Manray - a replicated workers program in Manifold

P. Spilling, F. Arbab

Computer Science/Department of Interactive Systems

Report CS-R9337 June 1993

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications. SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

P.O. Box 4079, 1009 AB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

manray - a replicated workers program in Manifold

P. Spilling and F. 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 computing. MANIFOLD is a *coordination* language whose sole purpose is to describe and manage complex interconnections among independent, concurrent processes. In the underlying paradigm of this language the primary concern is not with what *functionality* the individual processes in a parallel system provide. Instead, the emphasis is on these processes are *inter-connected* and how their *interaction patterns* change during the execution life of the system.
In this paper we briefly describe the language, and then, as an example of the application of MANIFOLD, we show how a replicated-workers program can be implemented in MANIFOLD. Our concern in this paper is to show the expressiveness of MANIFOLD, and its usefulness in practice. Issues regarding performance and optimization are beyond the scope of this paper.

AMS Subject Classification (1991): 68N99, 68Q10
Keywords & Phrases: replicated workers, parallel computing, coordination languages, parallel programming languages

1. INTRODUCTION

Specification and management of the communications among a set of concurrent processes is at the core of many problems of interest to a number of contemporary research trends. The theory of neural networks and the connectionist view of computation emphasize the significance of the concept of *management of connections* versus the local computation abilities of each node. The concept of dataflow programming has a certain resemblance with connectionism, albeit, it is closer to the discrete world of conventional programming than neural networks. Theoretical work on concurrency, e.g., CCS [1] and CSP [2, 3], is primarily concerned with the semantics of communications and interactions of concurrent sequential processes. Communication issues also come up in virtually every other type of computing, and have influenced the design (or at least, a few constructs) of most programming languages. However, not much effort has been spent on conceptual models and languages whose sole prime focus of attention is on the coordination of interactions among processes.

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

MANIFOLD is a language whose sole purpose is to manage complex interconnections among independent, concurrent processes. As such, like LINDA [5, 6], it is primarily a coordination language. However, there is no resemblance between LINDA and MANIFOLD, nor is there any similarity between the underlying models of these two languages. The details of the MANIFOLD model and the syntax and semantics of the MANIFOLD language are, of course, beyond the scope of this paper and are described in a separate document [7]. In this paper, we briefly describe the language, and show how

Report CS-R9337
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

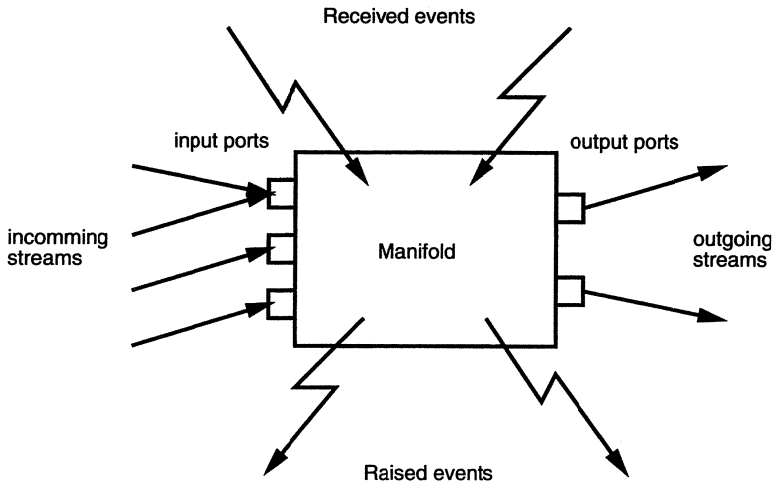

Figure 1: The model of a process in Manifold.

a replicated-workers program can be implemented in MANIFOLD. We summarize only enough of the description of the MANIFOLD model and language here, to make the examples and the significant programming issues presented in this paper understandable.

The rest of this paper is organized as follows. In §2 a more detailed description of the language is presented. In §3 we mention some of the application areas where MANIFOLD can prove to be a useful tool. In §4 we describe the implementation of the replicated-workers program, **manray**. Finally, §5 concludes this paper. The complete listing of the most important parts of the **manray** program is given in the appendix.

2. The Manifold Language

In this section we give a brief and informal overview of the MANIFOLD language. The sole purpose of the MANIFOLD language is to describe and manage complex communications and interconnections among independent, concurrent processes. As stated earlier, a detailed description of the syntax and the semantics of the MANIFOLD language and its underlying model is given elsewhere [7]. Other reports contain more examples of the use of the MANIFOLD language [8, 9, 10, 11].

