224, September 1991.
26. J. Peterson, R. Bogart, and S. Thomas, "The Utah Raster Toolkit," in *Proceedings of the Usenix Workshop on Graphics*, (Monterey, California), 1986.
27. S. Dyer, "A dataflow toolkit for visualization," *IEEE Computer Graphics & Applications*, vol. 10, July 1990.
28. C. Upson, "Scientific visualization environments for the computational sciences," in *Proceedings of the* 34^{th} *IEEE Computer Society International Conference*, (San Francisco), March 1989.
29. Silicon Graphics, Inc., "IRIS Explorer user's guide," tech. rep., Silicon Graphics, Inc., Mountain View, California, 1991.
30. J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics — Principles and Practice.* Reading: Adison–Wesley, 1990.
31. R. Gonzalez and P. Wintz, *Digital Image Processing.* Reading, Massachusetts: Addison–Wesley, 1983.
32. W. Bronsvoort, F. Jansen, and F. Post, "Design and display of solid models," in *Advances in Computer Graphics VI* (G. Garcia and I. Herman, eds.), Heidelberg: EurographicSeminar Series, Springer Verlag, 1991.
33. R. Bartels, J. Beatty, and B. Barsky, *An Introduction to Splines for Use in Computer Graphics & Geometric Modelling.* Los Altos, California: Morgan Kaufmann Publishers, Inc., 1987.
34. M. Cohen and J. Painter, "State of the art in image synthesis," in *Advances in Computer Graphics VI* (G. Garcia and I. Herman, eds.), Heidelberg: EurographicSeminar Series, Springer Verlag, 1991.
35. S. Green, *Parallel Processing for Computer Graphics.* Research Monographs in Parallel and Distributed Computing, London: Pitman, 1991.
36. F. Crow, "Parallel computing for graphics," in *Advances in Computer Graphics VI* (G. Garcia and I. Herman, eds.), Heidelberg: EurographicSeminar Series, Springer Verlag, 1991.
37. H. Siegel, J. Armstrong, and D. Watson, "Mapping computer-vision related tasks onto reconfigurable parallel processing systems," *IEEE Computer*, vol. 25, pp. 54–64, February 1992.
38. D. Laur and P. Hanrahan, "Hierarchical splatting: Progressive refinement algorithm for volume rendering," *Computer Graphics (SIGGRAPH'91)*, vol. 25, pp. 285–288, July 1991.
39. J. Browne, M. Azam, and S. Sobek, "CODE: A unified approach to parallel programming," IEEE *Software*, pp. 10–18, July 1989.
40. J. Browne, T. Lee, and J. Werth, "Experimental evaluation of a reusability-oriented parallel programming environment," *IEEE Transaction on Software Engineering*, vol. 16, pp. 111–120, 1990.
41. A. Veen, "Dataflow machine architecture," *ACM Computing Surveys*, vol. 18, pp. 365–396, 1986.

42. J. Herath, N. Saiko, and T. Yuba, "Dataflow computing models, languages and machines for intelligence computations," *IEEE Transactions on Software Engineering*, vol. 14, pp. 1805–1828, 1988.
43. P. Suhler, J. Bitwas, K. Korner, and J. Browne, "TDFL: A task-level dataflow language," *Journal of Parallel and Distributed Computing*, vol. 9, pp. 103–115, 1990.
44. P. ten Hagen, I. Herman, and J. de Vries, "A dataflow graphics workstation," *Computers and Graphics*, vol. 14, pp. 83–93, 1990.
45. International Organization for Standardization, Geneva, *Information Processing Systems — Open Systems Interconnections — LOTOS (Formal Description Technique based on the temporal ordering of observational behaviour) ISO/DIS 8807*, March 1988.
46. T. Bolognesi and E. Brinksma, "Introduction to the ISO specification language LOTOS," *Computer Networds and ISDN Systems*, vol. 14, pp. 25–59, 1986.
47. E. Brinksma, "A tutorial in LOTOS," in *Protocol Specification, Testing, and Verification V* (M. Diaz, ed.), pp. 171–194, Amsterdam: North-Holland, 1986.

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

Warnock()
{
        process test_and_color   is TestAndColor.
        process v                is variable.
        process divide_area      is DivideArea.
        process distribute       is Distribute.
        port  in  internal.

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

Listing 6. Program with return values I.

```
Distribute()
{
 port              in internal.
 process n         is variable.
 process merge     is Merger.

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

Listing 7. Program with return values II.