1991

F. Arbab, I. Herman, P. Spilling

An overview of Manifold and its implementation

Computer Science/Department of Interactive Systems Report CS-R9142 September

CWI, nationaal instituut voor onderzoek op het gebied van wiskunde en informatica

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

An Overview of Manifold and its Implementation

F. Arbab, I. Herman, P. Spilling
CWI
Department of Interactive Systems
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
Tel.: +31-20-592.4058, +31-20-592.4163, +31-20-592.4164
Fax.: +31-20-592.4199
Email: farhad@cwi.nl, ivan@cwi.nl, per@cwi.nl

SUMMARY

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 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 describe a simple window system and show how the communications between clients running on different windows and a window server can be described in this language.

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

1 Introduction

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

MANIFOLD is a language whose sole purpose is to manage complex interconnections among independent, concurrent processes. 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 [1]. In this paper, we give an overview of the MANIFOLD language and its implementation and present the skeleton of a window management system as an example of its application. We summarize only enough of the description of the MANIFOLD model and the language here to make the example and the significant implementation issues presented herein understandable.

The rest of this paper is organized as follows. In §2 the main motivations behind the 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 discuss the major issues in our implementation of MANIFOLD and sketch our solution approach. Our implementation scheme is based on the notion of an abstract machine that makes no assumptions about the actual architecture of the underlying parallel system. However, the material in §5 goes beyond this abstract machine model, and we explain the highlights of our real

Report CS-R9142
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

implementation (which makes a liberal use of shared memory) as well. In §6, we mention some of the similarities and major differences between MANIFOLD and certain related systems and models for parallel computing. In §7, we present a somewhat elaborate example of a window system in MANIFOLD. The purpose of this example is to illustrate the use of some of the features in the MANIFOLD language and to demonstrate the general applicability of MANIFOLD concepts. In §8 we mention some of the extensions and enhancements we plan to make to the MANIFOLD system in the future. Finally, §9 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 [2, 3], Concurrent C [4, 5], Concurrent C++ [6], Occam [7, 8] and many others (the interested reader may consult e.g., the survey of Bal et al. [9] 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 it 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 undeniable technical superiority of the NeWS windowing system over X Windows [10]; 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 [11], the so-called μSystem [12], 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 [13, 14] of Carnegie Mellon, or SunOS [15]).

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 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 [1]. Other reports contain more examples of the use of the MANIFOLD language [16, 17].

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 to observe 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.

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 constructed

[1] Some authors prefer the term "pseudo-parallelism" for such or similar forms of parallelism, see again Bal et al [9].

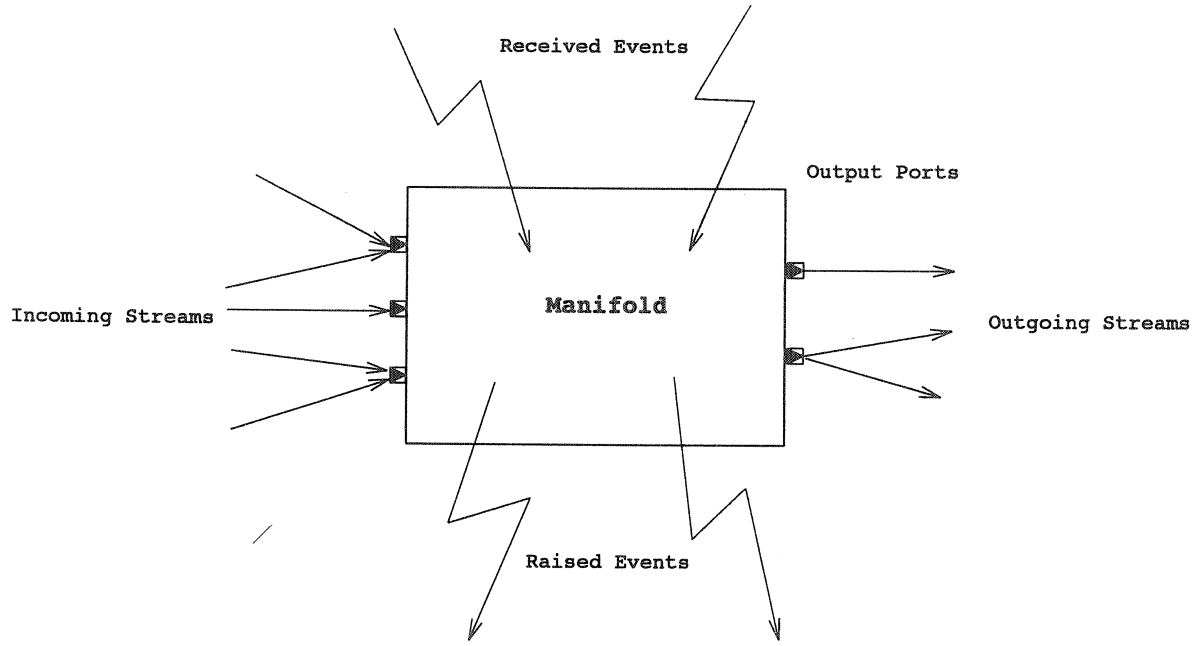


Figure 1. The model of a process in Manifold

and removed dynamically between ports of the processes that are to exchange some information. 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 3). 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.

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 such a broadcast event. In practice, usually only a few processes pick up occurrences of each event, because only they are "tuned in" to their sources. Occurrences of the same event from the same source can override each other from the point of view of some observer processes, depending on the difference between 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 manifold, see Figure 2 which shows the MANIFOLD source code for a not-particularly-interesting program.[2] 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 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 3 shows the connections set up by the manifold process `example` in Figure 2, while it is in the handling block for the event `e1` (for the details of event handling see §3.2). Figure 4 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 (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

---

[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.

[3] In fact, the semicolon operator is only an infix *manner call* (see §3.5) rather than an independent concept in MANIFOLD. However,

```
// 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 3):
      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 4):
      e2: C → B.
}
```

Figure 2. An example for a manifold process

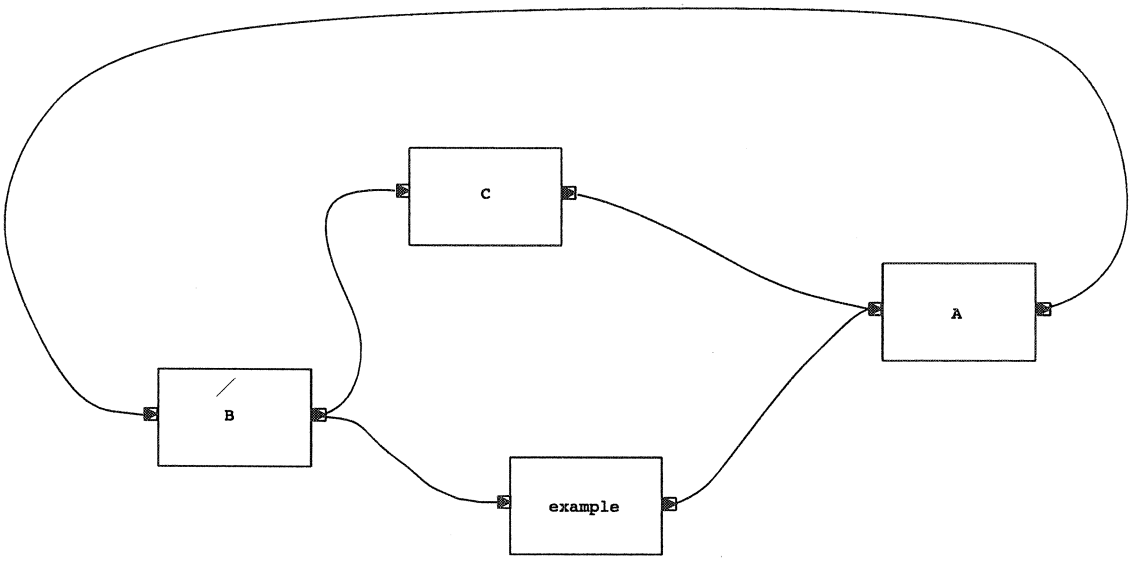


Figure 3. Connections set up by the manifold `example` *on event* `e1`

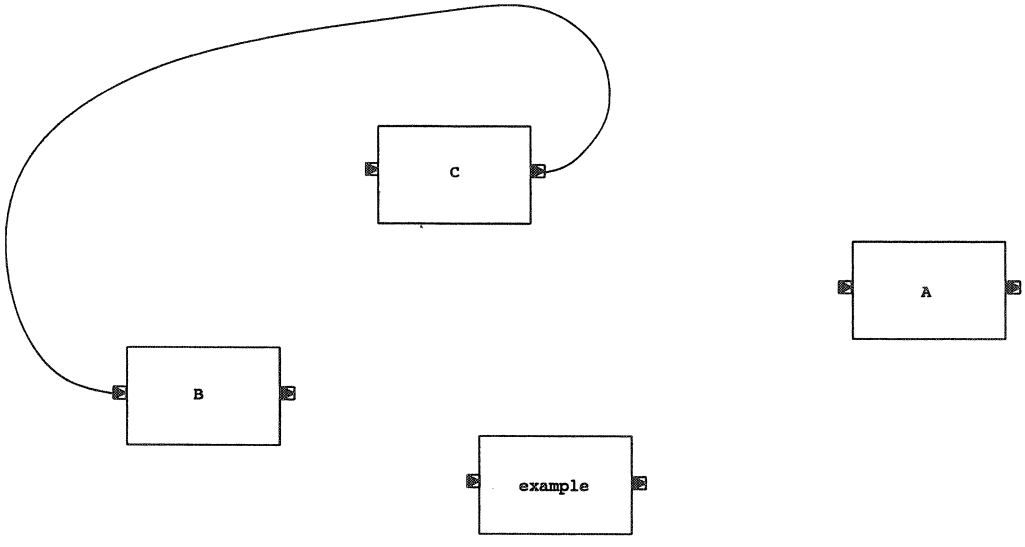


Figure 4. Connections set up by the manifold `example` *on event* `e2`

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` 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 [18, 19]. 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. 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.

