1992

F. Arbab

Specification of Manifold
Version 1.0

Computer Science/Department of Interactive Systems Report CS-R9220 June

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

Specification of MANIFOLD

Version 1.0

Farhad Arbab

CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

ABSTRACT

Management of the communications among a set of concurrent processes arises in many applications and is a central concern in parallel and distributed computing. In this report, we introduce MANIFOLD: a coordination language for managing complex interconnections among sets of independent, concurrent, cooperating processes. The MANIFOLD system is suitable for massively parallel environments. Both the conceptual model of MANIFOLD and the specification of the MANIFOLD language are presented here.

The conceptual model behind MANIFOLD is based on the principle of separating functionality from communication. This model immediately leads to a very simple, but non-conventional model of computation. Contrary to most other models, *computation* in MANIFOLD is built out of *communications*. As such it advocates a view point reminiscent of the connectionist view: that all (conventional) computation can be expressed as interactions.

The primary concern in the MANIFOLD language is not with *what* functionality the individual processes in a parallel system provide. Instead, the emphasis is on *how* these processes are inter-connected and how their interaction patterns change during the execution life of the system.

1987 CR Categories: C.1.2, C.1.3, C.2.m, D.1.3, D.3.2, F.1.2, I.1.3.
1985 Mathematics Subject Classification: 68N99, 68Q10.
Keywords and Phrases: parallel computing, coordination languages, MIMD, models of communication.

Report CS-R9220
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Table of Contents

Specification of MANIFOLD

Version 1.0

Farhad Arbab

Interactive Systems
Center for Mathematics and Computer Science
Kruislaan 413
1098 SJ Amsterdam
The Netherlands
Telephone: +31 20 5924056
Fax: +31 20 5924199
Telex: 12571 mactr nl
Email: farhad@cwi.nl

1. Introduction

Specification and management of the communications among a set of concurrent processes is at the core of many problems of interest to a number of contemporary research trends. The theory of neural networks and the connectionist view of computation emphasize the significance of the concept of *management of connections* versus the local computation abilities of each node. The concept of dataflow programming[9,19] has a certain resemblance with connectionism, albeit, it is closer to the discrete world of conventional programming than neural networks. Theoretical work on concurrency, e.g., CCS[13] and CSP[10], is primarily concerned with the semantics of communications and interactions of concurrent sequential processes. Communication issues also come up in virtually every other type of computing, and have influenced the design (or at least, a few constructs) of most programming languages. However, not much effort has been spent on conceptual models and languages whose sole prime focus of attention is on the coordination of interactions among processes.

It is advantageous to make a distinction between *computational models and languages* versus *coordination models and languages*[5]. Gelernter and Carriero correctly observe that relatively little serious attention has been paid in the past to the latter, and that "ensembles" of asynchronous processes (many of which are off-the-shelf programs) running on parallel and distributed platforms will soon become predominant[5].

In this paper, we introduce MANIFOLD: a language whose sole purpose is to describe and manage complex interconnections among independent, concurrent processes. As such, MANIFOLD is primarily a coordination language. We describe here both the conceptual model of MANIFOLD and the syntax and (the informal) semantics of the MANIFOLD language. Formal semantics of the MANIFOLD language is given elsewhere[14]. The highlights of our first implementation of the MANIFOLD system appears in another paper[3]. Examples of the use of the MANIFOLD language are given in various other reports and papers[1,3,16]. A comparison of MANIFOLD and some related models and systems is presented in a separate report[2].

The rest of this paper is organized as follows. The motivation for the MANIFOLD system is presented in §2. The purpose of the material in §3 is to give the reader an initial intuitive feeling for MANIFOLD programming. This can serve as a skeleton to which the details of the model and the language can later be attached. The MANIFOLD model is described in §4. A specific language based on the MANIFOLD model is described in §5. Because MANIFOLD is an open system, it is necessary to describe how non-MANIFOLD entities can be introduced and used within a MANIFOLD application. Such inherently implementation dependent specifications are left out of the MANIFOLD language proper, and are described as pragmas in §6. The MANIFOLD compiler directives, its pre-processor facilities, and the system's builtin processes are described, respectively, in §7,

§8, and §9. Finally, the description of the regular expressions used in the MANIFOLD language appears in Appendix A and the formal syntax of the language is given in Appendix B.

2. Motivation

Parallel programming is important as the means by which the computational power of several processors can be simultaneously concentrated on a single application. Conceptually, however, the significance of parallel programming goes beyond such "performance" issues: it is also a rich paradigm wherein even some of typically sequential programs can be expressed more naturally as the cooperation between a group of smaller sequential fragments running concurrently (on the same or different processors). We use the term *process* to refer to such a fragment.

Cooperative communication among a set of autonomous processes seems to lead to an effective and natural architecture in many complex systems. Scientific visualization, sophisticated user interface design, complex multidisciplinary systems such as intelligent CAD/CAM systems for concurrent engineering, and distributed open systems required for factory automation are but a few examples of emerging applications where parallelism is perhaps even more important as a conceptual tool to tackle their complexity, than as a means to achieve realistic performance.

While the hardware technology of parallel computers is still far from its maturity, it has clearly already passed its infancy. Today, tightly coupled architectures with a very large number of processors are a commercial reality, and loosely coupled networks of processors are commonplace. Major advances in parallel hardware technology are still necessary to meet the challenges of the new application areas with inherent parallelism and truly distributed scopes. Nevertheless, it is a fact that the potential of the existing parallel hardware technology is drastically underutilized in today's applications. One reason for this phenomenon is the usual lag between advances in hardware technology and the development of the proper software. However, in parallel and distributed computing, there are deeper impediments in the way of turning the full potential of parallel hardware into the reality of parallel applications.

One of the fundamental problems in parallel programming is coordination and control of the communications among the sequential fragments that comprise a parallel program. Programming of parallel systems is often considerably more difficult than (what intuitively seems to be) necessary. It is widely acknowledged that a major obstacle to a more widespread use of massive parallelism is the lack of a coherent model of how parallel systems must be organized and programmed. This is a conceptual problem that cannot be overcome by more advances in hardware technology alone. On the contrary, more advanced massive parallel hardware is likely to exacerbate this problem by widening the gap between the potential of the raw computing power made available, and the reality of how it is used to build parallel applications.

To complicate the situation, there is an important pragmatic concern with significant theoretical consequences on models of computation for parallel systems. Many user communities are unwilling and/or cannot afford to ignore their previous investment in existing algorithms and "off-the-shelf" software and migrate to a new and bare environment. This implies that a suitable model for parallel systems must be *open* in the sense that it can accommodate components that have been developed with little or no regards for their inclusion in an environment where they must interact and cooperate with other modules.

Many approaches to parallel programming are based on the same computation models as sequential programming, with added on features to deal with communications and control. This is the case for such concurrent programming languages like Ada[6, 18], Concurrent C[7, 8], and many others (the interested reader may consult, e.g., the survey of Bal et al[4] for more details on these languages). There is an inherent contradiction in such approaches which shows up in the form of complex semantics for these added on features. The fundamental assumption in sequential programming is that there is only one active entity, *the* processor, and the executing program is in control of this entity, and thus in charge of the application environment. In parallel programming, there are many active entities and a sequential fragment in a parallel application cannot, in general, make the convenient assumption that it can rely on its incrementally updated model of its environment.

To reconcile the "disorderly" dynamism of its environment with the orderly progression of a sequential fragment "quite a lot of things" need to happen at the explicit points in a sequential fragment when it uses one of the constructs to interact with its environment. Hiding all that needs to happen at such points in

the communication constructs makes their semantics complex. Inter-mixing the neat consecutive progression of a sequential fragment – focused on a specific function – with updating of its model of its environment and explicit communications with other such fragments, makes the dynamic behavior of the components of a parallel application program difficult to understand. This may be tolerable in applications that involve only small scale parallelism, but becomes an extremely difficult problem with massive parallelism.

Contrary to languages that try to hide as much of the "chaos of parallelism" as possible behind a facade of sequential programming, MANIFOLD is based on the idea that allowing programmers to see and feel all aspects of parallelism is actually beneficial. The idea of exposing the full view of parallelism is also shared by some other languages, notably LINDA[5]. Although LINDA is based on a very different model of communication than that of MANIFOLD, they both advocate the view that there often *is* a pay-off in using parallel or distributed programming, even if higher speeds are not (necessarily) achieved.

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

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

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

3. MANIFOLD Programming

Programming in MANIFOLD is organizing the activities of a set of concurrent processes and managing their mutual interactions. Orchestration of the interactions among a set of processes in MANIFOLD is done in an entity with multiple inlets and outlets, called a *manifold*, which itself is a process. As the conductor of such interactions, a manifold has a number of *states*, each defining a specific connection pattern. Connection patterns define buffered links between the input and output ports of various processes, called *streams*, through which the information produced by one process is made available for consumption to another.

The streams among processes in MANIFOLD form a network of links for the flow of information that is reminiscent of data-flow networks[19]. However, there are several major differences between MANIFOLD and data-flow programming. In MANIFOLD the connection patterns among processes change dynamically. Furthermore, processes are created and deleted dynamically as well. This by itself makes the connections graph of a MANIFOLD program, which is the combined effect of all its manifolds, very dynamic. However, there is more. The manifestation of a single manifold is of course a single (dynamically changing) process interconnection graph. Since manifolds too are processes, their input and output ports can be connected to other processes by other manifolds as well. The combined graph of the stream connections in a MANIFOLD program is indeed not a simple static graph.

A manifold goes through state transitions as a result of observing in its environment the occurrences of *events* in which it is interested. State transitions cause dismantling of the interconnections set up in pre-transition states, and establish the ones defined in the post-transition states. As such, events are the principal control mechanism in MANIFOLD, which makes it an event driven programming system. The coexistence

of event driven and data driven control gives MANIFOLD a unique flavor.

4. The MANIFOLD Model

The basic components in the MANIFOLD model are *processes*, *events*, *ports*, and *streams*. A process is a *black box* with well defined ports of connection through which it exchanges *units* of information with the other processes in its environment. The internal operation of some of these black boxes are indeed written in the MANIFOLD language, which makes it possible to open them up, and describe their internal behavior using the MANIFOLD model. These processes are called manifolds. 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 process in MANIFOLD.

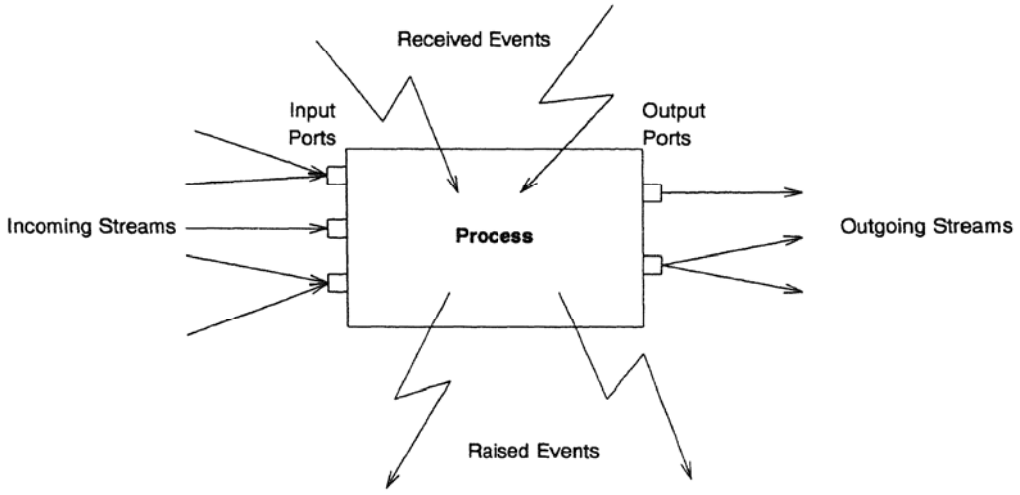


Figure 1 - The model of a process in MANIFOLD

The interconnections between the ports of processes are made with streams. A stream represents a flow of a sequence of units (of information) between two ports. Streams are constructed and removed dynamically between ports of the processes that are to exchange some information. The constructor of a stream need not be the sender 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. The flows of units of information in streams are automatically replicated and merged at outgoing and incoming port junctions, as necessary. The units of information exchanged through ports and streams, are *passive* pieces of information that are produced and consumed at the two ends of a stream with their relative order preserved. The consumption and production of units via ports by a process is analogous to read and write operations in conventional programming languages.

Independent of the stream mechanism, there is an event mechanism for information exchange in MANIFOLD. Contrary to units in streams, events are *atomic* pieces of information that are *broadcast* by their sources in their environment. In principle, *any* process in an environment can pick up such a broadcast event. In practice, usually only a few processes pick up occurrences of each event, because only they are *tuned in* to their sources.

Two state dependent sets determine how events are handled in a manifold process. First, the set of *observable events*, O, defines those events (and their sources) whose occurrences are to be picked up by the manifold process from its environment. These are the only events that a manifold can possibly react to in each of its states. Occurrences of all other events are completely ignored. Once an event occurrence is

picked up by an observer manifold, it may or may not cause an immediate reaction by the observer. The state-dependent *preemption set*, P, of a manifold process is a subset of its observable events and defines exactly those events whose occurrences are to be reacted to in that state.

The only way that a manifold process can react to an event occurrence is to preemptively make a state transition from its current state to a target state that is designated to handle that event occurrence. Determination of the appropriate target to handle an event occurrence is also state dependent. Once an event occurrence is handled (i.e., it has caused a state transition) it is cleared. Occurrences of events in $O - P$ are *saved* and may be handled later, once P is re-defined after an appropriate state transition to include them. Occurrences of events in O that are no longer in O after a state transition are lost.

When there is more than one event occurrence in P, a manifold process reacts to one of them non-deterministically, and all others are saved. 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 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.

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

4.1. Processes

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

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

The assumption that each process has its own private processor and memory, reinforces its autonomy. It also indicates that processes in MANIFOLD have no means of communications other than their port/stream connections and the event mechanism. The MANIFOLD model of communication (events and streams) is powerful enough to support all forms of interprocess communication. Therefore, in principle, there is no need for other forms of communication among processes. In practice, however, it may be desirable to allow other forms of inter-process communication. For example, processes may need to communicate and influence each other through other means for purposes such as resource management, job control, side effects (e.g., files), interaction with the real world, etc., and may use mechanisms such as message passing, shared memory, etc. While MANIFOLD implementations should not preclude such communications, *they assume that all communication of interest with a process takes place through its input and output streams and via events.*

There are two kinds of processes: *atomic processes* and *manifolds*. In general, a process in MANIFOLD does not, and need not, know the kind or identity of the processes with which it exchanges information. An atomic process is similar to a black box whose internal structure and behavior are unknown. In fact, an atomic process is any processing element whose external behavior is all that one is interested to observe at a given level of abstraction. The set of atomic processes is application dependent, and thus, is neither predefined nor fixed. Examples of atomic processes include processes written in some programming language other than MANIFOLD, a hardware device, and a person interacting with a program.

A manifold is a process whose behavior and structure are described in the MANIFOLD language by a *manifold definition* (see §5.2). Manifolds "orchestrate" the communication and interaction among processes (atomic processes and other manifolds alike), and provide a dynamic means of control over a multiprocessing environment. A processor that runs a manifold is called a *manifold processor*.

4.2. Streams

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

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

Conceptually, a stream in MANIFOLD has an infinite capacity that is used as a FIFO queue, enabling asynchronous production and consumption of units by its source and sink. The sink of a stream requiring a unit is suspended only if no units are available in the stream. The suspended sink is resumed as soon as the next unit becomes available for its consumption. The source of a stream is never suspended because the infinite buffer capacity of a stream is never filled. In practice, however, MANIFOLD implementations may behave unpredictably when memory is no longer available to accommodate additional units in streams.

Streams in MANIFOLD are dynamically constructed and dismantled (see §5.2 and §5.12). The MANIFOLD model supports two types of streams: *flushing* streams and *non-flushing* streams (see §5.8.5 for how the two types of streams are constructed in the MANIFOLD language). When a non-flushing stream is dismantled, the units that have already been produced by its source and not yet consumed by its sink, if any, remain queued up behind the sink for its eventual consumption. When a flushing stream is dismantled, all such leftover units inside the stream are discarded.

4.3. Ports

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

While streams are independent entities outside of processes, ports are properties of processes and are defined and owned by them. Information placed into one of its output ports by a process flows out of the port only when it is connected to a stream. This ensures that no information is lost if a process writes to its output port while it is not connected to any stream. A MANIFOLD implementation may provide internal buffers for input and output ports of processes to increase the parallelism between the execution of a process and its input/output operations.

Ports of a manifold process may raise a number of pre-defined events that can be observed only by that process. Thus, in addition to processes, ports can also be used to designate sources of events. Note that there is no way for a process to observe or react to an event raised by a port of another process.

4.4. Events

An event is an asynchronous, non-decomposable (atomic) message, broadcast by a process or port in its environment. The environment of a port event is its owner process. The environment of a process event includes all running processes in the same application. Broadcasting such a message is called *raising the event*. Events are identified by their names, and can also be distinguished based on their sources (except, perhaps, when they are raised by atomic processes; see §6.3).

Although conceptually, an event is broadcast when it is raised, only a subset of the processes in its environment can pick up the broadcast and react to it. A process that picks up an occurrence of an event is called an *observer* (of the event and of its source). For a manifold to pick up a broadcast event occurrence, the event and its source must be *observable* to the manifold.† In general, a manifold reacts only to a subset of the events from its observable sources. These are called the *observable events* and are the ones for which the manifold processor has an event handler.‡ Only the observable events can affect a manifold. All other events are ignored.

Reacting to an occurrence of an observed event always causes a preemptive change of state in an observer manifold. However, this does not necessarily happen *immediately* after the event occurrence is observed. Occurrences of all observable events are first saved in a state-independent memory of a manifold instance and comprise its set of pending event occurrences. Each state in a manifold defines a set of events whose occurrences may preempt that state. This set is called the *preemption set* (of that state). Preemption sets are subsets of the observable events of a manifold. Observable event occurrences that are not in the preemption set of the current state, are *saved* (i.e., they remain in the set of pending event occurrences). Because the pending event occurrences are kept in a set, further occurrences of the same event from the same source are lost. Thus, a mismatch between the occurrence frequency of an event from a given source and the reaction time of an observer creates an automatic *sampling* phenomenon. Occurrences of different events from the same source, or the same event from different sources, do not override each other.

A number of system events (e.g., **death**) in MANIFOLD have priority over other events. Occurrences of these events in the preemption set of a state are considered first for handling. Subject to this priority, selection of event occurrences in the preemption set for handling is non-deterministic.

Occurrences of events are synchronized with the flow of information in the streams through unit boundaries. Thus, streams always start and end with complete units. Events travel no slower than units. This means that if a process A produces an event e before it produces an output unit u, any other process B that receives both e and u, will see e no later than it can see u.

The concept of events in MANIFOLD is different than the concepts with the same name in most other models, notably simulation languages or CSP. The 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. This raised flag can potentially be seen by any process in the environment of its source. Indeed, it can be seen by any process for which the source of the event is *observable* (see §5.2.3 for observability rules). However, there are no guarantees that a raised flag will be observed by anyone, nor that if observed, it will make the observer react.

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

4.5. An Abstract Model of MANIFOLD

In this section we present an abstract model for the MANIFOLD system. The model presented here is a simplified one. Its purpose is to show only the basic behavior of a MANIFOLD process. In particular, the stack oriented facility for dynamic nesting of subroutine-like environments (called manners in the MANIFOLD language), are completely ignored.

---

† Observability rules in the MANIFOLD language are defined in §5.2.3.

‡ Event handling in the MANIFOLD language is described in §5.2.1. See also §5.2.

An executing MANIFOLD application consists of a medium for propagation of event occurrences wherein instances of two types of active entities interact: *processes* and *streams*. The MANIFOLD model defines the behavior of streams and the subset of non-atomic processes called manifolds. Instances of streams and manifolds can be created and deleted dynamically.

To create a stream instance a producer port and a consumer port must be specified. To create a manifold instance, a manifold and a set of actual parameters must be specified to substitute its formal parameters. Similarly, atomic process instances can also be created dynamically, but the internals of these processes are completely hidden from MANIFOLD.

4.5.1. An Abstract Model of a Stream

A stream is a simple finite state machine with two main states: *dormant* and *active*. Stream instances connect ports of processes. While any number of streams can exist between any pair of source and sink ports, only one stream instance connecting a particular pair of ports can be active at any time.

Once an instance of a stream is created, it checks to see if there are any other stream instances connecting its own source and sink ports. If so, the new stream instance goes into dormant mode, otherwise it becomes active. In active mode, a stream simply copies everything it receives from its source to its sink.

When an instance of a stream is to be deleted, it quietly disappears if it is in its dormant mode. When an active stream is to be deleted, it again checks to see if there are any other stream instances connecting its own source and sink ports. If so, it selects one of them non-deterministically, makes it active, and then disappears.

4.5.2. An Abstract Model of a Manifold Instance

A manifold instance is an abstract machine with a finite set of states Σ, and a memory E. Among the primitive operations that this machine can perform are creation and deletion of stream instances, creation of process instances, and broadcasting of events. Every primitive operation completes in finite time.

The memory of a manifold instance is used only to record the set of event occurrences it observes in its environment. Observed event occurrences are removed from this memory when they are handled. Otherwise, this memory persists through state transitions.

Each state in Σ is a tuple $<A,O,P,T>$. A is a list of primitive operations, called the action list of the state. O is a set of events (and their sources) called the observable events of the state. P is a subset of O and is called the preemption set of the state. $T: P \rightarrow \Sigma \bigcup \{\textbf{save}, \textbf{ignore}\}$ is the transition function of the state, defining the target state of a transition caused by occurrences of events in P. The function T is a total mapping: any event that is not explicitly bound to a target state is mapped to **ignore**.

In each state, occurrences of events in O are recorded in E. Because E is a set, multiple occurrences of the same event from the same source are recorded only once. A state transition is an atomic operation in the manifold abstract machine. After a transition to a new state $s=<A,O,P,T>$, a manifold abstract machine performs the primitive operations defined in A. Meanwhile, occurrences of observable events are recorded in E, but they cannot cause a state transition before all operations in A are performed.

E is the only history-sensitive part of a manifold abstract machine: after a state transition, it is only E whose value depends on its value in the previous state. All elements in the tuple that defines the current state of a manifold instance are reset to their new values after a transition. The new contents of the memory of a manifold instance after a transition, E', is $E' = E \bigcap O'$, where E is the set of outstanding event occurrences in the state prior to the transition, and O' is the set of observable events in the new state.†

Once all operations in A are complete, a manifold abstract machine and the state it is in become preemptable: an observed event occurrence in $e \in E \bigcap P$ causes a preemptive transition to the new state $s'=T(e)$. The transition from s to s' is preemptive in the sense that (assuming s' is neither **save** nor **ignore**), all

† In the present MANIFOLD language the set of observable events, O, is implicitly defined in each state of a manifold or manner. When a manifold processor enters a manner, O becomes the union of its previous value with that of the called manner. Returning from a manner call restores O to its previous value. Thus, with the existing MANIFOLD language, O changes only when a manifold processor enters or leaves manner invocations.

stream instances created in s are deleted before entering s'. A state transition is made if and only if the manifold abstract machine is preemptable and an event occurrence $e \in E$ is found to be in P. If there is more than one qualifying event occurrence in E, one is selected non-deterministically, subject to a fixed priority scheme. The details of this priority scheme are not important. We simply assume that there exists a fixed and finite number of priority classes for events, and that each event is assigned to a single priority class. There is a total ordering on the priority classes. The event selection criterion states that occurrences of events in a higher-priority class are considered before the ones in a lower-priority class, with non-deterministic selection among the occurrences of the events within the same class. A state transition caused by the event occurrence $e \in E$ involves removing e from E and finding a target state $s'=T(e)$. Event occurrences that are mapped to **save** or **ignore** by T cause no real change of state. When $T(e)$= **save**, e is put back in E for possible handling in a future state. Otherwise, all stream instances created in the state s are deleted and then the new state $s' \in \Sigma$ is entered.

4.5.2.1. State Transitions

A particular subset of actions A in a state $s = <A,O,P,T>$ consists of *pipeline* constructions. A pipeline is a (possibly empty) set of related stream instances and involves a (non-empty) set of processes. By definition, the relationship among the stream instances in a pipeline is such that all are deleted if one breaks. Generally, pipeline construction actions in A create a (possibly empty) set of pipelines L. Every stream instance created in a state s belongs to exactly one pipeline in L.

The three events λ, **break**, and **death** have a special role to play in state transitions. Any event internally raised by a manifold is, by definition, in P, and that includes λ. Furthermore, occurrences of **break** and **death** from any source are, also by definition, in P.

The event λ is internally raised by the manifold abstract machine when it reaches the end of a handling block. The name of this event is disguised (hence, "λ") so that a MANIFOLD programmer can never raise or handle it. Occurrences of **break** and **death** both cause the breakup of the pipelines involving their sources. Occurrences of **break** are handled internally within the same state by the manifold abstract machine and never cause a direct transition to a programmer-specified handler. Occurrences of **death** have a similar effect, but can cause direct preemptive transitions to other states as well.

Following is the sequence of actions that take place during a state transition:

1. Perform all actions specified in A in some non-deterministic order.
2. Let L be the set of pipelines created in step 1.
3. If L is empty, add λ to the set E.
4. Wait for an event occurrence $e \in E \bigcap P$. If there are more than one qualifying event occurrences, select one non-deterministically, subject to a pre-set priority scheme.
   - 4.1. If e is **break** or **death**, find the set B of all pipelines $l \in L$ such that the source of the event e is involved in l. Otherwise, let B be the empty set.
   - 4.2. Dismantle all pipelines in B: delete all streams in all pipelines in B.
   - 4.3. Subtract B from L.
   - 4.4. If e is **break**, remove e from the set E.
5. Find $s' = T(e)$.
6. If s' is **save**, go back to step 3.
7. Remove e from the set E.
8. If s' is **ignore**, go back to step 3.
9. Remove λ, if present, from the set E.
10. Delete all streams in all pipelines in L.
11. Enter the new state s'.

