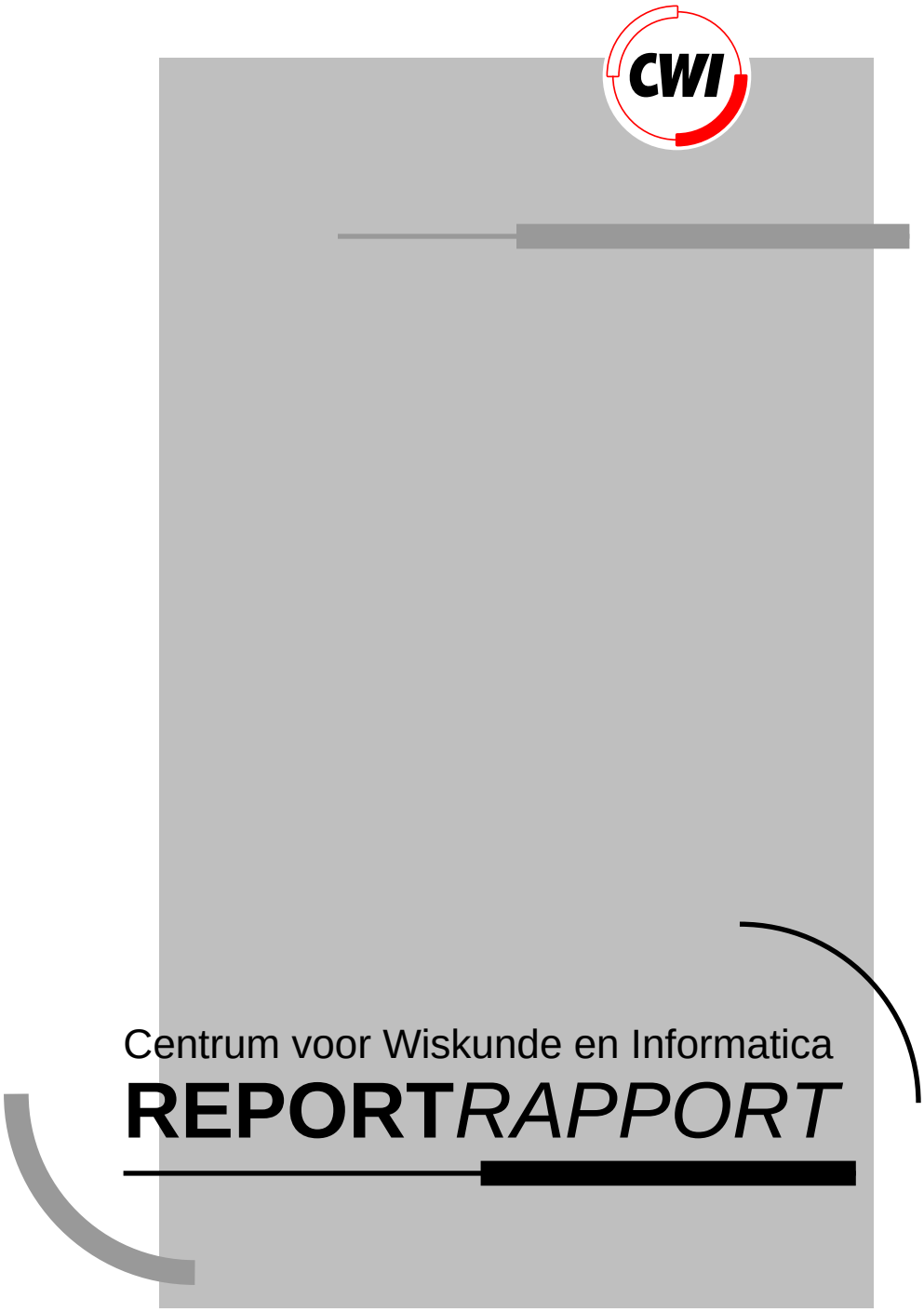

Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Formal Specification of Manifold: a preliminary study

E.P.B.M. Rutten, F. Arbab, I. Herman

Computer Science/Department of Interactive Systems

CS-R9215 1992

Formal Specification of MANIFOLD: a Preliminary Study

E.P.B.M. RUTTEN, F. ARBAB, I. HERMAN

CWI, Department of Interactive Systems
P.O. Box 4079, NL-1009 AB Amsterdam, The Netherlands
e-mail: rutten@cwi.nl, farhad@cwi.nl,ivan@cwi.nl

Abstract

This report is an initial version of a formal specification of the MANIFOLD language. A detailed informal specification of MANIFOLD already exists and has been used as the basis of its first implementation. The work on formal specification of MANIFOLD overlapped this implementation effort and they both affected the details of the informal specification of the language. In this report, we present an operational semantics of the event-driven mechanism of the MANIFOLD language.

MANIFOLD is a parallel programming language where processes called *manifolds* use an event-driven control mechanism to coordinate the communications among other processes (manifolds as well as external). Inter-process communication in MANIFOLD is through broadcast of events and a dynamic data-flow network, built out of *streams* carrying *units* of data.

In this report we consider only the event mechanism of MANIFOLD, which handles the control aspect of the language. The aspect of the language concerning the exchange of data, i.e., its streams and units, is not covered in this paper. The behavior of MANIFOLD is formally defined using transition systems at different structural levels. These transition systems define the control behavior of the MANIFOLD language constructs, from primitive actions to entire applications involving multitudes of concurrent processes.

This formal specification is intended as a preliminary study. Analysis of the properties of this model has not yet been carried out. The present formal specification constitutes a basis for further work on abstract models of MANIFOLD. Continuation of this work includes clarification of the complete behavior of MANIFOLD, development of programming assistance tools for analysis of MANIFOLD programs, further development of the MANIFOLD model, and possibly better models for its future implementations.

1991 Computing Reviews Classification: *C.1.2, C.1.3, C.2.m, D.1.3, D.3.1, F.1.2, I.1.3.*

1980 Mathematics Subject Classification: *68N15 [Software]: Programming Languages; 68Q05, 68U30.*

Key Words and Phrases: *formal specifications, parallel computing, models of computation, programming language semantics, coordination languages.*

Note: *This work was supported, in part, by the ERCIM (European Research Consortium in Informatics and Mathematics: CNR (Italy), CWI (The Netherlands), GMD (Germany), INESC (Portugal), INRIA (France), RAL (U.K.)).*

Contents

1 Introduction

This report is a preliminary version of a formal specification of the MANIFOLD language. It presents the operational semantics of its event-driven control mechanism. Starting with the original informal specification of MANIFOLD[1], we extracted the basic elements relevant to its event mechanism, and restricted the language to a simplified subset that corresponds to this mechanism. We then modelled the resulting kernel language using a set of transition systems.

A short presentation of MANIFOLD and its intuitive semantics appear in this report. However, the original specification must be consulted for a full description [1].

MANIFOLD is a parallel programming language, derived from a more general computational model. It is designed for the management of dynamic communication networks among concurrent processes, using two orthogonal control mechanisms: data-flow and broadcast of events. All communication in MANIFOLD is asynchronous. The event mechanism is used to sequence through different states, which imply different connection patterns in the data-flow network. The management of these connections (building them up, and breaking them down) is distributed between manifold processes, each concurrently controlling a sub-net of the global network.

The MANIFOLD language and its model have been described in a detailed informal specification [1]. On this base, an implementation of MANIFOLD (a compiler and a run-time system) is being finalized. Also, studies are currently underway on a visual programming interface and visual debugging tools for MANIFOLD programs.

One major motivation for developing a formal specification of MANIFOLD was to clarify the basic structures and constructs of the language as defined in its informal specification [1]. MANIFOLD is a full-sized programming language. As such, it is more complex in its structures and behaviors than "toy" languages used to examplify formal specification techniques. This, of course, adds to the complexity of the structure of the semantic model itself. Our formal semantics must eventually define all possible states of a MANIFOLD application, and all permissible transitions between these states.

The choice of the formalism for defining the semantics of MANIFOLD was motivated by the operational character of its original informal specification. We intended to remain closer to the intuitive semantics of the language.

This formal specification is an operational semantics in the form of transition systems, in the style of Plotkin [11]. Different states of a MANIFOLD application, of its individual processes, of their internal states, and, finally, each primitive action, are described. Transition relations define how it is allowed to go from one state to another, at each level of detail. Induction rules specify under which conditions a transition can be made, allowing an interpretation of the semantics of a program, describing its possible executions.

Inspiring examples of the application of this formal specification method can be found among the synchronous real-time programming languages [5]. The languages in this family are all founded on well-defined mathematical semantics, which gives a formal basis for their compilation. Compilation itself is a proof process in this formal system, and results in an implementation of compiled programs in the form of deterministic automata. This formal basis can also be exploited to develop powerful program analysis tools. Specifically ESTEREL [7, 6] uses communication through broadcast events and its formal specification of sequencing control structures is a good example for formal specification of some of the constructs in MANIFOLD. Other synchronous real-time programming languages have a more declarative, data-flow character [8, 9]. However, MANIFOLD is not a synchronous language, and data-flow oriented languages do not have the same notion of events. Nevertheless, the methodology used in the formal specification of these languages is useful for MANIFOLD too.

Other more abstract approaches to formal semantics may be used in future stages of this work to define and study the properties of MANIFOLD at a higher-level. The present operational semantics will still be useful and meaningful as a first formalization of MANIFOLD, from which other formal models can be drawn, alleviating the need to go back to the original informal specification.

One of the applications of the formal model presented here is in the implementation of a prototype interpreter. The transition systems that comprise this operational semantics can be used rather directly to build an interpreter in Prolog that simulates the parallelism of MANIFOLD, and its

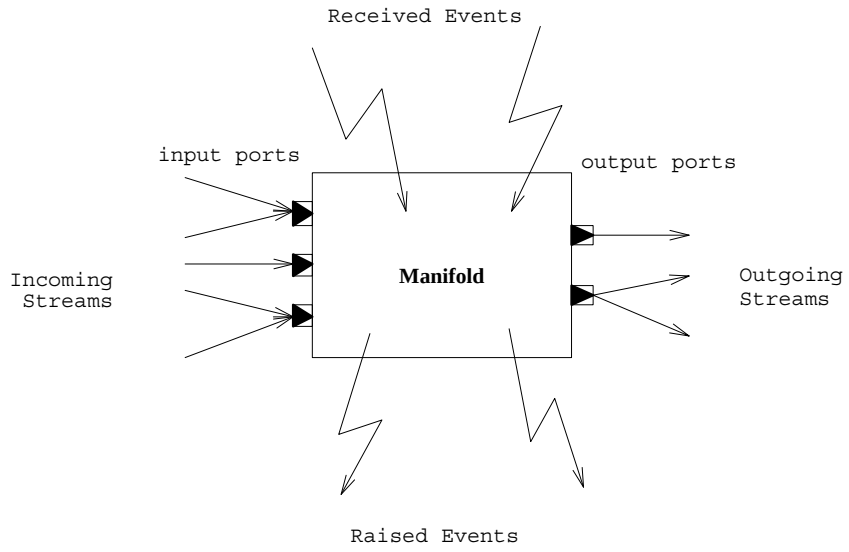

Figure 1: The model of a process in Manifold.

effects on an abstract model of an application. We have in fact built such an interpreter. Using this interpreter enables us to represent and compute all possible executions of a MANIFOLD program. It also allows us to experiment with the properties of the language, and, easily investigate the behavior of alternative language constructs by changing the interpreter.