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

Ports are regulated openings at the boundaries of processes through which they exchange units of information. The MANIFOLD language allows assigning special filters to ports for screening and

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

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

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

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

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

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

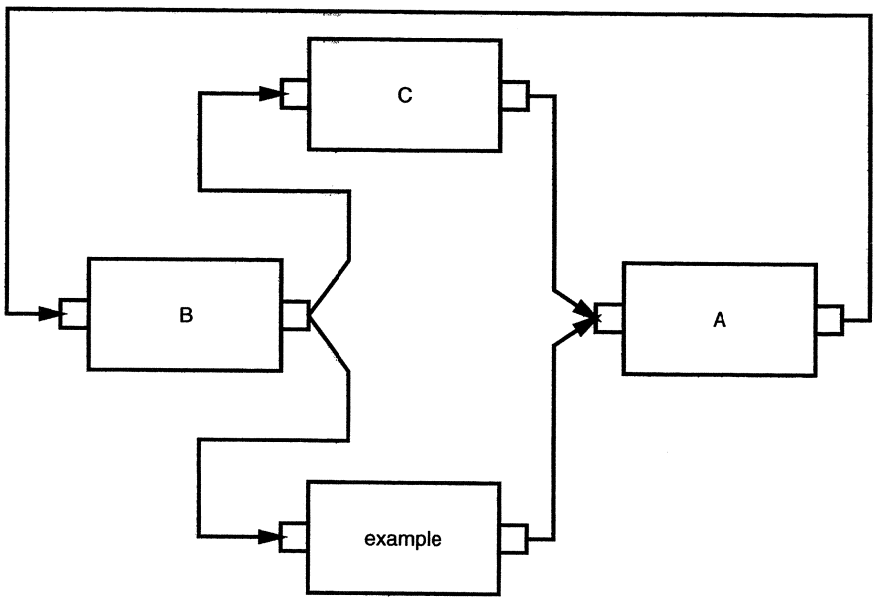

Figure 2: Connections set up by the manifold `example` on event `e1`.

2.1 Manifold Definition

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

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

Each event handler block describes a set of actions in the form of a *group* construct. The actions specified in a group are executed in some non-deterministic order. Usually, these actions lead to setting up *pipelines* between various ports of different processes. A *group* is a comma-separated list of members enclosed in a pair of parentheses. In the degenerate case of a singleton group (which contains only one member) the parentheses may be deleted. Members of a group are either primitive actions, pipelines, or groups. The setting up of pipelines within a group is simultaneous and atomic. No units flow through any of the streams inside a group before all of its pipelines are set up. Once set up, all pipelines in a group operate in parallel with each other.

A *primitive action* is typically *activating* or *deactivating* a process, *raising* an event, or a *do* action