Note that the function T maps **break** to **ignore**, because no explicit handler can be given for **break**. The only possible transition that **break** can cause, is through making L empty (in step 4.3), which after the

transition from step 8 to step 3, ''raises'' the special event λ. The event λ is in a priority class by itself, which has the lowest priority. Thus, the event λ is selected for handling only if there are no other event occurrences in E. A target state for λ always exists. Normally, this target state, $\tau(\lambda)$, is the termination state of the executing manifold process (but, see §5.6 for the special case of the ";" in the MANIFOLD language).

5. The MANIFOLD Language

This section contains the specification of a programming language based on the model described in §4. This language is only one of many possible languages conceivable in the MANIFOLD paradigm. Nevertheless, we refer to it as the MANIFOLD language in this document. The formal syntax of the MANIFOLD language appears in Appendix B. The formal semantics of this language is given in a separate document[14].

5.1. Comments

The text following a "//" on a source code line is a comment and, together with the "//" itself, is ignored by the compiler. In addition, "/*" and "*/" outside of string constants (§5.14.6) and comments starting with "//" symbols, signal start and end of comments, respectively. All text between a "/*" and its matching "*/" (inclusively) is ignored by the compiler.

5.2. 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. The body of a manifold may also contain additional declarative statements, defining its *private* entities. A manifold body primarily consists of a number of *event handler blocks*, representing its different execution-time states.

Conceptually, each activated instance of a manifold definition – a manifold for short – is an independent process with its own virtual processor. A manifold processor is capable of performing a limited set of actions. This includes a set of *primitive actions* (see §5.11), plus the primary action of setting up *pipelines* (see §5.12 and §5.12.5).

Each event handler block describes a set of actions (§5.2.2) and a set of preemptive events. In reaction to a preemptive event, a manifold processor finds its appropriate event handler block and executes the actions specified therein. Often, these actions lead to setting up *pipelines* between various ports of different processes (§5.12). The connections among processes is changed as a manifold changes its state in response to recognizable events. Once a manifold processor is through with the actions in its current block, it waits for the breakup of the pipelines set up in that block, if any, and then terminates.

While a manifold processor is waiting for the breakup of the pipelines set up in the current block, occurrence of a preemptive event diverts the processor from the current block to its corresponding handler block. This always results in the dismantling of all pipelines that were set up in the formerly active block. The processor then proceeds executing the actions specified in the new block.

5.2.1. Event Handling

Event handling in MANIFOLD refers to a 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 that event. An event handler is a labeled block of actions in a manifold (see §5.2.2). 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 pseudo-processes **event_name** and **event_source**, respectively (§5.15.2).

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 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 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. (In the terminology of §4.5.2, O is the set of all event occurrences that match any of the block labels in a manifold or manner definition.) Similarly, if the appropriate block found for an event is the keyword **ignore** (or **save**), the observed event is ignored (or saved, respectively) (see §5.11). Furthermore, events handled by the current block are also ignored (§5.2.3).

Conceptually, transition to a new state (event handling block) causes the previously saved event occurrences to be reconsidered as "just arrived" events, in some non-deterministic order. The first of such events that is allowed to preempt the new block will do so and divert the processor to its corresponding block. The remaining events are, again, saved. Note that once an event handling block is entered, *all* of the actions specified therein are performed before any preemption can occur: event occurrences never preempt execution of the actions in an event handling block; they only preempt the state by deleting the streams created in that state (see §5.12 and §5.12.4).

5.2.2. 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 (see the group construct in §5.12.1) or a single manner call (§5.3).

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 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.

Note that if *s* in a block label *e.s* is a port name, it must be a local port of the manifold. This implies that there is no way for a manifold to react on events raised by the ports of other processes. In MANIFOLD, events raised by ports are strictly local to the manifolds they belong to and are never broadcast to other processes. Consequently, port formal parameters are not allowed as block labels in manifolds, and they are meaningful in manners only if their corresponding actual parameter is indeed a local port of the calling manifold (see §5.10 and §5.8.7).

The label **start** is special. No source suffix can be specified for this event. Semantically, it marks the block for special handling necessary at the startup of a manifold or a manner (see §5.7).

5.2.3. Observability of Event Sources and Preemption Sets

Every process instance or port defined or used anywhere in a manner or manifold is an *observable* source of events for that manner or manifold. Event sources defined in permanent statements (§5.8.10) are also observable for all manifolds within their scope. An observable source simply means that occurrences of events raised by that source will be picked up by the executing manifold processor, provided that there is a handling block for them. A manifold is oblivious to events raised by sources other than its observable sources. 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. Events from observable sources that are not in the set of observable events are ignored (§5.11). The set of observable events of an executing manifold instance may expand and shrink dynamically due to manner calls and terminations (see §5.3). Depending on the state of a manifold processor (i.e., its current block), occurrences of observable events

are either ignored, or 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, or are mentioned in a permanent statement (§5.8.10). See §5.10 for preemption rules concerning actual parameters in manner invocations and manifold activations. A manifold can always internally raise an event that it will react to via the **do** primitive action (see §5.11).

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. All such event occurrences are ignored. 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 (see §4.4).

5.3. Manners

The state of a manifold is defined in terms of the events it is sensitive to, its preemption set, and the way in which it reacts to an observed event (i.e., the target block for each state transition). 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, 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 preserves it own current state in a push-down stack and then "enters and executes" the manner.

The distinction between manners and manifolds is similar to the distinction between procedures and tasks (or processes) in other 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. Ports, processes and events can be declared in the public declaration section of a manner with the keyword **dynamic**. References to all such entities in a manner are left unresolved until its invocation time. These 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. A **dynamic** entity in a new manner invocation binds to the first entity with the same name and compatible type found in this search (see §5.8.8).

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 preemptive 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 **dynamic** entities (§5.8.8), 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 of a called manner 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, what process to use, 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.

5.4. Atomic Process Specification

An atomic process specification defines the interface to an atomic process, its ports of communication, and the events that are to propagate between the instances of the atomic process and a MANIFOLD application program.

An atomic process specification consists of a name and a list of formal parameters, followed by declarative statements defining its events and ports of communication. The keyword **atomic** appears as the body of the process specification. See §6.1 for more detail on the mapping of atomic process specifications to their realizations. The correspondence between the ports of an atomic process and the input/output ports of its real implementation are described in §6.2. Propagation of events between a MANIFOLD application program and the real implementation of an atomic process is discussed in §6.3.

5.5. Functors and Signatures

A *functor* is the symbol designating a process or a manner as used for its activation or invocation. Usually, functors are the same as manner or process names, and they appear in a prefix notation before any actual parameters, which must be enclosed in a pair of parentheses. However, some special symbols are defined as prefix, infix, and match-fix functors and correspond to a number of predefined manners and processes. See, for example, §9.1 and §5.6.

A list of formal parameter types enclosed in a pair of parentheses following a functor is called the *signature* of the process or manner designated by that functor. The MANIFOLD language allows overloading of functors with different signatures. Thus it is possible to have two different manners or processes with the same name, provided that they accept different type or number of parameters.

To disambiguate an overloaded functor, the compiler searches through all manifold, atomic process, and manner definitions with the same name as the functor. If an exact match is found between the signature of one of these entities and the types of the actual parameters of the functor, the matching entity is taken as the intended target. If no exact match is possible, then one or more actual parameters are converted to allow a match. The rules for this conversion are as follows:

1- A process actual parameter, *P*, can be replaced by *P*.**input** or *P*.**output** in order to match a port **out** or a port **in** formal parameter, respectively.

2- A port or process actual parameter, *P*, can be converted to a pipeline, in order to match a group formal parameter of a manner (§5.10.2).

Rule 2, above, applies only if rule 1 fails to yield a match. If rule 2 fails to produce a match, or if more than one match is possible with rule 1 or rule 2, the compiler raises an error.

For example, a manner call m(proc1, proc2) can be converted to any of the following, in order to match the signature of an appropriate manner definition in the same source file:

```
m(proc1, proc2.input)
m(proc1, proc2.output)
```

```
m(proc.input, proc2)
m(proc.input, proc2.input)
m(proc.input, proc2.output)
m(proc1.output, proc2)
m(proc1.output, proc2.input)
m(proc1.output, proc2.output)
```

Now, assume that there is no manner definition in this file with the signature m(process, process), and consider the manner call m(proc1, proc2) and four manner definitions with the following signatures:

1- m(process, port in)

2- m(process, port out)

3- m(port in, process)

4- m(port out, process)

If any two of the above manner definitions are in the same file, then m(proc1, proc2) is ambiguous and results in a compile time error.

5.6. The Connective ";"

Normally, the actions specified in an event handling block are executed in a non-deterministic order, *not* in the sequential order of their appearance in the source text. The MANIFOLD language provides a special facility to support a common situation where a number of actions P_i must follow each other in a sequence. The connective ";" can be used to build the construct $P_1;P_2;\cdots;P_n$. Intuitively, this construct states that the processor will consider the sequence starting with P_i only after expiration of P_{i-1}, i.e., each P_i will be considered from left to right, one at a time.

The effect of the connective ";" in MANIFOLD is analogous to its meaning in many common programming languages. However, unlike other languages, ";" in MANIFOLD is *not* a fundamental language construct. It is in fact just a functor (§5.5) defined as an infix binary operator that corresponds to a manner, just like a user defined manner. This manner has the following definition (to be precise, there are similar definitions for all different signatures, but we consider only one here):

```
manner ;(x, y)
  ...
{
  start: x.
  λ, returned: y.
}
```

The only special thing about the above manner definition (aside from its name) is that it includes a handler for the special λ event which is generated by the run-time system (§4.5.2.1) and is unknown to MANIFOLD programmers.

The intuitive similarity in the meaning of ";" in MANIFOLD and in sequential programming languages is somewhat misleading. MANIFOLD is an event driven programming language, which implies a different style of programming than the sequential programming paradigm. Moving from left to right over a ";" is inherent in the sequential programming paradigm. In MANIFOLD, there is nothing special about this transition, and the sequential programming's notion of "continuation" does not exist: some specific event causes a specific change of state, which from a higher level is seen to mimic the consecutive progression of actions that takes place in sequential programming. However, there is a noticeable difference in the style of programming between sequential and event driven paradigms.

In sequential programming, there is an inherent sequential flow of control. A programmer must explicitly check for "exceptional" conditions in an algorithm and divert this flow appropriately, *before* it

reaches the default of handling the "normal" case in the algorithm.† In MANIFOLD, the ";" is used to define the sequence of actions in the "normal" case. Any "exceptional" case still preempts this normal sequence and causes a transition to another state to handle the exception. For example, consider the following MANIFOLD blocks:

e1: α;β;γ.

e2: δ.

They state that when in block e1, the actions α, β, and γ are to follow consecutively, each upon the completion of its previous one. This is the *normal* course of actions in this block. However, in case of any *exceptions*, e.g., occurrence of e2, this normal case is to be preempted and another state, e.g., the block labeled e2, must be entered. In other words, occurrence of e2 takes precedence over the semicolons in block e1.

5.7. Starting Up and Termination

The manifold name **main** is special in the MANIFOLD language. There must be a manifold definition with this name in every executing MANIFOLD application program. An instance of this manifold is automatically activated at the startup of the application program. The **main** manifold is just like any other manifold, with its own ports and parameters. The standard input, output, and error ports of this manifold are *not* special in any way (see §9.10 for access to the standard input and output channels provided by the operating system).

The actual parameters supplied on the command line activating the application program, if any, are passed on to the activated **main** manifold as string constants (§5.14.6), replacing their corresponding formal parameters in the **main** manifold definition. The **main** manifold, thus, has a special (restricted) parameter declaration: its parameters can be of type process (character string constants) only. Any formal parameter of **main** for which no actual parameter is supplied is set to the default empty character string.

The startup protocol of a manifold instance involves the special event **start** (see §5.15.4 and §5.2.2), and is as follows. Upon activation, the special startup event **start** is raised inside a manifold instance. Its processor, then, locates and enters the block labeled with **start**. Such a block must exist in every manifold and manner. Typically, this block is used to perform some initializations, before making a transition to another block. The event **start** is also raised as soon as a new invocation of a manner is entered by a manifold processor. Thus, the first block that becomes active in a manner call is its **start** block.

Termination of any process instance always raises the special event **death** (§5.15.4) by that process instance, to inform its observers, if any, of its imminent extinction. This event is raised irrespective of the conditions under which the termination of a process instance occurs. The cause of the termination can, for example, be successful completion, an error, or due to an **abort** (see below, and §5.15.4).

A requesting for termination of a manifold involves the special event **terminate** (§5.15.4), which can be raised directly by **raise**, or as the result of a **deactivate** or a **shutdown** primitive action (see §5.11). The default compiler provided handling block for this event terminates a manifold instance. This default is overridden if a manifold supplies its own handler for this event.

When a **terminate** block is entered, all events from all sources are ignored. However, it is possible for a manifold to leave its **terminate** block via a **do**; it can even completely ignore termination requests by specifying **ignore**, or save it for future handling by specifying **save** as the handling block for **terminate**.

Atomic processes that are the targets of a **terminate** event, receive proper interrupt signals that request their voluntary termination.

Immediate, inescapable termination of a process involves the special event **abort** (§5.15.4) which can be raised directly by **raise**, or as the result of the **cancel** primitive action (see §5.11). The compiler supplied default handler for **abort** immediately terminates the manifold instance. The **abort** event is properly communicated to atomic processes, when necessary, in the form of inescapable interrupts that cause their termination. It is an error to provide a handler block for **abort** in a manifold.

---

† Although programmers are quite used to this style, it is counter-intuitive to humans. Our natural style for giving instructions is to first state the default behavior, what is the normal case, and then add the modifiers necessary to take care of the exceptions.

5.8. Declarative Statements

The purpose of a declarative statement is to define a name and its associated referent. Declarative statements can appear in two places within a manifold or manner definition: before the body and inside the body. They can also appear in atomic process specifications (§5.4), and outside of any manifold, manner, or atomic process specification in source file (§5.8.1).

The declarative statements that appear before a manifold body or in an atomic process specification, comprise its *public* declarative section. They define its formal parameters and the events and ports through which it communicates with other processes in its environment. The declarative statements that appear before the body of a manner define its formal parameters only. The declarative statements that appear inside a manifold or manner body comprise its *private* declarative section. These entities are strictly private to the manifold or manner itself.

5.8.1. Global Names

Names defined by declarative statements in a source file outside of any manifold or manner definition are considered to be global names for all manifolds in that file. Names of manners, manifolds, and atomic processes defined in a file are also globally defined. This means that the name of a manifold or manner definition is known within its own definition, and thus recursion is possible. References to a global name from within any manifold or manner definition in a file are references to the same execution time entity, unless the global definition is overridden for that manifold or manner (§5.8.3).

5.8.2. External Names

The keywords **extern**, **import**, and **export** are used to define external names. External names are used to establish the correspondence between the object codes produced from separately compiled MANIFOLD source files. The compiler generates the proper instructions for the linker to make sure that all references to an external *name* correspond to the same execution time entity.

These keywords can appear only as modifiers to global declarations (§5.8.1). The entities defined in the private and public declaration sections of manners, manifolds, and atomic process specifications cannot be made external.

The keyword **extern** can appear as a prefix to global event declarations only. The keyword **import** can appear as a replacement for the body of a manifold or manner definition, or as a substitute for the keyword **atomic** in an atomic process specification. This indicates that the actual definition of the atomic process, manner, or manifold is contained in a separate source file. The actual atomic process specification or manifold or manner definition in that file must include the keyword **export** as a prefix before its name.

5.8.3. Scope Rules

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

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

The scope of the names defined in the declarative statements outside of any manifold or manner definition, is the entire source file (see §5.8.1).

Non-local names (i.e., used but not defined in a context), are statically bound to the entities with the same name in their enclosing contexts. It is a compile-time error if a non-local name remains unresolved. The binding of **dynamic** entities in manners is postponed until their invocation (see §5.8.8).

The scope of (non-parameter) events defined in the public declaration section of a manifold definition or atomic process specification is all manifold definitions and atomic process specifications in the same source file that define the same event in their public declaration sections. All instances of such atomic processes and manifolds share an identical event definition, which allows them to communicate

through the shared event. (This effectively creates a group of processes that privately share a global event definition.)

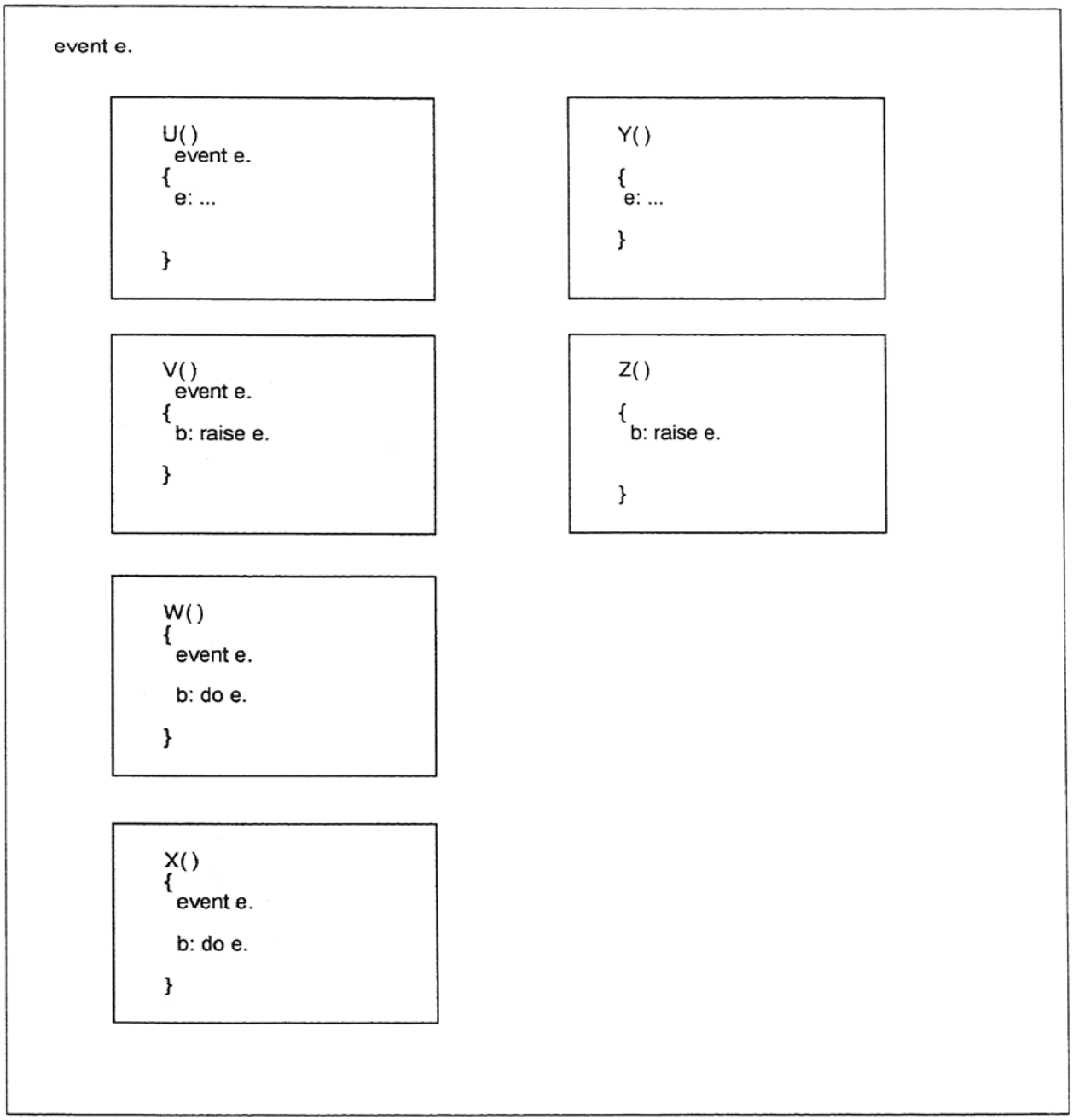


Figure 2 - Example of Scope rules for events

As an example, consider a MANIFOLD source file that contains a global declaration for an event *e*. This file also contains six manifold definitions, *U*, *V*, *W*, *X*, *Y*, and *Z*. *Y* and *Z* do not define, but use an event

named *e*. *W* and *X* each has a private declaration for an event *e*. *U* and *V* each has a declaration for an event *e* in its public declaration section, but *e* is not a parameter for *U* or *V*. These seven scopes are depicted in Figure 2. In this figure, boxes with the same shading represent scopes that share the same entity as event *e*. It is clear from this figure that in this example, there are actually four different events known as *e*.

The above scope rules lead to the following situation. The event *e* that the instances of *Y* and *Z* use is the global event *e* defined in the source file. Every instance of *W* and *X* has its own unique local event *e* that is unrelated to any other event known by anyone else: they cannot communicate through what they know as *e* with any other process. All instances of *U* and *V* know the *same* event as *e*, but this event is *different* than the global event *e* known to *X* and *Y*.

5.8.4. Process Instances

The construct:

process *X* **is** *Y*[(*argtype*$_1$,*argtype*$_2$, $\cdots$,*argtype*$_n$)].

(where square brackets enclose optional items) defines the name *X* as an instance of the process definition *Y*. The symbol *Y* is either the name of a manifold definition, or the name of an atomic process specification. The symbol *Y* can be optionally followed by a signature (§5.5). It is always permissible to supply the signature of the process in this declaration, and the compiler checks the signature for its validity. However, if the name *Y* unambiguously designates only one manifold definition or atomic process specification in the scope of this declaration, the signature can be left out.

More generally,

process *X*$_1$,*X*$_2$, $\cdots$,*X*$_n$ **is** *Y*[(*argtype*$_1$,*argtype*$_2$, $\cdots$,*argtype*$_n$)].

defines *X*$_1$ through *X*$_n$ as instances of *Y*.

A process instance can be defined as a place holder for a real process instance, a reference to which will be made available at execution time. For instance, the construct:

process *X*[(*argtype*$_1$,*argtype*$_2$, $\cdots$,*argtype*$_n$)] **deref** *P*.

defines *X* as the process (with the given signature) whose reference is found at execution time in the next unit obtained from *P*. The optional signature serves a different purpose in this declarative statement. If it is absent, any process reference produced by *P* will bind to *X* at execution time. However, in this case, the name *X* *cannot* be used to activate this process (see §5.11). When a signature is supplied, the execution time binding of *X* and the process reference produced by *P* will take place only if their signatures match. In this case, of course, the name *X* can be used in an **activate** primitive action. See §5.8.9 for more details on dereferencing.

See §5.8.7 for the declaration of process-type formal parameters in manifolds and manners. See also §5.8.8 for declaration of dynamically-binding process entities in manners.

Note that the **process** declarative statements only define a name *X* as a distinct instance of a process definition, or as an alias for a process instance created elsewhere. They do *not* activate these process instances. See **activate** primitive in §5.11.

5.8.5. Port Definitions

Port definitions can appear only in manifold definitions, in the public declaration section of manner definitions, or in atomic process specifications. They cannot appear in the private declarative section of a manner definition, nor can they appear among declarative statements that appear in a source file outside of all manifold and manner definition. Since manners have no ports of their own, port definitions in manners must refer to their formal parameters only (see §5.8.7 and §5.10.2).

The construct:

port *type name*$_1$, *name*$_2$, $\cdots$, *name*$_n$ [*mode*] [*buffer-spec*].

defines each *name*$_i$ as a port with type *type*, and the buffer specification *buffer-spec*. The port *type* is either **in** or **out**, designating every *name*$_i$ as an input or output port, from the view point of its owner process (see §5.12.4.1). The optional *buffer-spec* is a regular expression defining the units that are accepted by a port and a re-bundling of the information it receives into the units that it sends out. See §5.8.5.1 for buffer

specifications and Appendix A for the syntax of the regular expressions. When no *buffer-spec* is given for a port, it simply accepts and delivers every unit it receives without checking or rebundling.

The only acceptable value for the optional *mode* field is **autoflush**. If specified, it designates the port as a flushing port. Its significance is that any stream whose source or sink is a flushing port, will be a flushing stream. Streams are otherwise non-flushing. (See §4.2 for the definition of flushing and non-flushing streams.)

Port definitions that appear inside the body of a manifold definition define its *private* ports. The only difference between a public port and a private port in a manifold is that access to the private ports of manifolds is restricted. A public port of a manifold can be used by any other manifold (that knows its name and the name of its public port) to construct a pipeline. A private port of a manifold can be used exclusively by the manifold itself. However, a manifold can always produce a reference to one of its (public or private) ports using the **&** action (§5.11) and send the resulting port reference unit out on a stream. In this case, any manifold receiving this port reference unit can dereference it (§5.8.9) and access its port, regardless of its public/private status.

Port definitions in manifolds must specify at least the *type* and the *name* of a port. The other two fields, *mode* and *buffer-spec*, are optional. A port definition in a manner must specify only a *name* (which must be a formal parameter of the manner) and a *type* (see §5.3). The *mode* and the *buffer-spec* of formal parameter ports will be supplied by their corresponding actual parameters.

Three ports are automatically defined for every process: the reserved names **input, output,** and **error** are the standard input, standard output, and the standard error ports of a process (see §5.15.3). The default buffer specification for all three standard ports is such that they transfer every unit they receive intact. It is possible to redefine this default buffer specification for the standard ports in **port** statements, or to change their modes to **autoflush**, but their *type*s cannot be changed.

A port name can be defined as a place holder for a real port, a reference to which will be made available at execution time. For instance, the construct:

port *type name* **deref** *Y*.

defines *name* as a synonym for a port with type *type*, whose reference is found at execution time in the next unit obtained from *Y*. See §5.8.9 for more details on dereferencing.

See §5.8.7 for the declaration of port-type formal parameters in manifolds and manners. See also §5.8.8 for declaration of dynamically-binding port entities in manners.

5.8.5.1. Buffers

A *buffer* is a transit holding place for the information under transport through a port. Buffers are properties of ports, and thus belong to processes. A buffer defines meaningful groupings of the bits that flow through its port, as viewed by its owner process. The meaningful groups are specified using a regular expression grammar (see Appendix A).