Another application of this formal model is as a basis to build tools for analysis of MANIFOLD programs using a finite state automaton representation. Such a representation reflects the transitions among the various states of an application. It can be used, for example, to detect inaccessible states, or states from which no terminal state is accessible.

The rest of this paper is organized as follows. The next section, (§2), is a general overview of the MANIFOLD model and programming language. In section 3, we define the subset of the MANIFOLD language that we have considered in our work on formal semantics. In section 4 we give the operational semantics of this part of the MANIFOLD language. Finally, in section 5, we present the results of this preliminary study on the formal specification of MANIFOLD, describe our ongoing work on this subject, and our plans for its continuation.

2 An overview of MANIFOLD

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 [2, 3].

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

Figure 1 shows an abstract representation of a MANIFOLD process.

Ports are regulated openings at the boundaries of processes through which they exchange units of information. The MANIFOLD language allows assigning special filters to ports for screening and rebundling of the units of information exchanged through them. These filters are defined in a language of extended regular expressions. Any unit received by a port that does not match its regular expression is automatically diverted to the `error` port of its manifold and raises a `badunit` event (see later sections for the details of events and their handling in MANIFOLD). The regular expressions of ports are an effective means for "type checking" and can be used to assure that the units received by a manifold are "meaningful."

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

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

Once an event is raised by a source, it generally continues with its processing, while the event occurrence propagates through the environment independently. Event occurrences are active pieces of information in the sense that in general, they are observed asynchronously and once picked up, they preemptively cause a change of state in the observer. Communication of processes through events is thus inherently asynchronous in MANIFOLD.

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

Each state in a manifold defines a pattern of connections among the ports of some processes. The corresponding streams implementing these connections are created as soon as a manifold makes a state transition (caused by an event) to a new state, and are deleted as soon as it makes a transition from this state to another one. This is discussed in more detail in §2.2.

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

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

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

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

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

Table 1: An example for a manifold process.

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 very simple manifold, see table 1 which shows the MANIFOLD source code for a simple program.[1] More complete manifold programs are also presented, e.g., in [4]. 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

[1] 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.

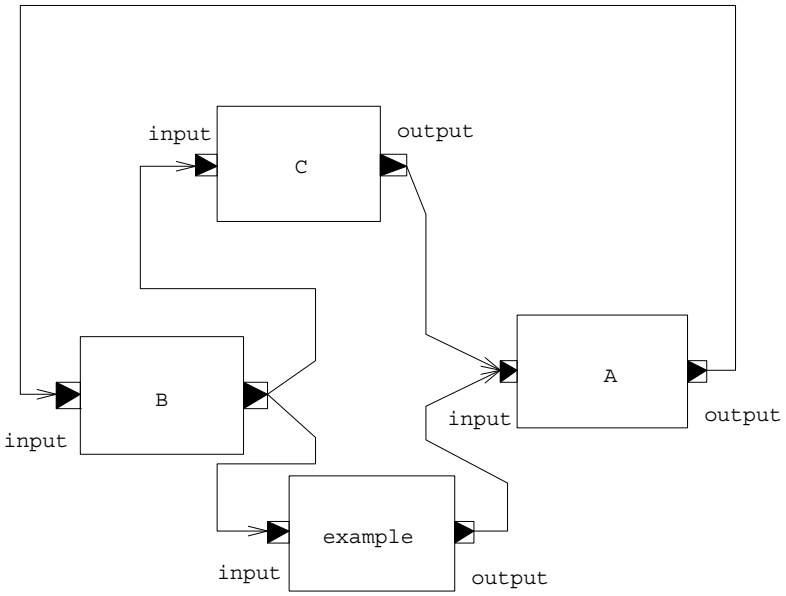

Figure 2: Connections set up by the manifold `example` on event `e1`.

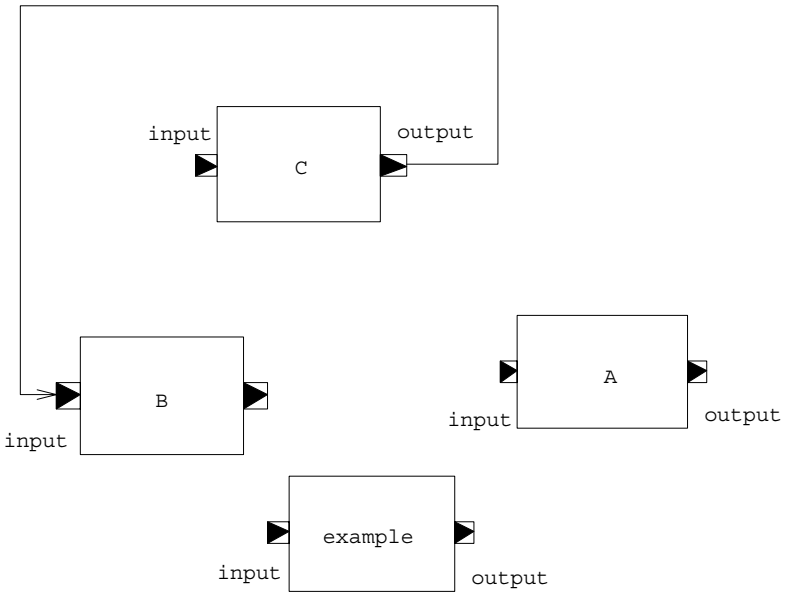

Figure 3: Connections set up by the manifold `example` on event `e2`.

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 2 shows the connections set up by the manifold process `example` on Listing 1, while it is in the handling block for the event `e1` (for the details of event handling see §2.2). Figure 3 shows the connections set up in the handling block for the event `e2`.

In its degenerate form, a pipeline consists of the name of a single port or process. Defining no useful connections, this degenerate form is nevertheless sometimes useful in event handler blocks because it has the effect of defining the named port or process as an observable source of events and a member of the preemption set of its containing block (see §2.4).

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[2]. In reaction to a recognized event, a manifold processor finds its appropriate event handler block and executes the list of sequential sets of actions specified therein. Once the manifold processor is through with the sequence in its current block, it terminates.

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. 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 §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. 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]In fact, the semicolon operator is only an infix *manner call* (see §2.5) rather than an independent concept in MANIFOLD. However, for our purposes, we can assume it to be the equivalent of the sequential composition operator of a language like Pascal.

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 member (i.e., an action, a pipeline, or a group), or a single manner call(see §2.5). If the body of a block is a pipeline, and it starts (ends) with a ->, 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.

2.4 Visibility of Event Sources

Every process instance or port 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.

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.

2.5 Manners

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

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

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

Semantically, there are two major differences between a manner and a manifold. First, manners have no ports of their own and therefore cannot be connected to streams. Second, a manner

invocation never creates a new processor. A manifold activation always creates a new processor to "execute" the new instance of the manifold. To invoke a manner, however, the invoking processor itself "enters and executes" the manner.

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

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

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

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

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

In principle, dynamic binding can be replaced by the use of (appropriately typed) parameters. Our preference for dynamic binding in manners is motivated by pragmatic considerations. Suppose a piece of information (e.g., how to handle a particular event, or where to return to) must be passed from a calling environment A, to a called environment B, through a number of intermediaries; i.e., B is *not* called directly by A, but rather, A calls some other "subroutine" which calls another one, which calls yet another one, ..., which eventually calls B. Passing this information from A to B using parameters means that all intermediaries must know about it and explicitly pass it along, although it has no functional significance for them. Dynamic binding alleviates the need for this explicit passing of irrelevant information and makes the intermediary routines more general, less susceptible to change, and more reusable.

3 A subset of MANIFOLD

In this section we define the subset of the MANIFOLD language that we use for our formal semantics study. This subset contains the event mechanism of MANIFOLD that comprises the control aspect of the language. We consider the communication and handling of events, management of the dynamic data-flow network, concurrent processes in an application, the behavior of processes receiving and handling an event, and the preemption of their current states.

The major issues in the general MANIFOLD model and language that are not considered here are streams and the transfer of units through them. For our purposes, the data-flow network is represented as a graph. The formal model presented here manipulates this graph to reflect construction and destruction of process instances and streams, but we do not consider what happens inside the streams.

In this section, we first present an informal description of the subset of the MANIFOLD language included in our formal specification. Next, we give an intuitive informal semantics for the constructs

of this kernel language, and conclude this section with a grammar for its abstract syntax.

3.1 The constructs and processes structure

A MANIFOLD *application* is consists of concurrent, independent *processes*, and it is "a medium for the propagation of event occurrences" among them [1]. The only means of communication with a process is through its input and output ports and through events. Each process has three standard ports: `input`, `output` and `error`. There are two kinks of processes in MANIFOLD: *atomic* processes and *manifolds*.

An atomic process is one whose behavior is *a priori* unknown. The behavior of a manifold process, on the other hand, is described using the MANIFOLD language. There are also a number of predefined processes, which are considered to be atomic, because we do not describe their behavior using the MANIFOLD language. For example, the predefined process `VOID` can appear as an action or part of a pipeline, and represents a process doing nothing, taking all input in, and producing no output.

A manifold is a reactive process: it performs actions only in response to observing an event occurrence. When an event occurrence is *observed* by a manifold, it may decide to handle the event occurrence only if it is a *preemptive* event. The body of a manifold is composed by a number of *handling blocks*, each specifying what actions must be performed (raising of events, (de-)activation of processes, building of pipelines, ...) when an event occurrence that matches the *label* of the block is selected for handling. This set of blocks is ordered, and can be seen as a list. This is important for deciding the next state when a preemption takes place. In the present version of the MANIFOLD language, the next handling block is determined by a circular search form the current block.

The syntax of a manifold program is as follows:

$$P_{name} \ \{ \ blocks \ \}$$

There is a specific manifold called `main`. An instance of the `main` manifold is activated at the start of an application. Other processes must be activated by `main` or by its successors in activation.

A handling block is of the form:

$$label : handler.$$

where the *label* is a list of event names that can be handled by this block, and the *handler* is the action that must be executed in reaction to this event. Each handling block represents a distinct state of its executing manifold process instance. Each state defines a set of *preemptive* events whose occurrences are allowed to cause a transition from that state to another.

A manifold instance selects one of the preemptive event occurrences it has observed for handling. This selection is done in accordance with a priority scheme that divides events into a finite number of priority classes. There is a strict total order on class priorities. The occurrence of an event is selected for handling by a manifold process only if there are no observed occurrences of events in higher-priority classes. Selection of event occurrences within the same priority class is non-deterministic.