---

for our purposes, we can assume it to be the equivalent of the sequential composition operator of a language like Pascal.

3.4 Visibility of Event Sources

Every process instance or port defined or used anywhere in a manner 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.

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 [16, 17]. 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., [20, 21]) 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 [22]; 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 [23] are 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 Center [24], the commercially available AVS Visualization Package of Stardent Computer Ltd. [25], and others, work on the basis of inter-connecting a whole

[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.

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 Implementation

When we began the implementation of the MANIFOLD system, we actually had two goals in mind. Quite naturally, we wanted to have a running experimental system which would serve as a basis for further applications, measurement, and evaluation; it is only through using such a system that one can really evaluate the usefulness and user-friendliness of such a language.

Our other, and not less important, goal was to have an existence proof of the feasibility of the computational model underlying the MANIFOLD language, as well as the language itself. It is of course true that the development of formal semantics for the MANIFOLD language is necessary (and this work is under way) but pragmatically, this would not be enough. Had the implementation effort raised serious fundamental issues or other kinds of difficulties, it would have shown that there was something erroneous or at least impractical in the specification of either the computing model or the language (or both).

This second goal had the practical consequence that we tried to find an implementation scheme that reflects the computational model directly, as far as possible. For instance, we ruled out any implementation scheme that involved some central management of the running manifolds and streams in favor of genuinely distributed schemes where each manifold is mapped directly onto a real process. (Of course, any actual implementation environment imposes some kind of central administration at the level of parallel hardware and/or the software supporting the concurrent processes. However, this is true in general and one should forget about it at a higher level of abstraction.)

In what follows, we give an overview of our implementation scheme without going too deeply into its environment dependent details. The interesting aspect of this description is that it gives general guidelines for how the MANIFOLD system can be implemented, provided that a support environment for some sort of cheap processes is available. It is only in §5.2 that we describe how the mapping of these general ideas onto a "real" implementation is done in our present experimental system.

5.1 The Manifold Stack Machine

The implementation model of the MANIFOLD system is centered around an abstract machine which we call the *Manifold Stack Machine* (MSM for short). This is a stack machine with a low-level instruction set. Although we do not define the exact low-level instruction set of MSM, we define a macro-assembly level instruction set which we use in our implementation.

An MSM contains the following components (see also Figure 5).

1. A *Manifold Processor* (MP)

   As its name suggests, this is the active entity within an MSM. Logically speaking, MP also contains a memory area which stores the instructions for the processor; this is not shown in the picture, and we do not deal with this detail in what follows. The only feature we make use of in this paper is that the processor can somehow be programmed.

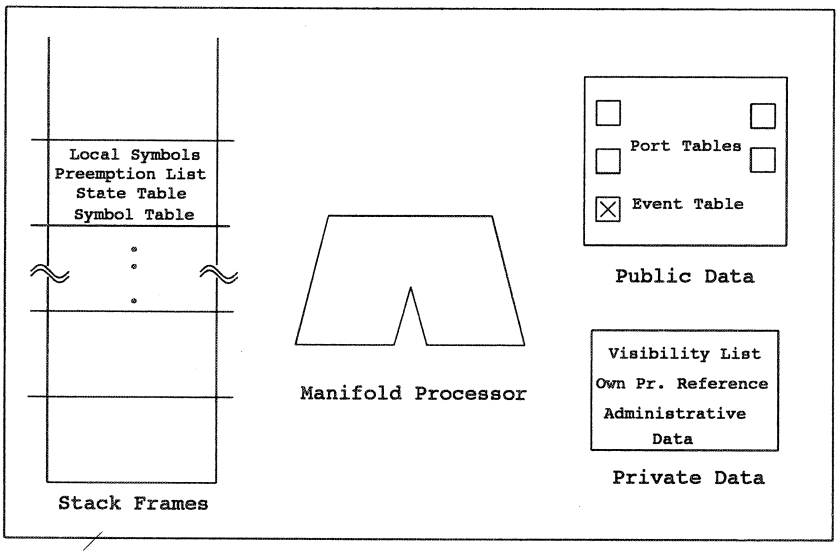


Figure 5. The Manifold Stack Machine (MSM)

2. A *Public Data Area* (PlD)

   This is a protected memory segment which can also be accessed by other processes. By "*protected*" we mean that a defense mechanism is provided to avoid concurrent access. Precisely how this is done is a matter of the underlying system environment. The PlD can be a shared memory area with some kind of mutual exclusion mechanism, or it can be yet another logical process to which transactions are sent. In what follows, we simply assume that a mechanism of abstract *operations* can be defined through which secured access and manipulation of the data contained in PlD is possible.

   A PlD contains the description of all the ports assigned to a given manifold instance as well as tables for all events it is sensitive to (these sub-units are depicted by small rectangles in all figures). Ports within the PlDs are separate structures; there is a separate instance of such a structure for each port defined in its corresponding manifold. Each such structure contains input and output queues for holding units as well as a table containing the process and port references for all other MSMs this port is connected to by a stream.

3. A *Private Data Area* (PrD)

   This is a plain memory segment which can be accessed by the MP only.

4. A *Stack* holding *Stack Frames* (SF)

   A stack frame holds all the data necessary to define the state of a manifold process. It defines all the events it is sensitive to, its visible event sources and the way in which it reacts to their occurrences. The top-most frame on the stack corresponds to the actual current state of a manifold process. Calling a manner results in pushing a new stack frame onto the stack and leaving a manner (preemptively, or due to normal return) results in popping its stack frame. This data segment is also private, in the sense that no other MSM can access its contents.

The main processing performed by an MSM consists of:

1. creation and initialization of all general lists on the stack, the private and the public memory areas;
2. setting up and/or deleting process reference lists for port and event structures;
3. performing unit transmission;

4. reacting on events and performing transitions to other blocks, if necessary;

5. performing primitive actions;

6. handling manner calls and terminations.

More details on these actions appear in the following sections. What is essential to note here is that the macro-assembly-level instructions defined for the MSM cover these and only these tasks. A fairly traditional compiler can then compile a MANIFOLD program with this macro-assembly as a target language. Although the precise description of these instructions is beyond the scope of this paper, it is worth mentioning that the major and the most complicated part of implementing these instructions concerns proper handling of different lists. In this sense, an MSM is not much more complex than, e.g., an abstract machine for Lisp.

One of the main activities of an MSM is transport of units. The MP of a manifold process must scan all of its port tables and send the units it has received in each port to every port it is currently connected to. The other main activity of an MSM is to react on event occurrences it receives. The basic operation scheme of an MSM consists of a cycle of suspension (as long as there are no units to transport and no event occurrences to react to), followed by event handling (if an interesting event occurrence has been received), or transport of the received units.

Once created, an MSM first initializes itself and then suspends itself waiting for an event occurrence, or for the arrival of a unit on any one of its ports. Once active, it first checks to see if there are any event occurrences to handle. If so, it selects and handles one of the pending event occurrences at this point in some non-deterministic order.

Handling an event occurrence results in either (a) ignoring the event occurrence, (b) saving the event occurrence so it can be considered after a transition is made to some other appropriate block, or (c) making a transition to a new block to handle the event occurrence. The deciding factors are the MSM's preemption list for the current block and the specification of the target handling blocks for each event occurrence.

Making a transition to a new block involves dismantling all streams that the executing MSM has set up in the present block, updating its various state lists to correspond to the new block, and performing all actions specified in the new block (including manner calls and stream constructions). The actions in a block are performed in some non-deterministic order[6]. When all actions in a new block are complete, the executing MSM checks for the existence of pending event occurrences, including any that may have been saved in previous states.

After zero or more transitions to new blocks, when none of the pending event occurrences (if any) are in the preemption list of its current block, an MSM performs its transport operation. This involves sending copies of units it receives on each of its currently active ports to all ports it is connected to[7]. By performing this operation, an MSM is acting on behalf of the streams that connect its source ports to other ports (see §5.1.1). The transport operation is either preempted by the arrival of an event occurrence (if it is in the current preemption list), or it ends with a suspension of the MSM when there are no more units to transfer.

5.1.1 Streams

Conceptually, streams are independent active entities in the MANIFOLD model. It is, of course, possible to actually implement them as such. However, because their functionality is so simple and their numbers so large, on most contemporary platforms, it is more appropriate to multiplex the functionality of streams into a smaller number of active entities (e.g., processes).

In our present implementation, streams are subsumed by MSMs: the functionality of streams is performed, primarily, by the processes that own their producer ports. Consequently, each MSM is responsible for delivering

---

[6]Note that the language level ; operator that implies sequential execution is only a manner call. MANIFOLD is an event driven paradigm and the conventional notion of sequentiality does *not* exist in this model.