The regular expression of a buffer is used to match against the stream of bits arriving on the incoming side of the port. The extracted value of this match (see Appendix A) is passed out as a new unit on the outgoing side of the port. The difference between the matched string and its extracted value, if any, is discarded. Units arriving on the incoming side of a port that cause a mismatch with the expected input according to the regular expression, are rejected in their entirety. Rejection of a unit raises the **badunit** event in the manifold (see §5.15.4) with the port as its event source.

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." The regular expression of a buffer implies a certain number of bits to represent the strings in the buffer. This is called the unit length. Note that the unit length of a port need not be a fixed number.

The notion of a buffer is not a fundamental concept in the MANIFOLD model. In the MANIFOLD model, the functionality of buffers can be trivially provided by placing atomic filter processes that perform equivalent regular expression re-bundling, before input ports and after output ports of each process. By explicitly associating this functionality with individual ports, the MANIFOLD language (1) makes it more convenient for users to specify the re-bundling of units, when necessary, and (2) makes it easier for the

MANIFOLD system to "optimize" these filter processes into their corresponding ports, when and if it is deemed appropriate in an implementation.

In the case of atomic processes, buffers provide a convenient means to specify the conversions necessary for exchange of information between them and the MANIFOLD system. These conversions are done by the MANIFOLD system on behalf of the atomic processes.

On output ports, buffers define the unit boundaries for the information placed into the streams. Streams guarantee that the contents of a unit, i.e., the sequence of bits representing the contents of an output buffer, is kept together. This is crucial if an input port is simultaneously connected to more than one output port: it prevents inter-mixing of bits coming from different ports that would garble the information from all sources.

On input ports, buffers make it more convenient for processes to read their input. The unit length of an input port need not be the same as the length of the units it receives. The incoming unit boundaries are detectable by port buffers and can be used for anchoring their regular expressions (see Appendix A). However, they have no other significance, and an input buffer is free to regroup the contents of several units from a stream (or select partial segments of units) to form units meaningful to its own process.

5.8.6. Event Definitions

There are two types of events in the MANIFOLD language: builtin and user defined. Builtin events are a set of predefined reserved names that correspond to specific conditions during the execution of a process, e.g., startup and termination of a process, various system interrupts, special error conditions, etc. (see §5.15.4). User defined events are identifier names designating event handler blocks within manifolds.

All user defined events must be explicitly defined in MANIFOLD. The only way for two manifolds to communicate via an event is for them to know the same event name in the same scope. The construct:

event $name_1, name_2, \cdots, name_n$.

defines each $name_i$ as an event in the scope of this declarative statement. This enables the manifolds in its scope to raise or handle each $name_i$.

An event name can be defined as a place holder for a real event, a reference to which will be made available at execution time. For instance, the construct:

event *name* **deref** *Y*.

defines *name* as the event whose reference is contained in the next unit that is obtained at execution time from *Y*. See §5.8.9 for more details on dereferencing.

Events of interest to an atomic process must be explicitly defined in its specification (see §5.4). The actual propagation of events between a MANIFOLD application program and its atomic processes is inherently implementation-environment dependent. See §6.3 for more detail on the specification of the necessary mappings.

See §5.8.7 for the declaration of event-type formal parameters in manifolds and manners. See also §5.8.8 for declaration of dynamically-binding event entities in manners.

5.8.7. Formal Parameter Declarations

The following constructs are used to define the formal parameters of manners and manifolds. The rules for parameter passing are explained in §5.10.

event $name_1, name_2, \cdots name_n$.

group $name_1, name_2, \cdots name_n$.

manner $name_1, name_2, \cdots name_n$.

port in $name_1, name_2, \cdots name_n$.

port out $name_1, name_2, \cdots name_n$.

process $name_1[(argtype_{1,1}, argtype_{1,2}, \cdots, argtype_{1,i})]$, $name_2[(argtype_{2,1}, argtype_{2,2}, \cdots, argtype_{2,j})]$, $\cdots$, $name_n[(argtype_{n,1}, argtype_{n,2}, \cdots, argtype_{n,k})]$.

The square brackets, above, indicate that the specification of a signature for a process formal

parameter is optional. If a signature is not specified for a process formal parameter X, any process instance can be passed as its actual parameter. However, in this case, the name X *cannot* be used to activate this process (see §5.9 and §5.11). When a signature is supplied for a process formal parameter X, its actual parameter is accepted only if its signatures matches that of X. In this case, of course, the name X can be used in an **activate** primitive action.

5.8.8. Dynamic Binding

Manners are dynamically nested in MANIFOLD. Thus, an entity used in a manner can get bound to an appropriate entity in the environment of its caller at execution time (see §5.3). The declarative constructs:

process $X_1[(argtype_{1,1}, argtype_{1,2}, \cdots, argtype_{1,i})]$, $X_2[(argtype_{2,1}, argtype_{2,2}, \cdots, argtype_{2,j})]$, $\cdots$, $X_n[(argtype_{n,1}, argtype_{n,2}, \cdots, argtype_{n,k})]$ **dynamic**.

event $X_1, X_2, \cdots, X_n$ **dynamic**.

port *type* $X_1, X_2, \cdots, X_n$ **dynamic**.

where *type* is either **in** or **out**, are used for dynamic binding in manner calls. Square brackets enclose optional items and these constructs are allowed only in the public declaration section of manners. They define X_i as place-holders for their corresponding entities that will be filled with appropriate real entities upon invocation of a manner. A target entity for such a place-holder is found 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. A target entity is the first (public or private) entity with the same name and compatible type found in this search. (Note that a target entity for a X_i can itself be a **dynamic** entity. Of course, at the time that it is considered as a potential target for dynamic binding of X_i, it clearly has a concrete binding and is thus no different than other entities defined using non-**dynamic** declarations.)

A **dynamic** event binds to the first event declaration found with the same name. A **dynamic** port, P, can bind to a target port or a process named P, depending on which is found first. A **dynamic in** port, P, binds to the first **in** port named P, or to the standard output port of the first process named P. Analogously, a **dynamic out** port, P, binds to the first **out** port named P, or to the standard input port of the first process named P.

A **dynamic** process binds to the first process instance defined with the same name. Specification of a signature for a **dynamic** process is optional. If a signature is not specified for a **dynamic** process X_i, it can bind to any process instance with the same name. However, in this case, the name X_i *cannot* be used to activate this process (see §5.9 and §5.11). When a signature is supplied for a **dynamic** process X_i, the target for its binding must have the same name and its signatures must match that of X_i. In this case, of course, the name X_i can be used in an **activate** primitive action.

If a **dynamic** entity remains unbound once the search through the dynamic chain of manner activations is complete, a message is placed in the **error** port, the event **unresolved** (§5.15.4) is raised, and it is bound to an appropriate default. The default is **void** (§5.15.2) if the **dynamic** entity is a process, **void.input** or **void.output** if it is a port, or **noevent** (§5.15.4) if it is an event.

5.8.9. Dereferencing

The declarative constructs:

process $X_1[(argtype_{1,1}, argtype_{1,2}, \cdots, argtype_{1,i})]$, $X_2[(argtype_{2,1}, argtype_{2,2}, \cdots, argtype_{2,j})]$, $\cdots$, $X_n[(argtype_{n,1}, argtype_{n,2}, \cdots, argtype_{n,k})]$ **deref** Y.

event $X_1, X_2, \cdots, X_n$ **deref** Y.

port *type* $X_1, X_2, \cdots, X_n$ **deref** Y.

where *type* is either **in** or **out**, are used for dereferencing. They declare X_i to be, respectively, a process instance with the given signature (if provided), an event, or a port of the specified *type*. However, the actual entity denoted here by X_i remains unknown until the activation of the manifold instance or invocation of the manner that contains these declaratives. It is expected that a reference to this entity will be contained in the next unit delivered by Y during the initialization of the manifold instance or the invoked manner. The compiler allows Y to be a (local or non-local) source port or a process. When Y is a process, it is taken to be a short-hand for Y.**output**. Reference units are produced by the **&** action (§5.11).

The dereferencing constructs are allowed only in the private declaration sections of manners and manifolds. The next unit produced by Y in each case is always consumed.

Specification of a signature for a dereferenced process is optional. If a signature is not specified for a dereferenced process X_i, any process instance will be accepted as its replacement at execution time. However, in this case, the name X_i *cannot* be used to activate this process (see §5.9 and §5.11). When a signature is supplied for a dereferenced process X_i, its execution-time replacement is accepted only if its signatures matches that of X_i. In this case, of course, the name X_i can be used in an **activate** primitive action.

Successful dereferencing of an **in**-type port requires a unit containing a source port reference, which is usually an output port of some process. Similarly, successful dereferencing of an **out**-type port requires a unit containing a sink port reference, which is usually an input port of some process. See §5.12.4.1 for more detail on source and sink ports.

If the unit produced by Y is not a reference unit, or if it is not compatible with the declaration of X_i, a message is placed in the **error** port, the event **badrefunit** (§5.15.4) is raised, and X_i is set to an appropriate default. The default is **void** (§5.15.2), **noevent** (§5.15.4), **void.input**, or **void.output**, depending on whether X_i is a process, an event, an **out**, or an **in** type port. The dereferencing declaratives that appear in a manner or manifold are resolved in the sequential order of their appearance in the MANIFOLD source program.

5.8.10. Permanent Event Sources

The statement:

permanent $name_1, name_2, \cdots name_n.$

makes each process name $name_i$ a permanent source of events for all blocks within the scope of this statement. (See §5.2.1 and §5.2.3.) The scope of a **permanent** statement is (1) all manifolds, but *not* manners (see below), defined in a source file, if the statement appears outside of all manifold definitions in the file; (2) the manifold containing the statement only, if it appears in the private declaration section of a manifold; or (3) the manner containing the statement, if it appears in a manner (§5.3). Note that ports cannot be mentioned in a **permanent** statement. The local ports of a manifold are always permanent sources of events in that manifold (and any manners it may call). Note also that any event source that is permanent within a manifold, also remains in effect as a permanent event source inside all its manner invocations, i.e., manner calls dynamically inherit permanent event sources from their callers.

Conceptually, the run-time MANIFOLD system is always defined as a permanent source of events in all application programs. The source of these events is the special process **system** (§5.15.2).

5.9. Process Activation

A manifold M can activate an instance of any atomic process or manifold definition whose name and signature is known to M. Because the name and signature of a manifold definition is known inside the definition itself, manifolds can also recursively activate instances of themselves. There are two forms of process activation: explicit and implicit.

Explicit activation refers to an explicit use of the **activate** action (§5.11), passing it the name of an already created process P together with its actual parameter values, if any. This activates the named process instance and passes the specified actual parameters to it. The process instance P must be defined in a **process** declarative statement that provides a signature for P (see §5.8.4). The activation fails if there is a mismatch between the signature of P and the type and the number of the supplied actual parameters.

Implicit activation refers to the use of a process definition name followed by a (possibly empty) comma-separated list of actual parameters in a pair of parentheses, in place of a process (instance) name. Every encounter with such an implicit activation creates a new instance of the process definition, activates it, and passes the specified actual parameters to it. The implicit activation is then replaced by the name of this newly created instance. The activation fails if there is a mismatch between the signature of the process definition and the type and the number of the supplied actual parameters. Since the names of implicitly activated process instances are not known at compile time, they are called anonymous instances. Named instances of processes (defined in **process** declarative statements), cannot be activated in this way; they must be explicitly activated by an **activate** primitive action.

5.10. Parameter Passing

Parameters can be passed to process instances upon their activation, or to manner instances upon their invocation. The actual parameter passed in place of a formal parameter cannot be changed in the process instance or the manner invocation: formal parameters take on their read-only values only once.

Groups and pipelines passed as actual parameters to a manifold or an atomic process are always first simplified (§5.12.3) in the calling environment before they are passed. Thus, local port names (§5.12.1), references to **self** and **parent** (§5.15.2), and the default **input** and **output** ports referenced by the dangling arrows at the ends of pipelines (§5.12.3.1), are all resolved in the environment of the caller.

Simplification of groups and pipelines passed as actual parameters to manners occurs, if necessary, when their corresponding manner parameters are combined in pipelines in the handling blocks of a manner. The actual parameters passed in a manner invocation are not evaluated unless and until they are actually used in a handling block of the invoked manner.

The types and the number of actual parameters in a manner call, or manifold and atomic process activation are used, if necessary, to disambiguate an overloaded functor. See §5.5 for the details and the relevant precedence rules.

The observability rule for the actual parameters of manifolds and atomic processes is that no process or port named in an actual parameter is automatically an observable source of events in the calling block. When an event source must be observable in a block where it is also (part of) an actual parameter, it must be made observable explicitly using a **permanent** (§5.8.10) or a group construction.

The observability rule for the actual parameters of manner calls is that an event source in an actual parameter is observable if it becomes an observable event source by recursively removing layers of manner calls and ignoring its sibling actual parameters, starting at the block level.

For example, consider the following event handling block:

e: m(n(q, r(a, b), p(c, d)), s(t, u(y, z)), x → v, w).

Assume that m, n, r, and u are manner names and p and s are implicit process activations. The rest of the symbols are either ports or processes. Event source w is observable because if we remove the top-level manner call, m, and ignore all of its parameters except w, then the result is the block e: w., where w is observable. Similarly, removing m and ignoring all of its parameters except the third, yields e: x → v. Thus, both v and x are observable event sources in the original block. Analogously, s is observable, but because it is a process, none of its actual parameters are observable, so t is not observable. Note that although u is a manner call, because s is a process, u cannot be reached by the recursive application of our rule, and therefore y and z are not observable. Recursively removing n shows that p and q are observable, but not c and d. Another recursive application of the rule to r reveals that a and b are also observable sources of events.

5.10.1. Manifold Parameters

The formal parameters of a manifold can be ports, events, or processes. The types of all formal parameters of a manifold must be explicitly defined in its public declaration section (§5.2) using proper declarative statements (§5.8.7). Table 1 shows a summary of parameter passing rules for manifolds.

An event formal parameter must be declared as such in an event definition (§5.8.6). The actual parameter corresponding to an event formal parameter must be an event. An event parameter of a manifold cannot appear in the labels of its blocks.

A port formal parameter must be declared as such in a port definition (§5.8.5). No *mode* or *buffer-spec* can be defined for port formal parameters. The actual parameter corresponding to a port formal parameter must be a port with compatible *type* (see Table 1). Port formal parameters of manifolds cannot be used in event handling block labels (see §5.2.2).

Formal parameter	Actual parameter	Notes
event	event	
port in	source port	The actual parameter can be *i*, *p.t*, or **self.***o*, where *i* and *o* are, respectively, a local input and a local output port of the calling manifold, *p* is some process known to the calling manifold, and *t* is an output port of *p*. See §5.12.4.1.
port out	sink port	The actual parameter can be *o*, *p.t*, or **self.***i*, where *i* and *o* are, respectively, a local input and a local output port of the calling manifold, *p* is some process known to the calling manifold, and *t* is an input port of *p*. See §5.12.4.1.
port in	process	The actual parameter is the standard output of the specified process
port out	process	The actual parameter is the standard input of the specified process
process	process	

Table 1 - Summary of parameter passing for manifolds

Process formal parameters must be explicitly defined as such in process parameter declaration statements. The statement

process $proc_1[(argtype_{1,1}, argtype_{1,2}, \cdots, argtype_{1,i})]$, $proc_2[(argtype_{2,1}, argtype_{2,2}, \cdots, argtype_{2,j})]$, $\cdots$, $proc_n[(argtype_{n,1}, argtype_{n,2}, \cdots, argtype_{n,k})]$.

defines formal parameters $proc_1$ through $proc_n$ as process parameters, with the specified signatures, if provided. The actual parameter corresponding to a process formal parameter must be a process instance. See §5.8.7 for the significance and the consequences of providing or omitting signatures for process formal parameters.

5.10.2. Manner Parameters

The formal parameters of a manner can be of type event, port, process, manner, or group. The types of all formal parameters of a manner must be explicitly declared in its header (§5.8.7). Table 2 shows a summary of parameter passing rules for manners.

An event formal parameter must be declared as such in an event definition (§5.8.6). The actual parameter corresponding to an event formal parameter must be an event.

Process formal parameters must be explicitly defined as such in process parameter declaration statements. The rules for process formal parameters are the same as the case of manifolds (see §5.10.1).

A port formal parameter must be declared as such in a port definition (§5.8.5). No *mode* or *buffer-spec* can be defined for port formal parameters. The actual parameter passed for a port formal parameter of a manner must be a port with compatible *type* (see Table 2). If a port formal parameter of a manner is used in a block label in that manner, its actual parameter is expected to be a local port of the executing manifold process (§5.2.2). This is checked on manner invocation. A message is placed in the **error** port and the event **needlocalport** (§5.15.4) is raised if the actual parameter does not belong to the executing manifold process. However, this is *not* a fatal error, and the manner invocation will proceed normally. The only other consequence of this "error" is that the invoked manner will be unable to react on any events raised by this port.

Process formal parameters must be explicitly defined as such in process parameter declaration

statements. The statement

process $proc_1[(argtype_{1,1}, argtype_{1,2}, \cdots, argtype_{1,i})]$, $proc_2[(argtype_{2,1}, argtype_{2,2}, \cdots, argtype_{2,j})]$, $\cdots$, $proc_n[(argtype_{n,1}, argtype_{n,2}, \cdots, argtype_{n,k})]$.

defines formal parameters $proc_1$ through $proc_n$ as process parameters, with the specified signatures, if provided. The actual parameter corresponding to a process formal parameter must be a process instance. See §5.8.7 for the significance and the consequences of providing or omitting signatures for process formal parameters.

Formal parameter	Actual parameter	Notes
event	event	
port in	source port	The actual parameter can be *i*, **self.***o*, or *p.t*, where *i* and *o* are, respectively, a local input and a local output port of the calling manifold, *p* is some process known to the calling manifold, and *t* is an output port of *p*. See §5.12.4.1.
port out	sink port	The actual parameter can be *o*, **self.***i*, or *p.t*, where *i* and *o* are, respectively, a local input and a local output port of the calling manifold, *p* is some process known to the calling manifold, and *t* is an input port of *p*. See §5.12.4.1.
port in	process	The actual parameter is the standard output of the specified process
port out	process	The actual parameter is the standard input of the specified process
process	process	
manner	manner call	
group	source port	The actual parameter, *P*, has the same syntax as above. The specified port, *P*, is converted to a group with an output, by enclosing it in a pair of parentheses. The effective value of the formal parameter in the called manner is $(P\rightarrow)$.
group	sink port	The actual parameter, *P*, has the same syntax as above. The specified port, *P*, is converted to a group with an input by enclosing it in a pair of parentheses. The effective value of the formal parameter in the called manner is $(\rightarrow P)$.
group	process	The specified process, *P*, is converted to a group with input and output by enclosing it in a pair of parentheses. The effective value of the formal parameter in the called manner is $(\rightarrow P\rightarrow)$.
group	pipeline	The specified pipeline, *P* is converted to a group by enclosing it in a pair of parentheses. The effective value of the formal parameter in the called manner is (P).
group	group	The specified group, *G* is converted to a group by enclosing it in a pair of parentheses. The effective value of the formal parameter in the called manner is (G).

Table 2 - Summary of parameter passing for manners

Group formal parameters must be explicitly defined as such in group parameter declaration statements. The statement **group** $param_1$, $param_2$, $\cdots$ $param_n$ defines formal parameters $param_1$ through $param_n$ as group parameters. The actual parameter corresponding to a group formal parameter is converted to a group. The actual parameter simply replaces its corresponding group formal parameter in a manner call.

The actions implied by an actual parameter (including construction of its pipelines and performing its primitive actions) are performed only if and when its corresponding group formal parameter is encountered during the execution of the invoked manner.

Note that effectively, an additional pair of parentheses is used to enclose any actual parameter corresponding to a group formal parameter. This allows the caller to have a control over whether or not connections to/from the group can be established inside the called manner. For example, consider a group formal parameter G in a manner M. Suppose G is used in M in a pipeline such as $A \rightarrow G \rightarrow B$. A call to M can specify $\rightarrow P \rightarrow$ or $(\rightarrow P \rightarrow)$ as the actual parameter for G, where P is some process or pipeline, possibly with nested groups. The effective value of G in this case is $(\rightarrow P \rightarrow)$ or $((\rightarrow P \rightarrow))$, respectively. Replacement of G in its context and its simplification in the called manner then implies that in the first instance, the output of A and the input of B are effectively connected to the input and output of P, respectively. In the second instance, the extra set of parentheses prevents effective connections between A, P, and B.

Manner call formal parameters must be explicitly defined as such in manner parameter declaration statements. The statement **manner** $param_1, param_2, \cdots param_n$ defines formal parameters $param_1$ through $param_n$ as manner parameters. The actual parameter corresponding to a manner formal parameter must be a manner call.

5.10.3. Atomic Process Parameters

All formal parameters of atomic processes are of type **port in**. Table 3 shows a summary of parameter passing rules for atomic processes. An actual parameter in an atomic process activation can be a unit-producing port or a process. The compiler supplied interface to atomic processes obtains the next unit produced by this port, or the next unit produced by the standard output of the process, and passes it as its corresponding actual parameter to the atomic process.

Formal parameter	Actual parameter	Notes
port in	source port	The actual parameter can be *i*, *p.t*, or **self.***o*, where *i* and *o* are, respectively, a local input and a local output port of the calling manifold, *p* is some process known to the calling manifold, and *t* is an output port of *p*. See §5.12.4.1.
port in	process	The actual parameter is the standard output of the specified process

Table 3 - Summary of parameter passing for atomic processes

Note that it is possible to produce units that are event, port, or process references and it is possible to pass them to atomic processes as actual parameters. However, generally, such units contain little useful information for an atomic process, and their format and contents are highly implementation dependent.

The activation of an atomic process instance is subject to availability of all of its actual parameters. The effective activation remains pending until all expected units are produced by their corresponding actual parameters.

5.11. Primitive Actions

Primitive actions are the basic operations of a MANIFOLD abstract machine. Thus, primitive actions are actually performed by the processor of a manifold instance. However, from a linguistic point of view, it is sometimes useful to pretend that primitive actions are also "special processes" that perform their function and then terminate. Some primitive actions have their own "ports" and can be used in pipelines, like normal processes. Primitive actions raise a **break** event upon their completion.

& The prefix operator **&** is a primitive action that produces a reference to its operand on its

standard output. The operand of a **&** can be a process, a local port name, or an event name. The resulting reference is a unit that comes out of the standard output port of the action and can pass through streams like other units. Such a reference unit can be dereferenced in some other manifold using an appropriate dereferencing construct (see §5.8.9).

Note that if the operand of **&** is a port, it must be a local port of the executing process. However, for a local port name *l*, both **&***l* and **&self.***l* are allowed, producing different results: if *l* is an input (output) port of the executing process, then **&***l* produces a port reference that can be used only as a source (sink) port, once it is dereferenced, while the port reference produced by **&self.***l* can be used only as a sink (source), after dereferencing. If *l* is an input port, the unit produced by **&***l* is suitable for dereferencing by a **port in** dereferencing construct, whereas **&self.***l* can be dereferenced only as a **port out**. Similarly, if *l* is an output port, the unit produced by **&***l* is suitable for dereferencing by a **port out** dereferencing construct, whereas **&self.***l* can be dereferenced only as a **port in**.

activate The action **activate** *process-ref* causes the activation of a new instance of the named process. The argument *process-ref* is a process name, optionally followed by a list of actual parameters enclosed in a pair of parenthesis. Process names are names defined in **process** declarative statements and their signature must be known in the scope where the **activate** action appears (§5.9, §5.8.4). The supplied actual parameters must match the signature of the process, and replace their corresponding formal parameters in the activated process instance. The process instance that successfully activates another process instance becomes the **parent** of the latter (§5.9, §5.15.2).

Activating an already active process is not an error and has no consequence. A deactivated process cannot be activated again. The **activate** action produces a boolean (§5.14) result on its standard output port, which is **true** if the process was activated by this action and **false** otherwise. Any errors encountered during process activation are placed in the standard error port of the **activate** action.

The argument *process-ref* is not automatically included as a source of events in the preemption set of the block containing the **activate** action.

cancel the action **cancel** broadcasts the special event **abort** (§5.15.4) to all processes in the MANIFOLD system. The source of the **abort** event initiated by **cancel** is **system** (§5.15.2). See §5.7 for process termination protocols.

deactivate The action **deactivate** *process-name* causes a **terminate** (§5.15.4) event to be raised inside the named process instance. The source of this **terminate** event is **system** (§5.15.2). See §5.7 for process termination protocols. The argument *process-name* is not automatically included as a source of events in the preemption set of the block containing the **deactivate** action.

It is not an error to deactivate a non-active process. The **deactivate** action returns a boolean result in its standard output port which is **true** if the process was active prior to the action and **false**, otherwise. Note that a **true** result does not necessarily mean that the target process was (or will be) actually deactivated; **terminate** events can be ignored by processes (§5.15.4).

do The action **do** *event* raises the named *event* inside of the manifold. No process outside of the manifold is affected by this action. Internally, the effect of this action is the same as if the manifold had observed the said event occurrence in its environment. The source of events raised by **do** is **self** (§5.15.2).

Unlike **raise**, the *event* in **do** can be any event for which a programmer defined handling block exists. For example, **do start** is permissible.

getunit The action **getunit**(*port*) waits, if necessary, for the availability of a unit in the departure side (§5.12.4.1) of the named *port* and delivers it as the output of the action. The named *port* must be a local port of the manifold. This action can be used in pipelines like normal processes, except that (since it has no standard input port) it can never appear on the

right had side of a →.

More than one **getunit** on the same port can be specified in a given event handling block. For example, the group (**getunit**(l) → B, **getunit**(l) → C, A → D) is valid. Multiple **getunits** on the same port in the same block deliver the same next unit out of the port on their respective standard output ports. Similarly, it is permissible to have a combination of normal stream connections and **getunits** involving the same port in the same block. For example, the group (**getunit**(l) → B, **getunit**(l) → C, l → D) is also valid. The first two pipelines in this group are broken after the first unit out of l is delivered to both B and C, while the third pipeline still remains to deliver additional units out of l to D after it has delivered the first.