Once an event occurrence is selected for handling, the manifold process then makes a circular search for an appropriate handling block for this event. The first handling block encountered in this circular search whose label matches the event occurrence is chosen as the target block. The event designators in block labels are generally of the form *event.source*, as explained in Section 2.3, with the possibility of using wild-card characters.

When a handling block is found, its *handler* is executed. The `start` handler is compulsary: it must be given in a manifold for it to be correct [1]. Other predefined events have default handlers, when no explicit handler is given for them by a MANIFOLD programmer.

3.2 Intuitive semantics of the actions

A *handler* can be one of the following:

Manner calls A manner is a sub-routine-like construct defined, like a manifold, by its own set of event handling blocks. When called, the local blocks of a manner take precedence over the previous event handling blocks and are used for event handling until the invoked manner returns. This *stacking* of handling blocks changes the behavior of the calling manifold to a different *manner*. A manner call has the form:

name (*parameters-list*)

Primitive actions :

- `DO` *event* raises the specified *event* locally (inside the manifold). The source of the *event* is the manifold process itself, which is denoted by `self`.
- `RAISE` *event* raises the specified *event* externally (outside the manifold). The source of the *event* is the manifold which executed the `RAISE` action.
- `BROADCAST` *event* raises the specified *event* inside and outside the executing manifold. The source of the *event* in this case is the special process `system`[3].
- `GUARD`(*port*,*event*) installs a process that guards the *port* and raises the *event* inside the manifold as soon as there is at least one unit ready for transfer inside the specified *port*.
- `IGNORE` does nothing, but when it is the only action in a handler, then the manifold goes back to the previous state it was in.
- `SAVE` saves the handled event in the observed events memory, but otherwise acts like `IGNORE` (i.e., goes back to the previous state).
- `RETURN` causes a return from a called manner.

Pipelines : One of the main activities of a manifold process is to dynamically build and dismantle pipelines. A pipeline consists of a set of streams, each of which establish a link between two ports of processes. What is special about a pipeline is that all streams in a pipeline are dismantled as soon as one of its streams is broken (e.g., one of the processes dies).

A pipeline has the following form[4].

$$[\ P_1 \texttt{ -> } P_2\ \texttt{ , } \ldots P_i \texttt{ -> } P_{i+1}\ \texttt{ , } \ldots]$$

The above construct means that the port P_i is connected to the port P_{i+1} by a stream. A port designator is generally *process.port*. The default for *process* is `self` when it is not given explicitely. If *port* is not given, it defaults to `input` when on the right-hand side of a stream operator `->`, or to `output` when on its left-hand side.

The first (or only) item in a pipeline can be one of the following special pseudo processes:

- `ACTIVATE` *process*, that activates the specified *process*, and delivers a boolean result on its own output port;
- `DEACTIVATE` *process*, that deactivates the specified *process*, and delivers a boolean result on its own output port;
- `GETUNIT`(*port*), that delivers on its output port the first unit of information coming out of the specified *port*.

Because these pseudo processes do not have `input` ports, it is syntactically incorrect for them to appear on the right hand side of the stream operator `->` [1].

---

[3] This action is not available to the MANIFOLD programmer directly. However, we use it here as a generalization of the actions `SHUTDOWN` and `CANCEL`.

[4] This syntax is a simplified adaptation of the original specification. Without loss of generality, we consider only the case where port connections are described explicitly [1].

Groups is a construct that allow associating several of these actions in one handler, by enclosing them in a pair of parentheses:

$$(\ a_1, \ \ldots a_n \ )$$

The above construct means that all the actions a_i are executed in a non-deterministic order, and the handler is terminated when they all terminate.

3.3 Grammar

In this section we present an abstract syntax for the MANIFOLD sub-language we consider in this report. In this sub-language we ignore all declarations, but assume that their effects are somehow known and properly remembered. For example:

- The set of all *atomic-processes* used by an application is known. For each atomic process P, *output(P)* is the set of events that can be raised by P.
- Every *manifold-process* is defined in a relation *manifold(* M_{name}, M_{blocks} *)* that associates the body M_{blocks} with the manifold name M_{name}.
- Every *manner* is defined in a relation *manner(* M_{name}, M_{blocks} *)* that associates the body M_{blocks} with the manner name M_{name}.

The grammar for the abstract syntax of the subset MANIFOLD language appears in table 2. In this grammar, non-terminals are in italics and are enclosed in $\langle$ and $\rangle$, i.e., $\langle a \rangle$. The construct a^+ means at least one a, and a^* means zero or more occurrences of a^+. The italic brackets *[* and *]* are used for grouping.

In this grammar, we treat $\langle event \rangle$ and $\langle name \rangle$ as identifiers. Furthermore, $\langle label \rangle$ and $\langle port \rangle$ designate block labels and port names as described in Sections 3.1 and 3.2, respectively.

4 Operational semantics

In this section we present the operational semantics of the kernel MANIFOLD language defined in Section 3. The notation and style used here are inspired by the now classical approach of Plotkin. Our operational semantics consists of four transition systems, each at a different level of abstraction: application, process, handler, and action.

The action level defines the semantics of each primitive action by describing its effect in four areas:

- locally raised events
- externally raised events
- installed pipelines
- (de)activated processes

The rules defining the action level semantics comprise our bottom-most transition system.

The handler level defines the semantics of an event handling block using a transition system on top of the action level semantics. The handler level semantics defines:

- the "normal" case of event handling: going to the next state
- the reactionnary actions (`IGNORE` and `SAVE`)
- the manner calls and returns

The process level defines the semantics of a process instance in terms of transitions among states whose semantics, in turn, are defined at the handler level. The transition system at this level describes:

⟨*action*⟩	::=	**DO** ⟨*event*⟩
		| **RAISE** ⟨*event*⟩
		| **BROADCAST** ⟨*event*⟩
		| **GUARD(**⟨*port*⟩**,**⟨*event*⟩**)**
		| **RETURN**
		| **IGNORE**
		| **SAVE**
⟨*pipeline*⟩	::=	⟨*procport*⟩ **->** ⟨*port*⟩ |
		[⟨*procport*⟩ **->** ⟨*port*⟩ *[* **,** ⟨*stream*⟩ *]** **]**
⟨*stream*⟩	::=	⟨*port*⟩ **->** ⟨*port*⟩
⟨*procport*⟩	::=	| **ACTIVATE** ⟨*process*⟩
		| **DEACTIVATE** ⟨*process*⟩
		| **GETUNIT(**⟨*port*⟩**)**
		| ⟨*port*⟩
⟨*group*⟩	::=	**(** ⟨*act*⟩ *[* **,** ⟨*act*⟩ *]** **)**
⟨*act*⟩	::=	⟨*group*⟩
		| ⟨*pipeline*⟩
		| ⟨*action*⟩
⟨*handler*⟩	::=	⟨*act*⟩
		| ⟨*manner*⟩
⟨*labels*⟩	::=	⟨*label*⟩ *[* **,** ⟨*label*⟩ *]**
⟨*block*⟩	::=	⟨*labels*⟩ **:** ⟨*handler*⟩ **.**
⟨*manifold*⟩	::=	**manifold**⟨*name*⟩ {⟨*block*⟩$^+$}
⟨*application*⟩	::=	⟨*manifold*⟩$^+$

Table 2: An abridged grammar of the considered sub-set of the MANIFOLD language.

- Atomic processes
- Manifold processes
  - reacting to priority events
  - reacting to preemptive events
  - evolution of the sub-network of streams under the control of a manifold
  - termination of a manifold or a manner

The application level is the top-most transition system. It defines the behavior of a set of concurrent processes, the semantics of each of which is defined using the process level transition system, above. It describes the global behavior of a MANIFOLD application and the state of the global data-flow network of streams.

Finally, we propose an extension to the semantics of MANIFOLD and recognize *stable* states of a MANIFOLD application. This extention is in the anticipation of our future work on higher-level analysis of MANIFOLD application.

4.1 A formal specification methodology

One of the ways to define an operational semantics for a programming language is to use the Structural Operational Semantics (SOS) method. This involves defining a set of states and rules for transition between these states. Plotkin proposes a widely-accepted framework and methodology for building transition systems [11]. This approach was used, for example, to define the formal semantics of the parallel language CSP [12]. It has also been used to define the operational semantics of ESTEREL, which is a synchronous real-time programming language [7]. The compilation and implementation of ESTEREL, and more recently, its direct hardware-compilation, are all based on this formal semantics. As in the case of other synchronous real-time programming languages [5], this formal semantics provides a mathematically well-founded basis for the language which can

be used to validate the correctness of its programs and its implementation. Our approach to the formal semantics of MANIFOLD is inspired by these works.

The essence of this approach to formal semantics is the definition of a number of states (or configurations) denoted by γ, γ', ..., and of a transition relation between these states. We use the notation:

$$\gamma \xrightarrow[\beta]{\alpha} \gamma'$$

to indicate that *within the context* α*, a transition can be made from the state* γ *to the state* γ'*, accompanied by effects* β*.*

Several transitions can lead from one state to another, passing through intermediary states:

$$\gamma \xrightarrow[\beta_0]{\alpha_0} \gamma_0 \xrightarrow[\beta_1]{\alpha_1} \gamma_1 \ldots \xrightarrow[\beta_n]{\alpha_n} \gamma'$$

and the transitive closure of the transition relation is denoted as:

$$\gamma \xrightarrow[\beta]{\alpha}{}^{*} \gamma'$$

A transition may be allowable only under certain conditions or premisses (involving α or aspects of γ). Rules for a transition with premisses are denoted as:

$$\frac{\textit{premisses}}{\gamma \xrightarrow[\beta]{\alpha} \gamma'}$$

meaning that the transition from γ to γ' is possible only if the *premisses* are true.