[7]In fact, ports can have filters defined as regular expressions that may change, combine, and split units that pass through them (see §3). However, we ignore this detail here.

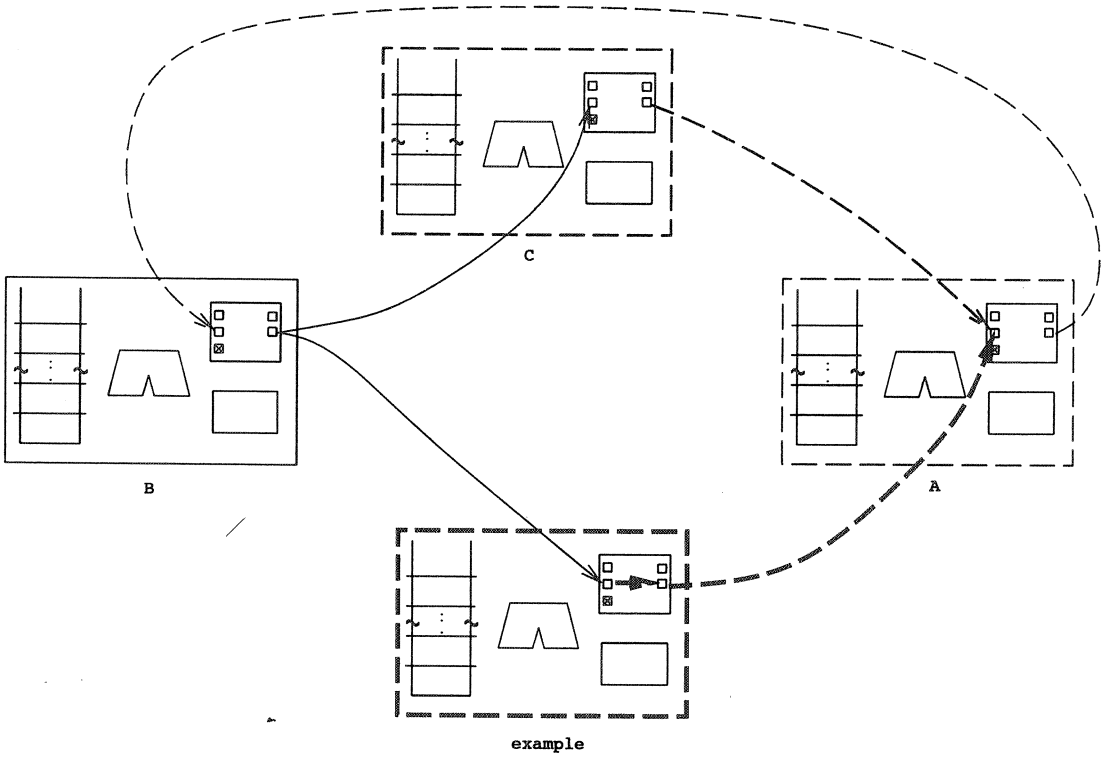


Figure 6. First set of connections set up by the manifold `example`

copies of units it receives on its ports to their corresponding destination ports. This is a reasonable scheme because generally, there is little overlap between the active intervals in the life of a stream and the process that owns its producer port.

The two main aspects of stream handling are creation/deletion of streams, and transport of units. Unit transport is performed by an MSM on behalf of every stream whose source is one of the ports of that MSM, as described in §5.1. To perform this function, an MSM needs to know the consumer port(s) connected to every one of its ports. This information is available in a data structure for each port, called the *port table*, in the `PID`. The port table for port p contains a list of process and port references, each describing the consumer of one of the streams flowing out of p. Port tables can be modified through a set of pre-defined operations that can be invoked by any MSM. Creation and deletion of a stream is performed by its constructor MSM through direct update of the port table of its source, using these operations. This fact closely corresponds to one of the fundamental concepts in the MANIFOLD model: an individual process is *not* aware of the connection network it is involved in.

Figure 6 shows the MSMs involved in the network of Figure 3 with their connections. Each port is depicted by a small rectangle inside the `PID` of its manifold processor. The different line styles used in Figure 6 symbolize the (concurrent) activity of the different processors during unit transport[8].

As an example, consider the activity of the `example` manifold, after a unit has arrived on its standard input. The statement `input` → `output` in the source program shown in Figure 2 results in an entry in the port table of the `input` port of the MSM `example` in Figure 6, designating its `output` port as a destination port.

Assuming no interesting event occurrences are detected by the `example` MSM to prevent its unit transport,

[8]Figures 6 and 7 show those connections which are set up by the manifold `example`. Other connections, not shown in the figures, may also exist, but they are not under the control of the `example` manifold.

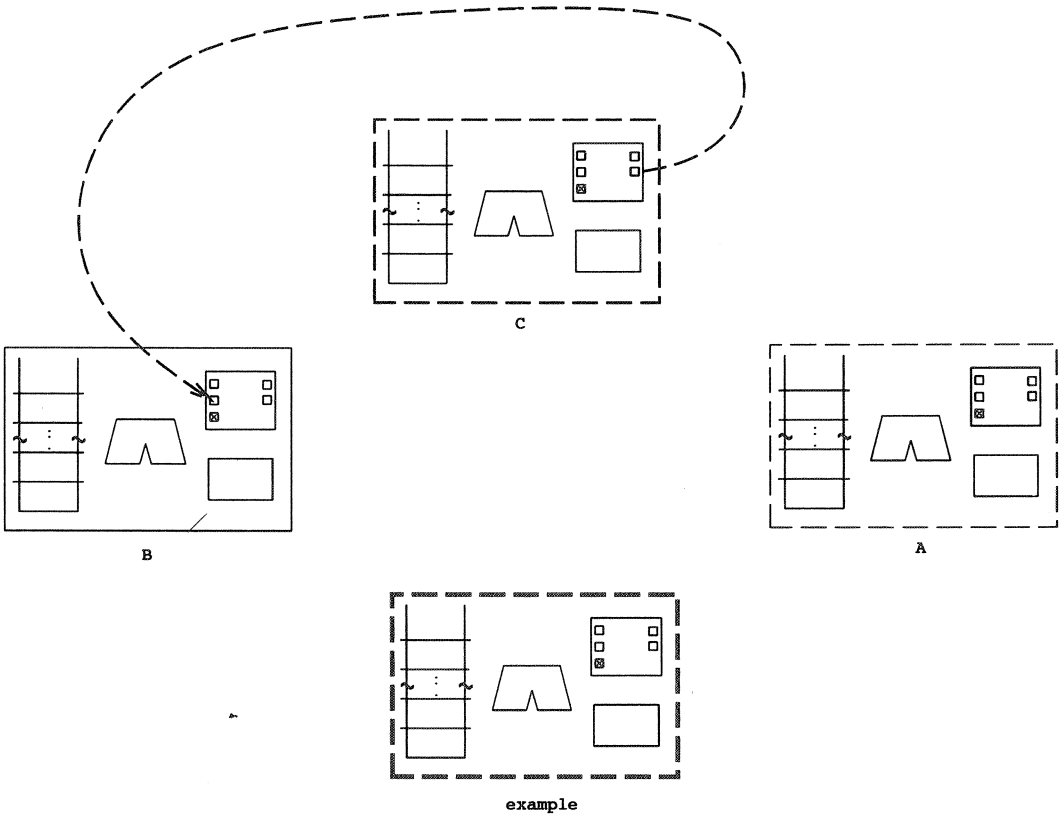


Figure 7. Second set of connections set up by the manifold `example`

it first transports the unit from its `input` port to its `output` port, according to the port table of `input`. Later on during its unit transport operation, the same MSM finds the unit in question in its `output` port. Consulting the port table for `output`, the `example` MSM finds that the `input` port of the MSM `A` is a destination. It thus transfers the unit to the input queue of the `input` port of the MSM `A`, using one of the well-defined operations on the `PID` of the MSM `A`. The MSM of `example` then notifies `A` of the presence of a new unit in its `input` port by another one of the well-defined operations on its `PID`. The responsibility of `example` is now over, as far as this unit is concerned: from this point on, this unit is handled by the processor of `A`. Assuming that `example` has nothing else to do (i.e., there are no more units in any of its ports), it simply returns to its idle state.

If a state transition occurs in `example`, e.g., to state `e2` (see, again, the listing in Figure 2), the network shown in Figure 6 must change to the one in Figure 7 (which corresponds to Figure 4). This network transformation is done by the processor of `example`: after all, it is a state transition within this MSM which is the cause of this change. The necessary network modification is done by the processor of `example`, performing a series of repetitive operations on the `PIDs` of the involved MSMs, to update their port tables.

This implementation scheme is, in fact, fairly simple. The operations on the `PIDs` must be defined very precisely. Because these operations can be performed simultaneously by more than one processor on the same `PID`, their implementation involves mutual exclusion issues analogous to the classical readers and writers problem. Given this set of operations, all that the `MP` of an MSM has to do for stream handling is to inspect its internal state lists describing connections and to perform the appropriate operations.

5.1.2 *Propagation of Events*

Event handling and state transitions involve a heavier use of the local memory of an MSM than streams. Most of the necessary information is kept in local lists, but transmission of event occurrences requires operations on PIDs, similar to stream handling. The various lists involved in event handling and state transitions are shown in Figure 8.