If **getunit** detects that there are no more units waiting for departure at its *port*, and the arrival side of *port* is not presently connected to the departure side of any other (source) port, it raises the special event **disconnected_i** (§5.15.4), but then proceeds normally and waits for the availability of the next unit. Note that units may still remain available for departure at a port after its connection with the source port that produced them is broken. See §4.2 and §5.8.5.

guard The action **guard**(*port*, *event*) installs a guard on the named *port*. The named *port* must be a local port of the executing manifold. The installed guard acts as an independent process, and the manifold processor can proceed with the following actions as soon as it is installed. The guard raises the named *event* inside the manifold as soon as a complete unit is available on the departure side (§5.12.4.1) of the named *port* for transport.

An installed guard remains on its port either until it fires its event, or until it is removed by a later guard installation on the same port. Note, in particular, the use of **guard**(*port*, **noevent**) which removes a previously installed guard on *port* without installing a new effective guard. If multiple **guards** are specified on the same port in an event handling block, in effect, only one is actually installed. Furthermore, since the actions in a block are executed in a non-deterministic order, it cannot be pre-determined which is the one that is actually installed.

halt This action raises a special priority event inside of the executing manifold. The handler for this event effectively terminates the executing manifold instance.

ignore The keyword **ignore** can be used in place of a block associated with an event. This will cause the corresponding event to be ignored. It is an error for **ignore** to appear in any other context (e.g., in a pipeline or a group). An ignored event causes no change of state in a manifold, and therefore, the existing pipelines and the state of the manifold are left intact. The ignored event occurrence itself is considered to be handled and is cleared.

raise The action **raise** *event* raises the named *event* outside of the manifold. Any process that can see the manifold can be affected as a consequence of the *event* being raised. This action has no effect on the manifold performing it.

It is an error to attempt to raise any one of the system reserved events. For example, **raise start** is not permissible. See §5.15.4.

return This action raises a special priority event that is caught by its handler in every manner. This handler causes the manifold processor to leave the environment of the current manner invocation and return to its caller. It, thus, terminates the manner invocation.

This action can appear only inside manners, not in manifolds. Upon return from a manner call, the **returned** event is raised in the environment of the caller with **self** as its source (see §5.15.4).

save The action **save** is almost identical to **ignore**. The only difference between the two is that in the case of **save**, the event occurrence itself is *not* considered to be handled: it is *saved* for future handling. Saved event occurrences can preempt a block just like a new event occurrence, as soon as the block is entered and all of its actions are performed (see §4, §5.2.1, and §5.2.3).

shutdown The action **shutdown** broadcasts the special event **terminate** (§5.15.4) to all processes in the MANIFOLD system. The source of this **terminate** event is **system** (§5.15.2). See **deactivate**, above, and §5.7 for process termination protocols.

5.12. Pipelines

The true purpose of most blocks in manifolds and manners is to set up a network of streams among a set of processes, connecting their various input and output ports to each other. This is done by defining *pipelines* that will be set up as soon as the event handling block containing them is entered at execution time (see §5.12.5). A pipeline is a (possibly empty) set of related stream instances and involves a (non-empty) set of processes. By definition, the relationship among the stream instances in a pipeline is such that all are deleted if one breaks.

The remainder of this section describes the basic rules for pipeline definition. First, pipelines and groups are defined as syntactic constructs in the MANIFOLD language in §5.12.1. Next, a set of rewrite rules are described in §5.12.3 that allow simplification of pipelines and groups. Finally, the semantics of pipelines is defined in §5.12.4.

5.12.1. Syntax of Pipelines

A pipeline is a non-empty expression with the syntax of a sequence of zero or more *terms* separated by right arrows (→). A pipeline may have a single leading and/or trailing → symbol. A term in a pipeline is any one of the following:

port:

A port term is either a single name, *P*, or a dotted pair of two names, *R.T*. *P* is either (1) the name of one of the local ports of the manifold instance evaluating the pipeline expression, or (2) a port parameter, dereferenced (§5.8.9) or dynamically bound port (§5.8.8) defined in the scope enclosing the pipeline expression. *R* is either the keyword **self**, designating the specific manifold instance resolving the pipeline expression at execution time (see §5.15.2), or a process name, designating some other process instance known to the manifold (see §5.8.4). *T* is the name of one of the ports of *R*.

In the current version of the MANIFOLD language neither *R* nor *T* in the construct *R.T* that designates a port, can be a parameter or a dereferenced entity (see §5.8.9). This restriction may be removed in a future version of the language.

process:

A process term is the name of a process instance known to the manifold (see §5.8.4).

implicit activation:

An implicit activation consists of a functor designating a manifold definition or an atomic process specification, and a number of actual parameters. The actual syntax of an implicit activation depends on the syntactic attributes of the functor (i.e., whether it is prefix, infix, postfix, or matchfix, as well as its associativity; see §5.5). For all practical purposes, such an implicit activation is equivalent to a unique compiler-generated name for a process instance, just as above, designating a new instance of the given manifold definition or the atomic process specification, invoked with the supplied actual parameters.

primitive action:

Any primitive action (see §5.11) can be used in a pipeline, analogous to implicit activation explained above. Indeed, as far as pipelines are concerned, primitive actions *are* the same as implicit activations designating predefined *internal* processes. However, their inclusion in pipelines follows their special syntax, and must be contained in their proper context.

The special context rules for inclusion of primitive actions in pipelines can be explained by noting that regarded as *special processes*, some primitive actions do not have standard input and/or standard output ports. Thus, they cannot appear in pipelines where they are preceded

and/or succeeded by a → symbol, respectively.

group

A group is a comma-separated list of group members in a pair of parentheses. A group member is either (1) a primitive action, (2) a pipeline, or (3) another group. The order of the members of a group is irrelevant.

5.12.2. Meta-pipelines

Simplification and semantic rules in §5.12.3 and §5.12.4 are defined on a superset of pipelines that we call *meta-pipelines*. A meta-pipeline has the same syntax as a pipeline, except that the operator ↛ is also allowed in a meta-pipeline wherever → is allowed. The operator ↛, and thus meta-pipelines, are *not* part of the MANIFOLD language, but they make it easier to deal with pipelines in our simplification and semantic rules. Intuitively, the operator ↛ denotes "no connection" between its operands.

5.12.3. Simplification of Pipelines

Simplification rules described below are term rewriting rules that transform (a group of) arbitrary meta-pipelines to (a single group of) meta-pipelines with no nested groups. Simplification consists of the application of the following transformations. Default substitution and aliasing are performed first, and only once. The rest of the rules are used as applicable.

5.12.3.1. Default Substitution

Every meta-pipeline that is not a member of a group (i.e., appears by itself in a block, or is an actual parameter) and starts with a leading →, is prepended with the keyword **input**. Analogously, every meta-pipeline that is not a member of a group and ends with a trailing →, is appended with the keyword **output**.

5.12.3.2. Aliasing

Every implicit activation or primitive action that appears in a group or meta-pipeline is replaced by a unique name which, for the purposes of simplification, is assumed to be a process name.

5.12.3.3. Subsumption

If a group member is subsumed by another member in the same group, it is deleted. A group member α is subsumed by another group member if they can be made identical by adding some (possibly empty) suffix and prefix to α. For example, A→B→C is subsumed by A→B→C, A→B→C→D, →A→B→C, and H→A→B→C→D.

5.12.3.4. Flattening

Let G represent the group $(g_0, g_1, \cdots, g_n)$ appearing as a member of another group. The following rules *flatten* such simply nested groups:

1- The group $(u_0, u_1, \cdots, u_{j-1}, \rightarrow G \rightarrow, u_{j+1}, \cdots, u_m)$ can be replaced by the group $(u_0, u_1, \cdots, u_{j-1}, g_0, g_1, \cdots, g_n, u_{j+1}, \cdots, u_m)$.

2- The group $(u_0, u_1, \cdots, u_{j-1}, \rightarrow G, u_{j+1}, \cdots, u_m)$ can be replaced by the group $(u_0, u_1, \cdots, u_{j-1}, g'_0, g'_1, \cdots, g'_n, u_{j+1}, \cdots, u_m)$, where each g'_i is obtained from g_i by dropping its trailing → symbol, if any.

3- The group $(u_0, u_1, \cdots, u_{j-1}, G \rightarrow, u_{j+1}, \cdots, u_m)$ can be replaced by the group $(u_0, u_1, \cdots, u_{j-1}, g'_0, g'_1, \cdots, g'_n, u_{j+1}, \cdots, u_m)$, where each g'_i is obtained from g_i by dropping its leading → symbol, if any.

4- The group $(u_0, u_1, \cdots, u_{j-1}, G, u_{j+1}, \cdots, u_m)$ can be replaced by the group $(u_0, u_1, \cdots, u_{j-1}, g'_0, g'_1, \cdots, g'_n, u_{j+1}, \cdots, u_m)$, where each g'_i is obtained from g_i by dropping its leading and trailing → symbols, if any.

5.12.3.5. Distribution

Let X be an arbitrary group member and G represent the group $(g_0, g_1, \cdots, g_n)$. The following rules *distribute* the connections between X and G over the members of G:

1- The construct $G \rightarrow X$ can be replaced by the group $(u_0, u_1, \cdots, u_n)$, where u_i is:

(a) $g_i \not\rightarrow X$ if g_i has no trailing $\rightarrow$ symbol; or

(b) $g'_i \rightarrow X$ if g_i is "$g'_i \rightarrow$" (i.e., it has a trailing $\rightarrow$ symbol).

2- The construct $X \rightarrow G$ can be replaced by the group $(u_0, u_1, \cdots, u_n)$, where u_i is:

(a) $X \not\rightarrow g_i$ if g_i has no leading $\rightarrow$ symbol; or

(b) $X \rightarrow g'_i$ if g_i is "$\rightarrow g'_i$" (i.e., it has a leading $\rightarrow$ symbol).

5.12.4. Semantics of Pipelines

When a manifold processor makes a transition to a new event handling block, it executes the actions specified in that block. While executing these actions, the manifold processor *cannot* be preempted by event occurrences. It is only after the completion of all the actions in a block that it becomes preemptable. The body of an event handling block is either a single manner call or, in general, a group (see §5.2.2). (A single pipeline is equivalent to a singleton group.) We are not concerned with manner calls here, and the simplification rules in §5.12.3 transform the latter to a simple group of meta-pipelines with no nested groups.

When the body of an event handling block is a group, the actions performed by a manifold processor upon transition to this block consist of an unordered list of the actions specified in the meta-pipelines of the group. Each meta-pipeline defines a list of actions. These actions include: (1) primitive actions, (2) implicit activations, and (3) stream constructions. The actions of type 1 and 2 are explicitly given in a meta-pipeline. To complete the list of actions that must be performed upon transition to a new block, all stream constructions implied by the meta-pipelines in a group must be found. In this section we define the rules for extracting stream connections given in a meta-pipeline.

A stream construction is a special action that connects two ports. An important side effect of stream construction is that when completed, units flow from the sources to the sinks of streams in a pipeline, making information produced by one process available to another. However, it is important to note that a manifold that constructs a stream is in fact oblivious to this flow of information through the stream: pipeline evaluation simply constructs the streams, but it does *not* "move the units" through them; the units flow through a stream by themselves so long as it exists.

The list of actions performed by a manifold when it enters an event handling block is the concatenation of the lists of actions specified in all of its pipelines, in some non-deterministic order. Streams constructed by the actions in this list remain until they are dismantled by the occurrence of an event that changes the state of its constructor manifold to another block (§5.2.1). No units flow through any of the streams in an event handling block before *all* implicit activations and stream constructions contained in that block are completed.

5.12.4.1. Sources and Sinks

Ports are transport mechanisms through which a process exchanges units with streams. The type designators **in** and **out** in **port** declarative statements (see §5.8.5), identify the usage of a port from the point of view of the process that sees the declaration. This holds for all **port** declarative statements: local port declarations, parameter port declarations, and dereferenced port declarations. Independent of its type, each port has two sides (arrival and departure) and can be used in pipelines both as a producer and a consumer of units.

Each process has the privilege of being able to access *both* sides of its local ports (i.e., its own ports), although, usually it only reads from its **in** ports and writes to its **out** ports. Thus, a local **in** port is one that its owner process (usually) obtains units from its departure side, and a local **out** port is one that its owner process (usually) places units into its arrival side. Access to parameter and dereferenced ports, however, are restricted. A manifold (or manner) that defines an **in**-parameter or an **in**-dereferenced port, can access only its departure side (i.e., it can only read from that port). Therefore, in the pipelines within this

manifold (or manner), such a port can appear only as a source (see below). Similarly, an **out**-parameter or an **out**-dereferenced port can be used only as a sink (see below).

A port used as a producer or consumer of units in a pipeline is called a *source* or *sink* port, respectively. The following rules are used to identify source and sink ports in pipelines. Syntactically, there are two ways to refer to ports in pipelines. First, is the construct *R.T* where *R* is the name of a process instance and *T* is the name of one of its ports. Second, is the occurrence of a name, *P*, defined as an input or output port of the executing manifold, or declared as a port formal parameter.

1- The construct *R.T* is a source port in a meta-pipeline if *T* is an output port of the process instance *R*.

2- The construct *R.T* is a sink port in a meta-pipeline if *T* is an input port of the process instance *R*.

3- *P* is a source port in a meta-pipeline if it is:

   (a) the name of an input port of the executing manifold (a local input port declared as **port in**);

   (b) a formal parameter, dynamically-bound port (§5.8.8), or a dereferenced port (§5.8.9) declared as **port in**.

4- *P* is a sink port in a meta-pipeline if it is:

   (a) the name of an output port of the executing manifold (a local output port declared as **port out**);

   (b) a formal parameter, dynamically-bound port (§5.8.8), or a dereferenced port (§5.8.9) declared as **port out**.

There are two forms in which a local port name, e.g., *l*, can appear in a pipeline. It can either appear as the port name itself, i.e., *l*, or as **self.***l*. Note that although both forms refer to the same port, according to the above rules, *l* and **self.***l* are semantically different: if one is a source, the other is a sink. If *l* is an input (output) port of the executing manifold instance, then *l* in a pipeline is a source (sink) port, whereas **self.***l* is a sink (source) port. This means that a manifold has access to both sides of its own ports and can both put units into, as well as get units from them. Only one side of the ports of other processes are accessible to a manifold: their input ports are sinks and their output ports are sources. The construct **self.***l* accesses the side of port *l* of a manifold that other processes can see.

5.12.4.2. Port Connections

A stream construction involves a *port connection*, which connects (the departure side of) a source port to (the arrival side of) a sink port. We use the construct $A \rightarrowtail B$ to denote the connection between the the two ports *A* and *B*. (Note the distinction between the syntactic stream symbol "$\rightarrow$" and the symbol "$\rightarrowtail$" designating its resulting port connection in our meta-language describing its semantics.)

A *connection element* is a triplet $\langle \gamma; \Sigma; \delta \rangle$. The components γ and δ are either empty or they designate sink and source ports, respectively. If γ is non-empty, it designates a port whose arrival side is made available for a connection. Similarly, if δ is non-empty, it designates a port whose departure side is made available for a connection. The component Σ is either empty, or it is an unordered, comma-separated list of port connections.

The port connections involved in a meta-pipeline are obtained by applying the following rules to its simplified meta-pipeline expression. They eventually convert a meta-pipeline expression to a single connection element. The Σ component of this final connection element gives the port connections that are actually created when a manifold processor evaluates the original pipeline expression.

1- Every occurrence of a process name *P* in a meta-pipeline expression is replaced by the connection element <*P*.**input** ;; *P*.**output**>. Occurrences of process names as components of port names (e.g., *P.portI*) are not affected by this rule.

2- Every occurrence of a sink port *X* is replaced by the connection element <*X*;;>.

3- Every occurrence of a source port *X* is replaced by the connection element <;;*X*>.

4- The construct $\langle \gamma_1; \Sigma_1; \delta_1 \rangle \rightarrow \langle \gamma_2; \Sigma_2; \delta_2 \rangle$ is replaced by:

(a) $\langle \gamma_1; \Sigma_1, \delta_1 \twoheadrightarrow \gamma_2, \Sigma_2; \delta_2 \rangle$ if both δ_1 and γ_2 are non-empty.

(b) $\langle \gamma_1; \Sigma_1, \Sigma_2; \delta_2 \rangle$ if δ_1 or γ_2 (or both) is (are) empty.

5- The construct $\langle \gamma_1; \Sigma_1; \delta_1 \rangle \not\rightarrow \langle \gamma_2; \Sigma_2; \delta_2 \rangle$ is replaced by $\langle \gamma_1; \Sigma_1, \Sigma_2; \delta_2 \rangle$.

Note that because the second component in a connection element is an unordered list, stream construction is associative: the order in which the → symbols in a meta-pipeline are transformed is irrelevant.

As an example, consider the meta-pipeline **input** → *A* → *B* → *C*.**input**. Applying rule 1, above, transforms process names *A* and *B* (but not *C*) to their corresponding connection elements and yields:

input → <*A*.**input**;; *A*.**output**> → <*B*.**input**;; *B*.**output**> → *C*.**input**.

Rule 2 now converts the last term in the meta-pipeline and produces:

input → <*A*.**input**;; *A*.**output**> → <*B*.**input**;; *B*.**output**> → <*C*.**input**;;>.

Rule 3 can convert the first term in the meta-pipeline and produce:

<;; **input**> → <*A*.**input**;; *A*.**output**> → <*B*.**input**;; *B*.**output**> → <*C*.**input**;;>.

The first → in the above expression and its operands can now be converted into a single connection element by rule 4, to produce:

<; **input** ↠ *A*.**input**; *A*.**output**> → <*B*.**input**;; *B*.**output**> → <*C*.**input**;;>.

Let us assume that now the second → in the resulting expression gets converted by rule 4. This produces:

<; **input** ↠ *A*.**input**; *A*.**output**> → <*B*.**input**; *B*.**output** ↠ *C*.**input**;>.

Finally, another application of rule 4 yields the final result:

<; **input** ↠ *A*.**input**, *A*.**output** ↠ *B*.**input**, *B*.**output** ↠ *C*.**input**;>.

The second component of this resulting connection element contains exactly the list of port connection actions represented by the original pipeline. (The first and third components of this connection element are empty.)

5.12.5. Setting up Pipelines

Setting up a pipeline involves creating streams that connect pairs of ports (see §5.12.4.2). Port connection is somewhat different than other actions performed by a manifold processor, in that the effect of this action may extend well beyond its completion. Entering a new event handling block, a manifold processor constructs the pipelines defined in that block. This merely creates the connections between the ports as specified in the pipeline. Once this is done, the port connection actions are of course complete. However, it is only upon its completion that a pipeline becomes effective: the corresponding processes now communicate accordingly. A pipeline remains in effect until an event diverts the processor to another handler, at which point it is broken up (see §5.12.6 and §5.2.1). No units flow through any of the streams in a pipeline before *all* streams in its containing block are constructed.

After simplification, the meta-pipeline members of the group that define an event handling block are considered independently during pipeline construction. If an error occurs during construction of one such meta-pipeline (e.g., connecting to a dead process), all connections in that meta-pipeline are ignored. However, this does not affect the construction of other meta-pipelines in the group.

Note that stream setups in pipelines, as well as across different manifolds, are additive. This means that the construct A → B specifies a stream between A and B, in addition to any other connections that A and B may presently be involved in.

The only exception to the additivity of streams is that time-overlapped multiple definitions of the same connection are *not* additive. This means that if we have A → B in two different places and they overlap in time, one of them is superfluous. The significance of this exception is that it prevents double streams between two processes that would result in two copies of everything going from the source to the target process. When multiple definitions of the same connection overlap in time, the connection will exist from the time of the first set-up request, until the time of the last breakup request.

For example, consider two manifolds *P* and *Q*, both defining the connection A → B. Suppose *P* defines the connection first. The connection is established at this time on *P*'s request. Later, *Q* requests the same

connection, while it already exists. *Q*'s request is superfluous and the connection is not duplicated. Next, *P* requests the breakup of the connection A → B. The connection is *not* severed at this time, unbeknownst to *P*, because it must exist to satisfy *Q*. Later, when *Q* requests the breakup of the connection, it will be dismantled.

A process instance must exist (i.e., must have been defined), but need not be active before it can participate in a pipeline. Setting up a pipeline among the ports of a set of processes does not affect the status of the processes. Specifically, participating in a pipeline will not automatically activate a process. Similarly, breaking up of a pipeline has no direct effect on the status of a running process.

5.12.6. Breakup of Pipelines and Groups

Once a group or a pipeline is installed, it can either *expire* or get preempted. Preemption occurs when an occurrence of a preemptive event (see §5.2.3) other than **break** is caught by an event handling block (other than **save** or **ignore**) and the manifold processor makes a transition to its new state. Expiration of a group occurs when all of its pipelines have expired. A pipeline expires when the preemptive event in question is a **break** or a **death**, whose source is a process that participates in that pipeline.

The MANIFOLD language does not allow explicit handlers for **break**: this event is caught by its handler in the MANIFOLD run-time system, on behalf of the MANIFOLD abstract machine (§4.5.2.1).

The MANIFOLD processor treats occurrences of the **death** event in a similar way. The only difference between **break** and **death** is that once all pipelines involving the source of the event are broken up, a **break** event is considered as handled. In the case of **death**, it is *not* considered to be handled, and is then given a chance to cause a state transition to its proper handling block, if one exists. (As with other events, when no handler is found for an occurrence of **death**, a manifold processor ignores it.)

This means that when a process in a pipeline dies, its **death** event first has a chance to preempt the pipeline (and the group it is a member of). This preemption occurs if a matching handler block is found for that **death** event anywhere in the dynamic chain of manner calls between the environment enclosing the pipeline and the environment of the manifold instance. If no such handler is found, the un-caught **death** event converts to a **break**, that will result in the expiration of the pipeline. Note that the expiration of this pipeline may or may not terminate its containing group (depending on whether or not there are other active pipelines in the group), although, had the original **death** been caught by a handler, the group would have been preempted.

5.13. The Void Process

There is a special built-in process in MANIFOLD called **void** that never terminates, never generates any events, never produces any units on its standard output port, and immediately consumes every unit arriving on its standard input port. The **void** process has several practical uses in MANIFOLD programming. Although it is only a built-in process and not part of the language, the uses of **void** are important enough for us to discuss them here.

1- The built-in process **void** can be used as an action in a block. As such, it has the effect of an "idle action" causing the executing manifold processor to wait indefinitely (for it to terminate). Once in this state, the manifold processor can be diverted to some other block, only by events coming from permanent sources (see §5.8.10).

2- Used as a member of a group, **void** will prevent the termination of the group. The enclosing group (block) can be preempted by events coming from permanent sources, or from other observable sources in the preemption set of the block.

3- Used as a process in a pipeline, **void** effectively blocks the flow of units, but does not affect termination of the pipeline. The pipeline A → B → C → **void** → D → E is a single pipeline where nothing flows between C and D. Note that this pipeline is different than the group (A → B → C → **void**, **void** → D → E): an un-caught **death** of any of the processes in the original pipeline will result in the expiration of the pipeline and breakup of all of its connections. In case of the group, an un-caught **death** of any of its processes will result in the expiration of its containing pipeline only, and the break up of the connection in the other pipeline of the group will occur only after the (un-caught) **death** of one of its processes occurs.

Incidentally, note also that the above group is different than the group (A → B → C, D → E) (see 4, below).

4- Connections to the standard input and output ports of **void** can sometimes be used as a substitute for a more useful connection. Although **void** never produces any units and immediately consumes all input units, having a connection to one of its ports may make a big difference in the behavior of a process that is sensitive to its port connections. In the example above, C and D can behave differently (e.g., decide they should terminate) if they were not connected to the ports of **void**.

5.14. Values and Constants

In MANIFOLD, values are produced and consumed by processes and flow through streams. There are only four types of values in MANIFOLD: process references, port references, event references, and bit strings. Event occurrences themselves are *not* considered as values in MANIFOLD, because they do not flow through streams. Process, port, and event references have implementation dependent fixed formats. They are internal identifiers of specific events, ports, and process instances in a running MANIFOLD application. MANIFOLD imposes no interpretation, nor any format restrictions, on what it considers to be a bit string.

Special facilities are provided in the MANIFOLD language to produce and dereference event, process, and port reference values (§5.8.9, §5.11). It is sometimes necessary to produce a specific bit string at a certain place in a MANIFOLD program. Although MANIFOLD places no interpretation on bit strings, the MANIFOLD language, nevertheless, contains special syntax for processes that produce some specific (perhaps, implementation dependent) bit strings; these are called constants.

A constant in MANIFOLD is a process that when (implicitly) activated, produces its value on its output port and then terminates. Termination of a constant raises a **break** event. The name of such a process is the appearance of the constant (determined by the appropriate syntax rules), and its value is a unit containing the execution time internal representation of the constant.

The following subsections describe the proper syntax for some useful bit string values. Note that while individual processes may impose interpretations compatible with the following classification on the values they receive and produce, to the MANIFOLD language, they are only strings of bits.

5.14.1. Bit strings

A bit string is a unit containing an arbitrary sequence of bits. A bit string constant is represented by a sequence of characters 1 and 0 enclosed in a pair of back-quotes, e.g., '010011'. The character B can optionally appear as the first character after the first back-quote to emphasize that this is a binary number.

Bit string constants can also be represented in octal and hexadecimal notation. The constructs 'O*octal-digits*' and '*Xhex-digits*' are bit strings represented in octal and hexadecimal notation, respectively. In this case *octal-digits* is one or more digits 0 through 7, and *hex-digits* is one or more digits 0 through 9, A through F, or a through f. The length of a bit string represented in octal (hexadecimal) notation is always a multiple of 3 (4).

5.14.2. Boolean Values

Boolean values are units whose contents are the same as the internal representation of the boolean constants **true** and **false**. The boolean constants **true** and **false** have their own implementation dependent internal representation, which is, presumably, meaningful to the atomic processes running on the same platform.