Special cases of *premisses* consist of transitions involing parts of γ, as when the transition of a construct or structure depends on the transitions of its parts. For example, to show that a transition from γ to γ' is possible only if there is a transition between δ (a part of the structure γ) and δ' (a part of the structure γ'), we write:

$$\frac{\delta \xrightarrow[\beta_2]{\alpha_2} \delta'}{\gamma \xrightarrow[\beta_1]{\alpha_1} \gamma'}$$

We also use abbreviated notations for sets of rules, inspired by those used by Plotkin in his study of CSP [12, p.207,215]. When several transitions from the same state are possible, we write:

$$\frac{\gamma_0 \xrightarrow[\beta]{\alpha} \gamma_1 | \ldots \gamma_k}{\gamma_0' \xrightarrow[\beta']{\alpha'} \gamma_1' | \ldots \gamma_k'} \qquad \text{instead of the } k \text{ rules} \qquad i : 1..k, \frac{\gamma_0 \xrightarrow[\beta]{\alpha} \gamma_i}{\gamma_0' \xrightarrow[\beta']{\alpha'} \gamma_i'}$$

When several combinations of several transitions are possible, we write:

$$\frac{\gamma_0 \xrightarrow[\beta]{\alpha} \gamma_1 | \ldots \gamma_m, \qquad \gamma_0' \xrightarrow[\beta']{\alpha'} \gamma_1' | \ldots \gamma_n'}{\gamma_0'' \xrightarrow[\beta'']{\alpha''} \gamma_{11}'' | \ldots \gamma_{1n}'' | \ldots \gamma_{m1}'' | \ldots \gamma_{mn}''}$$

$$\text{instead of the } mn \text{ rules } i : 1..m, j : 1..n, \frac{\gamma_0 \xrightarrow[\beta]{\alpha} \gamma_i, \quad \gamma_0' \xrightarrow[\beta']{\alpha'} \gamma_j'}{\gamma_0'' \xrightarrow[\beta'']{\alpha''} \gamma_{ij}''}$$

4.2 The actions level

At this level the semantics of each primitive action is described. We define the effect of each primitive action on the transmission of events, observed events, pipelines, and processes. A primitive action is assumed to be within the context of an event handling block for the event occurrence e, in a manifold P_{name}.

The transition system that defines the action level semantics of MANIFOLD consists of a number of states and a transition relation described below. States are of the form $\langle A, \sigma \rangle$, or, for terminal states, σ. A is an action and σ is the environment in which it is executed. An environment σ is a four-tuple:

$$\langle E_{loc}, E_{ext}, Pipe, Proc \rangle$$

where

- E_{loc} is the set of events raised within the executing manifold.
- E_{ext} is the set of events raised outside of the executing manifold.
- $Pipe$ is the set of current pipelines under the control of the executing manifold.
- $Proc$ is the set of references to processes p that must be activated or deactivated, denoted as p^+ and p^-, respectively.

Two of the components of an environment four-tuple, E_{loc} and $Pipe$, represent information local to an executing manifold process. The other two, E_{ext} and $Proc$, represent information meaningful at the application level and are treated there.

The transition relation is:

$$\langle A, \sigma \rangle \xrightarrow{e, P_{name}}_{act} \langle A', \sigma' \rangle | \sigma'$$

where:

- e is the event occurrence that caused a transition to the event handling block containing the action A, and
- P_{name} is the name of the manifold process instance in which the action is executed.

The semantics of an action is evaluated in one or several transitions from an environment σ, terminating in an environment σ':

$$\langle A, \sigma \rangle \xrightarrow{e, P_{name}}{}^{*}_{act} \sigma'$$

In order to simplify our notation which uses four-tuples of sets, we define a constant neutral element $\sigma_0 = \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$, and an operation of union. For $\sigma = \langle a, b, c, d \rangle$ and $\sigma' = \langle a', b', c', d' \rangle$, we define:

$$\sigma \cup \sigma' = \langle a \cup a', b \cup b', c \cup c', d \cup d' \rangle.$$

4.2.1 The Primitive Actions Level

In this section we define the semantics of the primitive actions that cannot appear within pipelines.

4.2.1.1 `DO`

The action `DO` e_{local} raises the event e_{local} inside of the executing manifold process only. To raise an event locally, the event is added to the set of locally raised events, with source `self`. The action `DO` terminates. The manifold will be preempted by e_{local}. A special case of this action is `HALT`, which is equivalent to the action `DO abort`, resulting in the termination of the manifold instance within which it is executed.

Rule 1 *: raising an event locally*

$$\langle \texttt{DO}\ e_{local},\ \sigma \rangle \xrightarrow{e, P_{name}}_{act} \sigma\ \cup \langle \{e_{local}.\texttt{self}\}, \emptyset, \emptyset, \emptyset \rangle$$

4.2.1.2 RAISE

The action RAISE e_{ext} raises the event e_{ext} outside of the executing manifold. The event e_{ext} is added to the set of externally raised events and will be transmitted to its proper observer processes by the application level semantics. The action RAISE terminates. This may terminate the handler. If this happens and there is no preemptive event to be handled, then the current manner, or executing manifold if at that level, will be exited (see process level rules).

Rule 2 *: raising an event globally*

$$\langle \texttt{RAISE} \quad e_{ext}, \sigma \rangle \xrightarrow{e, P_{name}}_{act} \sigma \cup \langle \emptyset, \{e_{ext}.P_{name}\}, \emptyset, \emptyset \rangle$$

4.2.1.3 BROADCAST

A broadcast event is added to both event sets, combining the effects of DO and RAISE with the special source system. Special cases of broadcasting of events are the actions SHUTDOWN and CANCEL as defined in the original MANIFOLD language (which are, respectively, equivalent to the actions BROADCAST terminate and BROADCAST abort).

Rule 3 *: broadcasting a system event*

$$\langle \texttt{BROADCAST} \quad e_{broad}, \sigma \rangle \xrightarrow{e, P_{name}}_{act} \sigma \cup \langle \{e_{broad}.\texttt{system}\}, \{e_{broad}.\texttt{system}\}, \emptyset, \emptyset \rangle$$

4.2.1.4 IGNORE

When an event occurrence causes a transition to a block that contains an IGNORE action, it is consumed. The IGNORE action does not change its environment; it is neutral and has an effect σ_0. The new environment is thus $\sigma \cup \sigma_0 = \sigma$.

Rule 4 *: ignoring the handled event*

$$\langle \texttt{IGNORE}, \sigma \rangle \xrightarrow{e, P_{name}}_{act} \sigma$$

4.2.1.5 SAVE

When an event occurrence causes a transition to a block that contains a SAVE action, it is not consumed. Saving the handled event puts it back in the local events set. The SAVE action terminates, and like IGNORE, it has no other effect on its environment.

Rule 5 *: saving the handled event*

$$\langle \texttt{SAVE}, \sigma \rangle \xrightarrow{e, P_{name}}_{act} \sigma \cup \langle \{e\}, \emptyset, \emptyset, \emptyset \rangle$$

For the case where IGNORE or SAVE is the *only* constituent of the handler, other effects (i.e., the fact that the previous state of the process is returned to) are described further, at the handler level; in a group, their effect to that respect is neutral[5]. In fact, even SAVE has no side-effect at all when in a group: the saved event e will not be in the set of *observable* events, because it is part of the labels of the current block, and thus will be discarded of the events memory E.

[5] In the present version of the specification [1], however, IGNORE and SAVE are not allowed in groups i.e., they are allowed only as full handlers; but as the proposed semantics gives them a coherent meaning even in groups, it offers us a generalisation.

4.2.1.6 `GUARD`

Guarding a port of a process, whose complete name is $P_{name}.port$, starts a new process and adds it to the set of process activations.

Rule 6 *: setting a guard on a port*

$$\langle \texttt{GUARD}(port,\ e_{guard}), \sigma \rangle \xrightarrow[act]{e, P_{name}} \sigma \ \cup \langle \emptyset, \emptyset, \emptyset, \{\texttt{guard}(P_{name}.port, e_{guard})^+\} \rangle$$

4.2.2 Processes in pipelines

The following actions can be used in pipelines as processes with streams from their output ports, or they can appear by themselves as actions. As for other processes, their termination will be described further, at the process level.

4.2.2.1 `ACTIVATE`

Activating a process p consists of adding two processes to the set of activated processes: p^+ and $\texttt{activate}(p)^+$ are added to the set of to-be-(de-)activated processes. This means that p and $\texttt{activate}(p)^+$ must be activated at the application level as new processes. The $\texttt{activate}(p)^+$ itself is a process that produces a result on its standard output port. The termination of this primitive action follows the rules of process termination.

Rule 7 *: activating a process*

$$\langle \texttt{ACTIVATE}\ p, \sigma \rangle \xrightarrow[act]{e, P_{name}} \sigma \ \cup \langle \emptyset, \emptyset, \{\texttt{ACTIVATE}(p).\texttt{output}\}, \{p^+, \texttt{activate}(p)^+\} \rangle$$

4.2.2.2 `DEACTIVATE`

Deactivating a process p adds p^- and $\texttt{deactivate}(p)^+$ to the set of (de-)activated processes. This means that the process p will be deactivated at the application level and a new process, $\texttt{deactivate}(p)^+$, will be activated to produce a result on the standard output of the `DEACTIVATE` primitive action. As in the case of the `ACTIVATE`, the termination of a `DEACTIVATE` action is subject to the process level rules.

Rule 8 *: deactivating a process*

$$\langle \texttt{DEACTIVATE}\ p, \sigma \rangle \xrightarrow[act]{e, P_{name}} \sigma \ \cup \langle \emptyset, \emptyset, \{\texttt{DEACTIVATE}(p).\texttt{output}\}, \{p^-, \texttt{deactivate}(p)^+\} \rangle$$

4.2.2.3 `GETUNIT`

Getting a unit from a port activates a process `GETUNIT`(*port*). This adds $\texttt{getunit}(port)^+$ to the process (de-)activations set. As in the case of the `ACTIVATE`, the termination of a `GETUNIT` action is subject to the process level rules. The union operation defined on the four-tuples implies that if several `GETUNIT` actions are used in the same block on the same port, only one `getunit` process will actually be activated at the application level, as stated in the specification [1].

Rule 9 *: getting a unit from a port*

$$\langle \texttt{GETUNIT}(port), \sigma \rangle \xrightarrow[act]{e, P_{name}} \sigma \ \cup \langle \emptyset, \emptyset, \{\texttt{GETUNIT}(P_{name}.port).\texttt{output}\}, \{\texttt{getunit}(P_{name}.port)^+\} \rangle$$

4.2.2.4 Breakup of a pipeline

When a process involved in a pipeline raises a **death** or **break** event, the pipeline breaks up. The following rule takes care of this situation.

Rule 10 *: treating the death or break of a process in a pipeline*

$$\frac{e \in \{\texttt{death}.P, \texttt{break}.P\}}{\langle P.Port, \sigma\rangle \xrightarrow{e, P_{name}}_{act} \sigma}$$

When the process is not terminated, the port is taken in the $Pipe$ set of σ:

Rule 11 *: process in a pipeline*

$$\frac{e \notin \{\texttt{death}.P, \texttt{break}.P\}}{\langle P.Port, \sigma\rangle \xrightarrow{e, P_{name}}_{act} \sigma \cup \langle \emptyset, \emptyset, \{P.Port\}, \emptyset\rangle}$$

4.2.3 Pipelines

The essence of a pipeline expression is to establish sets of port connections. Pipelines as defined in the original specification [1] can be rather complex. Simplification of pipeline expressions involves various meta-notation and naming. The simplification rules in [1] transform arbitrary pipelines into simple basic pipeline expressions. In this section, we consider only such basic pipeline expressions and assume that the function *simplified-pipe* is available at the handler level to return the simplified equivalent of an arbitrary pipeline expression. We describe here only the aspect of pipelines that concerns port connections.

