Reusability of coordination programs

F. Arbab, C.L. Blom, F.J. Burger, C.T.H. Everaars

Computer Science/Department of Interactive Systems

Report CS-R9621 June 1996

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 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 0169-118X

Reusability of Coordination Programs

F. Arbab, C.L. Blom, F.J. Burger and C.T.H. Everaars

CWI

P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

Isolating computation and communication concerns into separate pure computation and pure coordination modules enhances modularity, understandability, and reusability of parallel and/or distributed software. This can be achieved by moving communication primitives (such as SendMessage and ReceiveMessage), which are now commonly scattered in application codes, into separate modules written in a language dedicated to the coordination of processes and the flow of information among them.

MANIFOLD is a pure coordination language that encourages the separation of communication and computation concerns, We use real, concrete, running MANIFOLD programs to demonstrate the concept of pure coordination modules and the advantage of their reuse in applications of different nature.

CR Subject Classification (1991): D3.3, D.1.3, D.3.2, F.1.2, I.1.3.
AMS Subject Classification (1991): 68N15, 68Q10.
Keywords and Phrases: software reusability, distributed computing, parallel computing, coordination languages, models of communication.

1. INTRODUCTION

Some of the shortcomings of the common approaches to the design and development of parallel and distributed applications stem from the fundamental properties of the various models of communication used to construct this software[5]. Without a proper programming paradigm for expressing the coordination of the cooperation of various active components that comprise a single concurrent application, programmers are forced to use low-level communication constructs, such as message passing, directly in their code. Because these primitives are generally scattered throughout the source code of the application and are typically intermixed with non-communication application code, the protocols of coordination generally never manifest themselves in a tangible form as easily identifiable pieces of source code. Thus, in spite of the fact that the coordination protocols are often the most complex and expensive-to-develop part of a non-trivial parallel or distributed application, they are not treated as a separate commodity that can be designed, developed, debugged, maintained, and reused, in isolation from the rest of the application code.

Intermixing communication concerns with computation decreases the comprehensibility, maintainability, and reusability of software modules. Moreover, the targeted send primitives used in message passing models of communication strengthen the dependence of individual processes on their environment. This too diminishes the reusability and maintainability of processes. It also complicates debugging and proving the correctness of programs because a process that depends on the existence and certain expected "valid" behavior of some other processes for its own correctness, by itself is not a well encapsulated concept.

Coordination languages[1] ameliorate these problems to some extent, but existing coordination languages, such as Linda, still do not go all the way to support reusable, pure coordination program modules. The goal of this paper is to demonstrate the concept of pure coordination modules and their reusability in different applications through real, concrete, running examples using the coordination language MANIFOLD, which is based on the IWIM model of communication developed at CWI. A

summary of the IWIM model is presented here in §2. An introduction to the MANIFOLD language which is based on that model appears in §3 in this paper. A more detailed description of IWIM and MANIFOLD appear [5], which also contains a review of some related work and a comparison with other coordination models and systems. An overview of an earlier version of the MANIFOLD language and its implementation was published in [2], which contains a series of other examples. The complete syntax and semantics of the current version of the MANIFOLD coordination language can be found in [4]. For our purpose in this paper, some of the relevant language constructs in MANIFOLD are introduced in §4 through a trivial example program. In §5, we discuss a quite non-trivial example of sorting and show how MANIFOLD encourages isolating communication and computation concerns into separate modules. The reusability of the pure coordination module developed for sorting is then demonstrated in §6, where the same MANIFOLD program is applied to coordinate a parallel/distributed numerical optimization application using a domain decomposition algorithm. All examples presented in this paper run, without any change to any source code, on a variety of parallel and/or heterogeneous distributed computing platforms. We close this paper with a short conclusion in §7.

2. Two Models of Communication

In this section we recapitulate the argument for having the *Idealized Worker Idealized Manager* (IWIM), Model of Communication, in contrast with the *Targeted-Send/Receive* Model (TSR), which describes the essentials of many commonly used communication models[5].