5.14.3. Integer Values

Integer values are bit strings that are valid internal representations for integer numbers in a given implementation. Integer constants have the syntax of signed decimal integers. MANIFOLD places no restrictions on the size or internal representation of integer constants. However, such restrictions may be imposed by MANIFOLD implementations.

5.14.4. Floating Point Values

Floating point values are bit strings that are valid internal representations for floating point numbers in a given implementation. Floating point constants are binary approximations of decimal real numbers expressed in the usual signed decimal notation, or in the power-of-ten "scientific notation." MANIFOLD implementations may impose additional restrictions on the range and precision of floating point numbers.

5.14.5. Character Values

Character values are bit strings that are valid internal representations according to the character set used in a MANIFOLD implementation. A character constant is a bit string with the appearance of a quoted character, e.g., 'A'. The actual value of the bit string represented by a specific character is implementation dependent. For example, if an implementation of MANIFOLD uses the ASCII character set, then the bit string value represented by 'A' is '01000001'.

In addition to the printable characters, some non-printable (control) characters have "special names" denoted by escape sequences. For example, if an implementation of MANIFOLD uses the ASCII character set, then '\t' and '\n' represent the bit strings for the ASCII characters TAB and the NEWLINE, respectively.

5.14.6. String Constants

A (character) string constant is a bit string with the appearance of a character string of arbitrary length, enclosed in a pair of double-quotes ("). The actual value of the bit string represented by a string is implementation dependent. It is obtained by concatenating the bit string representations of the characters appearing in the string, preserving their order. The bit string does *not* contain any extraneous information such as length or end-markers.

In addition to printable characters and "special names" of the non-printable characters mentioned above, arbitrary bit strings may also appear inside a string constant. The bit string can appear in its binary, octal, or hexadecimal appearance, but its length must be an integer multiple of the length of the bit string representation of a single character in the implementation.

5.15. Reserved Names

The following names are reserved and they are defined in all manifolds. The definitions associated with these names cannot be overridden.

5.15.1. Manifold Names

The name **main** is a reserved word with a special meaning. It designates the manifold that is to be activated automatically at the start up of a MANIFOLD application program. See §5.7.

5.15.2. Process Instance Names

The following names are analogous to constants. Their encounter implicitly activates an instance of a process which produces one unit containing a value. This value then substitutes the original name.

atomic_source This is the name of a fictitious process instance that is the source of all events raised by any atomic process in certain implementations of MANIFOLD where it is impossible to distinguish the sources of interrupt signals (§6.3).

event_name The process **event_name** is a read-only variable that like a constant (§5.14) sends its value out on its standard output port only once. However, the value of this ''constant'' can be different every time an event handling block is entered. It is the real event name that caused the transition to the current block. See §5.2.1.

event_source The process **event_source** is a read-only variable that like a constant (§5.14) sends its value out on its standard output port only once. However, the value of this ''constant'' can be different every time an event handling block is entered. It is the source (port or process instance) of the event that caused the transition to the current block.

Note that the implementation environment may make it impossible for the

MANIFOLD system to know the value of **event_source** if it is an atomic process. In this case, the value of **event_source** will be a unique special value that will match the predefined constant **atomic_source**. See §5.2.1.

parent The name **parent** is a reference to the activator of the present instance of the executing manifold.

self The name **self** is a reference to the individual instance of the executing manifold.

system This is a special fictitious process instance that represents the MANIFOLD run-time system. It is the source of certain events, e.g., in **shutdown** (§5.11), and is the parent process of the first instance of the **main** manifold process that is activated when a MANIFOLD application runs (§5.7).

While it is not forbidden to construct streams involving the ports of **system**, the behavior of this process with respect to the units transferred through such streams is undefined at this time. Various MANIFOLD implementations may use this channel of communication with running application programs to alter certain execution environment parameters, e.g., for debugging purposes.

void This is a special *black hole* filter process instance. It consumes every unit it receives on its standard input immediately, and never produces a unit on its standard output. A **void** process instance never raises any events and it never dies. See §5.13 for uses of **void** in MANIFOLD programming.

5.15.3. Port Names

The three port names **input**, **output**, and **error** refer to the standard input, standard output, and the standard error ports of the local manifold. See §5.8.5.

5.15.4. Event Names

The following event names are reserved and have special meanings in the MANIFOLD language.

abort This is a priority event. Receiving this event immediately stops the observing process. The default immediate termination effect of this event cannot be overridden by any event handling block. This event can be raised directly by a manifold, which causes its observers to abort, or it can be raised as the result of a manifold executing the **cancel** primitive action, causing the entire application program to abort. See §5.7.

badrefunit This event is raised during the initialization phase of a manner invocation or a manifold activation, if problems are encountered while resolving its dereferencing declarative statements. See §5.8.9. The source of this event is **self**.

badunit This is a port event. This event is raised inside a manifold to indicate that its source port was forced to reject a unit it received on its incoming side, because its contents caused a mismatch with the buffer definition of the port (see §5.8.5.1). This event cannot be raised explicitly by a raise primitive action.

break This is a priority event. This event can be raised explicitly, but it can never be caught by a programmer defined handling block; it is always handled by the run-time MANIFOLD system on behalf of the MANIFOLD abstract machine (see §4.5.2.1). The effect of raising this event is to break up any pipeline its source process participates in.

connected_i This is a port event. It is raised inside a manifold when the arrival side of one of its ports gets connected to a stream. Let n be the number of streams connected to the arrival side of a port p. The event **connected_i**.p is raised when $n = 0$ changes to $n > 0$.

It is *not* an error to attempt a transfer information through a disconnected port. See the explanation under **disconnected_i**.

connected_o This is a port event. It is raised inside a manifold when the departure side of one of

its ports gets connected to a stream. Let n be the number of streams connected to the departure side of a port p. The event **connected_o**.p is raised when $n = 0$ changes to $n > 0$.

It is *not* an error to attempt a transfer information through a disconnected port. See the explanation under **disconnected_i**.

dead This is a priority event. This event is raised when a process instance is found to be dead (i.e., one that has already been deactivated) during the construction of a pipeline. The source of **dead** is always **self**. This event cannot be raised by a **raise** action.

death This is a priority event. This event is always raised by a process to inform its observers, if any, of its death. The death of a process in MANIFOLD simply means its termination due to any cause, including its successful completion (see §5.7).

Regardless of whether an occurrence of **death** is caught by a handling block or not, it always has the same effect on the pipelines that involve its source (the dying process) as a **break** event (see §4.5.2.1). The event **death** cannot be raised explicitly by a **raise** primitive.

disconnected_i This is a port event. It is raised inside a manifold when it attempts to transfer information through a port which is not connected to any other port on its arrival side. Let n be the number of streams connected to the arrival side of a port p. The event **disconnected_i**.p is raised when $n > 0$ changes to $n = 0$ and there are no (whole) units pending at the port for consumption.

It is *not* an error to attempt a transfer through a disconnected port. The transfer attempt will simply raise the **disconnected_i** event, but will proceed normally, otherwise. A transfer from a disconnected input port will simply wait for the arrival of the next unit (see, for example, **getunit** in §5.11). A transfer to a disconnected output port will *not* result in a loss of data: the information may be buffered, or the transferring process may be suspended if the buffer is full, depending on the implementation.

Of course, the **disconnected_i** event raised by a transfer attempt can be caught by a different event handler, in which case the transfer attempt may be canceled as a side effect of the resulting state transition. This event cannot be raised explicitly by a **raise** primitive.

disconnected_o This is a port event. It is raised inside a manifold when it attempts to transfer information through a port which is not connected to any other port on its departure side. Let n be the number of streams connected to the departure side of a port p. The event **disconnected_o**.p is raised when $n > 0$ changes to $n = 0$ and there is at least one whole unit pending at the port for consumption.

It is *not* an error to attempt a transfer information through a disconnected port. See the explanation under **disconnected_i**.

needlocalport This event is raised during a manner invocation for every port formal parameter that is used in a block label in the manner but receives an actual parameter that does not belong to the executing manifold process. See §5.10.2.

noevent This is a non-existing event. It can never have a handling block and raising it is a no-operation.

returned This is a priority event. This event is raised in the environment of a caller upon return (i.e., normal termination via a **return** action) from a manner call. The source of a **returned** event occurrence is the executing process, i.e., **self**.

This event is always caught by a handler in every manner or manifold and thus, is *never* propagated out of the environment of the caller. This event can never be **saved**: an attempt to save a **returned** event results in **ignore**-ing of the event.

start Activation of an instance of every manifold raises the event **start** inside the newly created manifold instance. Similarly, invocation of a manner raises this event inside the manner as well. See §5.7. This event cannot be raised explicitly by a **raise** primitive.

terminate Observing this event informs a process instance that it is being asked to terminate itself. This can be the consequence of a **deactivate** or a **shutdown** primitive action (see §5.11). This event can also be raised explicitly by a manifold executing a **raise** primitive action, requesting its observers to terminate. See §5.7.

unresolved This event is raised during the initialization of a manner invocation every time a **dynamic** entity remains unresolved after the search through the dynamic chain of manner calls is exhausted. See §5.3. The source of this event is **self**.

6. Pragmas

Pragmas are declarative statements that define the interface between the MANIFOLD language proper with the specific features of an environment enclosing a particular implementation of the MANIFOLD system. Functionally, pragmas provide mappings from features of the environment to concepts in the MANIFOLD system. The syntax and the semantics of such mappings are inherently highly implementation environment dependent. As such, it is more appropriate to place such functionality in the category of compiler supplied *convenience* features (that may change from one compiler to another, and from one implementation environment to the next), instead of providing first class constructs for them in the language proper itself.

Pragmas are subject to the scope rules defined in §5.8.3. A pragma must appear in the proper scope wherein the MANIFOLD entity to which it refers is declared. Pragmas *attach* to the MANIFOLD entities they refer to, and therefore have the same scope.

6.1. Mapping Atomic Processes to Programs

An *atomic process pragma* establishes the correspondence between an atomic process specification in MANIFOLD (§5.4) and the code of a program that is its actual realization. The properties of the target of this mapping, the realization of an atomic process, are inherently dependent on certain particular characteristics of the system environment wherein a MANIFOLD implementation runs.

The specific atomic process pragma described here assumes a multi-tasking implementation environment, with possible support for (cheaper than a task) intra-task light-weight processes, and possible support for loosely coupled (networked) multiple processors. This pragma has one of the following forms, where square brackets enclose optional items:

pragma atomic internal *procname* [*funcname*]($argtype_1$,$argtype_2$,...,$argtype_n$).

pragma atomic external *procname* [*mode*] *"filename"*.

pragma atomic network *procname* [*mode*] *"filename" processor*.

Each of these pragmas maps the MANIFOLD atomic process specification *procname* to its realization, i.e., either a function called *funcname* that will be linked together with the MANIFOLD application program, or the independent executable program contained in file *filename*. When *funcname* is missing, it is assumed to be the same as *procname*. The keywords **internal**, **external**, and **network** determine the *domain* wherein this realization is to run. The optional *mode* indicates the mode of behavior of an external or network atomic process. its acceptable values are **compliant** and **noncompliant**, with the default of **noncompliant** if omitted.

The *domain* value **internal** indicates that the atomic process is to run as a light-weight process inside the MANIFOLD system task. A subprogram with the given *funcname* must exist at link time as the realization of such an atomic process. The MANIFOLD system will create an internal light-weight process every time an instance of this atomic process is created by a MANIFOLD application. This light-weight process will, in due course, invoke the specified subprogram, passing it the actual arguments supplied in its activation.

In the current implementation of the MANIFOLD language, $argtype_1$,$argtype_2$,...,$argtype_n$ can be any basic C type or a pointer to void (i.e., void * in C). The function *funcname* must be defined in such a way as to accept basic C types by value and pointers to void (for all other types) as its corresponding parameters. The indirection in parameter passing used for non-basic type arguments, however, is *not* the same as the usual

call-by-reference in C: changes made to its actual parameters by the function *funcname* are in fact made to its own private copy of their values and will be lost upon its termination. The memory allocated to these non-basic type arguments may be released upon return from the function *funcname*, invalidating their corresponding $argtype_i$ pointers.

The *domain* value **external** indicates that the atomic process is to run as an independent operating system level task outside of the MANIFOLD system task. The following *filename* must then contain the executable code of an independent program. The content of such a file is, of course, operating system dependent.

In the current implementation of the MANIFOLD language, *filename* must be a shell-level executable command. This program will be started in a separate Unix process that inherits the shell environment of the executing MANIFOLD application. The units that constitute the actual parameters of an **external** atomic process must be valid character strings (see §5.10.3).

The *domain* value **network** is almost the same as **external**, except that it also indicates that the program in *filename* must run on the specified *processor*, which may be different that the one that executes the MANIFOLD application.

A *mode* value can be specified only for **external** and **network** atomic processes. (All **internal** atomic processes are in effect forced to behave in a **compliant** mode.) If no *mode* is explicitly specified, the default value of **noncompliant** is assumed.

The *mode* value **compliant** indicates that the program which is the realization of the atomic process uses calls to special library routines, supplied by the MANIFOLD implementation, to communicate with the MANIFOLD system (i.e., raise and poll event occurrences and exchange units).

The *mode* value **noncompliant** indicates that the program which is the realization of the atomic process is totally unaware of the fact that it is to cooperate with the MANIFOLD system. This mode of operation allows incorporating any program that can run in the implementation environment of a MANIFOLD system, into a MANIFOLD application program as an atomic process. However, in some implementations, using **noncompliant** atomic processes may impose a performance penalty on their communication links. More importantly, it may also limit the communications possibilities between the MANIFOLD system and the atomic process, especially with regards to event propagation (see §6.3).

6.2. Atomic Process Ports

A MANIFOLD implementation can allow **internal** and **compliant** atomic processes to exchange units with other processes in a MANIFOLD application program through the named ports defined in its atomic process specification, by making calls to special library routines (see §5.4 and §6.1). Mapping the ports of **noncompliant** atomic processes to the input/output ports of their corresponding running programs is influenced by the conventions in the operating system environment wherein a MANIFOLD application program runs. In most platforms, the input/output ports of a running program are managed by functional equivalents of file handling routines, often optimized to bypass the slow secondary storage, whenever possible.

Ports of **noncompliant** atomic processes are mapped to their corresponding input/output ports of their implementing programs by *port pragmas*. The port pragma described here, assumes that (1) the input/output ports of a running program have unique identifiers, and that (2) the identifier designating a specific input/output port of a program at run time can be predetermined and remains the same in different runs. The pragma:

pragma port *proc.port port_id*

maps the mentioned *port* of the atomic process *proc* in a MANIFOLD application program, to the specified *port_id* of its corresponding running program.

In most platforms, **input**, **output**, and **error** ports of atomic processes naturally correspond to certain input/output ports of running programs. No port pragmas should be necessary to explicitly specify such obvious mappings.

6.3. Mapping Events to Interrupts

A MANIFOLD implementation can allow **internal** and **compliant** atomic processes to effectively observe and raise events just as manifolds do, through calls to special library routines (see §6.1). Event propagation between **noncompliant** atomic processes and a MANIFOLD application program is highly dependent on the operating system environment in which a MANIFOLD implementation is embedded. In most platforms, the equivalent of interrupts will be used as the means by which **noncompliant** atomic processes raise and observe events in a MANIFOLD application. *Event pragmas* are used to specify the mapping between MANIFOLD events and their corresponding interrupts.

The event pragma described here assumes an operating system environment wherein it is possible to map incoming and outgoing MANIFOLD events, into interrupt signals exchanged between a MANIFOLD application program and its **external** and **network** type atomic processes. This assumption requires:

1- Identifying an individual **noncompliant** atomic process that runs as an independent operating system level task, as the source of an incoming interrupt in a MANIFOLD application program.

2- Mapping an incoming interrupt signal to a MANIFOLD event.

3- Mapping an event of interest to a **noncompliant** atomic process, when raised in a MANIFOLD application program, into an outgoing interrupt targeted for that process.

The event pragma:

pragma event *name direction proc int_spec*.

specifies a mapping between the MANIFOLD event *name* and the interrupt signal *int_spec*, when it propagates in the specified *direction* between a MANIFOLD application program and the **noncompliant-external** or **noncompliant-network** atomic process *proc*. The *direction* is either **from**, for interrupts coming from the **noncompliant** atomic process to a MANIFOLD application program, or **to**, for events going from a MANIFOLD application program to the **noncompliant** atomic process. The syntax and details of the interrupt specification *int_spec* are implementation dependent.

It is worth noting that many platforms make it impossible for MANIFOLD implementations to fulfill the first requirement, above, i.e., it may be impossible for the MANIFOLD system to know which **external** or **network** atomic process instance is the source of an incoming interrupt. In such environments, the following additional constraints are imposed on the event pragmas.

1- In each scope, a MANIFOLD application program expects the union of all interrupts mentioned in the **from** event pragmas for all atomic processes contained in that scope.

2- Different **from** event pragmas for different atomic processes within the same scope cannot map the same interrupt to different events.

3- The source of all incoming interrupts into the MANIFOLD system is assumed to be the unique fictitious external atomic process **atomic_source**.

7. Compiler Directives

Directives are instructions to the MANIFOLD compiler that affect the way in which it interprets its source code, or that result in further instructions for the link/load stage.

7.1. Forced Child Termination

In MANIFOLD, each process is an independent active entity that is not obliged to terminate upon the death of its parent (activator). If a manifold must terminate upon termination of its parent, this must be explicitly arranged for via an event handler block expecting its parent's death. The parent process must also be an active source of interrupts in all blocks within the child manifold as well. This can be done via a **permanent** directive in the child manifold (see §5.8.10).

In applications where all child processes must terminate upon death of their parents, repeating the above explicit arrangements becomes cumbersome. Turning the *terminate_on_death_of_parent* option on, causes the compiler to automatically insert the necessary source code in all manifold definitions in a source file, causing their instances to terminate upon the death of their activators.

8. Pre-processor Facilities

The pre-processor to the MANIFOLD compiler supports some limited text processing capabilities for modifying the input source program before it is passed on to the compiler. It supports inclusion of source files, a restricted form of macros, and conditional inclusion of source text. The present MANIFOLD pre-processor is the same as the C pre-processor[11].

8.1. Include Files

The pre-processor statement:

#include *file*$_1$ *file*$_2$ $\cdots$ *file*$_n$

puts the contents of the named files in place of the **#include** statement.

8.2. Define Macros

The pre-processor statement:

#define *name* [*value*]

defines *name* as a symbolic name for *value*. Value is any string of characters up to the end of the line. All subsequent encounters with *name* are replaced with *value*. If *value* is missing in the **#define** statement, *name* will be defined but will have no substitution value.

Parameterized macros can be defined by specifying a list of parameters enclosed in parentheses, following the *name*. Occurrences of these parameters in *value* are replaced with their corresponding actual parameters when the macro is used.

8.3. Conditional Text Inclusion

The following construct accepts the input lines between the **#if** and the **#else**, if present, or the **#if** and the **#endif**, if no **#else** alternative is present, when the specified *condition* is true. If the *condition* is false, the input lines between the **#else** and the **#endif** (none, if the **#else** alternative is absent) are accepted.

#if *condition*
[**#else**]
#endif

A limited form of expression evaluation is supported for evaluating *condition*.

9. Builtin Processes and Manners

The following subsections list the builtin manners and (manifold and atomic) processes of the MANIFOLD system. The symbols representing the builtin processes and manners are recognized by the MANIFOLD compiler properly when used in their usual prefix/infix syntax.

9.1. Arithmetic

The symbols **+**, **-**, *****, **/**, **%**, and **mod**, used as binary infix operators represent atomic processes that perform addition, subtraction, multiplication, division, integer division, and modulus on numeric values (see §5.14 for numeric values and constants). The symbol **-** is also recognized as a unary prefix operator, representing negation.

The operands of these atomic processes are passed to them as parameters. The result of each process is produced on its **output** port. Errors, if any, are reported on the **error** port of these atomic processes and raise the **bad_number** event. These atomic processes terminate after they produce their results.

Expressions can be built out of these atomic processes, using the usual syntax for arithmetic expressions. The use of parentheses in arithmetic expressions for grouping is not the same as in group constructs. The role of the parentheses in arithmetic expressions is tied to the infix arithmetic operators. Note that the special notation for arithmetic expressions is nothing more that a "fancy" syntax for implicit process activation (and parameter passing) in MANIFOLD.

9.2. Logical Operators

The symbols <, >, ==, <=, >=, and != used as binary infix operators represent processes implementing boolean functions that compare their operands and produce boolean **true** or **false** values on their **output** ports. The unary prefix operator ! represents a process that performs logical negation of its parameter. Errors, if any, are reported on the **error** port of these atomic processes and raise the **bad_boolean** event.

Just as in the case of arithmetic operations, logical operations can be combined in logical expressions using the usual syntax, with optional parentheses used for grouping. Note, again, that such expressions are simply an alternative syntax for construction of pipelines in MANIFOLD.

9.3. Assignment

Assignment is an infix operator that corresponds to a predefined manner call. The construct A = B is a synonym for B → **pass1**() → A (see §9.6 for **pass1**). A can be a port or a process, and B can be a port, a process, a group, or a pipeline.

9.4. Variable

The declaration:

process *V* **is variable.**

defines *V* as an instance of the predefined process ''variable().'' A variable is a special process that reads the input units it receives on its standard input, and copies each of them to its standard output, at least once. A variable retains the last input unit it receives and repeats it on its standard output as long as it receives no new input units. A permanent connection always exists from the standard output of a variable to **void**. Thus, the units produced by a variable are lost if it is not connected to the input port of some other process to receive them. Consequently, every fresh connection to a variable receives its ''most recent'' value as the first unit.

9.5. Alarm

The builtin process **alarm**(*time, event*) activates an alarm. The activated alarm is an independent process, and the manifold processor proceeds with the following actions as soon as it is activated. The alarm will raise the named *event* after the specified *time* has elapsed (since the alarm activation time).

Caveat: The actual elapsed time is only approximately equal to the specified *time*. Depending on the underlying implementation platform, this approximation may be very poor, especially for small values of *time*.

9.6. Filters

The following (manifold or atomic) processes basically pass the units they receive from their **input** port to their **output** port. Some, perform additional functions such as counting, conversion, etc.

count(*limit, event*)

This filter passes every input unit it receives on its standard input, to its standard output. It also counts the number of units it passes on, and as soon as *limit* number of units have passed through, it raises the specified *event*. It then continues to pass on the units it receives, and raises the specified *event* every time an additional *limit* number of units pass through.

count1(*limit, event*)

This is a different version of **count** which terminates after the specified *event* is raised for the first time.

pass This filter passes on all input units it receives on its standard input, to its standard output.

pass1 This is a different version of **pass** which terminates after the first unit is passed through.

trigger(*pattern, event*)

This filter passes all units received on its input port to its output port, up to the first unit that matches the specified *pattern*. When a unit that matches *pattern* arrives, **trigger** raises the

specified *event*.

The syntax of *pattern* is the same as the syntax of the regular expressions used for port buffer definitions (see Appendix A).

9.7. Control Structures

The following manners make it more convenient to incorporate complex flow of control into MANIFOLD programs.

if(*cond*, *then_part* [, *else_part*])

This manner obtains the next unit produced by the pipeline *cond*. Depending on whether it is identical to **true** or **false** (see §5.14.2), the *then_part* action or the *else_part* action is performed. If the *else_part* action is omitted and the *cond* is **false**, the manner simply returns.

; This infix binary operator supports sequential execution of its action parameters from left to right. See §5.6.

9.8. Type Checking Filters

The following processes act as filters that pass their input units to their standard output. As long as they receive units of the right type, they pass them on silently. When they see an incompatible unit, they raise an appropriate event and ignore the offending unit.

check_eventref	Pass on event reference units. Raise **bad_eventref** for offending units.
check_bool	Pass on boolean units (§5.14.2). Raise **bad_boolean** for offending units.
check_char	Pass on character units. Raise **bad_char** for offending units.
check_float	Pass on floating point units. Raise **bad_float** for offending units.
check_integer	Pass on integer units. Raise **bad_integer** for offending units.
check_numeric	Pass on numeric units of all types. Raise **bad_number** for offending units.
check_procref	Pass on process reference units. Raise **bad_procref** for offending units.
check_portref	Pass on port reference units. Raise **bad_portref** for offending units.
check_string	Pass on character string units. Raise **bad_string** for offending units.

9.9. Miscellaneous

port_of(*proc*)

This process produces a boolean value on its standard output for every unit it receives on its standard input. An output unit is **true** if its corresponding input unit is a reference to one of the ports of the specified process *proc*. Otherwise, it is **false**.

9.10. Interface with the Environment

The following atomic processes provide an interface for MANIFOLD application programs to communicate with the operating system environment in which it runs.

read(*file*, *mode* [, *size*])

This atomic process opens the specified *file* and sequentially delivers its contents on its standard output. The argument *mode* indicates the desired grouping of the contents of *file* into units. Some typical values for *mode* are **char**, **line**, **token**, **string**, and **bits**, indicating that the contents should be delivered as units containing, respectively, individual characters, lines, tokens, character strings with the fixed specified *size*, and bit strings of the fixed specified *size*. Deactivating this process closes its *file*. This process never reads from its standard **input** port.

shell(*command*)

This atomic process starts up the specified operating system command as a separate process and makes available its input/output ports as its own standard input/output ports.

sysinput(*mode* [, *size*])

This is a special version of the builtin process **read** that obtains its input from a special file. It is in fact an alias for the standard input channel of communication between a MANIFOLD application and the operating system under which it runs.

sysoutput() This is a special version of the builtin process **write** that places its output into a special file. It is in fact an alias for the standard output channel of communication between a MANIFOLD application and the operating system under which it runs.

syserror() This is a special version of the builtin process **write** that places its output into a special file. It is in fact an alias for the standard error channel of communication between a MANIFOLD application and the operating system under which it runs.

write(*file*)

This atomic process is the complement of **read**, above. It writes the units it receives on its standard input to the designated *file*. Deactivating this process closes its *file*. This process never writes to its standard **output** port.