The rule for the set-up of a stream between P_1 and P_2 combines their effects encoded as the four-tuples $\langle E_{loci}, E_{exti}, Pipe_i, Proc_i\rangle$. Note that $Proc_i$ may contain process activations. Event sets E_{exti} and E_{loci} are not used yet, but are combined by a union for generality. If neither $Pipe_i$ is empty, the resulting pipeline is the pipeline expression between the two transformed forms of P_1 and P_2.

Rule 12 *: stream set-up*

$$\frac{\begin{array}{c}\langle P_1, \sigma_0\rangle \xrightarrow{e, P_{name}}_{act} \langle E_{loc1}, E_{ext1}, \{P_1'\}, Proc_1\rangle, \\ \langle P_2, \sigma_0\rangle \xrightarrow{e, P_{name}}_{act} \langle E_{loc2}, E_{ext2}, \{P_2'\}, Proc_2\rangle\end{array}}{\begin{array}{l}\langle P_1\texttt{->}P_2, \sigma\rangle \xrightarrow{e, P_{name}}_{act} \\ \qquad \sigma \cup \langle E_{loc1} \cup E_{loc2}, E_{ext1} \cup E_{ext2}, \{P_1'\texttt{->}P_2'\}, Proc_1 \cup Proc_2\rangle\end{array}}$$

A stream breaks if any of its processes dies, raises a **break** event, or has already been deactivated. In all these cases for some i, the component $Pipe_i = \emptyset$. Other components of the four-tuples are nonetheless taken into account.

Rule 13 *: stream breaking*

$$\frac{\begin{array}{c}\langle P_1, \sigma_0\rangle \xrightarrow{e, P_{name}}_{act} \langle E_{loc1}, E_{ext1}, Pipe_1, Proc_1\rangle, \\ \langle P_2, \sigma_0\rangle \xrightarrow{e, P_{name}}_{act} \langle E_{loc2}, E_{ext2}, Pipe_2, Proc_2\rangle, \\ Pipe_1 = \emptyset \vee Pipe_2 = \emptyset\end{array}}{\begin{array}{l}\langle P_1\texttt{->}P_2, \sigma\rangle \xrightarrow{e, P_{name}}_{act} \\ \qquad \sigma \cup \langle E_{loc1} \cup E_{loc2}, E_{ext1} \cup E_{ext2}, \emptyset, Proc_1 \cup Proc_2\rangle\end{array}}$$

A pipeline is a chain of streams. The following rule defines the effect of a pipeline as the union of the effects of all of its streams.

Rule 14 *: pipeline set-up*

$$\frac{i:1..n:\ \langle S_i,\sigma_0\rangle \xrightarrow[act]{e,P_{name}} \langle E_{loci},E_{exti},\{S_i'\},Proc_i\rangle}{\langle [\ S_1,\ \ldots S_i\ldots,\ S_n],\sigma\rangle \xrightarrow[act]{e,P_{name}} \sigma\ \cup\langle \bigcup_{i:1..n} E_{loci},\ \bigcup_{i:1..n} E_{exti},\ \{[\ S_1',\ \ldots S_i'\ldots,\ S_n']\},\ \bigcup_{i:1..n} Proc_i\rangle}$$

A pipeline expires if any of its streams breaks, i.e., if for any i, $Pipe_i = \emptyset$. Other components of the four-tuple are nonetheless taken into account.

Rule 15 *: pipeline breaking*

$$\frac{i:1..n:\ \langle S_i,\sigma_0\rangle \xrightarrow[act]{e,P_{name}} \langle E_{loci},E_{exti},Pipe_i,Proc_i\rangle \qquad \bigvee_{i:1..n}(Pipe_i=\emptyset)}{\langle [\ S_1,\ \ldots S_i\ldots,\ S_n],\sigma\rangle \xrightarrow[act]{e,P_{name}} \sigma\ \cup\langle \bigcup_{i:1..n} E_{loci},\ \bigcup_{i:1..n} E_{exti},\ \emptyset,\ \bigcup_{i:1..n} Proc_i\rangle}$$

4.2.4 Groups

The effect of a group is the union of the effects of its members. A group terminates when all of its members have expired. The order of the evaluation of actions in a group is non-deterministic. We can thus use symmetric rules each a_i member of a group. Since the net effect of a group is obtained as union of the σ's of its members, and because union is both commutative and associative, the non-deterministic order of evaluation of the a_i's does not affect the semantics of a group. In this sense, just as $\cup$, the group operator "," is both associative and commutative.

The action `IGNORE` is completely neutral inside a group.

Rule 16 *: group*

$$\frac{\langle a_1,\sigma\rangle \xrightarrow[act]{e,P_{name}} \langle a_1',\sigma'\rangle |\ \sigma'}{\langle (a_1,\ a_2\ ,\ \ldots a_i\ldots a_n\ ),\sigma\rangle \xrightarrow[act]{e,P_{name}} \langle (\ a_1',\ a_2\ ,\ \ldots a_i\ldots a_n\ ),\sigma'\rangle |\langle (\ a_2\ ,\ \ldots a_i\ldots a_n\ ),\sigma'\rangle}$$

4.3 The Handler Level

The state of a process at the handler level is a triple $\langle P,C,E\rangle$, where:

- P is the state of the program of the process. P is represented as `manifold` P_{name} $\{blocks\}$[6], where *blocks* is a list of the form: $manner^* block^+$. This means that there is at least one block in *blocks* , and that it begins with 0 or more called manners (representing the "stacking" of manner calls). A called manner *manner* in this list has the form `manner`$M_{name}\{block^+\}$[7]. The list of blocks $block^+$ is ordered, and the current block is always in the first position. As we will see later, this is important for the search of the next current block in event handling.

[6] The functions $name(P)$ and $blocks(P)$ give, respectively, P_{name} and *blocks*.

[7] This is a kind of addition to the grammar presented in table 2, as these constructs are not available to MANIFOLD programmers, but are here for the internal representation of the effect of manner calls on the set of handling blocks of a manifold.

- C is the state of the current block of the process, and can be one of:
  - `inactive`, before activation and the handling of the `start` event (in this state, `start` is the only observable and preemptive event).
  - the current state of the current block: *labels* : *action*, where *action* might have different successive states.
  - `END` when the current handler is terminated,
  - `deactivated` when the whole process is terminated.
- E is the local memory of events for this process, containing all the received occurrences of observable events, that have not yet been handled.

The transition relation is: $\xrightarrow[E_{ext},\, Proc]{e,\, P_p,\, C_p}_h$, where:

- e is the handled event.
- P_p and C_p are the previous states of the program and the current block[8].
- E_{ext} is the set of externally raised events.
- $Proc$ is the set of (de-)activation of the processes.

This transition relation details the computation of the next state of a process handling an event. It is used for one-step transitions, without particular terminal states. This level is intended to operate more as a "choice" rather than for an effective computation. The net effect of this level is to discriminate between several cases of event handling by a manifold process.

4.3.1 Event handling

This is the "normal" case, of simple handler actions not involving a return to a previous state. The action a_c of the current block is evaluated, with its effects in the four-tuple $\langle E_{loc}, E_{ext}, Pipe, Proc\rangle$. If the action a_c is a pipeline, it must first be simplified according the rules given in the original specification [1]. We assume that this is done by the function *simplified-pipe*, which leaves all non-pipeline actions unchanged.

Rule 17 *: event handling*

$$\frac{\begin{array}{c} a_c \notin \textit{reactionary-actions-set} \cup \textit{manners} \ \cup\{\texttt{RETURN}\}, \\ a'_c = \textit{simplified-pipe}(a_c), \\ \langle a'_c, \sigma_0\rangle \ \xrightarrow{e,\, name(P)}{}^{*}_{act} \ \langle E_{loc}, E_{ext}, Pipe, Proc\rangle, \end{array}}{\langle P, l_c{:}a_c, E\rangle \ \xrightarrow[E_{ext},\, Proc]{e,\, P_p,\, C_p}_h \ \langle P, \textit{pipe-to-block(Pipe,}l_c\textit{)}, E \cup E_{loc}\rangle}$$

At the actions level, where they are calculated, the sets of externally raised events E_{ext} and (de-)activated processes $Proc$ are part of the evaluation environment. However, at the handler level, these sets are part of the labels of the transitions only.

There is a special event in MANIFOLD, called `noevent`, whose semantics states that it must be ignored. This can be done by filtering out this innocuous event from E_{ext} and E_{loc} at this level. The above rule does not reflect this filtering. It is trivial to replace E_{ext} and E_{loc} in the lower part of the rule with $E_{ext} \setminus \{\texttt{noevent}.P_{name}\}$ and $E_{loc} \setminus \{\texttt{noevent.self}\}$, respectively.

The set $Pipe$ is transformed by the function *pipe-to-block(Pipe,*l_c*)* into a block with the labels set l_c of the old current block and a minimum action denoting the pipelines in $Pipe$ (a block with a group: l_c: (`a1 , ... an`) of all $a_i \in Pipe, i : 1..n$; or the pipeline block: $l_c : a$ if $Pipe = \{a\}$; or `END` if $Pipe = \emptyset$).

[8] This is necessary because reactionary actions like `IGNORE` and `SAVE` "go back" to the previous state of the process.

4.3.2 Reactionary actions

These actions are characterized by the fact that they cause a return to the previous state. According to the present specification [1], the two actions `IGNORE` and `SAVE` have this behavior if they appear as the only action in an event handling block.

$$\textit{reactionary-actions-set} = \{\ \texttt{IGNORE},\ \texttt{SAVE}\ \}^{9}$$

We define a generalized framework for reactionary primitive actions and allow the possibility of adding other actions to this set. Each reactionary action can have a particular effect on a particular dimension of the four-tuple $\langle E_{loc}, E_{ext}, Pipe, Proc\rangle$[10].

Rule 18 *: reactionary actions*

$$\frac{a_c \in \textit{reactionary-actions-set},\quad \langle a_c, \sigma_0\rangle \xrightarrow{e,\,name(P)}{}^{*}_{act} \langle E_{loc}, E_{ext}, Pipe, Proc\rangle}{\langle P, l_c{:}a_c, E\rangle \xrightarrow[E_{ext},\,Proc]{e,\,P_p,\,C_p}{}_{h} \langle P_p, C_p, E \cup E_{loc}\rangle}$$

4.3.3 Manner calls

A call to a manner has the effect of adding its list of blocks in front of the caller's, in a push-down stack style. The net effect of this is that the list of event handling blocks of the manner take precedence over those of its caller, until the called manner returns.

When a handler M_{name} is a manner call, it has the form $name(parameters)$. A static relation derived from the declarations, denoted $manner(M_{name}, M_{blocks})$, gives the list of blocks M_{blocks} for a manner call. This blocks list is put at the beginning of the caller's blocks list as `manner` $M_{name}\{M_{blocks}\}$. This means that the manner is pushed onto the manner calls stack of the manifold. The local event `start.self` is raised locally in the environment of the called manner. The process level transition from the state `inactive` gives the new state of the process P' and the new current block C'. This, and other transitions, may involve nested manner calls in the handling blocks of the called manner.