In the MANIFOLD model, events are broadcast so that all active processes in the system may receive them. Thus, conceptually, an event source merely raises an event for the general public, and it is the responsibility of each process to pick up event occurrences that it is interested in from its environment. Consequently, an event source does not even know the processes, if any, that are interested in its events.

This abstract model gives a clean description of the event mechanism in MANIFOLD. A direct implementation of this model involves some central resource to play the role of the over-all environment: a central events-registry where raised events are recorded by their sources, and which is queried by all observers. This scheme involves intensive communication and turns the central events-registry into a bottle-neck, resulting in an inefficient implementation on existing platforms.

In our implementation model, we decided to distribute and merge the functionality of the events-registry with all observer processes. This means that each MSM has its own events-registry area where occurrences of all events it is interested in are recorded. Given that usually, each process is interested in the events of only a few other processes, the size of the private registry of an MSM is typically small. Generally, checking for event occurrences by observers is far more frequent than the incidents of raising them by event sources. This scheme allows the more frequent checking to be done by each observer with its own events-registry: it increases concurrency, and reduces communication and contention.

The draw-back of this distribution of the events-registry is that each occurrence of an event raised by a source must now be recorded in many private registries. Typically, an event is meaningful to only a small subset of the processes involved in a MANIFOLD application. Although there may be a few events of more general, or even universal, interest in an application, the substantial majority of event occurrences need to be recorded into only a small number of private events-registries. Several schemes are possible for recording a raised event in its proper events-registries. Our implementation model uses one of the simplest ones: it requires an event source to record its event occurrence in the events-registry of every observer.

Thus, contrary to the conceptual model of MANIFOLD, each MSM *knows* all other MSMs that are interested in the events it raises. This information is also recorded in the events-registry of MSMs. The events-registry of an MSM consists of a *receive table* and a *broadcast table* that are located in its PID area. Each MANIFOLD process that is interested in an event that can be raised by another MANIFOLD process must sign itself in, in the broadcast table of the latter's MSM. This is accomplished by the observer process executing the proper operation on the PID of the source process.

The set of all processes that a manifold (or manner) can possibly be interested in observing their events, is known at its activation (or invocation) time. This set constitutes the *visibility list* of the manifold or manner. The visibility list of an MSM is maintained in its local memory and is used by the MSM to sign into (and out of) the broadcast tables of other MSMs at its activation time (and just prior to its termination). The visibility list of an MSM is *not* part of its stack frames. However, invocation and termination of manners called by the MP of an MSM dynamically add (set union) and subtract the visibility lists of the called manners to and from the visibility list of the MSM (see §3.5). This is necessary because, while inside a called manner, the MSM must still receive the events raised by sources visible to the ancestor(s) of the manner in its chain of manner calls. The corresponding signing in and out operations to update the broadcast tables of the involved sources are also done at invocation and termination of manners.

The event broadcast table itself is very similar to the port tables discussed earlier. Raising an event is a primitive action in MSM. Execution of this action involves consulting the event broadcast table of the MSM, and performing the proper operation on the PID of every observer MSM found in there, to record the raised event in its events-registry. The PID operation to record an event into the events-registry of an MSM simply ignores the event if it does not match any of the entries in its event receive table.

5.1.3 *Event Handling*

In the MANIFOLD model of events, it is the observer that is responsible for picking up its events of interest from its environment. Event sources are oblivious to who, or if anyone at all, is picking up the events they raise. Furthermore, they cannot assume that their observers pick up and react to the events they raise in the chronological order that they were raised.

Reacting to an occurrence of an observed event always causes a preemptive transition to a new event handling block in an observer manifold. However, this does not necessarily happen *immediately* after the event occurrence is observed. Each event handling block in a manner or manifold defines a set of events whose occurrences may preempt that block. This set is called the *preemption set* (of that block). Preemption sets are subsets of the observable events of a manifold. Observable event occurrences that are not in the preemption set of the current state, are *saved.* Further occurrences of a saved event from the same source are lost. Thus, a mismatch between the occurrence frequency of an event from a given source and the reaction time of an observer creates an automatic *sampling* phenomenon. Occurrences of different events from the same source, or the same event from different sources, do not override each other.

In our implementation, there is a *state table* for every manner or manifold that contains an entry for each of its event handling blocks. Each state table entry contains a pointer to the sequence of instructions for an event handling block, together with a list of events (and their sources) that are to be handled by that block.

An executing MSM has a logical stack of state tables. The bottom-most state table in this stack is that of the MSM's owning manifold. Every manner call pushes the state table of the invoked manner on the top of this stack, making it the current state table of the MSM (see §5.1.4). Termination of a manner pops its state table from this stack.

Reacting on an event, an MSM searches through its current state table, starting with the entry for the block following its current block, and continues circularly, until it reaches the entry for the block preceding its current block. If a matching handling block is found for the event in question, a transition is made to its corresponding block. Otherwise, the state tables below the current one in the state table stack are searched (see §5.1.4).

Transition from one event handling block to another in MANIFOLD involves dismantling all streams created by the former, and executing the actions specified in the latter. An MSM, thus, must keep a list of all streams it sets up in each block, and delete them as it leave that block.

5.1.4 *Implementation of Manners*

Manners in MANIFOLD are subprograms that are dynamically nested (similar to APL functions), as opposed to the statically nested subprograms of Algol-type languages (e.g., C or Pascal). This means that the names (of processes, ports, and events) that are not locally bound to a referent inside a manner definition must be resolved upon invocation of the manner, by searching backwards through the dynamic chain of manner calls. This search starts in the environment of the caller of the manner and terminates in the environment of the manifold instance to which the executing processor belongs. Remaining unresolved names are then bound to appropriate benign defaults (the `void` process, its standard input or standard output port, or the special event `noevent`).

A manner invocation (see §3.5) pushes a new stack frame onto the stack of the executing MSM. This stack frame is popped to restore the caller's environment as the active environment for the executing MSM, upon exit from the called manner. Exit from a manner occurs either due to normal termination of the manner or by preemption (see §3.5).

Dynamic binding of non-local entities to undefined names in manners requires the existence of a run-time symbol table in each manner and manifold. This symbol table (see Figure 9) contains the name of every (locally defined or undefined) entity used in a manner or manifold, together with an offset value. This offset value gives the stack frame location where the actual value of its corresponding symbol is stored at execution time. Thus, all instances of a manner or manifold share the same symbol table, while each instance has its own value for the names in this symbol table in its stack frame. One of the entries in each symbol table is for a stack frame slot that points to the symbol table of its calling environment.

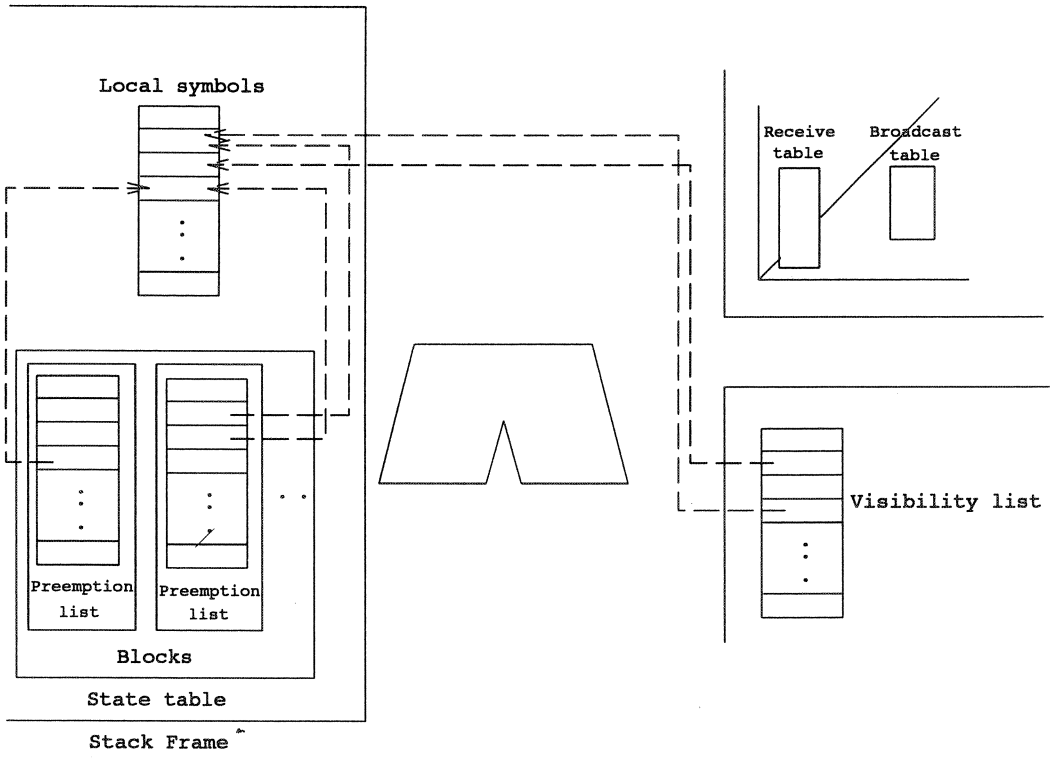


Figure 8. Lists related to event handling

Dynamic binding of non-local variables in an invoked manner, if any, is done during its initialization stage. The compiler generates the proper instructions to direct the executing MSM in its search for the value of each non-local variable down the chain of symbol tables, using the *previous symbol table* link. The found value (or a default mentioned above) is then copied into the "empty" slot of the variable in the manner's stack frame. At the end of this phase, all symbols used in a just-invoked manner have acquired a proper value. All lists (e.g., visibility lists, preemption lists, etc.) and operands of the MSM instructions use references to local slots in a stack frame. Consequently, once the contents of these slots are set correctly, all lists and operands of MSM instructions automatically become correct.