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

```
example()                          // This is the header (there are no arguments):

    port in  input.                // These are the public declarations:
    port out output.               // Two ports are visible from the outside of the manifold
                                   // "example"; one is an input port and the other is an output one.

{                                  // The body of the manifold begins here.

    process A  is A_type.          // private declarations:
    process B  is B_type.          // three process instances are defined:
    process C  is C_type.

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


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


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


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

Listing 1: An example of a manifold process.

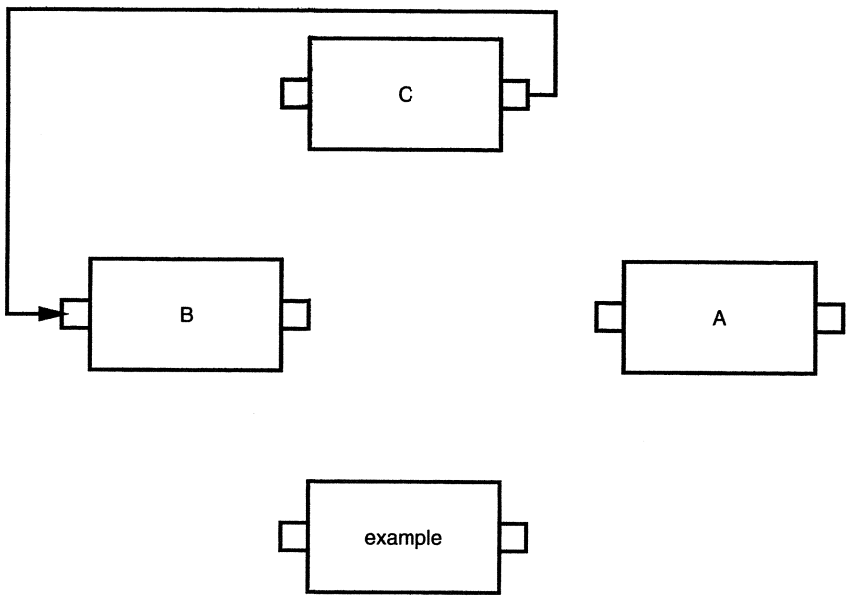

Figure 3: Connections set up by the manifold `example` on event `e2`.

which causes a transition to another handler block without an event occurrence from outside. A *pipeline* is an expression defining a tandem of streams, represented as a sequence of one or more groups, processes, or ports, separated by right arrows. It defines a set of simultaneous connections among the ports of the specified groups and processes. If the initial (final) name in such a sequence is omitted, the initial (final) connection is made to the current input (output) port. Inside a group, the current input and output ports are the input and output ports of the group. Elsewhere, the current input and output ports are `input` and `output`, i.e., the executing manifold's standard input and output ports. As an example, Figure 2 shows the connections set up by the manifold process `example` on Listing 1, while it is in the handling block for the event `e1` (for the details of event handling see §2.2). Figure 3 shows the connections set up in the handling block for the event `e2`.

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

An event handler block may also describe sequential execution of a series of (sets of) actions, by specifying a list of pipelines and groups, separated by the semicolon (;) operator[2]. In reaction to a recognized event, a manifold processor finds its appropriate event handler block and executes the list of sequential sets of actions specified therein. Once the manifold processor is through with the sequence in its current block, it terminates.

2.2 Event Handling

Event handling in MANIFOLD refers to a preemptive change of state in a manifold that observes an event of interest. This is done by its manifold processor which locates a proper event handler for the

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

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

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

The manifold processor in a given manifold is sensitive to (i.e., interested in) only those events for which the manifold has a handler. All other events are to be ignored. Thus, events that do not match any label in this search do not affect the manifold in any way (however, see §2.5 for the case of called manners). Similarly, if the appropriate block found for an event is the keyword `ignore`, the observed event is ignored. Normally, events handled by the current block are also ignored.

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

2.3 Event Handling Blocks

An event handling block consists of a comma-separated list of one or more block labels followed by a colon (`:`) and a single body. The body of an event handling block is either a group member (i.e., an action, a pipeline, or a group), or a single manner call(see §2.5). If the body of a block is a pipeline, and it starts (ends) with a $\rightarrow$, the port name `input` (respectively, `output`) is prepended (appended) to the pipeline.

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

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

2.4 Visibility of Event Sources

Every process instance or port defined or used anywhere in a manner (see §2.5) or manifold is an *observable* source of events for that manner or manifold. This simply means that occurrences of

events raised by such sources (only) will be picked up by the executing manifold processor, provided that there is a handling block for them. The set of all events from observable sources that match any of the block labels in a manner or manifold is the set of observable events for that manner or manifold. The set of-observable events of an executing manifold instance may expand and shrink dynamically due to manner calls and terminations (see §2.5). Depending on the state of a manifold processor (i.e., its current block), occurrences of observable events cause one of two possible actions: preemption of the current block, or saving of the event occurrence.

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

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

2.5 Manners

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

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

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

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

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

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

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

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

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

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

2.6 Scope Rules

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

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

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

MANIFOLD supports separate compilation. This is a very effective mechanism for modularization of large applications. In principle, all names defined and used in a source file are strictly local to that

file. Names (of events, manners, manifolds, or atomic processes) that are used in different source files and must indeed designate the same entity at execution time, must be explicitly declared as such using `extern`, `import`, and `export` constructs (see [7]).

3. Applications

The MANIFOLD language has already been used to describe some simple examples, like a parallel bucket sort algorithm, a simplified version of a (graphics) resource management and the like. The interested reader is referred to the reports published elsewhere [8, 9]. These examples were primarily meant to test the MANIFOLD concepts themselves. In this section we mention some of the possible application areas for MANIFOLD in large-scale and non-trivial parallel systems.

MANIFOLD is an effective tool for describing interactions of autonomous active agents that communicate in an environment through address-less messages and global broadcast of events. For example, elaborate user interface design means planning the cooperation of different entities (the human operator being one of them) where the event driven paradigm seems particularly useful. In our view, the central issue in a user interface is the design and implementation of the communication patterns among a set of modules[3]. Some of these modules are generic (application independent) programs for acquisition and presentation of information expressed in forms appealing to humans. Others are, ideally, acquisition/presentation-independent modules that implement various functional components of a specific application. Previous experience with User Interface Management Systems (see, e.g., [12]) has shown that concurrency, event driven control mechanisms, and general interconnection networks are all necessary for effective graphical user interface systems. MANIFOLD supports all of that and, in addition, provides a level of dynamism that goes beyond many other user interface design tools. As an example, it has recently been used to successfully reformulate the GKS[4] input model [13]; this work is regarded as a starting point in the development of new concepts for highly flexible, reconfigurable graphics systems suitable for parallel environments.