Rule 19 *: manner call*

$$\frac{M_{name} \in \textit{manners},\ \ manner(M_{name}, M_{blocks}),\quad \langle \texttt{manifold}\ name(P)\ \{\texttt{manner}M_{name}\{M_{blocks}\}\ blocks(P)\}, \texttt{inactive}, \{\texttt{self.start}\}\rangle \xrightarrow[E_{ext},\,Proc]{}{}_{p} \langle P', C', E'\rangle}{\langle P, l_c{:}M_{name}, E\rangle \xrightarrow[E_{ext},\,Proc]{e,\,P_p,\,C_p}{}_{h} \langle P', C', E \cup E'\rangle}$$

The blocks of the manner M_{name} as given by the relation $manner(M_{name}, M_{blocks})$ are complete with certain default blocks for a number of predefined events, e.g., to handle its termination and return to its caller.

4.3.4 Manner returns

Returning from a manner M_{name} consists of removing its blocks list, M_{blocks}, from the top of the stack of called manners. The special event `returned.self` must also be raised in the caller's environment. As we are exiting from the manner call, the resulting current block state is `END`. This

[9] In particular, this makes the transitivity $*$ of the action level relation quite useless, as it is made in one step anyway. But leaving it there keeps open the possibility of having more complex reactionary actions.

[10] In particular, *Pipe* is ignored in the present version of the rule. But we can have something like a group combining the action of the previous state, a_{Cp}, and the effect of a_c i.e., *Pipe*, in the following way: (a_{Cp} , *pipe-to-block(Pipe,*l_c*)*). The effect of such an action can be described as installing a supplementary pipeline and going back to the previous current block.

rule reflects the semantics of normal return from a called manner. Termination of a called manner is dealt with elsewhere (rule 32).

Rule 20 *: manner return*

$$\langle \texttt{manifold}\ P_{name}\ \{\texttt{manner}M_{name}\{M_{blocks}\}\ blocks\},\ l_c\texttt{:RETURN}, E\rangle \xrightarrow[\emptyset,\emptyset]{e,P_p,C_p}{}_h \langle \texttt{manifold}\ P_{name}\{blocks\},\ \texttt{END},\ E \cup \{\texttt{self.returned}\}\rangle$$

The set of received event occurrences E may contain events that are observable only inside of a called manner. This will be dealt with at the application level, where E is intersected with the current set of observable events, *observable(P)*. This set is dependent on the current state, and is thus sensitive to calls to and returns from manners. This way, event occurrences that are not observable any more after returning from a called manner are filtered out.

4.4 The Process Level

The state of a process is represented as a triple $\langle P, C, E\rangle$ where:

- P is the state of the program of the process. For manifold processes, this is the same representation as used earlier at the handler level (section 4.3). For an atomic process (predefined or otherwise) we use its name.
- C is the state of the current block of the process. For a manifold process, it is the same representation as used earlier at the handler level. For an atomic process, it is either `inactive`, `active` or `deactivated`.
- E is the local memory of observed event occurrences for this process. This component is also the same as in the handler level.

In our semantic model for MANIFOLD, each process has its own local representation of the observed event occurrences that it is interested in. This reflects the fundamental notion in MANIFOLD that events are handled asynchronously.

The process level transition relation is defined as $\xrightarrow[E_{ext}, Proc]{}{}_p$ where:

- E_{ext} is the set of events to be raised outside of the process as it makes the transition, and
- $Proc$ is the set of processes that must be (de-)activated.

This is a one-step transition whose purpose is to select one of the possible alternative transitions for a process. The `deactivated` states of all processes are terminal states, because there are no transitions defined from these states to any other state.

4.4.1 Atomic processes

The guard on a port is a special atomic process, because it raises a specified event with *proc.port* as its source[11], and then terminates.

Rule 21 *: termination of the guard on a port*

$$\langle \texttt{guard}(proc.port, e), \texttt{active}, E\rangle \xrightarrow[\{e.proc.port\},\emptyset]{}{}_p \langle \texttt{guard}(proc.port, e), \texttt{deactivated}, E\rangle$$

[11] This is a local event that follows the same path as the external events. However, in our semantic model for MANIFOLD, an event source *proc.port* is observable only by the process instance *proc*.

The process `getunit(`*proc.port*`)` may raise a `disconnected` event with the source *proc.port*, but it does not terminate in this case.

Rule 22 *: attempt to get a unit from a disconnected port*

$$\langle \texttt{getunit}(proc.port), \texttt{active}, E\rangle \xrightarrow[\{\texttt{disconnected}.proc.port\},\emptyset]{p} \langle \texttt{getunit}(proc.port), \texttt{active}, E\rangle$$

An atomic process starts when it receives the event `start.self`:

Rule 23 *: starting an atomic process*

$$\frac{P \in atomic\text{-}processes}{\langle P, \texttt{inactive}, \{\texttt{start.self}\}\rangle \xrightarrow[\emptyset,\emptyset]{p} \langle P, \texttt{active}, \emptyset\rangle}$$

A programmer defined atomic process P can raise any event in its declared set of output events. We assume the set of output events of a process p is given by $output(P)$. The state of an atomic process, $\langle P, C, E\rangle$ does not visibly change, because the internal behavior of an atomic process is completely unknown to MANIFOLD. From the viewpoint of the MANIFOLD system, the choice of an atomic process to raise a specific event is completely arbitrary.

Rule 24 *: raising of an event by an atomic process*

$$\frac{P \in atomic\text{-}processes\ , e \in output(P)}{\langle P, \texttt{active}, E\rangle \xrightarrow[\{e.P\},\emptyset]{p} \langle P, \texttt{active}, E\rangle}$$

As far as MANIFOLDis concerned, the external atomic process P remains unchanged: it remains in state $\langle P, \texttt{active}, E\rangle$.

Termination of an atomic process P involves raising the `death` event externally and changing the activation state of P to `deactivated`.

Rule 25 *: atomic process terminating*

$$\frac{P \in atomic\text{-}processes}{\langle P, \texttt{active}, E\rangle \xrightarrow[\{\texttt{death}.P\},\emptyset]{p} \langle P, \texttt{deactivated}, E\rangle}$$

Similar transitions can be easily defined to deal with certain special events, like `abort`, `terminate`, and `start`. Likewise, other similar rules can express transitions, e.g., directly from `inactive` to `deactivated` in reaction to an `abort` event.

4.4.2 Manifold processes

A manifold process can make several types of transitions, depending on its set of observed events in E:

- Occurrences of priority events, if any, are handled first. The actual handling of priority events is no different than other preemptive events. It is just that the priority events are considered first.
- Occurrences of preemptive events are handled using rule 29.
- With no events to preempt the current state, some events can still cause an evolution of the current state (i.e. the pipelines network), without causing a transition to a different handler.
- In the absence of any of the above three possibilites, and if the current state is empty (i.e. the action is terminated, the pipeline is broken), then the manifold process has nothing left to do. This means that the manifold must terminate.

Handling of events (with or without priority) involves a search to find its appropriate handling block. This is done here using a relation denoted:

$$\xrightarrow{e}_{search}$$

This relation is defined and elaborated in Section 4.4.3.

Priority events are pre-defined in a set with a total ordering, $\langle_p$, defined on them.

$$priority\text{-}events = \{\ \texttt{abort}, \texttt{break}, \texttt{dead}, \texttt{end}, \texttt{returned} \ldots\}$$

The event `abort` has the effect of terminating the process immediately:

Rule 26 *: handling* `abort`

$$\frac{\texttt{abort}.* \in E}{\langle P, C, E\rangle \xrightarrow[\emptyset, \emptyset]{}_p \langle P, \texttt{deactivated}, E\rangle}$$

A `break` event has the effect of breaking the pipelines in which its source process P_{break} is involved:

Rule 27 *: handling* `break`

$$\frac{\begin{array}{c}\texttt{break}.P_{break} \in E,\\ \langle a_c, \sigma_0\rangle \xrightarrow{\texttt{break}.P_{break},\, name(P)}{}^*_{act} \langle E_{loc}, E_{ext}, Pipe, Proc\rangle\end{array}}{\langle P, l_c{:}a_c, E\rangle \xrightarrow[E_{ext}, \emptyset]{}_p \langle P,\ pipe\text{-}to\text{-}block(Pipe, l_c),\ (E \setminus \{\texttt{break}.P_{break}\}) \cup E_{loc}\rangle}$$

In other cases, the *greatest* element according to $\langle_p$ is the event occurrence with highest prioritary, and is the one that is selected for handling.

Rule 28 *: handling the highest priority event*

$$\frac{\begin{array}{c}P \in manifold\text{-}processes,\\ \exists e,\ e = max_{\langle_p}(E \cap priority\text{-}events \setminus \{*.\texttt{abort}, *.\texttt{break}\}),\\ blocks(P) \xrightarrow{e}{}^*_{search} \langle \{blocks'\}, C'\rangle,\\ \langle \texttt{manifold}\ name(P)\{blocks'\}, C', E \setminus \{e\}\rangle \xrightarrow[E_{ext}, Proc]{e, P, C}_h \langle P'', C'', E''\rangle\end{array}}{\langle P, C, E\rangle \xrightarrow[E_{ext}, Proc]{}_p \langle P'', C'', E''\rangle}$$

In this rule, it is assumed that these special events have handlers and that, their priority aside, their handling follows the normal rules.

Preemptive events are handled in the absence of occurrences of priority events. Whether or not an observed event occurrence is a preemptive event depends on the state of the process P. An event is preemptive in a given state of a process P if it is in the set $preemptive\text{-}events(P)$, defined as follows:

$$\begin{array}{rl} \textit{preemptive-events(P)} = & \textit{events of which the source process} \\ & \textit{is referenced in the current handler} \\ \cap & \textit{events for which there is a handler} \\ \setminus & \textit{events in the labels set of the current block} \end{array}$$

The preemptive events of P are then exactly those events in the intersection of its observed event occurrences and *preemptive-events*(P).

Rule 29 *: handling a preemptive event*

$$\frac{\begin{array}{c} P \in \textit{manifold-processes}, \\ E \cap \textit{priority-events} = \emptyset, \qquad \exists e \in E \cap \textit{preemptive-events(P)}, \\ \textit{blocks}(P) \xrightarrow{e}{}^{*}_{search} \ \langle \textit{blocks}', C' \rangle, \\ \langle \texttt{manifold}\ \textit{name}(P)\{\textit{blocks}'\}, C', E \setminus \{e\} \rangle \xrightarrow[E_{ext}, Proc]{e, P, C}{}_{h} \ \langle P'', C'', E'' \rangle \end{array}}{\langle P, C, E \rangle \xrightarrow[E_{ext}, Proc]{}{}_{p} \ \langle P'', C'', E'' \rangle}$$

The choice of the handled event is non-deterministic ($\exists$ in the set) [1].