In MANIFOLD, when an event occurrence is detected inside a called manner, first, the event handling blocks defined in the manner are checked. If a match is possible, a transition is made to the corresponding block. When the event occurrence does not match any handling block in the manner, the environment of its caller is searched for a matching event handling block. The search for a matching handling block continues recursively down the chain of manner calls until either (1) a matching handling block is found, or (2) the environment of the manifold instance that owns the processor is reached. In case (2), the event occurrence is ignored and the processor continues in the environment of the called manner where it was detected. In case (1), all called manners above the environment of the matching event handling block are preempted, their stack frames are popped, and the processor makes a transition to this block.

To support the preemption of called manners by transition to non-local event handling blocks, there is an entry in each symbol table for a pointer to the state table of the calling environment. The event handling mechanism uses this link and the *previous symbol table* link to locate the proper non-local handling blocks.

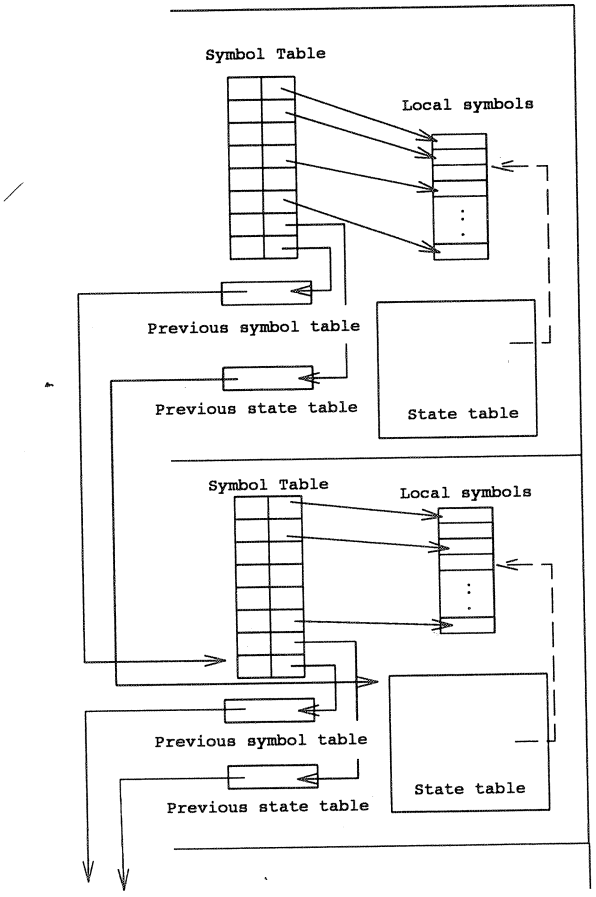


Figure 9. Dynamic nesting

5.2 Manifold Stack Machines in Concurrent C++

Our implementation platform is a Concurrent C++ [6] environment running on a network of Sun SparcStations or SGI Personal Iris workstations. Our present implementation is based on the MSM model described in the previous sections. The abstract notion of an MSM had to be mapped onto some real process in order to have a running environment.

The advantage of the Concurrent C++ environment developed at AT&T is that within a single program, one may have logical processes running on remote processors (i.e., nodes on a local network) as well as local light-weight processes. Processes are really cheap in this environment (some of them are only light-weight processes *within* one UNIX[9] process) and can also be created dynamically. The Concurrent C++ environment includes its own preemptive scheduler to manage its light-weight processes. Thus, a light-weight process is conceptually almost identical to a full-fledged UNIX process. All of these points were among our important concerns in selecting our implementation platform.

Each abstract MSM is mapped onto a Concurrent C++ logical process. The normal stack of the process plays the role of the MSM stack. Manners are just appropriately defined functions; the normal C++ invocation of functions automatically takes care of the stack handling for manner calls.

The `PID` and the `PrD` of the MSMs are collected in Concurrent C++ *capsules*. It is beyond the scope of this paper to give a detailed description of capsules (see, *e.g.*, [26]). Essentially, capsules are protected, shared memory segments reminiscent of monitors [27, 28]. Syntactically, capsules look very much like C++ classes [29] with the notable difference that calls to public member functions of capsules are mutually exclusive. Also, processes can be suspended on member function calls, using programmer-defined logical conditions on the private variables of a capsule. We use this feature to suspend an MSM as described in §5.1. The attractive feature of capsules is that they offer a very fast and reliable message passing facility through shared memory. This suits our MSM model very well[10].

Using Concurrent C++, we defined all the necessary C++ classes and Concurrent C++ capsules to realize the run-time environment for the MSMs. This ended in a runtime library which includes all the necessary methods, list handling utilities and the like.

How does a manifold program meet this run-time environment? First, a fairly traditional compiler compiles the manifold program into the abstract macro-assembly instructions of the MSMs. These instructions include statements like `begin_manifold`, `end_manifold`, `declare_variable`, `define_visibility_list` and the like. The result of the compilation of each manifold program file is an MSM macro-assembly file which contains the description of possibly several manifold and/or manner specifications.

In the next step, a macro processor [30] turns this macro-assembly file into a real Concurrent C++ source program file. Each manifold becomes a separate Concurrent C++ process type; the macro-assembly instructions are either turned into plain Concurrent C++ instructions or into calls to the run-time library functions. Finally, the Concurrent C++ code is compiled by its own compiler and the usual linker links the output with the run-time MANIFOLD library.

6 Related Work

The general concerns which led to the design of MANIFOLD are not new. The CODE system [31, 32] 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.

---

[9] UNIX is a Trademark of Bell Laboratories.

[10] In fact, special arrangements must be made when communication crosses over network node boundaries. Special Concurrent C++ processes are used to "simulate" capsule member function calls in this case.

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 [33] or Herath et. al [34] 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. [35].

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 [36]. 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 [35] 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 [37, 38] were also clearly concerned with the reusability of existing software. LINDA uses a so called generative communication model, based on a *tuple space*. The tuple space of LINDA is a centrally managed space which contains all pieces of information that processes want to communicate. A process in LINDA is a black box. The tuple space exists outside of these black boxes which, effectively, do the real computing. LINDA processes can be written in any language. The semantics of the tuples is independent of the underlying programming language used. As such, LINDA supports reusability of existing software as components in a parallel system, much like MANIFOLD.

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

One of the best known paradigms for organizing a set of sequential processes into a parallel system is the Communicating Sequential Processes model formalized by Hoare [18, 19] which served also as a basis for the development of the language Occam [7]. 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. In contrast, MANIFOLD clearly separates the functionality of a process from the concerns about its communication with its environment, and places the latter entirely outside of the process. It then completely takes over the responsibility for establishing and managing the interactions among processes in a parallel system.

Another significant difference between CSP 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)[39, 40, 41]. It is a formal description technique based on the temporal ordering of observable behavior of concurrent processes. The LOTOS language is based on a model of parallelism very similar to CSP. 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.

The implementation scheme described in §5, and specifically, our use of Concurrent C++ (see §5.2) as a platform, shows the relationship between the MANIFOLD environment and more traditional distributed programming languages. As mentioned in §2, these distributed languages can become fairly complicated to use for 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 A Window Server Example

The interaction between a window server and its clients is a typical example of client-server type applications. It is also a good example of a problem which is most naturally solved using parallel programming techniques. First, we define some terms. A window system can roughly be divided into four layers: a user interface toolkit, a window manager, a window server, and an imaging/graphics library.

The meaning of the terms "window manager" and "window server" are often confused; a *window manager* deals with the user interface to windows on a computer display, i.e., their "look and feel", while a *window server* deals with resource allocation and input distribution, and services the output requests of its clients. It is also important to understand that contrary to most other types of servers in a computer network, the window server runs on the computer the user is using, while the client applications may be running on other computers.

A window is a piece of real estate on a computer monitor which is used by a human user to give input to an application, and by a client application to output its results. Input from a user is received by a window system as low level input units[11] in the form of mouse movements, mouse button presses, keyboard input, and other types of low level input.

A client application in this context is some computer program that needs to communicate with a human user via a window on a computer display. It can be any program, e.g., a word processor or a CAD system. Client

[11] Input units are often called input events in window systems. To not confuse them with the events as defined in the MANIFOLD, we use the term "input units" instead.

