Interaction Management of a Window Manager in Manifold

F. Arbab I. Herman P. Spilling

Centre for Mathematics and Computer Sciences
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

Abstract

Management of the communications among a set of concurrent processes arises in many applications and is a central concern in parallel computing. In this paper we introduce 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; the emphasis is on how these processes are interconnected and how their interaction patterns change during the execution life of the system.

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.

1 Introduction

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]. Another report contains more examples of the use of the MANIFOLD language [2]. Strict space limitations do not allow us to give the motivation for MANIFOLD and a comparison with related work here. The interested reader is encouraged to find them in [3] and in [4]. In this paper, we give an overview of the MANIFOLD language 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 understandable.

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 [5], Concurrent C++ [6], Occam [7] and many others. Contrary to languages that try to hide as much of the "chaos of parallelism" as possible behind a facade of sequential programming, MANIFOLD is based on the idea that allowing programmers to see and feel this chaos is actually beneficial.

2 The Manifold Language

In this section we give a brief and informal overview of the MANIFOLD language. The basic components in the MANIFOLD model 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 describe their internal behavior using the MANIFOLD model. These processes are called manifolds. Other processes may be a piece of hardware, programs written in other programming languages, or humans. These processes are called atomic processes. In fact, an atomic process is a 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.

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 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 2). The flows of units of information in streams are automatically replicated and merged at outgoing and incoming port junctions, as necessary. 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.

Once an event is raised by a source, it generally continues with its processing, while the event occurrence propagates through the environment independently. Communication of processes through events is 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 §2.2.

2.1 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 3 which shows the MANIFOLD source code for a simple program.

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. 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
This is the header (there are no arguments):

```java
example()
```

These are the public declarations:

- Two ports are visible from the outside;
- One is an input port and the other is an output.

```java
port in input.
port out output.
```

The body of the manifold begins here.

```java
private declarations:
```

Three process instances are defined:

```java
process A is A.type.
process B is B.type.
process C is C.type.
```

First block (activated when "example" becomes active)

The other processes are activated in a "group" construct:

```java
start: ( activate A,
   activate B,
   activate C);
do begin. // "goto" begin
```

Three pipelines in a group are set up:

```java
begin: (A → B, output → C).
```

Event handler for the event "e1"

```java
e1: (B → input, C → A, A → B,
   output → A, B → C).
```

Event handler for the event "e2"

```java
e2: (C → B).
```

Figure 3: An example for a manifold process

Figure 4: Connections set up by the manifold example on event e2

2.2 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. 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 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 §2.5 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 [8]. 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.

2.3 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 or a single manner call.

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.

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.

2.4 Visibility of Event Sources

Every process instance defined or used anywhere in a manner (see §2.5) or manifold is an observable source of events for that manner or manifold. This simply means that occurrences of events raised by such sources (only) will be picked up by the executing manifold processor, provided that there is a handling block for them. The set of all events from observable sources that match any of the block labels in a manner or manifold is the set of observable events for that manner or manifold. The set of observable events of an executing manifold instance may expand and shrink dynamically due to manner calls and terminations (see §2.5). 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.

2.5 Manners

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. 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.

3 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 three layers: a user interface toolkit, a window server, and an imaging/graphics library.

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 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 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
long duration connections
transient connection

WF: Window Filter
C1: Clients

Figure 5: The two-step solution. Manifold and atomic processes have thin and thick borders, respectively.

3.1 Process Specifications

We now look into the problem in a little bit more detail (see also Figure 5). Recall that the input and output units are sent and received via streams. The next issue is the initial communication of a client with the WindowServer of its choice. In our solution, we 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.

Process name: WindowServer
Process type: manifold
Parameters: none

<table>
<thead>
<tr>
<th>Port type</th>
<th>Port name</th>
<th>Unit description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>ii</td>
<td>Hardware input units.</td>
</tr>
<tr>
<td>in</td>
<td>request</td>
<td>Request for a service.</td>
</tr>
<tr>
<td>out</td>
<td>reply</td>
<td>Reply to a request.</td>
</tr>
<tr>
<td>in</td>
<td>dci</td>
<td>Drawing commands.</td>
</tr>
<tr>
<td>out</td>
<td>io</td>
<td>Processed input units.</td>
</tr>
</tbody>
</table>

Figure 6: External Specification of the WindowServer manifold

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 manifolds.

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 is described on Figure 6.

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.

The WindowFilter manifold behaves as follows. When it receives a unit from the WindowServer manifold, it checks its "address" to see if the unit is addressed 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 to know which window the unit comes from (see also Figure 7).

### 3.2 The Implementation

Due to the limited size of the present paper, we cannot present the detailed code of the program here; the reader should refer to [3] for the details. Suffices it to say that 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 IOHandler. In other words, the WindowServer manifold manages the communication structure between the atomic process performing the effective calculations and the other processes responsible for the communication with the clients.

### 4 Conclusions

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

The 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, [4], we describe an implementation of the GKS logical input device in MANIFOLD.

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

Our present implementation of MANIFOLD uses the Concurrent C++ [6] environment running on a network of Sun SparcStations or SGI Personal Iris workstations. An overview of our implementation scheme is described in a separate paper[3]. This paper also contains a more detailed description of the window manager example. The general concerns which led to the design of MANIFOLD are not new; a comparison of MANIFOLD with some related work appears elsewhere[2, 3].

### References