All streams in the pipelines of the previous state are deleted: C is lost, and is replaced by C''.

The handler for e is found in the blocks list of P', which are rearranged into $\{\textit{blocks}'\}$ by the circular search mechanism (see Section 4.4.4).

The transition is made in one step of the $\xrightarrow[E_{ext}, Proc]{}{}_{p}$ relation. This reflects the atomicity of event handling actions (and manner *calls*).

Note that if an event like a `death` is preemptive (i.e., there is a handler for it), then it is handled at this level, and preempts the current state, instead of causing a pipeline evolution.

Evolution of the network of pipelines takes place in the absence of preemption, by the death of a process or breaking of a pipeline. Evolution occurs in reaction to a non-preemptive event in the *evolution-events-set*:

$$\textit{evolution-events-set} = \{\texttt{death.*}, \texttt{break.*}\}$$

Rule 30 *: evolution of a pipeline*

$$\frac{\begin{array}{c} E \cap (\textit{preemptive-events}(P) \cup \textit{priority-events}) = \emptyset, \\ \exists e \in E \cap \textit{evolution-events-set}, \\ \langle a_c, \sigma_0 \rangle \xrightarrow{e, \textit{name}(P)}{}^{*}_{act} \langle E_{loc}, E_{ext}, Pipe, Proc \rangle \end{array}}{\langle P, l_c{:}a_c, E \rangle \xrightarrow[E_{ext}, \emptyset]{}{}_{p} \ \langle P, \ \textit{pipe-to-block(Pipe,}l_c\textit{)}, \ (E \setminus \{e\}) \cup E_{loc} \rangle}$$

The action level rules are used to evaluate the evolution of pipelines and groups. Their evaluated results is $Pipe$, from which the new state of the current block is obtained, using the function *pipe-to-block* discussed earlier (see Section 4.3.1). Events raised in E_{loc} and E_{ext} are in fact not presently used; they will always be empty in the case of evolutions. The set of process (de-)activations $Proc$ is ignored here, because activations have been done already when installing the pipeline or group[12].

[12]However, redundant activations are harmless; redundant deactivations shouldn't be different. So $Proc$ may as well be transmitted to the above levels, allowing for this chain of effects.

An interesting case is of course that of the termination of the current handler, i.e., when $Pipe = \emptyset$ and the new current block is `END`. An interesting way to make this transition is to fake a `break` event inside the manifold and add it to the local events: $E_{loc} \cup \{\texttt{break.self}\}$. In most cases, this event will be filtered out of E at application level, because there is usually no handler for `break`, which makes it unobservable. However, this may be useful for a predefined construct like the sequencial operator manner. This manner, for example, captures the `break` event and handles it [1].

Termination takes places when there is nothing left to do. This is when the current handler is terminated (i.e., $C =$`END`) and there is no event occurrence in E that can preempt this state.

Rule 31 *: termination of a manifold*

$$\frac{E \cap (\textit{preemptive-events}(P) \cup \textit{priority-events}) = \emptyset}{\langle P, \texttt{END}, E\rangle \xrightarrow[\{\texttt{death}.P_{name}\},\emptyset]{}_p \langle P, \texttt{deactivated}, E\rangle}$$

Termination in a manner means a return from the manner only. In this case, a `returned.self` event is raised in the environment of the caller. This event may be a preemptive event in the caller's environment.

Rule 32 *: termination of a manner*

$$\frac{E \cap (\textit{preemptive-event}(P) \cup \textit{priority-events}) = \emptyset}{\langle\ \texttt{manifold}\ P_{name}\{\texttt{manner}M_{name}\{M_{blocks}\}\ blocks\}, \texttt{END}, E\rangle \xrightarrow[\emptyset,\emptyset]{}_p \langle \texttt{manifold}\ P_{name}\{blocks\}, \texttt{END}, E \cup \{\texttt{self.returned}\}\rangle}$$

4.4.3 Search of the handling block for an event

The search to find an appropriate handler for an event occurernce is defined using a new transition system. The states of this transition system have one of the following three forms.

- A list of 0 or more manner calls, "stacked" as described in rule 19 on top of the blocks of a manifold:
$$\{manner^* block^+\}$$
.
- The terminal state when the search is successful is a pair that contains a new arrangement of the list of blocks, together with the new current block:
$$\langle\{manner^* block^+\}, block\rangle$$
.
- The terminal state when the search is unsuccessful is an empty list of blocks: {}.

The transition relation is $\xrightarrow{e}_{search}$, where e is the event for which a handling block must be found (i.e., whose labels contain e).

The target handler must be looked for in the deepest called manner. If it is there, then the corresponding terminal state is reached:

Rule 33 *: successful search in manner*

$$\frac{\{M_{blocks}\} \xrightarrow{e}_{search} \langle\{M'_{blocks}\}, l_c{:}a_c\rangle}{\{\texttt{manner}\ M_{name}\{M_{blocks}\}\ blocks\} \xrightarrow{e}_{search} \langle\{\texttt{manner}\ M_{name}\{M'_{blocks}\}\ blocks\}, l_c{:}a_c\rangle}$$

If the target handler is not found in the deepest called manner, the manner is preempted by the event. This means that the event `returned.self` is *not* raised in the environment of its caller, but the manner is terminated and the search continues in the environment of its caller [1].

Rule 34 *: unsuccessful search in manner*

$$\frac{\{M_{blocks}\} \xrightarrow{e}_{search} \{\}}{\{\texttt{manner}\ M_{name}\{M_{blocks}\}\ blocks\} \xrightarrow{e}_{search} \{blocks\}}$$

At each level, a circular search is performed among the event handling blocks of the current environment to find the appropriate handler. We denote this search by the relation $\xrightarrow{e}_{circ}$ defined in Section 4.4.4. The success or failure of our search depends on the success or failure of this circular search at each level.

Rule 35 *: successful search*

$$\frac{\langle\{l_1{:}b_1.\ blocks\}, \{\}\rangle \xrightarrow{e}{}^{*}_{circ} \{l_c{:}b_c.\ blocks'\}}{\{l_1{:}b_1.\ blocks\} \xrightarrow{e}_{search} \langle\{l_c{:}b_c.\ blocks'\}, l_c{:}b_c.\rangle}$$

Rule 36 *: unsuccessful search*

$$\frac{\langle\{l_1{:}b_1.\ blocks\}, \{\}\rangle \xrightarrow{e}{}^{*}_{circ} \{\}}{\{l_1{:}b_1.\ blocks\} \xrightarrow{e}_{search} \{\}}$$

4.4.4 The circular search

The circular search is also defined by a transition system. The states of this transition system are one of the following.

- A pair of lists of blocks $\langle\{block^*\}, \{block^*\}\rangle$, the first of which contains the blocks to be searched, and the second one the blocks already searched.
- The terminal state $\{block^*\}$, reflecting the new arrangement of the list of blocks, where the new current block just found is the first element. If the search fails, the list of blocks is empty, and the terminal state is { }.

The transition relation is $\xrightarrow{e}_{circ}$, where e is the event for which a handling block must be found. The event e is matched against the labels of each block, considering the wild-card character $*$ (see Section 3.1). We use $e \in labels(b_1)$ to denote a successful match of the event e with the labels of the block b_1.

Circular search: if the event for which we are looking for a handler is not in the labels set of the first block b_1, then the first block is appended at the end of the list of the already searched blocks, which reflects the *circularity* of the search.

Rule 37 *: circular search*

$$\frac{e \notin labels(b_1)}{\langle \{b_1\ blocks\}, \{blocks'\} \rangle \xrightarrow{e}_{circ} \langle \{blocks\}, \{blocks'\ b_1\} \rangle}$$

Handler found: if the event e is in the labels of the first block, then the search ends successfully:

Rule 38 *: successful circular search*

$$\frac{e \in labels(b_1)}{\langle \{b_1\ blocks\}, \{blocks'\} \rangle \xrightarrow{e}_{circ} \{b_1\ blocks\ blocks'\}}$$

The new arrangement of the list of blocks is obtained by placing the mathed block in the first position, followed by the remaining blocks that were not searched, and then by the blocks that were searched before.

Unsuccessful search : The end of the search is reached when the list of blocks to be searched is exhausted.

Rule 39 *: unsuccessful circular search*

$$\langle \{\}, blocks' \rangle \xrightarrow{e}_{circ} \{\}$$

4.5 The Application Level

The state of an application is the set of the states of all the processes (manifolds and atomic processes) in that application. The application level transition relation is denoted by $\longrightarrow_a$.

4.5.1 Concurrency of processes

An application changes when any of its *active* processes causes a change, possibly by raising events that must then be propagated to the rest of the processes in the application:

Rule 40 *: transition at application level*

$$\frac{\exists p \in A,\ p \xrightarrow[E_{ext}, Proc]{} p',\ A' = (A \setminus \{p\}) \cup \{p'\}}{A \longrightarrow_a \{\ p'' \mid \exists p' \in A',\ p'' = diffact(p', E_{ext}, Proc)\ \}}$$

The MANIFOLD application state A is changed by:

- replacing previous state p of the process by its new state p' (this can also be written as $A[p'/p]$). The operator $\exists$ represents non-determinism and asynchrony.
- diffusing the possibly empty set of raised events E_{ext} and activating or deactivating the processes in $Proc$.

As an extension, the *diffact* function can also verify that the current pipelines installed in p' can be actually built, i.e., that the processes involved actually exist and are active in the application. If this verification fails, the event **dead** is added to the events memory of p'.

$diffact(\langle P, C, E\rangle, E_{ext}, Proc) =$

```
   if   C= inactive
then      if   name(P)+ ∈ Procs
          then  ⟨P, inactive, E ∪ {start.self}⟩
           end
 else      if   C = deactivated
          then  ⟨P, C, E⟩
           else      if   name(P)− ∈ Procs
                    then  ⟨P, C,
                            (E ∪ E_ext ∪ {terminate.self})
                                  ∩observable(P)⟩
                     else  ⟨P, C,
                            (E ∪ E_ext)∩observable(P)⟩
                     end
           end
 end
```

Table 3: Diffusion of events in a process: the function *diffact*.

4.5.2 Diffusion of events and (de-)activation of processes

The function $diffact(P, E_{ext}, Proc)$ in table 3, describes the reception of events in E_{ext} by each process, considering what this latter can *observe* and can currently handle. It also performs the activation or deactivation of processes named in $Proc$.

The set of observable events is a function of the state of a manifold process P [1]. Observable event sources for a manifold process P are:

- All processes referenced in the blocks of the manifold. Because of manner calls, this set is state dependent.
- The ports of the process P.
- The keyword `self` designating the process P itself.

Observable events are those that:

- come from an observable source, and,
- match the label of one of the blocks other than the current block (i.e., they have a handler, different than the current block).

Intersecting the whole event memory E of a process with the set of observable events has the effect that event occurrences in E that have been received earlier and not yet handled, and that are not observable any more (e.g., due to a manner exit), are filtered out. This corresponds to the informal specification declaring them lost [1].