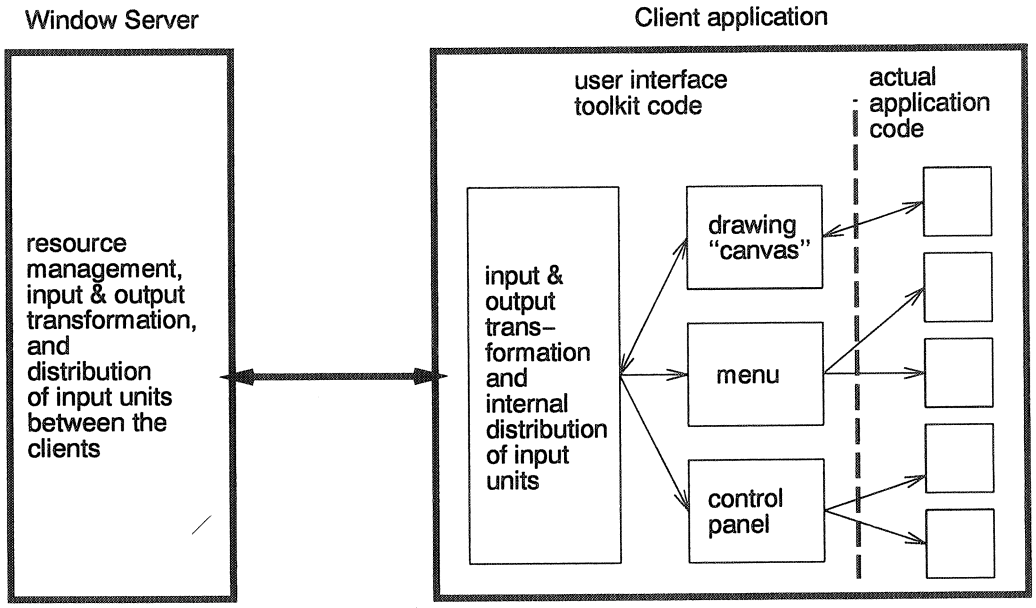


Figure 10. Overview of a window server and one of its clients. The figure also shows how the client application interfaces with the window system via a user interface toolkit.

applications usually interface with a window system via a user interface toolkit. A user interface toolkit contains abstractions for creating user interface elements such as windows, menus, buttons, etc.

A typical scenario is the following: when a (human) user starts up an application (a client) that needs to have a window on a screen, the first thing the client does is to make contact with the window server running on the computer designated by the user and ask the window server to create a window for it. The window server then allocates the resources for this window and notifies the window manager that a new window has been created. The window manager then updates its internal tables, resets the color of the screen area covered by the window, and draws the border around it. The window is now ready to display output from the client and receive input from the user.

To simplify the problem to be solved in this paper, we forget about the window manager in this example and concentrate on the interaction between a window server and client applications.

There are several ways to solve this problem in the MANIFOLD language. We must use streams to transport input units: we cannot direct events to any particular client, and *if* we were to use events, we could loose important input "events" since MANIFOLD events are not queued.

One way to implement our window server is to use a manifold with one, or a fixed number of, port(s) available per client, so that one port of the server is connected to only one client. The server can then redirect the input coming on its in-port to the appropriate out-port depending on which client should receive the input. The problem with this solution is that the number of ports a manifold can have is static. This means that the window server would have a fixed maximum number of clients it can serve. This is not necessarily bad in practice, but we prefer a more elegant solution that does not impose such static limits.

To implement a window server with a theoretically unbounded number of serviceable clients, we can use a two-step solution: The window server will process the input units it receives from all clients. All input units to the window server and all replies from the window server to its clients go through the same input and output ports of the window server, respectively. All input units and replies contain proper identification tags to show their originator or target clients.

To spare the clients the burden of generating such address tags and sieving through all replies to recognize their

own, we use special filter processes that act as intermediaries between an individual client and the window server. On the client-to-window-server direction, a filter process simply puts a tag on passing input units that contains the identification of its corresponding client. On the window-server-to-client direction, a filter process screens all replies produced by a window server and lets only those units through which have the address of its client.

The scheme described above has the advantage of imposing no communications-related concerns on the clients: they produce their input units and receive their replies without any knowledge of the addressing protocol. Without the intermediary filters, the clients would have to, at least, know *how* they must talk to the window server. This scheme is especially attractive for massively parallel environments. The use of filter processes to multiplex/de-multiplex communication between all clients and the window server through the same ports may not be very efficient without low-overhead processes and cheap inter-process communication. On the other hand, in a massively parallel system it makes no difference if processors are kept busy or left idle.

In the rest of this section we elaborate on the unbounded server scheme described above. The MANIFOLD source code for the major components of our program appears in Appendix A. This solution can easily be adapted to the bounded server model mentioned earlier as well. The three most important processes in this solution are: the window server, the window filter, and the client application. The window server and the window filter are manifold processes. We assume that the client applications are implemented in some other language and treat them as atomic processes. In addition, there is an atomic process used internally by the window server manifold process to manage its data structures and computation intensive processing. In the following subsections, we first give a more detailed specification of the individual processes in the system. Then, we discuss their implementation, and give the actual MANIFOLD code for the main modules in this example.

7.1 Process Specifications

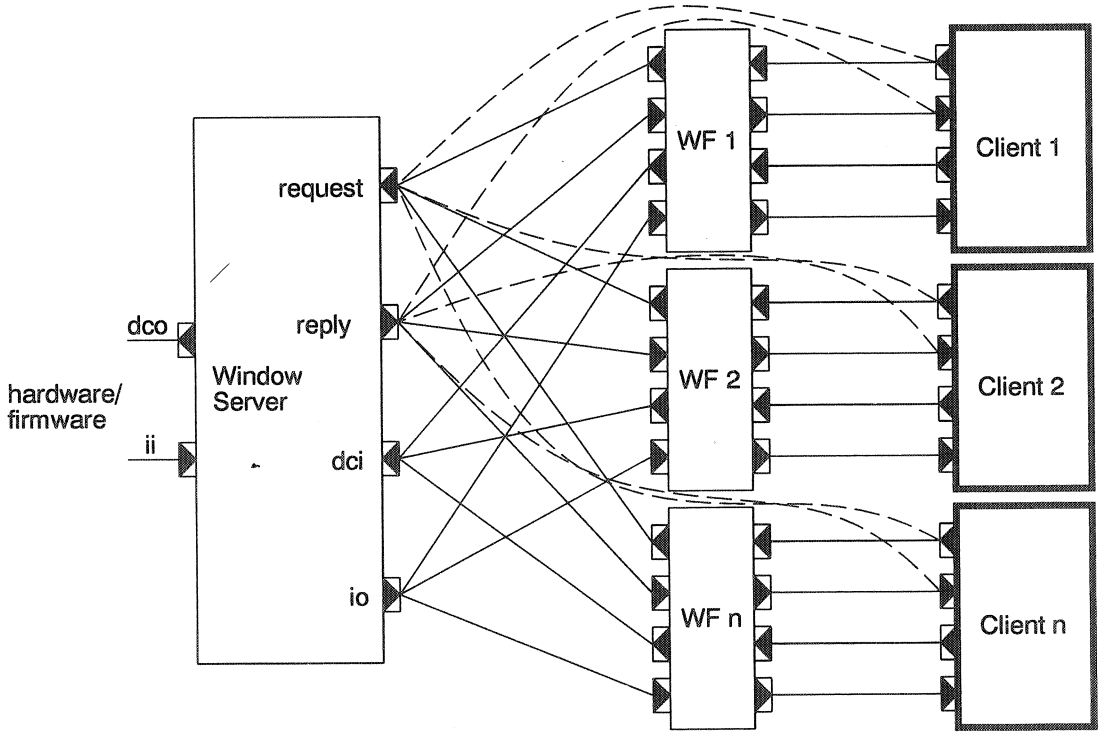
We now look into the problem in more detail. Recall that the input- and output units are sent and received via streams. We must decide how a client should "sign on" and "sign off" with the window server. It may seem that one possibility is for the client to raise "sign on" and "sign off" events that are received and reacted to by the WindowServer. The main problem with this scheme is that in the MANIFOLD language we do not presently have the ability to direct an event at a specific target process. This is necessary here because a user, of course, wants a new window to be opened on the computer display were he/she is sitting at. A broadcast "sign on" event can be picked up by all servers, which is clearly not appropriate in this case. Another smaller problem with this scheme is that a manifold process responds to pending events in a nondeterministic fashion. Consequently, "sign on" and "sign off" events from different clients may not be served on a first-come-first-served basis.

Our solution is to have a client set up a temporary stream connection and send a unit containing its sign on (i.e., "create window") request to the server. To do this the client needs to know the process identification of the server. This must be supplied by the user, usually via an environment variable, such as the *DISPLAY* variable used in the X window system.

Typically, a client must wait for an acknowledgement of the acceptance or completion of its requests (especially, its initial sign on request). Thus, in general, a client is simultaneously involved in *two* two-way conversations: one with the window server and one through the window server with a user. It is, of course, possible to multiplex these two conversations through the same input and output ports on the client's side. This requires some form of identification scheme (e.g., a type designator prefix) to separate the units that belong to the two conversations. We use a simpler scheme instead, that requires the client to carry on each conversation through a different pair of input and output ports.

With one exception, both conversations a client is engaged in go through its window filter process. The exception is the client's very first sign on request, which is made directly with the window server. The window server's reply to this request also goes directly to the client. A change of state in the client then breaks up these initial streams and the rest of the two conversations will be conducted through the newly created window filter, which also constructs all necessary streams.

To make our MANIFOLD programs more structured it is helpful to let the ports reflect the functionality of their manifolds. The following is a specification of the external interfaces of the WindowServer and the WindowFilter



The following types of units are received/sent by the ports of the window server:

in–ports:		*out–ports:*	
ii	– "raw" input units	dco	– (low level) drawing commands
dci	– (high level) drawing commands	io	– processed input units
request	– request for a service	reply	– reply on an inquiry

Figure 11. The two-step solution. Manifold and atomic processes have thin and thick borders, respectively.

manifolds.