Separating the specification of the dynamically changing communication patterns among a set of concurrent modules from the modules themselves, seems to lead to better user interface architectures. A similar approach can also be useful in applications of real time computing where dynamic change of interconnection patterns (e.g., between measurement and monitoring devices and actuators) is crucial. For example, complex process control systems must orchestrate the cooperation of various programs, digital and/or analogue hardware, electronic sensors, human operators, etc. Such interactions may be more easily expressed and managed in MANIFOLD.

Coordination of the interactions among a set of cooperating autonomous intelligent experts is also relevant in Distributed Artificial Intelligence applications, open systems such as Computer Integrated Manufacturing applications, and the complex control components of systems such as Intelligent Computer Aided Design.

Recently, scientific visualization has raised similar issues as well. The problems here typically involve a combination of massive numerical calculations (sometimes performed on supercomputers) and very advanced graphics. Such functionality can best be achieved through a distributed approach, using segregated software and hardware tools. Tool sets like the Utah Raster Toolkit [14] were already a first step in this direction, although in the case of this toolkit the individual processes can be connected in a pipeline fashion only. More recently, software systems like the apE system of the Ohio Supercomputer Center [15], the commercially available AVS Visualization Package of Stardent Computer Ltd. [16], the IRIS Explorer system [17] and others, work on the basis of inter-connecting a whole set of different software/hardware components in a more sophisticated communication network. The successes of

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

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

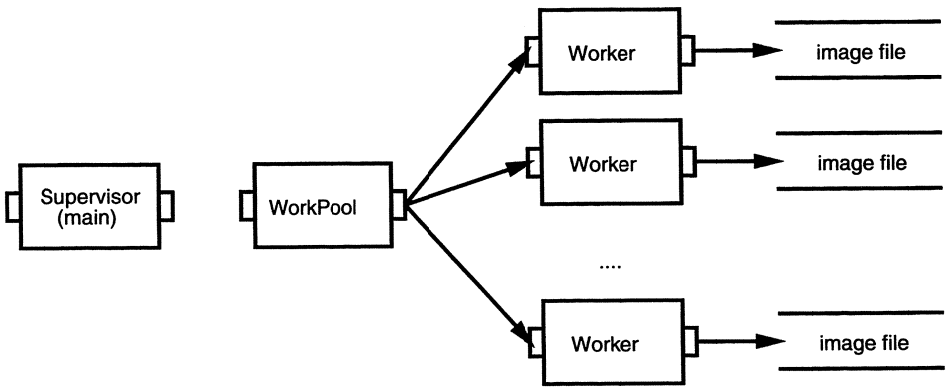

Figure 4: Overview of the **manray** replicated-workers program.

these packages, and mainly the general ideas behind them, point toward a more general development trend which leads to reconsideration of the software architecture used for graphics packages in general.

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

4. A REPLICATED WORKERS EXAMPLE - MANRAY

One of the commonly used paradigms in parallel programming is the **replicated workers paradigm** (also called agenda parallelism [18]). **manray** is an example of how this paradigm can be programmed in MANIFOLD.

There are three main components in a replicated workers program: a **supervisor** process, a set of identical **worker** processes, and a **work-pool** process, see Figure 4. Each worker is capable of performing any of the tasks in the work-pool. The dynamics of such a program is as follows: the supervisor initializes the work-pool process, starts up N workers, where N depends on the available resources, and waits for the workers to finish. The workers request work from the work-pool process, perform the task, and repeat, until there is no more work to do.

The replicated workers paradigm has several nice characteristics:

- A replicated-workers program executes the same way no matter how many workers there are, as long as there is at least one.
- Natural load-balancing is achieved by having (many) more tasks than there are workers. In this way one worker might be struggling with one big task while other workers might be performing many smaller tasks.
- It is fairly simple to implement.

- It has a broad area of applicability.

In the **manray** example the workers are instances of the public domain **rayshade** raytracing program [19], and the work to be done is contained in a rayshade input-file. The tasks handed out by the `WorkPool` process are sub-parts (a horizontal slice) of the image to raytrace. The output produced by the workers is a run-length-encoded description of the bitmap of their slice. The overall structure of the program has been inspired by an Orca version of a replicated workers program [20].

4.1 The supervisor process

The supervisor is the core of the program. The outline of the MANIFOLD code for this process is given in Listing 2. The supervisor starts up as many workers as indicated by the special atomic process `NPROCESSORS`. The description of the scene to be raytraced is contained in a special rayshade input-file. This name of this file is requested via the `GetFilename` manner. When all the initialization is done, the supervisor uses the `loop` manner to start up the workers. The body of the `loop` manner calls the `CreateWorker` manner that actually starts the `Worker` process instances.

The `active_workers` process is used by the supervisor to wait for the workers to finish. This is a *trigger-* or *condition* variable which is initialized with the number of workers to be started up. When the value of `active_workers` reaches a certain point, 0 in this case, it raises an event. This event is the signal to the supervisor process that all the workers have finished.

4.2 The work-pool process

The source code for the work-pool process is shown in Listing 5. The code of the two manners is also shown in listings (3 and 4) . The `GetTask` manner is part of the "public" interface to the `WorkPool` process. It is used by the worker processes when they want to receive a new task to work on. The reference operator, &, is used to generate a unit containing the reference to one of the ports of the worker process. This port is then used in the `WorkPool` process to return a task to the worker.

The `GiveTask` is a "private" manner used by the `WorkPool` process when it wants to reply to a get-task request from a worker process. The dereference primitive action, `deref`, gets the next unit from the port and dereferences its contents.

The `WorkPool` process itself is very simple. Since the workers get the name of the rayshade input-file on startup it only needs to give the workers a number to tell the worker which slice of the image it must raytrace. The get-task requests arrive on the input port of the `WorkPool` process. The **guard** installed on the input port raises the `give_task` event when a get-task request arrives. When there are no more tasks, the `WorkPool` process enters the `no_more_tasks` state. Workers that make a get-task request then receive a special value (0 in this case) to indicate that there are no more tasks to work on.

4.3 The worker process

The outline for the `Worker` process is shown in Listing 6. The `MY_PROCESSOR` process is a special atomic process which returns the identification of the CPU on which this worker runs. The worker repeatedly requests a task, then works, until there are no more tasks. The body of the loop is contained in the `work` event handler block. It uses the `Work` manner to start the `rayshade` raytracer.

5. Conclusions

In this paper we overview the MANIFOLD language and describe the implementation of a replicated-workers program in MANIFOLD. MANIFOLD uses the concepts of modern programming languages to describe and manage connections among a set of independent processes. The unique blend of