10. Acknowledgment

The work on MANIFOLD started as an individual attempt to formalize some of the requirements for a model of communication to describe complex user interface definition systems. In the span of several months, it became a group project with some colleagues showing an interest in the model. The present version of the MANIFOLD specification contains direct and indirect contributions by several people, whose work in the group and contributions to the project must be mentioned.

In particular, Paul ten Hagen inspired some of the original concerns and motivation for MANIFOLD by his earlier work on the Dialogue Cells[12,15], and through our numerous discussions about its follow-up Transaction Cells. Many of the original concepts in MANIFOLD were fleshed out, revised and shaped in discussions with Ivan Herman. It is a pleasure to acknowledge his contributions as a co-designer/developer of the MANIFOLD system. Kees Blom helped to refine the formal syntax for the MANIFOLD language, produced the MANIFOLD compiler, and prepared the syntax diagrams in Appendix B. Eric Rutten is developing the formal semantics of MANIFOLD. Anco Smit's work on a visual interface to MANIFOLD, Per Spilling and Dirk Soede's exercises in MANIFOLD, and especially, Freek Burger's programming effort, and their ongoing contributions to the project are also acknowledged and much appreciated.

References

1. ARBAB, F. AND I. HERMAN, "Examples in Manifold," Technical Report CS-R9066, Centre for Mathematics and Computer Science (CWI), Amsterdam (1990).
2. ARBAB, F. AND I. HERMAN, "Manifold: A Language for Specification of Inter-process Communication," in *Proceedings of EurOpen Autumn Conference*, Budapest (September 16-20, 1991).
3. ARBAB, F., I. HERMAN, AND P. SPILLING, "An Overview of Manifold and its Implementation," Technical Report CS-R9142, Centre for Mathematics and Computer Science (CWI), Amsterdam (1991).
4. BAL, H. E., J. G. STEINER, AND A. S. TANENBAUM, "Programming Languages for Distributed Computing Systems," *ACM Computing Surveys* **21**(3), pp. 261-322 (September 1989).
5. CARRIERO, N. AND D. GELERNTER, "Linda in Context," *Communication of the ACM* **32**, pp. 444-458 (1989).
6. GEHANI, N. H., *Ada: Concurrent Programming,* Prentice-Hall, London - Sydney - Toronto - New Delhi - Tokyo (1984).
7. GEHANI, N. H. AND W. D. ROOME, "Concurrent C," *Software – Practice and Experience* **16**, pp. 821-844 (1986).
8. GEHANI, N. H. AND W. D. ROOME, *The Concurrent C Programming Language,* Silicon Press, Summit, NJ (1989).
9. HERATH, J., Y. YAMAGUCHI, N. SAITO, AND T. YUBA, "Dataflow Computing Models, Languages, and Machines for Intelligence Computations," *IEEE Transactions on Software Engineering* **14**, pp. 1805-1828 (1988).
10. HOARE, C. A. R., *Communicating Sequential Processes,* Prentice–Hall, New Jersey (1985).
11. KERNIGHAN, B. W. AND D. M. RITCHIE, *The C Programming Language (Second Edition),* Prentice-Hall, New Jersey (1988).
12. LIERE, R. VAN AND P. J. W. TEN HAGEN, "Introduction to Dialogue Cells," Technical Report, Centre for Mathematics and Computer Science (CWI), No. CS-R8703, Amsterdam (1987).
13. MILNER, R., *Communication and Concurrency,* Prentice-Hall, New Jersey (1989).
14. RUTTEN, E. P. B. M., F. ARBAB, AND I. HERMAN, "Formal Specification of Manifold: A Preliminary Study," Technical Report (in preparation), Centre for Mathematics and Computer Science (CWI), Amsterdam (1992).
15. SCHOUTEN, H. J. AND P. J. W. TEN HAGEN, "Dialogue Cell Resource Model and Basic Dialogue Cells," *Computer Graphics Forum* **7**, pp. 311-322 (1988).
16. SOEDE, D., F. ARBAB, I. HERMAN, AND P. J. W. TEN HAGEN, "The GKS Input Model in Manifold," *Computer Graphics Forum* **10**(3) (September 1991).
17. STALLMAN, RICHARD, *GNU Emacs Manual,* Free Software Foundation, Cambridge, MA (1987).
18. UNITED STATES DEPARTMENT OF DEFENSE,, *Reference Manual for the ADA Programming Language*, November 1980.
19. VEEN, A. H., "Dataflow Machine Architecture," *ACM Computing Surveys* **18** (1986).

Appendix A: Regular Expressions

Regular expressions are convenient means for specifying patterns for string matching. They are used in numerous utilities, e.g., text editors, running on many platforms for searching and filtering. In the MANIFOLD language, regular expressions are used to construct units for transfer through a port. They are also used as input patterns for triggers (see §9.6). The particular syntax for regular expressions presented below is an adaptation of the one used in the GNU Emacs text editor[17].

A regular expression (RE) specifies a set of (character or bit) strings to match against — such as *any line containing digits 5 through 9*, *only units containing upper-case letters*, or *any sequence of bits starting with three 1's and ending with two 0's*. A member of this set of strings is said to be *matched* by the regular expression. When multiple matches are possible, a regular expression matches the *longest* of the *leftmost* matching strings. The main differences between the use of regular expressions in MANIFOLD port buffers and for conventional matching, e.g., in an editor, are:

1- The matching in MANIFOLD port buffers is a *continuous* match. The matcher always expects more input to come through the port.

2- Every successful match produces a result, which is the extracted value of the matched input.

3- Literals can be inserted into the extracted value of a match.

When no match is possible, the leftmost (chronologically oldest) input unit is discarded and the match continues with the next unit.

Special characters and operators are provided to constrain a match to the beginning or end of units and lines. A match of this type is called an *anchored match* because it is *anchored* to a specific place in a unit or line. A *line*, in this context, is any sequence of bits starting with the first bit in a unit or after a NEWLINE, up to the last bit before the next NEWLINE or the last bit in the same unit, whichever comes first.

Atomic Regular Expressions

Regular expressions can be built up from the following atomic RE's:

c — Any ordinary character not listed below. An ordinary character matches itself. It never skips over the null character at the end of a unit.

\ — Backslash. When followed by a special character, the RE matches the *quoted* character. Special characters are: **$**, **^**, **.**, *****, **+**, **?**, **[**, **]**, ****, **'**, and **‘**. Any other character appearing in an RE is an ordinary character. A backslash followed by digits 1 through 9, or one of **b**, **e**, **w**, **W**, **"**, **<**, **>**, **(**, **)**, **|**, **{**, or **}**, represents an *operator* in a regular expression, as described below. Backslash followed by any other character is an RE that matches the character itself.

. — Dot. Matches any single character except NEWLINE or the null character at the end of a unit.

‘*b...***‘**

‘B*b...***‘**

‘O*o...***‘**

‘X*x...***‘** — These constructs allow including arbitrary bit strings as atomic RE's in regular expressions. There is no restriction on the length of such bit strings nor on their alignment within the input strings. However, non-judicious use of arbitrary bit strings with strange alignments may result in less efficient matches.

The two forms **‘***b...***‘** and **‘B***b...***‘** are identical, and define *bit strings*. Each *b* in these RE's is either **1**, **0**, or **.** (dot), matching the bit 1, 0, or either, respectively.

The form **‘O***o...***‘** is an octal representation of a bit string. Each *o* is either an octal digit (**0** through **7**), or **.** (dot), representing its corresponding three-bit string, or any three-bit string, respectively.

The form **‘X***x...***‘** is a hexa-decimal representation of a bit string. Each *x* is either a

hexadecimal digit (**0** through **9**, **a** through **f**, or **A** through **F**), or **.** (dot), representing its corresponding four-bit string, or any four-bit string, respectively. The binary, octal, or hexadecimal dot character (**.**) does not skip over the null character at the end of units.

'bool' This atomic RE matches any bit string that is a valid representation of a boolean (**true** or **false**) value in the underlying implementation.

'eventref' This atomic RE matches any bit string that is a valid representation of a reference to an event. The matched bit string is guaranteed to have only the proper format for an event reference. Its actual value may or may not be valid as an event reference.

'float' This atomic RE matches any bit string that is a valid representation of a floating point number in the underlying implementation.

'int' This atomic RE matches any bit string that is a valid representation of an integer value in the underlying implementation.

'portref' This atomic RE matches any bit string that is a valid representation of a reference to a port. The matched bit string is guaranteed to have only the proper format for a port reference. Its actual value may or may not be valid as a port reference.

'procref' This atomic RE matches any bit string that is a valid representation of a reference to a process. The matched bit string is guaranteed to have only the proper format for a process reference. Its actual value may or may not be valid as a process reference.

^ A caret (or circumflex) matches the null character at the beginning of a line. It thus constrains the RE on its right to match the leftmost portion of a line.

$ A dollar sign matches the null character at the end of a line. It thus constrains the RE on its left to match the rightmost portion of a line.

^RE**$** The construction **^**RE**$** constrains the RE to match the entire line.

\< This operator matches the null character at the beginning of a unit. It thus constrains the RE on its right to match the leftmost portion of a unit.

\> This operator matches the null character at the end of a unit. It thus constrains the RE on its left to match the rightmost portion of a unit.

The **\>** anchoring operator cannot appear in the buffer definitions for the output ports of atomic processes.

\<RE**\>** The construction **\<**RE**\>** constrains the RE to match the entire unit.

\b The sequence **\b** in an RE constrains the atomic RE immediately following it only to match something at the beginning of a *word*; that is, either at the beginning of a line, or just before a letter, digit, or underline and after a character not one of these.

\e The sequence **\e** in an RE constrains the atomic RE immediately following it only to match something at the end of a *word*.

\w The sequence **\w** matches any character that can appear in a word.

\W The sequence **\W** matches any character that cannot appear in a word.

[c...**]** A nonempty string of characters, enclosed in square brackets matches any single character in the string. For example, **[abcxyz]** matches any single character from the set '**abcxyz**'. When the first character of the string is a caret (**^**), then the RE matches any character *except* NEWLINE and those in the remainder of the string. For example, '**[^45678]**' matches any character except '**45678**'. A caret in any other position is interpreted as an ordinary character.

[]c... **]** The right square bracket does not terminate the enclosed string if it is the first character (after an initial '**^**', if any), in the bracketed string. In this position it is treated as an ordinary character.

[*l***–***r***]** The minus sign, between two characters, indicates a range of consecutive ASCII characters to match. For example, the range '**[0–9]**' is equivalent to the string '**[0123456789]**'. Such a bracketed string of characters is known as a *character class*. The '**–**' is treated as

an ordinary character if it occurs first (or first after an initial ^) or last in the string. Also, '---' is a range that contains '-' only.

Regular Expression Constructors

The following rules and special characters allow for constructing RE's from atomic RE's:

A concatenation of RE's matches a concatenation of text strings, each of which is a match for a successive RE in the search pattern.

\(...\) An RE enclosed between the character sequences \(and \) matches whatever the unadorned RE matches, but it also:

1- encapsulates the RE to bound the applicability of the other RE constructors, below.

2- saves the string matched by the enclosed RE in a numbered substring register. Up to nine such substrings are accessible at a time in an RE (see *\n*, below).

The parenthesized RE's can be nested.

\| This operator specifies an alternative. Two RE's *a* and *b* with a '\|' between them matches anything that either *a* or *b* can match. For example, '**Tom\|Jerry**' matches the string 'Tom' or the string 'Jerry' and '**\(Tom\|Jerry\)X** matches either 'TomX' or 'JerryX'. Note that using the pair of parentheses has the additional side effect of saving the matched string ('Tom' or 'Jerry') in a numbered substring register.

***** An atomic or encapsulated (i.e., in a pair of parentheses) RE, followed by an asterisk (*) matches *zero* or more occurrences of the RE. Such a pattern is called a *closure*. For example, **[a–z][a–z]*** matches any string of one or more lower-case letters.

+ An atomic or encapsulated (i.e., in a pair of parentheses) RE, followed by a plus (+) matches *one* or more occurrences of the RE.

? An atomic or encapsulated (i.e., in a pair of parentheses) RE, followed by a question mark (?) matches zero or one occurrence of the RE.

\{*m*\}
\{*m*,\}
\{,*n*\}
\{*m*,*n*\} An atomic or encapsulated (i.e., in a pair of parentheses) RE followed by \{*m*\}, \{*m*,\}, \{,*n*\}, or \{*m*,*n*\} is an RE that matches a *range* of occurrences of the RE. The values of *m* and *n* must be nonnegative integers less than 256; \{*m*\} matches *exactly m* occurrences; \{*m*,\} matches *at least m* occurrences; \{,*n*\} matches *at most n* occurrences; \{*m*,*n*\} matches *any number* of occurrences *between m* and *n*, inclusively. Whenever a choice exists, the RE matches as many occurrences as possible.

n Matches the contents of the *n*th substring register from the current RE. This provides a mechanism for extracting matched substrings. For example, the expression **^\(..*\)\1$** matches a unit consisting entirely of two adjacent non-null appearances of the same string. When nested parenthesized substrings are present, *n* is determined by counting occurrences of \(starting from the left.

Extracted Values

An RE has one or more extracted values. Every extracted value of an RE is a separate unit by itself. By default, the *extracted value* of a RE is the string it matches. The extracted value of a RE can be restricted to a substring of the string it matches, using the extraction operator \":

\" An RE, *u*, which includes a pair of extraction operators matches whatever string its equivalent RE without the extraction operators, *v*, would match. The extracted value of *u*, however, is the substring of the extracted value of *v* that matches the portion of *u* between the pair of \" operators.

The extracted value of an RE with more than one pair of extraction operators is the

sequence of units produced by its sub-RE's, each of which contain only one pair of extraction operators.

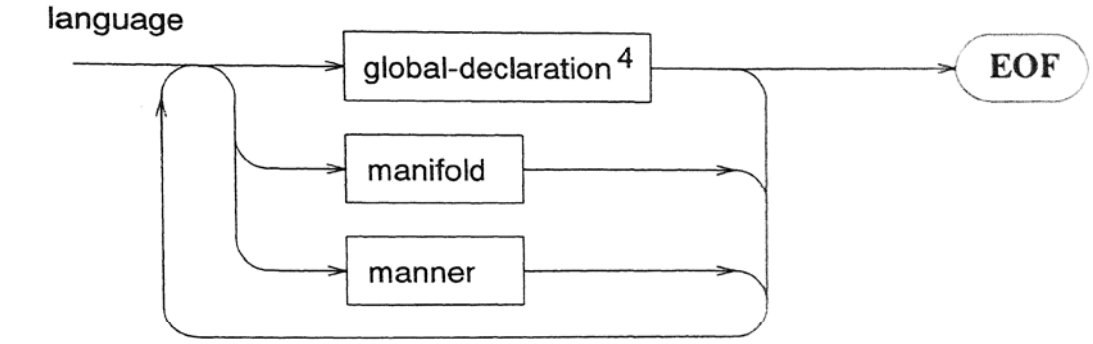
Insertion of Literals

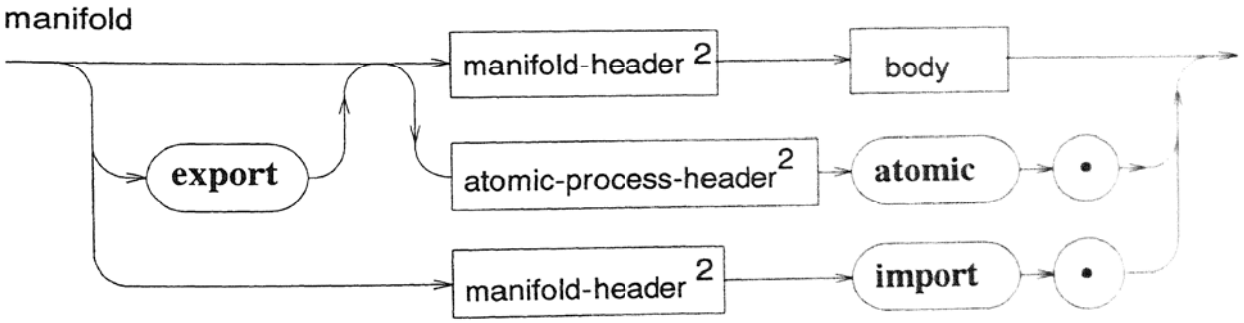
Arbitrary constants can be inserted as literals in the extracted value of a match.

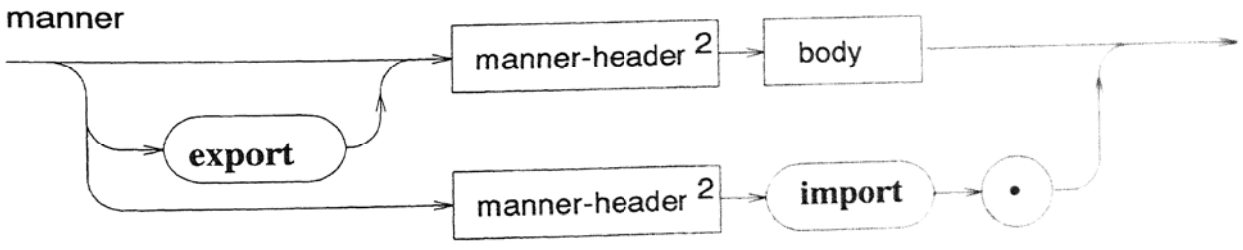
' Any string enclosed between a pair of single quotes is inserted in its place in the extracted value of an RE. Literals do not affect the RE matching in any way.

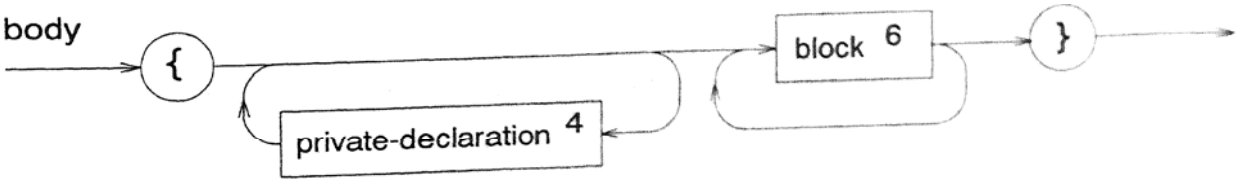
Appendix B: Language Syntax

Terminal symbols are enclosed in circles and rounded boxes. Literals are in bold type. Non-terminal symbols are in rectangular boxes. Superscript numbers indicate pages where their corresponding non-terminals are defined (omitted if on the same page).