The *WindowServer* manifold:

The WindowServer provides the following services: creation of windows, deletion of windows, servicing of client drawing commands, and distribution of input units to its clients. The external interface of the WindowServer manifold looks like this:

Process name: WindowServer		
Process type: manifold		
Parameters: none		
Port type	Port name	Unit description
out	dco	The drawing commands to be sent to the hardware/firmware
in	ii	The input units coming from the hardware/firmware
in	request	Request for a service, for example the creation of a window. The unit should contain the process reference (pref) of the client, the requested service, and the parameters for the requested service.
out	reply	A unit containg a reply to a request, such as an acknowledgement that a window has been created.
in	dci	Drawing commands received from the clients
out	io	Processed input units sent out to the clients

The *WindowFilter* manifold:

An instance of the WindowFilter is created by the WindowServer when a client sends the latter a "create window" request. The function of a WindowFilter is to set up the connections between a WindowServer and a client, and to act as a two-way filter between them.

To set up these connections, a WindowFilter needs to know the process reference of its client and the process reference of the WindowServer it wishes to connect to. Since in our scheme, the WindowServer is the activator of the WindowFilter, the process reference of the server need not be passed to it as a parameter: the process reference of the activator of a manifold (its parent) can be accessed via a builtin facility in the MANIFOLD language. The process reference of its client, however, must be passed on to a window filter as a parameter.

The WindowFilter manifold behaves as follows. When it receives a unit from the WindowServer manifold, it checks its "adress" to see if the unit is adressed to its client. If so, it strips off the address and lets the unit through. Otherwise, it throws away the unit. On the opposite direction, when it receives a unit from its client, it "stamps" a "window reference" on the unit to allow the WindowServer know which window the unit comes from. In our program, we actually use a reference to the client process as this window reference.

Process name: WindowFilter		
Process type: manifold		
Parameters: client process reference		
Port type	Port name	Unit description
out	request_out	"Stamped" requests from the client to the server
in	reply_in	Replies from the server to the client
in	ii	Input units for all clients of the window server
out	dco	"Stamped" drawing commands to be sent to the window server
in	request_in	Requests from the client to the server
out	reply_out	"Filtered" replies from the server to the client
in	dci	Drawing commands received from the client
out	io	"Filtered" input units meant for its own client

7.2 The Implementation

The WindowServer manifold itself simply creates and deletes WindowFilters, while the fairly complex computation or data structure maintenance is done by an atomic process called IO_Handler. The following is the external specification of the IO_Handler process:

The *IO_Handler* atomic process:

The IO_Handler maintains a directed acyclic graph of the windows of all clients (i.e., a screen-map). Its tasks are to:

- Update its screen-map when a window is created, deleted, pushed, popped, or moved on the screen. This data structure also contains the process reference of the WindowFilter manifold associated with the window.
- Identify the window that should receive an incoming input unit.
- Service client drawing requests.

In a real window system, the window manager generates window push, pop, and move requests, in addition to create and delete requests, as a user manipulates the entities that appear on his/her screen. We do not get into the details of the window manager in this paper. However, note that such requests from a window manager can trivially be added to the input of IO_Handler via additional streams. Additivity of streams in MANIFOLD and the fact that IO_Handler knows nothing about its communication links with other processes are both important in this case.

To find the window that is to receive an incoming unit from the hardware/firmware side, the IO_Handler must consult its screen map and other state variables and, perhaps, consider the current cursor position as well. Once it finds the target window, it places the process reference for its corresponding WindowFilter as a tag in the unit. This tag indicates to each WindowFilter process which input units are meant for its client.

The task of servicing client drawing requests mainly consists of transformation of the coordinates of the client drawing commands from window coordinates to screen coordinates, and transformation of the high-level drawing commands from the client to the lower level drawing commands of the underlying hardware/firmware.

Process name: IO_Handler		
Process type: atomic		
Parameters: none		
Port type	Port name	Unit description
in	c_win	The client process reference, the window size and position, and the process reference of its WindowFilter process to be added to the screen-map
in	d_win	A "window reference" of the window to be deleted from the screen-map
in	ii	Input units from the hardware/firmware
out	io	Composite units containing the process reference of the WindowFilter process of the target client and the input unit

When a client wants to create a window the following actions take place:

1. The client application accesses an environment variable to get the "address" of the server the user wishes to use.
2. The client application uses this "address" to set up a temporary pipeline to the "request" port of the WindowServer and send over its process reference, a "create window" request, and the window parameters.[12]

[12]To make it easy to use existing applications, it may not be the actual application code that follows this protocol, but some MANIFOLD wrapper process which, from the viewpoint of the application, transparently encapsulates it.

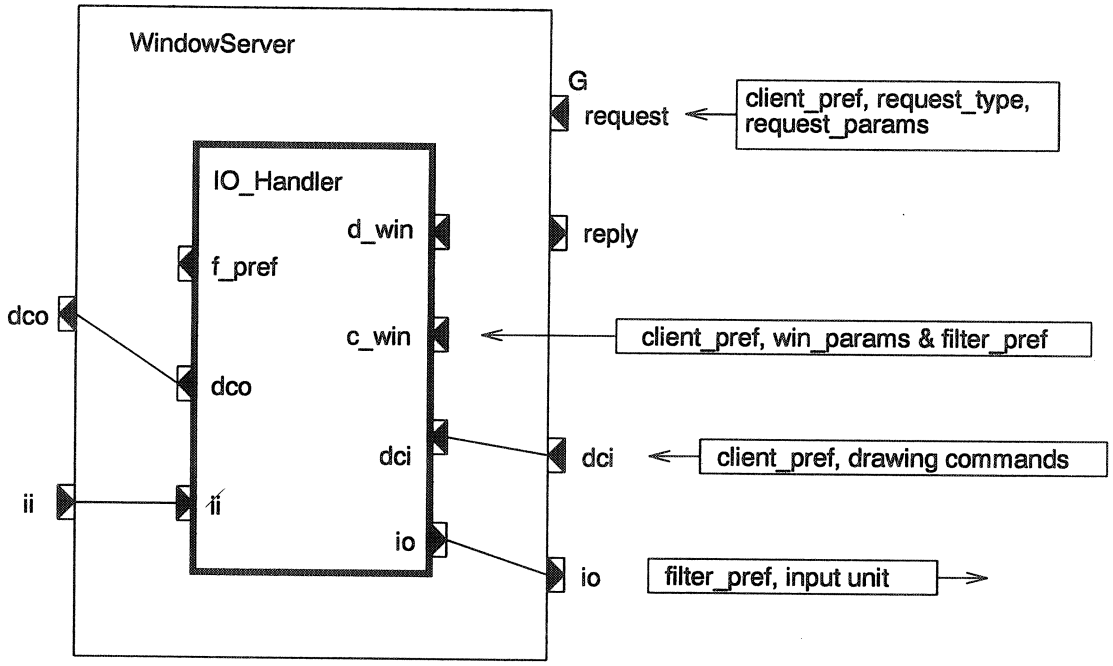


Figure 12. The internal structure of the WindowServer manifold. The ports with a capital letter G assigned to them are guarded ports. At some of the ports the unit type sent/received is also shown. The internal variable processes are not shown.

3. When the create-window unit is received by the guarded "request" port of the WindowServer, this port raises a *serve_request* internal event.
4. The WindowServer now assigns the three parts of the create-window unit (the client process reference, the create window request, and the window parameters), to three internal variable-processes.
5. The WindowServer then activates an anonymous instance of the WindowFilter manifold, giving it the client process reference as a parameter.
6. The WindowServer then sends a composite unit to its internal IO_Handler process so that it can update its screen-map.
7. Finally, the WindowServer sends a reply to the client telling it that a window has been created, and then reinstalls the guard on its request port.

When a client wants to delete a window the following actions take place:

1. The client application sends a delete-window unit containing its process reference, a "delete window" request, and empty an set of parameters to the "request" port of the WindowServer.
2. When the delete-window unit is received by the guarded "request" port of the WindowServer, this port raises a *delete_window* internal event.
3. The WindowServer places the client process reference in one of its internal variable processes, and passes it to the "d_win" port of the IO_Handler process.
4. The IO_Handler process then removes the window from its screen-map and puts a unit containing the process reference of the corresponding WindowFilter process on its f_pref port.

5. This WindowFilter process reference is used by the WindowServer manifold to deactivate this WindowFilter.

6. Finally, the WindowServer sends a reply to the client, notifying it that its window has been deleted, and then reinstalls the guard on its "request" port.

8 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 extentions 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.

9 Conclusions

This paper gives 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.

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.

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 window system example discussed in this paper, implements only the top layer of input/output handling between the window server and the client. 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, [22], we describe an implementation of the GKS logical input device in MANIFOLD.

The code of the window server can in fact be reused by the window "process" in the user interface toolkit since, a window does more or less the same tasks for its subwindows. The advantages of using MANIFOLD for these types of problems are: easy specification of interaction between processes because of the declarative nature of MANIFOLD, reduction of complexity because of easy decomposition in manageable modules, and high degree of

reusablity of the different modules because of their autonomous nature and little assumptions about their execution environment.

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.

References

[1] F. Arbab, "Specification of manifold," Tech. Rep. to appear, Centrum voor Wiskunde en Informatica, Amsterdam, 1991.

[2] N. Gehani, *Ada: Concurrent Programming*. London — Sydney — Toronto — New Delhi — Tokyo: Prentice-Hall, 1984.