```
main()
{
    process print           is perm_sysoutput.
    process NP              is NPROCESSORS.
    process sh_workerid     is sh_variable.
    process workpool        is WorkPool.
    process ntasks          is variable.
    process active_workers  is sh_trigger_variable. //  condition variable
    process filename        is variable.
    process display         is variable.
    event   wait, all_dead.

    start:
        (  activate print, activate NP, activate sh_workerid,
           activate filename, activate active_workers( 0, &all_dead ),
           activate display, activate ntasks );

        GetFilename( filename, print );              //  the rayshade input file
        display        = Display();                  //  X-window display
        active_workers = NP;                         //  the number of workers
        ntasks         = NP * 2;                     //  the number of tasks
        activate workpool( ntasks, print );
        sh_workerid    = 0;

        //  Start up as many workers as there are processors

        loop( NP, CreateWorker( active_workers, ntasks,  workpool,
                                sh_workerid, filename, display, print ) );
        do wait.

    wait:                       //  Wait for the "all dead" event from "active workers"
        active_workers.

    all_dead.active_workers: //  Show the images and terminate on the users request.
        ...

}
```

Listing 2: The supervisor process.

```
export manner GetTask( work_pool, task_port, task )
    process work_pool.
    port in task_port.
    process task.
{
    // make a unit containing a reference to "self.task_port"
    // send it to "work_pool", and wait for a new task.

    start:
        &self.task_port -> work_pool;
        getunit( task_port ) -> task.
}
```

Listing 3: Manner used by worker processes to get a task from the work-pool process.

```
manner GiveTask( port, task )
    port in port.
    process task.
{
    // get a unit from "port", derefence it, and assign the next task to "worker"

    port out worker deref port.

    start:
        worker = task.
}
```

Listing 4: Manner used by the work-pool process.

```
export WorkPool( ntasks, print )
   process ntasks.
   process print.
{
   process n  is variable.
   event   give_task, no_more_tasks, give_no_task, serve.

   start:
       activate n; n = 1; do serve.

   serve:
       ( guard( input, give_task ), void ).

   give_task:
       GiveTask( input, n );
       n = n + 1;
       if ( n > ntasks, do no_more_tasks, do serve ).

   no_more_tasks:
       ( guard( input, give_no_task ), void ).

   give_no_task:
       GiveTask( input, 0 );            // task 0 means no more tasks
       do no_more_tasks.

   terminate:
       deactivate n.
}
```

Listing 5: The work-pool process.