2.1 The TSR Model of Communication

In many models of communication, two active parties can be distinguished: the *Sender* and the *Receiver*. In general, a sender s sends a message m to a receiver r. The roles played by s and r are significantly different. For example, consider two cooperating processes p and q, which must communicate with each other. At some point, the process p produces two values to be passed on to q. The process q, in turn, after performing some additional computation using the input it receives from p, passes the results of its computation back to p. The source code for this concurrent application may look like the following:

```
process p:                          process q:

compute m1                          receive m1
send m1 to q                        let z be the sender of m1
compute m2                          receive m2
send m2 to q                        compute m using m1 and m2
do other things                     send m to z
receive m
do other computation using m
```

We make a few important observations in this example:

- The source code of both p and q contains not only the computations performed by these processes, but also a description of how they must cooperate with each other: computation code is *intermixed* with communication code.
- The operations "Send" and "Receive" are *asymmetric:* each "Send" needs to know the identity of its destination, whereas "Receive" can receive a message from any anonymous source.

  Especially because of the "Targeted Send", the resulting code is not as flexible as one may want it to be. Ror instance, in a different application environment, the result of q may be needed by another process x instead of the sending process z (whose identity was provided with the

messages $m1$ and $m2$). Then we have no choice but to modify the source code for q, thereby hampering the reusability of q.

2.2 The IWIM Model of Communication

The basic concepts in the IWIM model are *processes*, *events*, *ports*, and *channels*. A *process* is a *black box* with well defined *ports* which are openings in the bounding walls of the black box.

Channels connect ports and their existence enables processes to exchange *units* of information through their ports with other processes in their environment.

We use the notation $p.i$ to refer to the port i of the process instance p. The IWIM model supports *anonymous communication*: in general, a process does not, and need not, know the identity of the processes with which it exchanges information. This concept reduces the dependence of a process on its environment and makes processes more reusable.

Independent of the channels, there is an event mechanism for information exchange in IWIM. Events are broadcast by their sources in their environment, yielding an *event occurrence* in specific processes: those processes which are *tuned in* to certain event sources may pick up these event occurrences and then perform actions, such as creating or destroying processes, establishing channels between ports of processes, and generating more event occurences.

A process in IWIM can be regarded as a worker process or a manager (or coordinator) process. The responsibility of a worker process is to perform a certain (computational) task. A worker process is not responsible for the communication that is necessary for it to obtain the proper input it requires to perform its task, nor is it responsible for the communication that is necessary to deliver the results it produces to their proper recipients. In general, *no process in* IWIM *is responsible for its own communication with other processes.* It is always the responsibility of a manager process to arrange for and to coordinate the necessary communications among a set of worker processes.

There is always a bottom layer of worker processes, called *atomic workers*, in an application. In the IWIM model, an application is built as a (dynamic) hierarchy of (worker and manager) processes on top of this layer. A manager process may itself be considered as a worker process by another manager process. In this model, the example above can be formulated as follows:

```
process p:                          process q:                      process c:

compute m1                          read m1 from input port i1      ...
write m1 to output port o1          read m2 from input port i2      create the channel p.o1 →q.i1
compute m2                          compute m using m1 and m2       create the channel p.o2 →q.i2
write m2 to output port o2          write m to output port o1       create the channel q.o1 →p.i1
do other things                                                     ...
read m from input port i1
do other computation using m
```

Now the processes p and q are "ideal" workers. They do not know and do not care where their input comes from, nor where their output goes to. They always do their job provided that they receive the right input at the right time. Under these provisions, they can trivially be *reused* in any other application.

The process c is an "ideal" manager. It knows nothing about the details of the tasks performed by p and q. Its only concern is to ensure that they are created at the right time, receive the right input from the right sources, and deliver their results to the right sinks.

It is likely that some of such ideal manager processes may be used in other applications, coordinating very different worker processes, producing very different results; as long as their cooperation follows the same protocol, the same coordinator processes can be reused in very different applications.

The purpose of this paper is to demonstrate how the latter type of reusability (of *coordinator modules*) can be utilized in practice.

3. The Manifold Coordination Language

In this section, we briefly introduce MANIFOLD: a coordination language for managing complex, dynamically changing interconnections among sets of independent, concurrent, cooperating processes[2], which is based on the IWIM model, described in §2.

A MANIFOLD application consists of a (potentially very large) number of (light- and/or heavy-weight) processes running on a network of heterogeneous hosts, some of which may be parallel systems. Processes in the same application may be written in different programming languages. Some of them may not know anything about MANIFOLD, nor the fact that they are cooperating with other processes through MANIFOLD in a concurrent application.

The MANIFOLD system consists of a compiler, a run-time system library, a number of utility programs, libraries of builtin and predefined processes[4], a link file generator called MLINK and a run-time configurator called CONFIG. The system has been ported to several different platforms (e.g., SGI 5.3, SUN 4, Solaris 5.2, IBM SP/1). MLINK uses the object files produced by the (MANIFOLD and other language) compilers to produce link files needed to compose the application executable files for each required platform. At run time of an application, CONFIG determines the actual host(s) where the processes which are created in the MANIFOLD application will run.