[3] United States Department of Defense, *Reference Manual for the Ada Programming Language*, November 1980.

[4] N. Gehani and W. Roome, "Concurrent C," *Software —Practice and Experience*, vol. 16, pp. 821–844, 1986.

[5] N. Gehani and W. Roome, *The Concurrent C Programming Language*. Summit NJ: Silicon Press, 1989.

[6] N. Gehani and W. Roome, "Concurrent C++: Concurrent programming with class(es)," *Software —Practice and Experience*, vol. 18, pp. 1157–1177, 1988.

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

[8] D. May, "OCCAM," *Sigplan Notices*, vol. 18, April 1983.

[9] H. Bal, J. Steiner, and A. Tanenbaum, "Programming languages for distributed computing systems," ACM *Computing Surveys*, vol. 21, pp. 261–322, 1989.

[10] 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.

[11] T. Doeppner Jr., "A threads tutorial," Tech. Rep. CS-87-06, Brown University, 1988.

[12] 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.

[13] 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.

[14] D. Black, "Scheduling support for concurrency and parallelism in the Mach operating system," IEEE *Computer*, vol. 23, pp. 35–43, May 1990.

[15] SUN Microsystems, *SunOS Manuals, Lightweight Processes*, revision A ed., 1990.

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

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

[18] C. Hoare, "Communication sequential processes," *Communications of the* ACM, vol. 21, August 1978.

[19] C. Hoare, *Communicating Sequential Processes*. New Jersey: Prentice-Hall, 1985.

[20] R. van Liere and P. ten Hagen, "Introduction to dialogue cells," Tech. Rep. CS-R8703, Centrum voor Wiskunde en Informatica, Amsterdam, 1987.

[21] 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.

[22] 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.

[23] J. Peterson, R. Bogart, and S. Thomas, "The Utah Raster Toolkit," in *Proceedings of the Usenix Workshop on Graphics*, (Monterey, California), 1986.

[24] S. Dyer, "A dataflow toolkit for visualization," IEEE *Computer Graphics & Applications*, vol. 10, July 1990.

[25] C. Upson, "Scientific visualization environments for the computational sciences," in *Proceedings of the* 34^{th} IEEE *Computer Society International Conference*, (San Francisco), March 1989.

[26] N. Gehani, "Capsules: a shared memory access mechanism," tech. rep., AT&T Bell Laboratories, Computer Technology Research, Murray Hill, New Jersey, 1990.

[27] P. Hansen, *Operating System Principles*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

[28] C. Hoare, "Monitors: an operating system structuring concept," *Communications of the* ACM, vol. 17, pp. 549–557, October 1974.

[29] B. Stroustrup, *The C++ Programming Language*. Reading, Massachusetts: Addison-Wesley, 1986.

[30] R. Seindal, GNU M4. Free Software Foundation, Inc., 0.07 ed., November 1990.

[31] J. Browne, M. Azam, and S. Sobek, "CODE: A unified approach to parallel programming," IEEE *Software*, pp. 10–18, July 1989.

[32] 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.

[33] A. Veen, "Dataflow machine architecture," ACM *Computing Surveys*, vol. 18, pp. 365–396, 1986.

[34] 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.

[35] 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.

[36] P. ten Hagen, I. Herman, and J. de Vries, "A dataflow graphics workstation," *Computers and Graphics*, vol. 14, pp. 83–93, 1990.

[37] N. Carriero and D. Gelenter, "LINDA in context," *Communications of the* ACM, vol. 32, pp. 444–458, 1989.

[38] W. Leler, "LINDA meets unix," *IEEE Computer*, vol. 23, pp. 43–54, February 1990.

[39] 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.

[40] T. Bolognesi and E. Brinksma, "Introduction to the iso specification language LOTOS," *Computer Networds and* ISDN *Systems*, vol. 14, pp. 25–59, 1986.

[41] E. Brinksma, "A tutorial in LOTOS," in *Protocol Specification, Testing, and Verification V* (M. Diaz, ed.), pp. 171–194, Amsterdam: North-Holland, 1986.

Appendix A Manifold source code for the Window Server example

The names of the process *types* are spelled with mixed case letters, while the names of process *instances* are spelled with lower case letters with underscores as the word dividers.

```
#define request_type   create window\|delete window
#define request_params .*
#define input_unit     .*
#define draw_command   .*
#define reply_unit     .*

WinServer()
port out dco.
port in  ii.
port in  request  "\<procref\>\<request_type\>\<request_params\>".
port out reply.
port in  dci.
port out io.
// The regular expression on the "request" port will chop the incomming
// unit in three, one part containing the process reference of the
// client, the next part containing the type of request, and the last
// part containing the request parameters.
{
   process io_handler          is IO_Handler.
   process client_pref         is variable.
   process request_type        is variable.
   process request_params      is variable.
   process filter_pref         is variable.

   event serve_request, create_window, delete_window.

   permanent ( io_handler.dco -> dco,
               ii             -> io_handler.ii,
               dci            -> io_handler.dci,
               io_handler.io  -> io ).

   start:
       ( activate io_handler,  activate client_pref,
         activate request_type, activate request_params,
         activate filter_pref );
         guard( request, serve_request ).

   serve_request:
       client_pref    = getunit( request );
       request_type   = getunit( request );
       request_params = getunit( request );

       // "case" is a special manner with the usual semantics.

       case(  $request_type,
             "create window",  do create_window,
             "delete window",  do delete_window );
```

```
        // This list could be continued with all kinds of
        // requests, but in this exercise we limit the
        // types of requests to creation and deletion of
        // windows.

    create_window:
        // Activate a new WindowFilter and assign its pref to
        // filter_pref by using the reference operator (ampersand).
        // The dereference operator (dollar) is used to pass the
        // client_pref to the WindowFilter manifold.

        filter_pref = &WindowFilter( $client_pref );

        // Assemble a composite unit consisting of the client pref,
        // the window parameters, and the filter pref, and send it
        // the io_handler process. The "pass1" is a builtin
        // manifold which terminates itself after it has let one
        // unit through.

        [ $client_pref, $request_params, $filter_pref ] ->
                  pass1 -> io_handler.c_win;

        // Tell the client that a window has been created, and then
        // reinstall the guard on the request port.

        "window created" -> reply -> pass1 -> $client_pref.reply;
        guard( request, serve_request ).

    delete_window:
        $client_pref -> io_handler.d_win;

        // When the io_handler receives the "delete window"
        // request from the client on its d_win port it will update
        // its screen-map, and put the pref of the clients' Window-
        // Filter process on its f_pref port. In this way the Window-
        // Server manifold can deactivate the WindowFilter manifold.

        filter_pref = io_handler.f_pref;
         deactivate $filter_pref;

        // Tell the client that its window has been deleted, and then
        // reinstall the guard on the request port.

        "window deleted" -> reply -> pass1 -> $client_pref.reply;
        guard( request, serve_request ).
}
```

```
WindowFilter( client )
process  client.
port out request_out.
port in  reply_in  "\<procref\>\<reply_unit\>".
port out dco  "\<procref draw_command\>".
port in  ii  "\<procref\>\<input_unit\>".
port in  request_in.
port out reply_out.
port in  dci.
port out io.
// The regular expression on the "ii" port "chops" the incomming
// units in two, while the regular expression on the "dco" port
// glues two units together.
{
   event filter_input, pass_on_input, not_pass_on_input,
         filter_reply, pass_on_reply, not_pass_on_reply,
         stamp_output, stamp_request.

   permanent ( request_out    -> parent.request,
               parent.reply   -> reply_in,
               dco            -> parent.dci,
               parent.io      -> ii,
               io             -> client.ii,
               client.request -> request_in,
               reply_out      -> client.reply
               client.dco     -> dci ).

   start:
        ( guard( ii, filter_input ),
          guard( dci, stamp_output ),
          guard( reply_in, filter_reply ),
          guard( request_in, stamp_request ) ).

   // ****************************************
   //  Input-unit event handler blocks
   // ****************************************

   filter_input:
        // The control-flow is done with the if-then-else manner. It
        // has the following syntax:
        //     if ( cond, then_part [, else_part] ) .
        // The "self" process reference is used to check if a
        // unit received on the "ii" port should be passed on to
        // the client. Similarly the "self" process reference is
        // "stamped" on the units being sent out of the "dco"
        // port.

        if ( $getunit( ii ) ==  $self,  do pass_on_input,
                                        do not_pass_on_input ).
```

```
    pass_on_input:
        getunit( ii ) -> output; guard( ii, filter_input ).

    not_pass_on_input:
        getunit( ii ) -> void; guard( ii, filter_input ).

    // ****************************************
    //  Reply-unit event handler blocks
    // ****************************************

    filter_reply:
        if (  $getunit( reply_in ) ==  $self,
              do pass_on_reply,  do not_pass_on_reply ).

    pass_on_reply:
        getunit( reply_in ) -> reply_out; guard( reply_in, filter_reply ).

    not_pass_on_reply:
        getunit( reply_in ) -> void; guard( reply_in, filter_reply ).

    // ****************************************
    //  Request- and output-unit event handler blocks
    // ****************************************

    stamp_output:
         &self -> dco; getunit( dci ) -> dco; guard( dci, stamp_output).

    stamp_request:
         &self -> request_out; getunit( request_in ) -> request_out;
        guard( request_in, stamp_request).
}
```