```
Worker( active_workers, ntasks, workpool, workerid, filename, display, print )
     process active_workers.                    // shared condition variable
     process ntasks.
     process workpool.
     process workerid.
     process filename.
     process display.
     process print.
// ports:
    port in task_port.
{
    process my_processor  is MY_PROCESSOR.      // just for status information
    process task          is variable.
    process win           is winoutput_w_args. // output window for this worker
    process rayargs       is variable.
    process winargs       is variable.

    event   get_task, work, stop.

    start:
        ( activate my_processor, activate task,
          activate rayargs, activate winargs );

        // Open a window for this worker and print a start message
        ...

        // Repeatedly do: {get a task; work} until: no more tasks

        do get_task.

    get_task:
        GetTask( workpool, task_port, task );
        if ( task != 0, do work, do stop ).

    work:
        WorkMessage( workerid, task, win );
        rayargs = MakeArg( "-n ", ntasks, " -w ", task, " ", filename );
        Work( rayargs, win );                   // Start raytracing
        do get_task.

    stop:
        // Decrement the the shared trigger variable "active workers"
        // and wait for "main" to kill me.

        decrement( active_workers ); void.

    terminate:
        ( deactivate my_processor, deactivate rayargs,
          deactivate winargs, deactivate win, deactivate task ).
}
```

Listing 6: The worker process.

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

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

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

REFERENCES

1. R. Milner, *Communication and Concurrency.* Prentice Hall International Series in Computer Science, New Jersey: Prentice Hall, 1989.
2. C. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, August 1978.
3. C. Hoare, *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science, New Jersey: Prentice-Hall, 1985.
4. D. Gelernter and N. Carriero, "Coordination languages and their significance," *Communication of the ACM*, vol. 35, pp. 97–107, February 1992.
5. N. Carriero and D. Gelenter, "LINDA in context," *Communications of the ACM*, vol. 32, pp. 444–458, 1989.
6. W. Leler, "LINDA meets UNIX," *IEEE Computer*, vol. 23, pp. 43–54, February 1990.
7. F. Arbab, "Specification of manifold version 1.0," Tech. Rep. CS-R9220, Centrum voor Wiskunde en Informatica, Amsterdam, 1992.
8. F. Arbab and I. Herman, "Examples in Manifold," Tech. Rep. CS-R9066, Centrum voor Wiskunde en Informatica, Amsterdam, 1990.
9. F. Arbab and I. Herman, "Manifold: A language for specification of inter-process communication," in *Proceedings of the EurOpen Autumn Conference* (A. Finlay, ed.), (Budapest), pp. 127–144, September 1991.
10. F. Arbab, I. Herman, and P. Spilling, "Interaction management of a window manager in Manifold," in *Computing and Information ICCI'92* (W. Koczkodaj, P. Lauer, and A. Toptsis, eds.), (Toronto), IEEE Press, June 1992.
11. I. Herman and F. Arbab, "More examples in examples in Manifold," Tech. Rep. CS-R9214, Cen-

trum voor Wiskunde en Informatica, Amsterdam, 1992.

12. H. Schouten and P. ten Hagen, "Dialogue cell resource model and basic dialogue cells," *Computer Graphics Forum*, vol. 7, no. 3, pp. 311–322, 1988.
13. D. Soede, F. Arbab, I. Herman, and P. ten Hagen, "The GKS input model in manifold," *Computer Graphics Forum*, vol. 10, pp. 209–224, September 1991.
14. J. Peterson, R. Bogart, and S. Thomas, "The Utah Raster Toolkit," in *Proceedings of the Usenix Workshop on Graphics*, (Monterey, California), 1986.
15. S. Dyer, "A dataflow toolkit for visualization," *IEEE Computer Graphics & Applications*, vol. 10, July 1990.
16. C. Upson, "Scientific visualization environments for the computational sciences," in *Proceedings of the* 34^{th} *IEEE Computer Society International Conference*, (San Francisco), March 1989.
17. Silicon Graphics, Inc., "IRIS Explorer user's guide," tech. rep., Silicon Graphics, Inc., Mountain View, California, 1991.
18. N. Carriero and D. Gelenter, "How to write parallel programs: A guide to the perplexed," *ACM Computing Surveys*, vol. 21, no. 3, pp. 323–357, 1989.
19. C. E. Kolb, *Rayshade User's Guide and Reference Manual*, January 1992.
20. H. Bal, M. Kaashoek, and A. Tanenbaum, "Experience with distributed programming in orca," tech. rep., Vrije Universiteit, Amsterdam, The Netherlands, 1989.

A. Code listing

A.1 The supervisor process (main)

File name: supervisor.m

This file contains the code for the supervisor-part of a replicated-workers program in Manifold.

Import statements:

```
#include "misc_manifolds.i"
#include "sh_vars.i"
#include "loop_manner.i"
#include "os_interface.i"
#include "print_manners.i"
#include "network_utils.i"
#include "winoutput.i"
#include "arg.i"

manner CreateWorker( active_workers, ntasks, workpool,
                     sh_workerid, filename, display, print )
    process active_workers.
    process ntasks.
    process workpool.
    process sh_workerid.
    process filename.
    process display.
    process print.
import.

WorkPool( ntasks, print )
    process ntasks.
    process print.
import.

GetMaxYval( rayshade_ifile )
    port in rayshade_ifile.
import.

ShowImages( rayshade_ifile, max_yval, nparts, print )
    process rayshade_ifile.
    process max_yval.
    process nparts.
    process print.
import.
```

Manners which are part of the supervisor process definition:

```
manner GetFilename( filename, print )
    process filename.
    process print.
{
    process tmp is variable.

    start:
        activate tmp;
        GetArg( "Give rayshade input-filename: ", tmp, print );
        filename = CompleteFilename( tmp );
        PrintVar( "Filename is ", filename, print );
        deactivate tmp.
}

manner StartMessage( NP, print )
    process NP.
    process print.
{
    start:
        "main: There are " -> print; print = NP;
        "processors available. Starting workers..\n" -> print.
}
```

Supervisor (main) process definition:
This is the heart of the program. The supervisor uses the **NPROCESSORS** internal atomic process to inquire how many processors which are available.
The number of tasks is currently chosen to be 2 times the number of workers. This means that some workers might perform several light tasks while other workers might only perform 1 heavy task. In this way we get a a simple natural load-balancing. A more advanced sort of load-balancing might be done by analysing the image to be raytraced and split it up in more or less equally big tasks.
The **loop** manner is given two arguments: a value (**NP**), and a manner (**CreateWorker**). It will go through the loop NP times, calling the manner every time it executes the body of the loop. **ShowImages** is a manifold process which will combine the sub-parts of the image and start up an image-viewer process. The whole program will terminate when **ShowImages** dies (this happens when the user quits the image-viewer process).

```
main()
{
    process print          is perm_sysoutput.
    process NP             is NPROCESSORS.
    process sh_workerid    is sh_variable.
    process workpool       is WorkPool.
    process ntasks         is variable.
    process active_workers is sh_trigger_variable.  // guard variable
    process filename       is variable.
    process display        is variable.
    process max_yval       is variable.
    event   wait, all_dead.
```

```
    start:
        ( activate print, activate NP, activate sh_workerid,
          activate filename, activate active_workers( 0, &all_dead ),
          activate display, activate max_yval, activate ntasks );

        GetFilename( filename, print );
        max_yval       = GetMaxYval( filename );
        display        = Display();
        active_workers = NP;
        ntasks         = NP * 2;                    // number of tasks
        activate workpool( ntasks, print );
        sh_workerid    = 0;

        StartMessage( NP, print );

        // Start up as many workers as there are processors

        loop( NP, CreateWorker( active_workers, ntasks,  workpool,
                                sh_workerid, filename, display, print ) );
        do wait.

    wait:
        // Wait for the "all_dead" event from "active_workers"
        active_workers.

    all_dead.active_workers:
        "main: All workers should be finished now.\n" -> print;
        "main: Here are the images.\n" -> print;

        ShowImages( filename, max_yval, ntasks, print );

        ( deactivate print, deactivate NP, deactivate sh_workerid,
          deactivate filename, deactivate active_workers,
          deactivate display, deactivate max_yval, deactivate ntasks,
          deactivate workpool );

        ( print, NP, sh_workerid, filename,
          active_workers, display, max_yval ); shutdown.
}
```

A.2 The worker process
File name: worker.m

This is an example of a worker which can be used in a replictated-worker program in Manifold. The workers are instances of the RayShade program which are each given a sub-section of the scene to be raytraced.

Import statements:

```
#include "misc_manifolds.i"
```

```
#include "misc.i"
#include "sh_vars.i"
#include "loop_manner.i"
#include "os_interface.i"
#include "network_utils.i"
#include "winoutput.i"
#include "arg.i"
#include "print_manners.i"

manner GetTask( work_pool, task_port, task )
    process work_pool.
    port in task_port.
    process task.
import.

rayshade( args )
    port in args.
import.
```

Manners which are part of the Worker process definition:

```
manner StartMessage( workerid, my_processor, print )
    process workerid.
    process my_processor.
    process print.
{
    start:
        "Worker " -> print; print = workerid;
        PrintVar( " running on processor ", my_processor, print ).
}

manner WorkMessage( workerid, task, print )
    process workerid.
    process task.
    process print.
{
    start:
        "Worker " -> print; print = workerid;
        PrintVar( " working on task ", task, print ).
}

manner Work( rayargs, win )
    process rayargs.
    process win.
{
    process rs   is rayshade.

    start:
        // Activate the rayshade worker process. Connect the error port
        // of rs (stderr of the unix process) to win so that status
```

```
        // messages will be displayed in the window.

        ( activate rs( rayargs ), rs.error -> win ).
}
```

Worker process definition:
The "weight" pragma will cause the Manifold compiler to create code which will inform the RTS that "Worker" should be distributed.

```
pragma weight 10000 distribute Worker.

Worker( active_workers, ntasks, workpool, workerid, filename, display, print )
    process active_workers.                    // shared guard variable
    process ntasks.
    process workpool.
    process workerid.
    process filename.
    process display.
    process print.
// ports:
    port in task_port.
{
    process my_processor is MY_PROCESSOR.
    process task         is variable.
    process win          is winoutput_w_args.
    process rayargs      is variable.
    process winargs      is variable.
    event   get_task, work, stop.

    start:
        ( activate my_processor, activate task,
          activate rayargs, activate winargs );

        winargs = MakeArg( "-display ", display, " -title Worker_",
                           workerid, "_processor_", my_processor );
        PrintVar( "Worker: winargs are ", winargs, print );

        activate win( winargs );
        StartMessage( workerid, my_processor, win );
        do get_task.

    get_task:
        GetTask( workpool, task_port, task );
        if ( task != 0, do work, do stop ).

    work:
        WorkMessage( workerid, task, win );
        rayargs = MakeArg( "-n ", ntasks, " -w ", task, " ", filename );
        PrintVar( "Worker: rayargs are ", rayargs, win );
        Work( rayargs, win );
        do get_task.
```

```
    stop:
        "Worker finished..\n" -> win;

        // Decrement the the shared trigger variable "active_workers"
        // and wait for "main" kill me.

        decrement( active_workers ); void.

    terminate:
        ( deactivate my_processor, deactivate rayargs,
          deactivate winargs, deactivate win, deactivate task ).
}
```

CreateWorker manner definition:
This manner is used by the supervisor to create new workers.

```
export manner CreateWorker( active_workers, ntasks, workpool,
                            sh_workerid, filename, display, print )
    process active_workers.
    process ntasks.
    process workpool.
    process sh_workerid.
    process filename.
    process display.
    process print.
{
    // NB! Every worker must get their own copy of "sh_workerid"

    process worker    is Worker.
    process workerid  is variable.

    start:
        activate workerid; increment( sh_workerid ); workerid = sh_workerid;
        "creating worker...\n" -> print;
        activate worker( active_workers, ntasks, workpool,
                         workerid, filename, display, print ).
}
```

A.3 The work-pool process
File name: work_pool.m

This file contains the code for the **WorkPool** process. The workers must request tasks to work on from the **WorkPool** process by using the **GetTask** manner. A special value (0 in this case) will be returned when there is nothing more to do.

Import statements:

```
#include "built_in.i"
#include "print_manners.i"
```

GetTask manner definition:

```
export manner GetTask( work_pool, task_port, task )
    process work_pool.
    port in task_port.
    process task.
{
    start:
        &self.task_port -> work_pool;
        getunit( task_port ) -> task.
}
```

WorkPool process definition:

```
manner GiveTask( deref_port, task )
    port in deref_port.
    process task.
{
    port out worker deref deref_port.

    start:
        worker = task.
}

export WorkPool( ntasks, print )
    process ntasks.
    process print.
// ports:
    port in deref_port.
{
    process n is variable.
    event   give_task, no_more_tasks, give_no_task, serve.

    start:
        activate n; n = 1; do serve.

    serve:
        ( guard( input, give_task ), void ).

    give_task:
        getunit( input ) -> self.deref_port;
        GiveTask( deref_port, n );
        n = n + 1;
        if ( n > ntasks, do no_more_tasks, do serve ).

    no_more_tasks:
        ( guard( input, give_no_task ), void ).

    give_no_task:
```

```
        getunit( input ) -> self.deref_port;
        GiveTask( deref_port, 0 );            // task 0 means no more tasks
        do no_more_tasks.

    terminate:
        deactivate n.
}
```