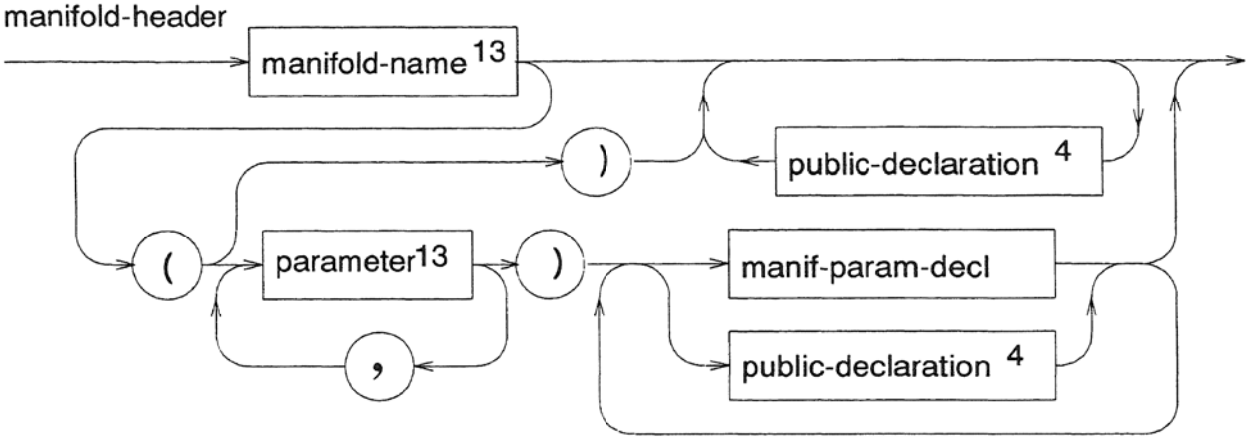




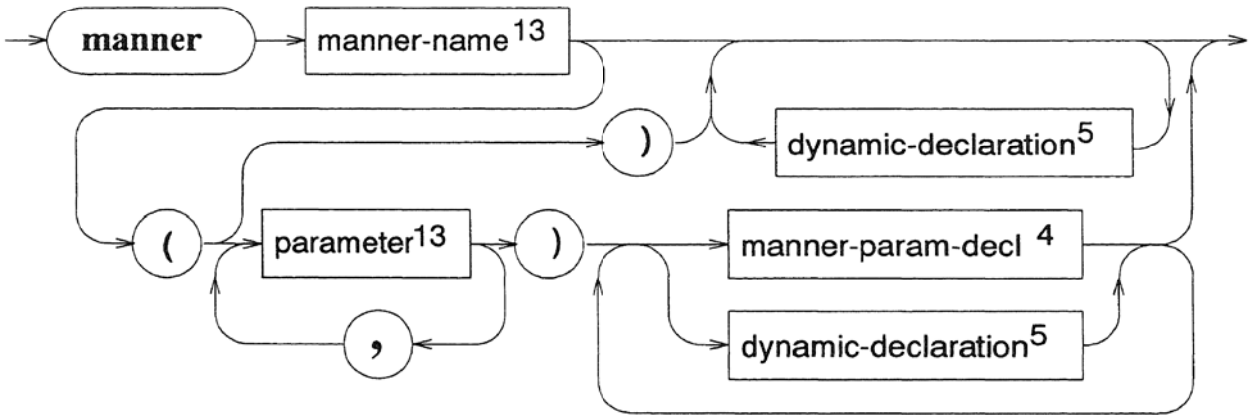




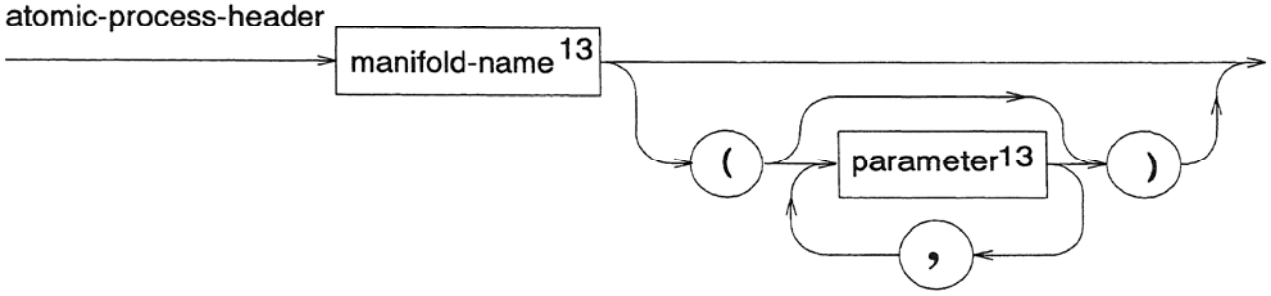
manifold-header



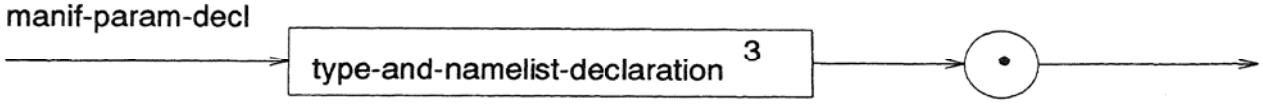
manner-header



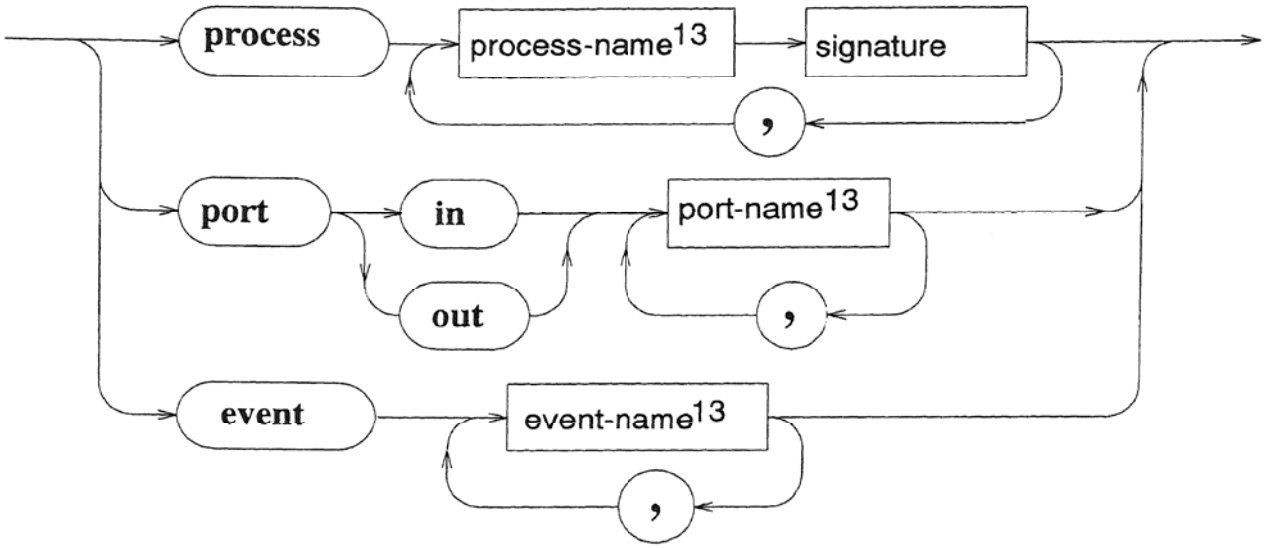
atomic-process-header



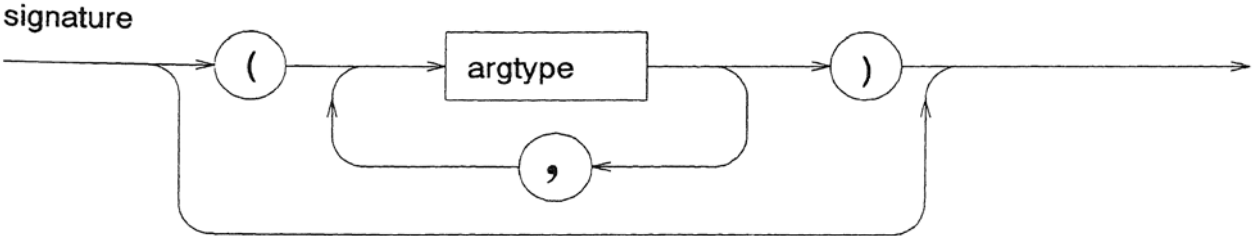
manif-param-decl



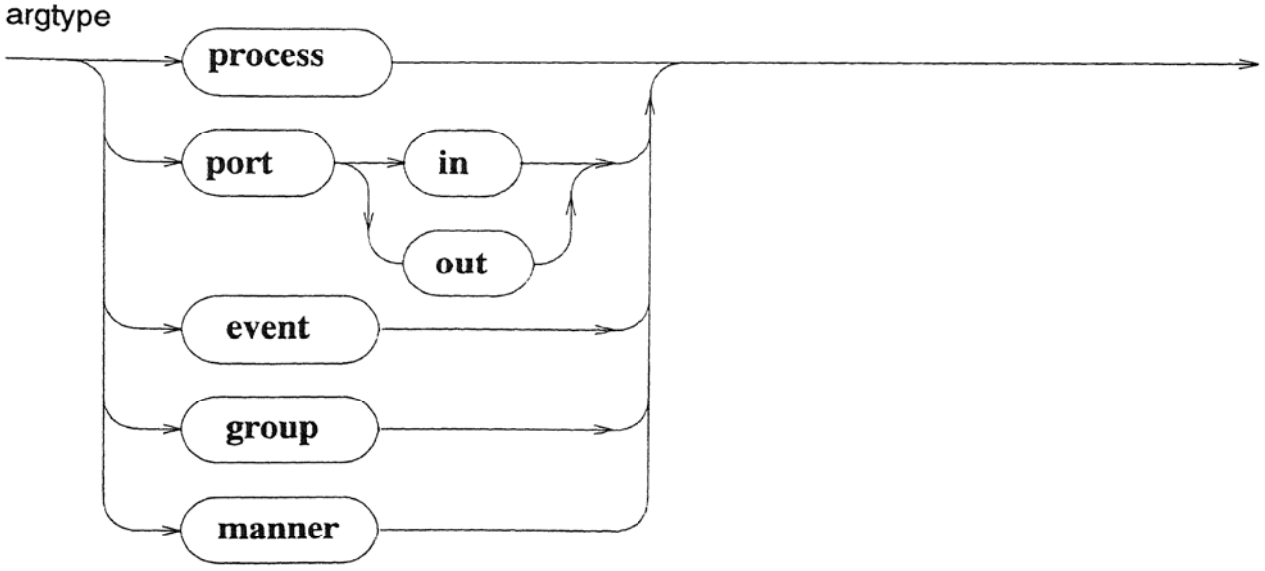
type-and-namelist-declaration

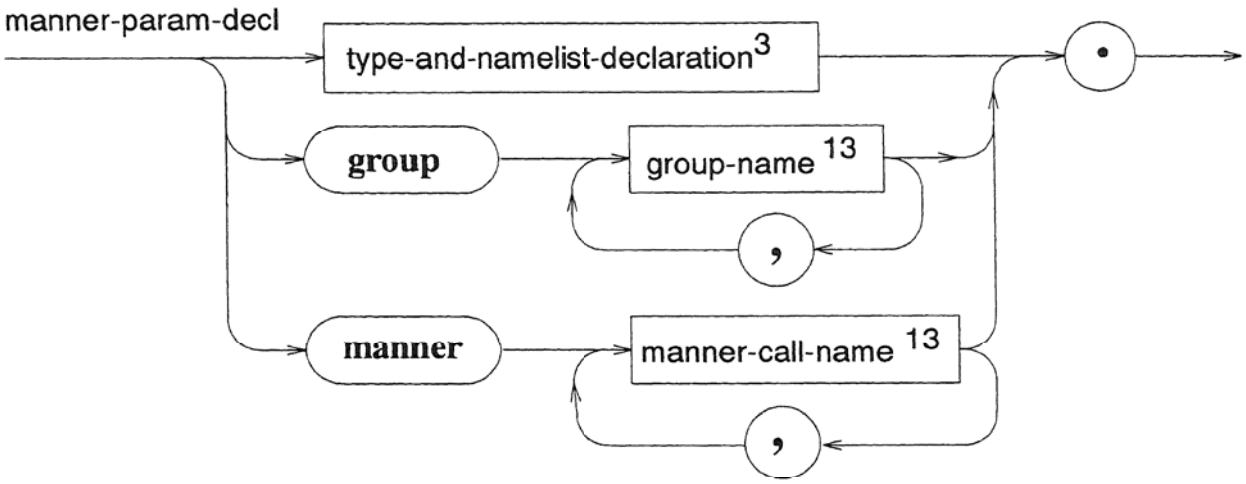


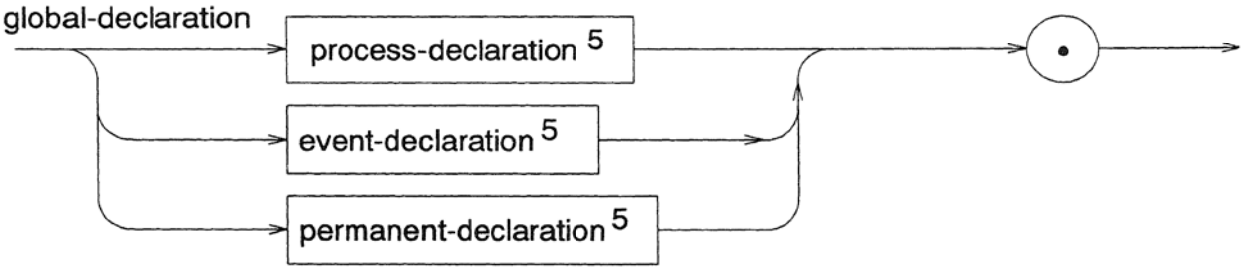
signature

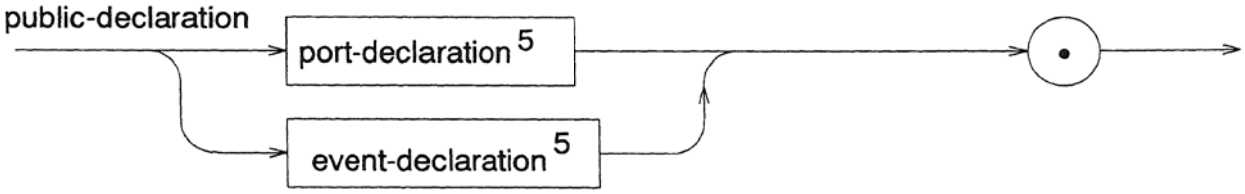


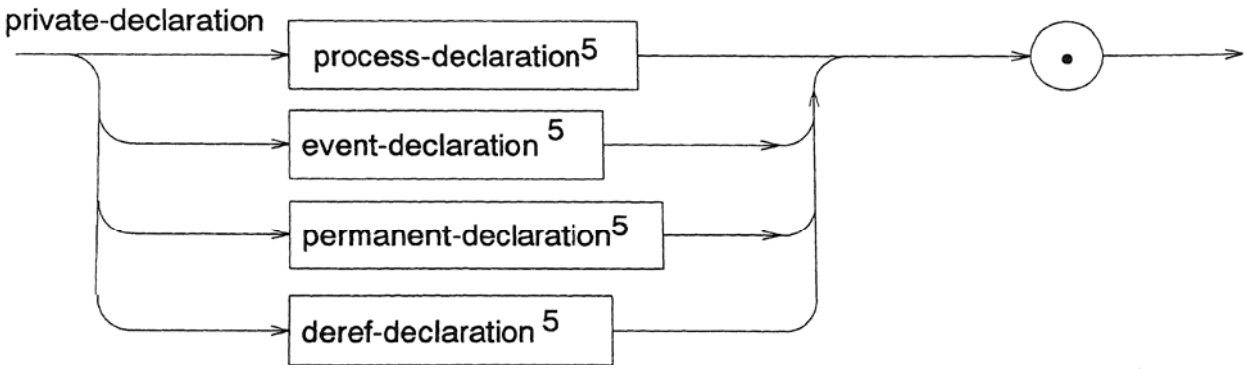
argtype



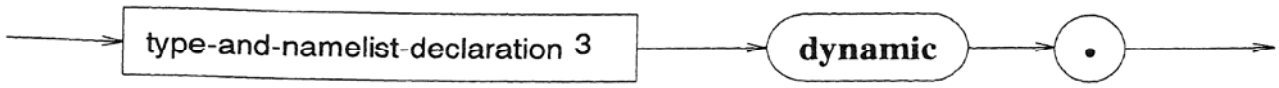

manner-param-decl
type-and-namelist-declaration 3
group
group-name 13
,
manner
manner-call-name 13
,


global-declaration
process-declaration 5
event-declaration 5
permanent-declaration 5

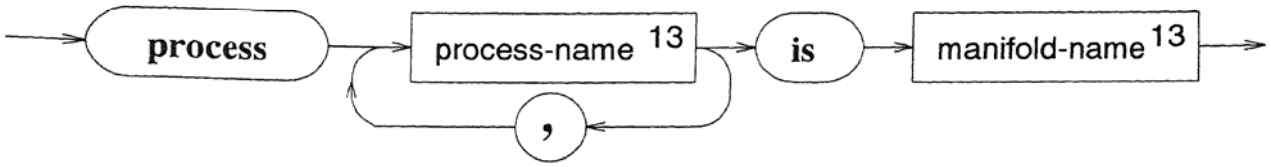

public-declaration
port-declaration 5
event-declaration 5


private-declaration
process-declaration 5
event-declaration 5
permanent-declaration 5
deref-declaration 5

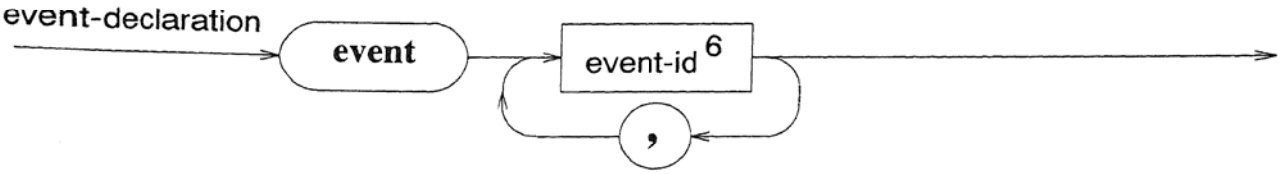
dynamic-declaration



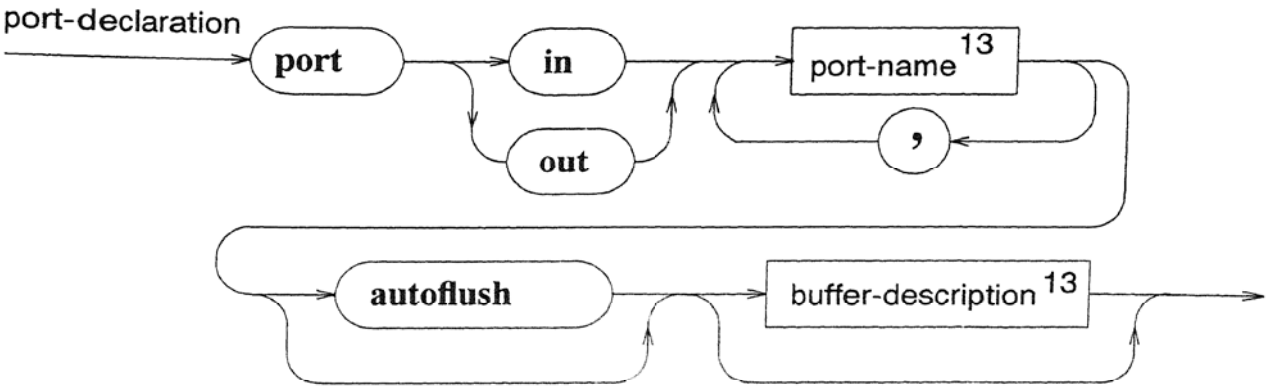
process-declaration



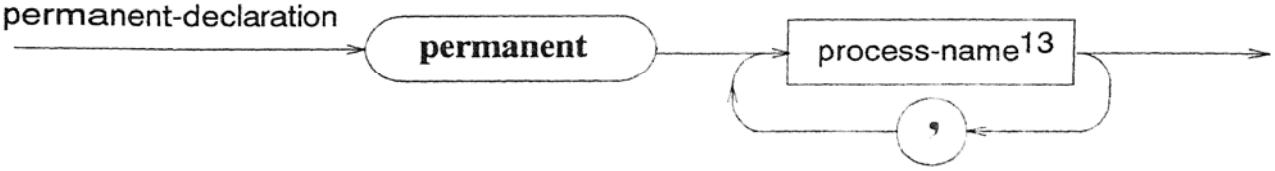
event-declaration



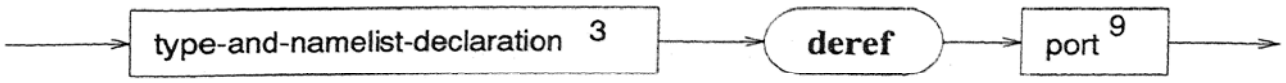
port-declaration

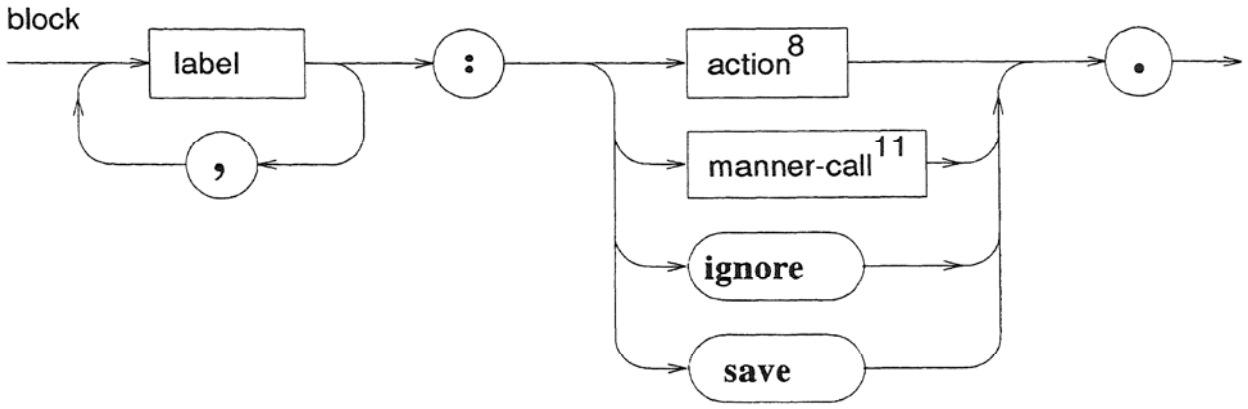


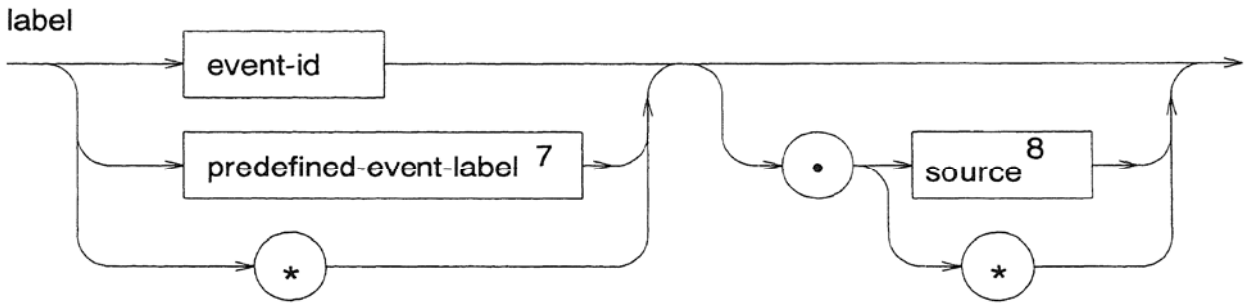
permanent-declaration

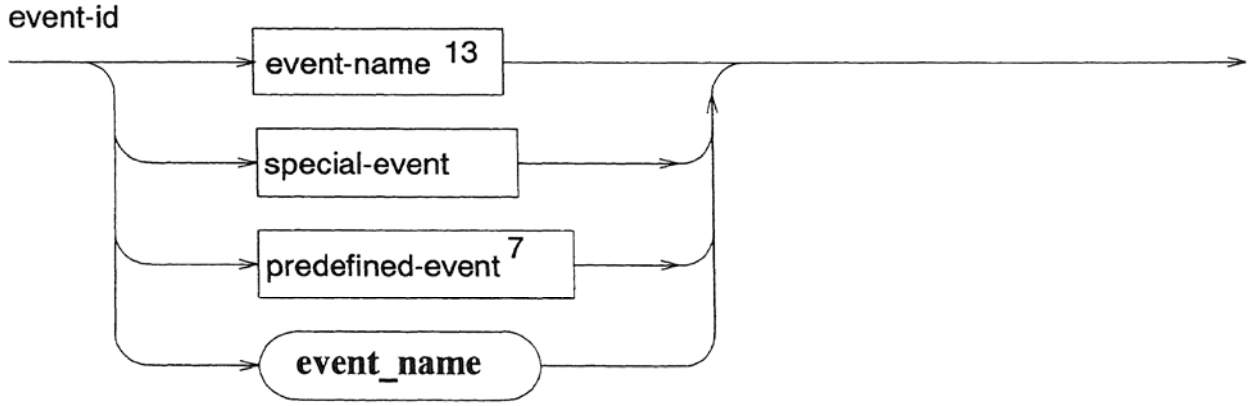


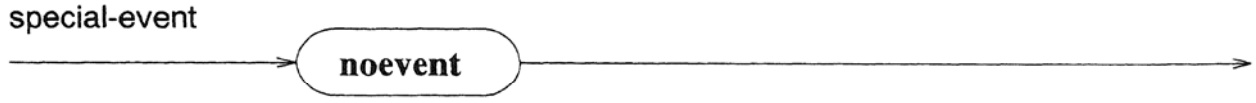
deref-declaration

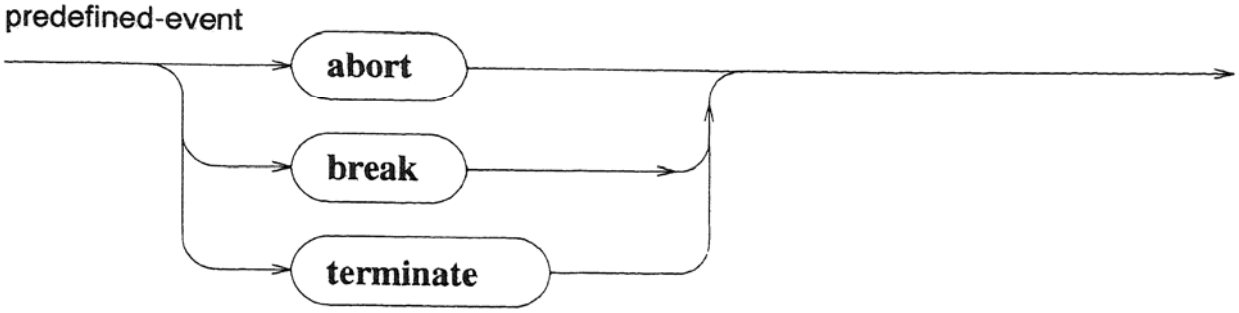


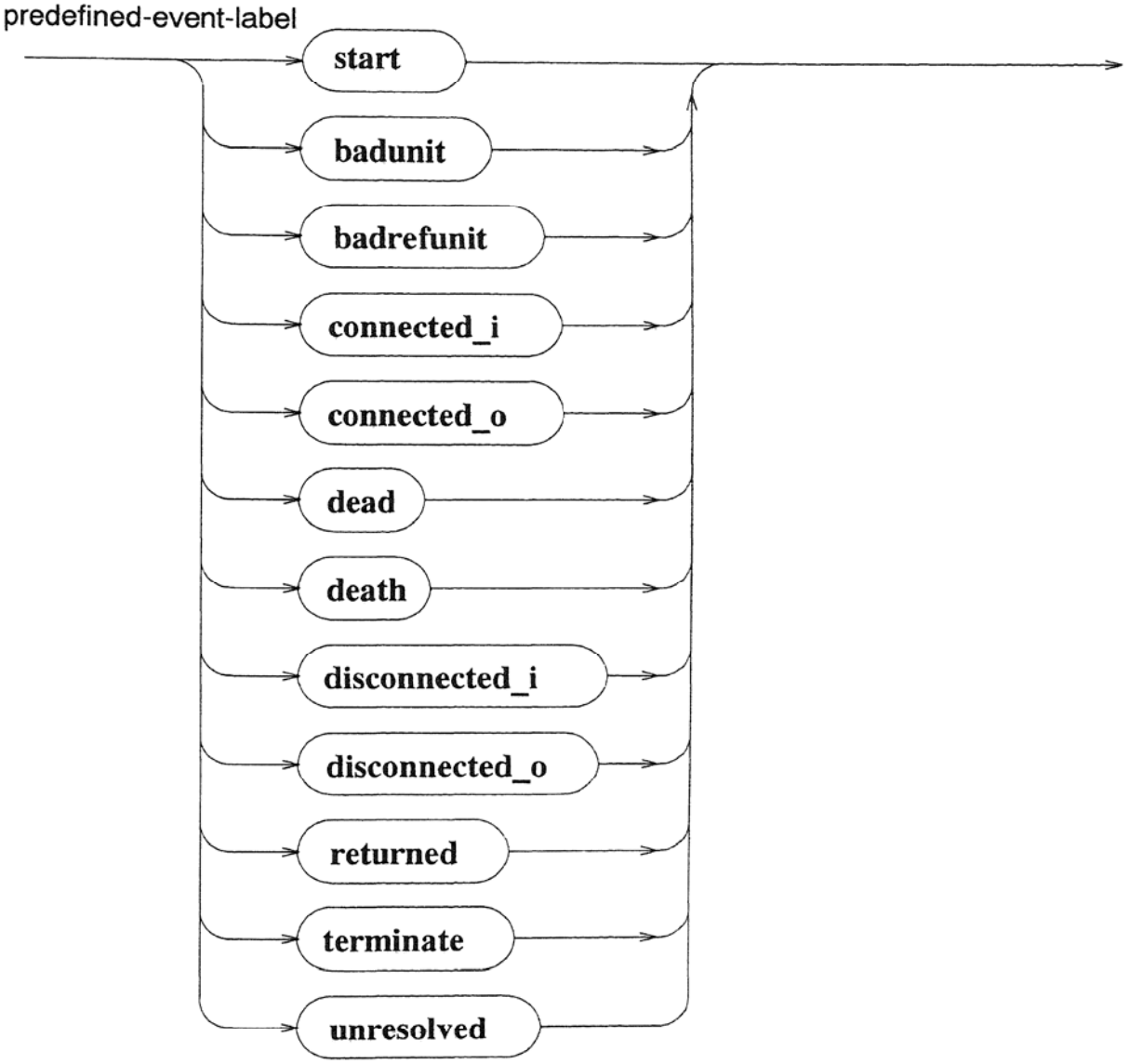

block
label
,
:
action 8
manner-call 11
ignore
save
.