4.6 Global Network

The state of the global network connecting all processes can be obtained from the state of a MANIFOLD application by a *multi-union* of the local networks of its individual processes. This results in a *multi-set* where an element can be redundantly present. Redundant elements in this multi-set mean that a stream is redundantly installed [1] (see Figure 4). Each local network is a function of the current state of its manifold.

The elements of the network of streams are pairs $\langle port_1, port_2\rangle$ of port names[13].

[13] In a later stage of the formal specification work, when we consider the data-flow aspect of MANIFOLD and the transfer of units, we must also include the stream type tags flushing/non-flushing.

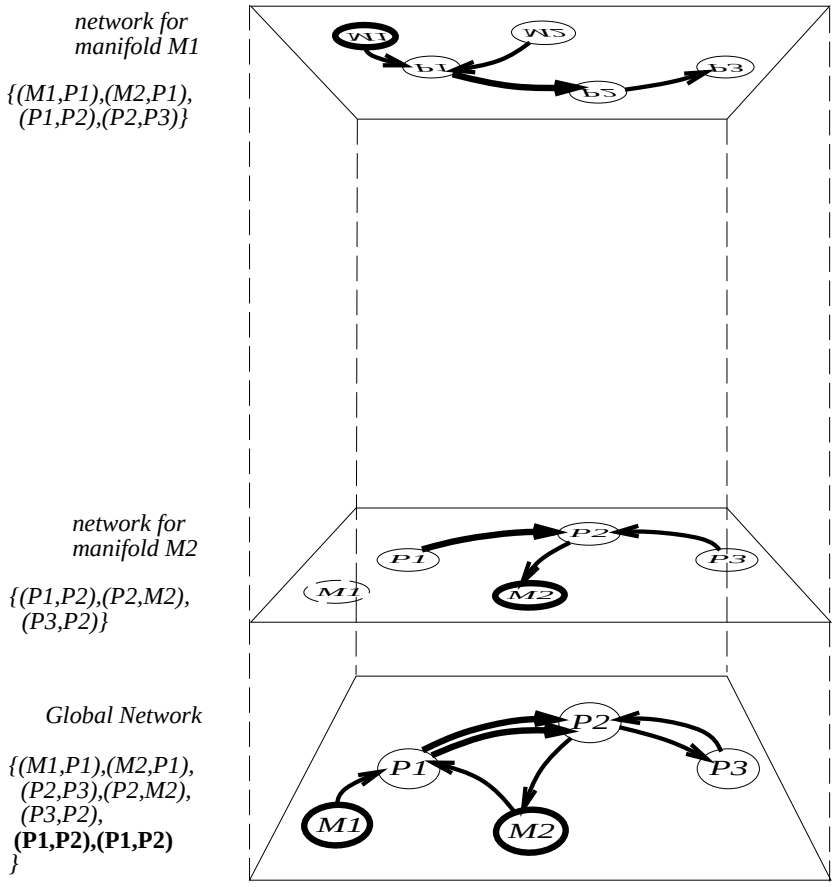

Figure 4: Local and global networks of streams in a MANIFOLD application.

The network is a *multi-set* in order to represent the property that a stream disappears only when dismantled as many times as it was installed. The global network is defined by the function $net(A)$. It is the *multi-union* $\overset{*}{\cup}$ of local networks controlled by individual manifold processes.

$$net(A) = \overset{*}{\bigcup}_{p \in A \cap manifolds} net(p)$$

The local network controlled by a manifold process is given by the state of its current handler.

$$net(\langle \texttt{manifold}\ P_{name}\ l_c : a_c\{blocks\}, E, \texttt{active}\rangle) = net(a_c)$$

The network implemented by a group is the multiunion of the streams constructed in the pipeline members of the group. For a binary group, the following rule applies:

$$net(\ (a_1, \ \ldots\ a_i\ , \ \ldots\ a_n)\ ) = \overset{*}{\bigcup}_{i \in [1..n]} net(a_i)$$

If an action is a pipeline construction, then the result is its corresponding set of streams, and otherwise the empty set $\emptyset$:

$$net(a) = \begin{cases} \bigcup_{i:1..n} net(S_i) & \text{if } a = [\ S_1\ ,\ \ldots\ S_i\ \ldots\ S_n\ ] \\ \emptyset & \text{otherwise} \end{cases}$$

For a single stream, the net is its corresponding singleton:

$$net(Port_1\text{->}Port_2) = \{\langle Port_1, Port_2\rangle\}$$

5 Results and Current Activity

The language considered in this report is an approximation of the original language [1]. It does not include the data-flow aspects of MANIFOLD (the inside view of streams, transfer of units, and their interaction with the event mechanism of MANIFOLD). Some of the details of the event mechanism of MANIFOLD are also left out of this report.

One of the results of our work on the formal specification of MANIFOLD is the implementation of an interpreter. This interpreter enabled us to "validate" our formal specification. We have also used it as a tool to experiment with the definition of the language itself.

Our present ongoing work on formal specification of MANIFOLD includes:

An extension of the application level transitions As presently defined, the application level transitions linearize the concurrent reactions of parallel processes. However, there is no consideration for the time it takes to react, i.e., the time needed to find the handler for an event. Between the selection of an event occurrence to reacted to, and the effects of its handling, another process may make a transition.

Thus, it may be interesting to dissociate selection of an event occurrence for handling (as a kind of read operation on the state of the application) and effects of its handling (as a kind of write operation). This would be closer to the reality of the execution and communication model of MANIFOLD.

Stable states of an application The stable states of an application can be characterized as those states wherein the netwok of streams is stable and no preemption is possible by the present set of all event occurrences. No transition is possible from a stable state unless a new event occurrence arrives from an external source, e.g., an atomic process or a pseudo process primitive action such as a **guard**. In a stable state, the data-flow network is active and units are travel through streams. There is some resemblence between our notion of stable states and the concept of stability in Grafcet [10].

Using the notion of stable states, a MANIFOLD application can be seen as a system that makes "global transitions" between stable states. These transitions depend on termination of the lower-level (application-level) transition relation $\longrightarrow_a$. The property of being *stable*, for an application A, can be defined as:

$$stable(A) = \bigwedge_{\langle P,E\rangle \in A} (E \cap (preemptive(P) \cup \ priority\text{-}events \cup \ evolution\text{-}events\text{-}set) = \emptyset)$$

The global transition relation between stable states is defined in terms of several steps of the "unstable" application transitions, as:

$$\frac{stable(A), A \longrightarrow_a^* A', stable(A')}{A \longrightarrow_s A'}$$

It may not always be possible to attain a stable state. Some programs may go into a kind of loop through a tandem of "unstable" states. An interesting area for future work is to define a method to detect such "unstable" programs.

A prototype interpreter We are implementing a prototype interpreter for the subset of MANIFOLD considered in our work on its formal specification. The interpreter is being used to validate and "debug" the rules presented in this report. It consists of a quite direct encoding of the rules in QUINTUS PROLOG, where non-deterministic choices are resolved through interaction with the user. In a future version of the interpreter, we may want to explore the whole space of possible executions. However, this requires that termination must be assured.

A graphical user interface We plan to use the above mentioned interpreter to feed a prototype graphical user interface that is under development for MANIFOLD. This, on the one hand, gives the interpretor a user-friendly interface, and on the other hand, the new concepts of the interface (e.g., the 3-D representations of MANIFOLD applications, various debugging and tracing functionalities, etc.) can be experimented with using the interpretor. Given that the interpreter is much smaller and simpler than the actual implementation of the MANIFOLD run-time system, this facilitates developement of the graphical user interface.

6 Conclusions and Future Work

This report documents a first step towards a complete and formal specification of the MANIFOLD model and language. It presents the specification of a subset of the MANIFOLD language as defined in its original informal specification [1].

We intend to continue our formal specification work to cover the whole MANIFOLD language, including streams and the transportation of units. The interactions between the streams and the event mechanism of MANIFOLD is indeed an important and complex aspect of the language. We may also examine other formal specification methodologies in order to evaluate their adequacy compared to our present choice of hierarchical transition systems. Also, a systematic exploration of the definition of the primitive actions can be made in an orthogonal way. For example, we may consider primitive actions based on their effects on the four-tuple $\langle E_{loc}, E_{ext}, Pipe, Proc \rangle$.

Another direction for future work is to concentrate on the modeling methodology itself, and study alternative ways to analyse the behavior and properties of MANIFOLD programs based on their formal semantics. The semantics presented in this report gives a clear view of the behavior of the language. Other representations, such as a finite-state automata encoding of the states and transitions of MANIFOLD, can be derived from this model.

Acknowledgement

We appreciate the effort of all members of the MANIFOLD group (Paul ten Hagen, Freek Burger, Per Spilling, Kees Blum, Anco Smit, and Dirk Soede) for their direct and indirect contributions to this paper.

7 References

[1] F. Arbab. *Specification of* MANIFOLD. CWI Report, Interactive Systems Dept., to be published, 1992.

[2] F. Arbab, I. Herman. *Examples in* MANIFOLD. CWI Report, Interactive Systems Dept., CS-R 9066, 1990.

[3] F. Arbab, 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.

[4] F. Arbab, I. Herman, P. Spilling. *An overview of* MANIFOLD *and its implementation.* CWI Report, Interactive Systems Dept., CS-R 9142, 1991.

[5] A. Benveniste, G. Berry. The synchronous approach to reactive and real-time systems. *Another look at real-time programming, Proceedings of the IEEE*, special section, September 1991.

[6] G. Berry. *A hardware implementation of pure* ESTEREL. DEC Paris Research Laboratory, Report n^0 15, July 1991.

[7] G. Berry, G. Gonthier. *The* ESTEREL *synchronous programming language: design, semantics and implementation.* INRIA Research Report n^0 842, 1988.

[8] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud. The synchronous data-flow programming language LUSTRE. *Another look at real-time programming, Proceedings of the IEEE*, special section, September 1991.

[9] P. Le Guernic, T. Gautier, M. Le Borgne, C. Le Maire. Programming real-time applications with SIGNAL. *Another look at real-time programming, Proceedings of the IEEE*, special section, September 1991.

[10] L. Marcé, P. Le Parc. Modélisation de la sémantique du Grafcet à l'aide de processus synchrones. In *Proceedings of Grafcet '92*, Paris, 25 – 26 Mars 1992.

[11] G.D. Plotkin. *A structural approach to operational semantics.* Aarhus University, Computer Science Dept., Report DAIMI FN–19, Sept. 1981.

[12] G.D. Plotkin. An operational semantics for CSP. In D.Bjørner (ed.), *Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts – II*, Garmisch-Partenkirchen, Germany, 1–4 June 1982, North-Holland.

[13] D. Soede, F. Arbab, I. Herman, P. ten Hagen. *The GKS input model in* MANIFOLD. CWI Report, Interactive Systems Dept., CS-R 9127, 1991.