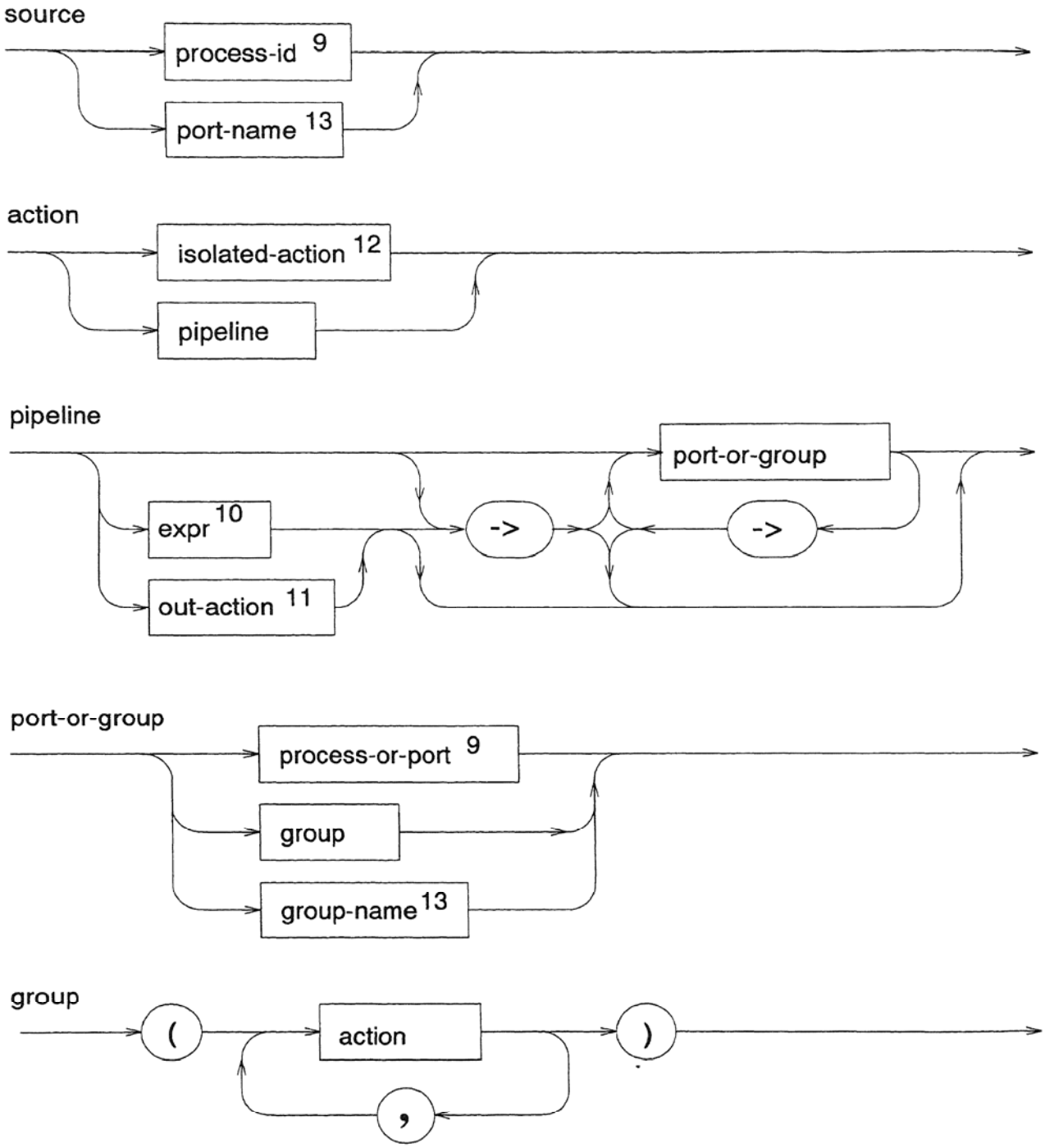

label
event-id
predefined-event-label 7
*
.
source 8
*


event-id
event-name 13
special-event
predefined-event 7
event_name

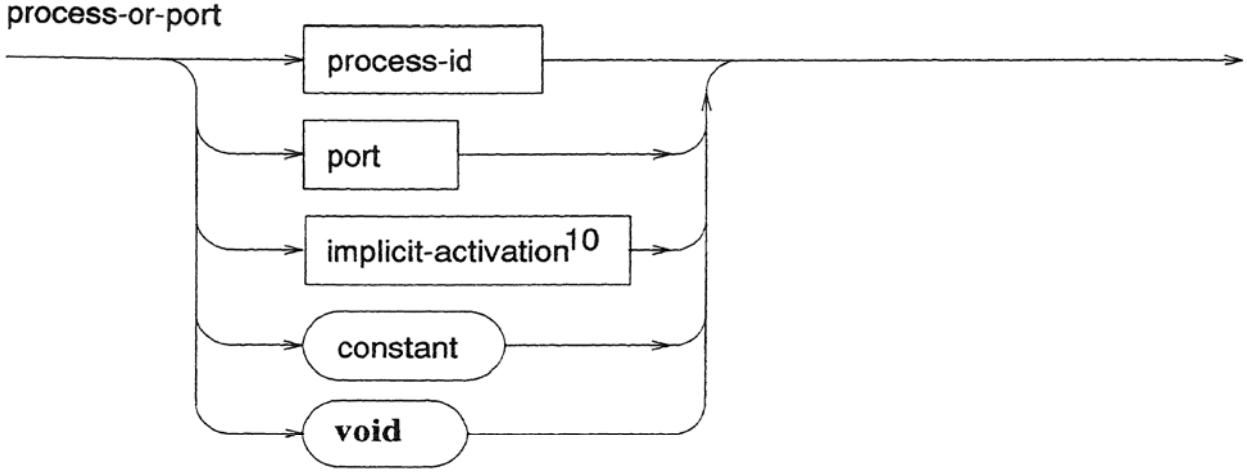

special-event
noevent


predefined-event
abort
break
terminate

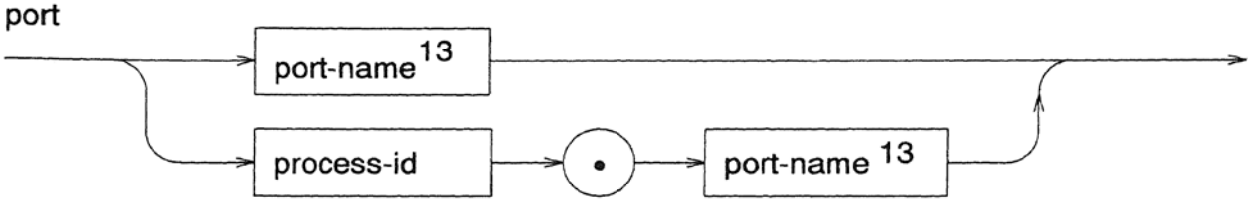

predefined-event-label
start
badunit
badrefunit
connected_i
connected_o
dead
death
disconnected_i
disconnected_o
returned
terminate
unresolved


source
process-id 9
port-name 13
action
isolated-action 12
pipeline
pipeline
port-or-group
expr 10
->
->
out-action 11
port-or-group
process-or-port 9
group
group-name 13
group
(
action
)
,

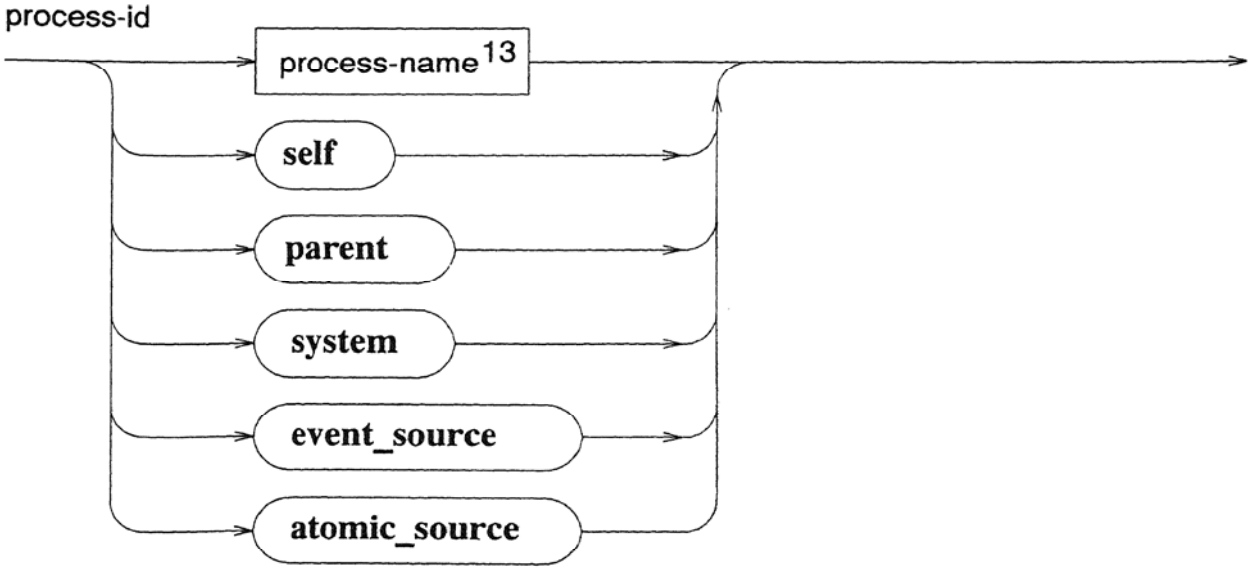
process-or-port

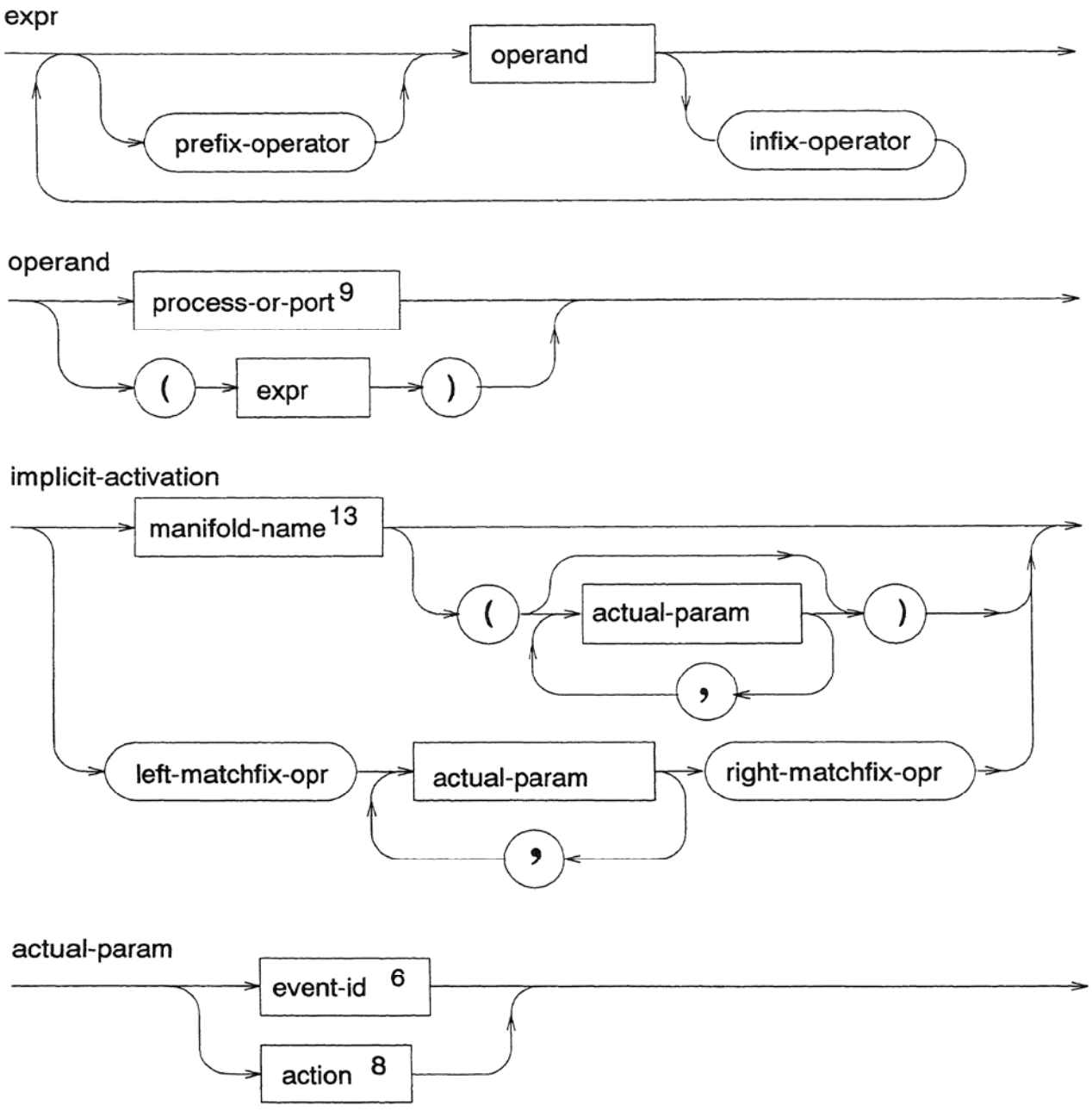


port

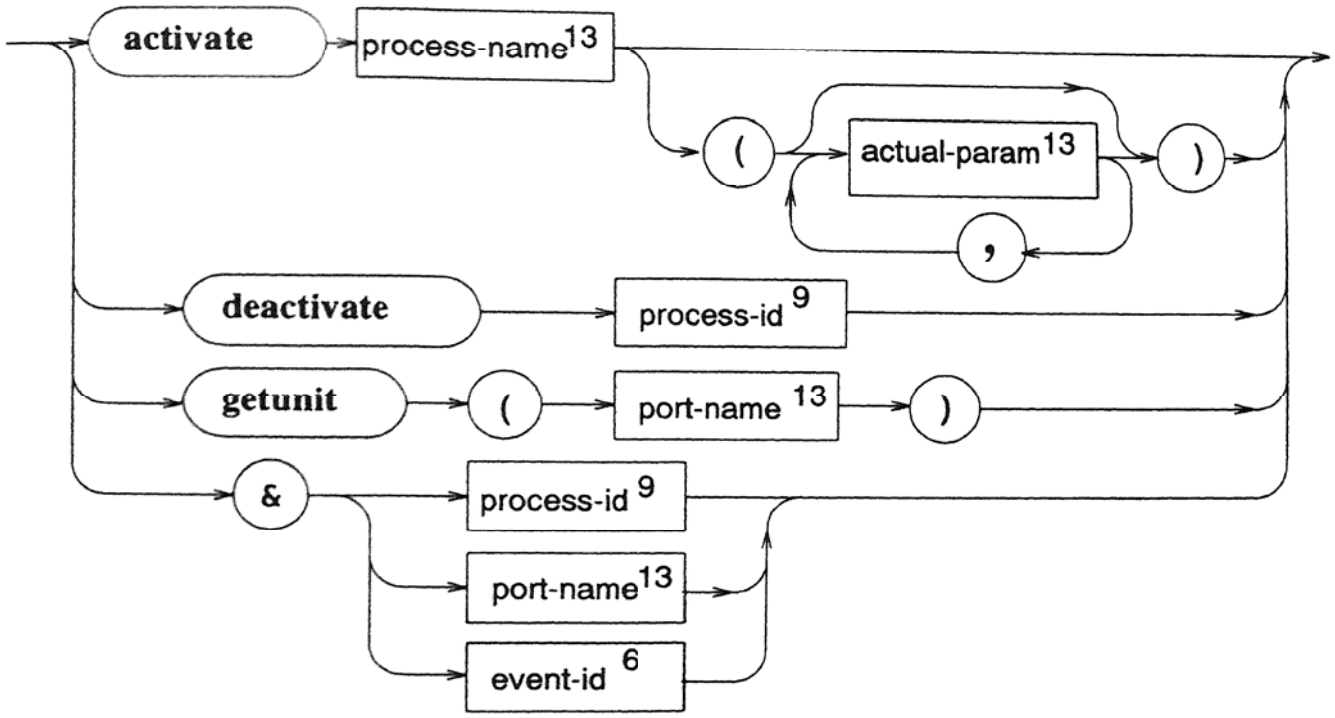


process-id

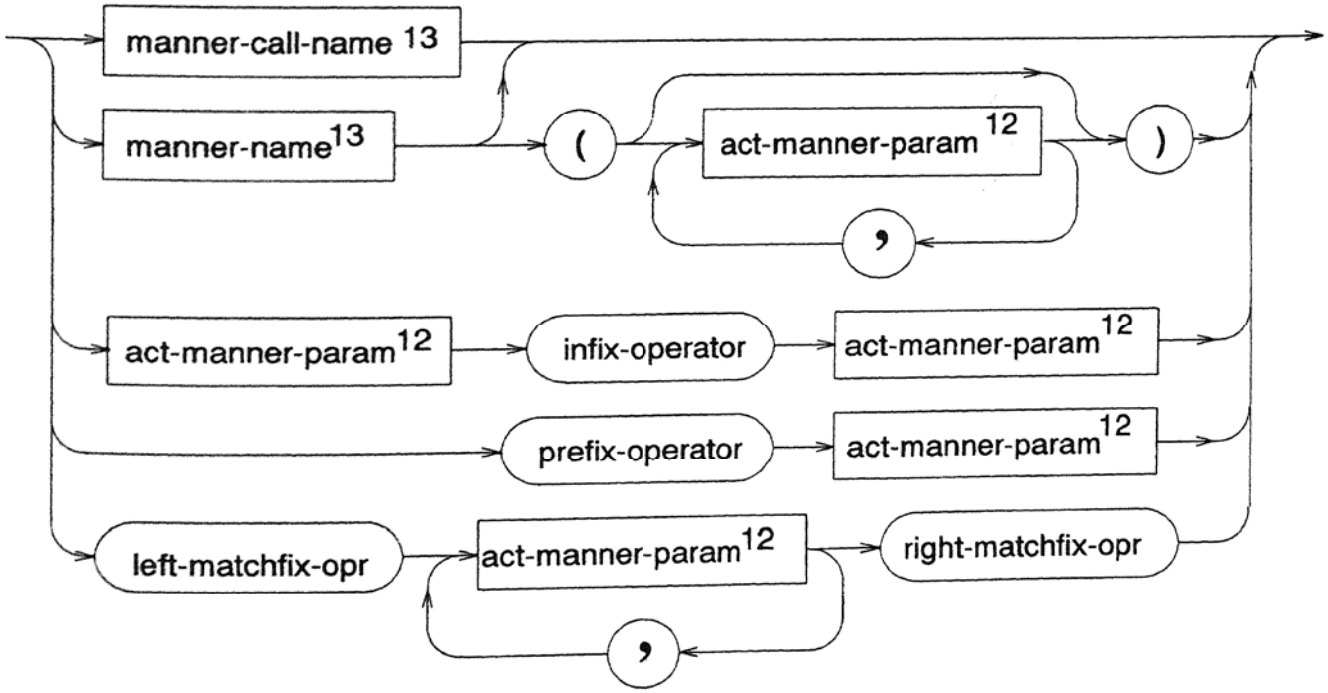



expr
operand
prefix-operator
infix-operator
operand
process-or-port 9
(
expr
)
implicit-activation
manifold-name 13
(
actual-param
)
,
left-matchfix-opr
actual-param
right-matchfix-opr
,
actual-param
event-id 6
action 8

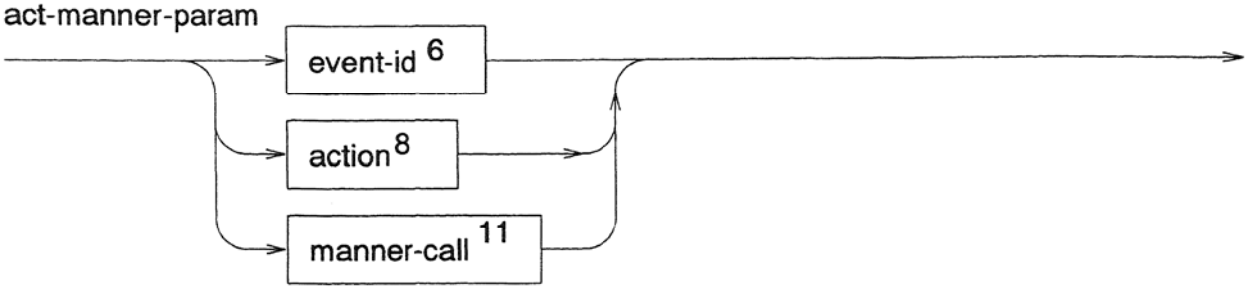
out-action



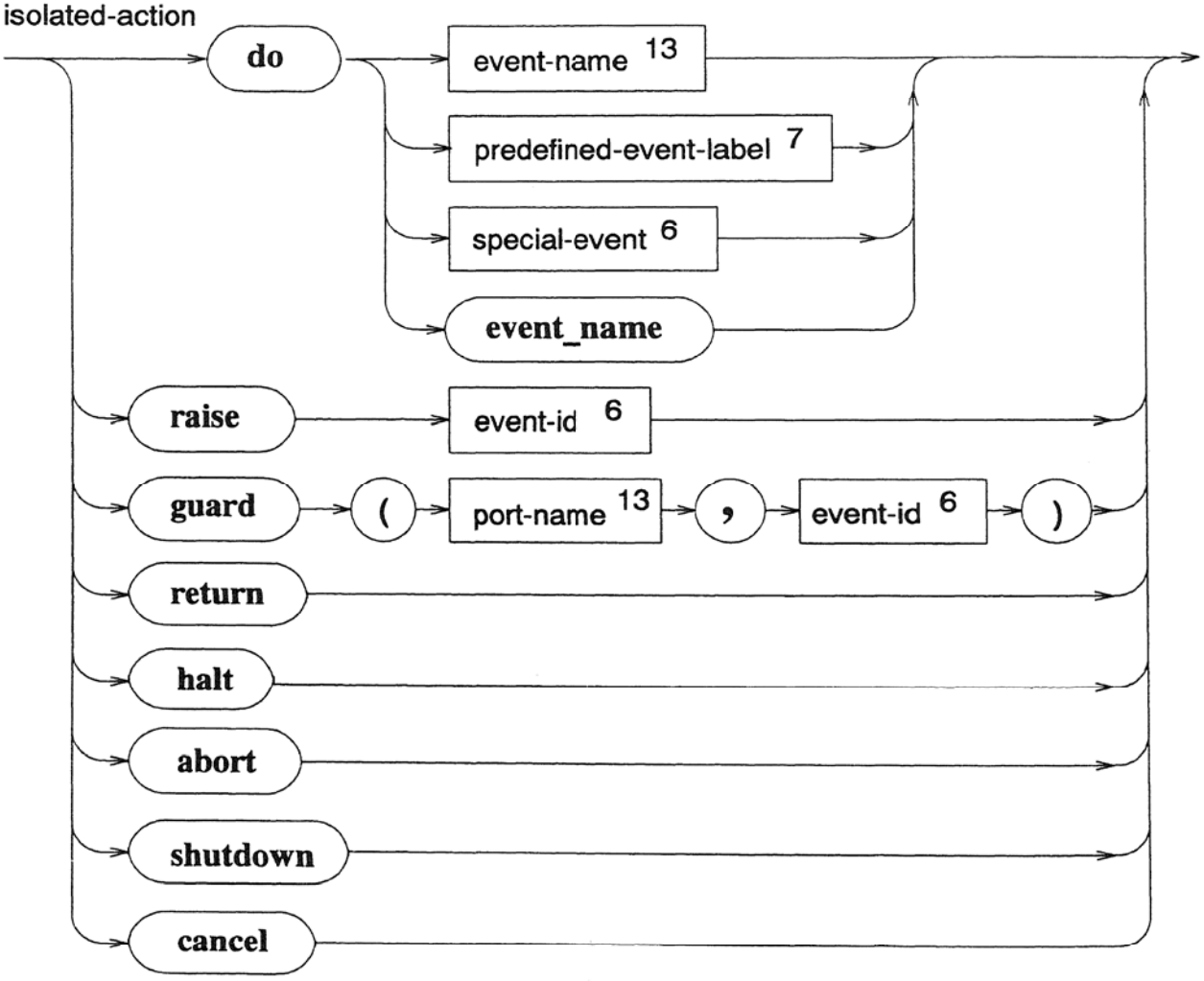
manner-call

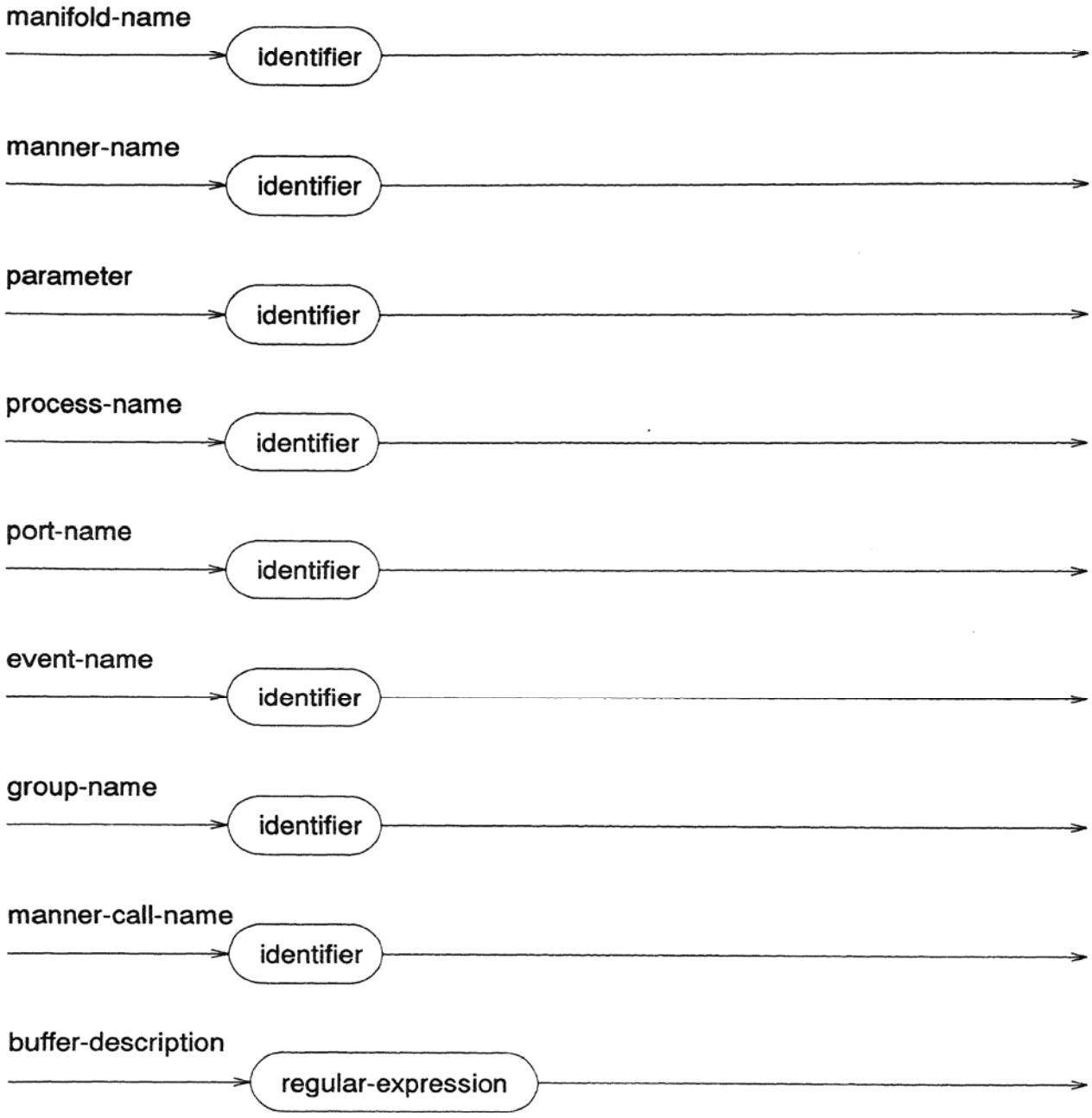


act-manner-param



isolated-action




manifold-name
identifier
manner-name
identifier
parameter
identifier
process-name
identifier
port-name
identifier
event-name
identifier
group-name
identifier
manner-call-name
identifier
buffer-description
regular-expression