The library routines that comprise the interface between MANIFOLD and processes written in other languages (e.g. C), automatically perform the necessary data format conversions when data is routed between various different machines.

3.1 Processes

In MANIFOLD, the atomic workers of the IWIM model are called atomic processes. Any operating system-level process can be used as an atomic process in MANIFOLD. However, MANIFOLD also provides a library of functions that can be called from a regular C function running as an atomic process, to support a more appropriate interface between the atomic processes and the MANIFOLD world. Atomic processes can only produce and consume units through their ports, generate and receive events, and compute. In this way, the desired separation of computation and coordination is achieved.

Coordination processes are written in the MANIFOLD language and are called manifolds. The MANIFOLD language is a block-structured, declarative, event driven language. A manifold definition consists of a header and a body. The header of a manifold gives its name, the number and types of its parameters, and the names of its input and output ports. The body of a manifold definition is a block. A block consists of a finite number of states. Each state has a label and a body. The label of a state defines the condition under which a transition to that state is possible. It is an expression that can match observed event occurrences in the event memory of the manifold. The body of a simple state defines the set of actions that are to be performed upon transition to that state. The body of a compound state is either a (nested) block, or a call to a parameterized subprogram known as a *manner* in MANIFOLD. A manner consists of a header and a body. As for the subprograms in other languages, the header of a manner essentially defines its name and the types and the number of its parameters. A manner is either atomic or regular. The body of a regular manner is a block. The body of an atomic manner is a C function that can interface with the MANIFOLD world through the same interface library as for the compliant atomic processes.

3.2 Streams

All communication in MANIFOLD is asynchronous. In MANIFOLD, the asynchronous IWIM channels are called streams. A stream is a communication link that transports a sequence of bits, grouped into (variable length) *units*.

A stream represents a reliable and directed flow of information from its *source* to its *sink*. Once a stream is established between a producer process and a consumer process, it operates autonomously and transfers the units from its source to its 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.

There are four basic stream types designated as BB, BK, KB, and KK, each behaving according to a slightly different protocol with regards to its automatic disconnection from its source or sink. Furthermore, in MANIFOLD, the BK and KB type streams can be declared to be *reconnectable*. See [4] or [5] for details.

Note that as in the IWIM model, the constructor of a stream between two processes is, in general, a third process. Stream definitions in MANIFOLD are generally additive. This means that a port can simultaneously be connected to many different ports through different streams.

3.3 Events

In MANIFOLD, once an event is *raised* by a process, it continues with its processing, while the event occurrence propagates through the environment independently. Any receiver process that is interested in such an event occurrence will automatically receive it in its *event memory*. The observed event occurrences in the event memory of a process can be examined and reacted on by this process at its own leisure. The event memory of a process behaves as a set: there can be at most one copy of the occurrence of the same event generated by the same source in an event memory.

3.4 State Transitions

The only control structure in the MANIFOLD language is an event-driven state transition mechanism. More familiar control structures, such as the sequential flow of control represented by the connective ";" (as in Pascal and C), conditional (i.e., "if") constructs, and loop constructs can be built out of this event mechanism, and are also available in the MANIFOLD language as convenience features.

Upon transition to a state, the primitive actions specified in its body are performed atomically in some non-deterministic order. Then, the state becomes *preemptable*: if the conditions for transition to another state are satisfied, the current state is preempted, meaning that all streams that have been constructed are dismantled and a transition to a new state takes place. The most important primitive actions in a simple state body are (1) creating and activating processes, (2) generating event occurrences, and (3) connecting streams to the ports of various processes.

4. HELLO WORLD!

For our first example, consider a simple program to print a message such as "Hello World!" on the standard output. The MANIFOLD source file for this program contains the following:

```
1 manifold printunits import.
2
3 auto process print is printunits
4
5 manifold Main
6 (
7   begin: "Hello World!" -> print.
8 )
```

The first line of this code defines a manifold named `printunits` that takes no arguments, and states (through the keyword `import`) that the real definition of its body is contained in another source file. This defines the "interface" to a process type definition, whose actual "implementation" is given elsewhere. Whether the actual implementation of this process is an atomic process (e.g., a C function) or it is itself another manifold is indeed irrelevant in this source file. We assume that `printunits` waits to receive units through its standard input port and prints them. When `printunits` detects that there are no incoming streams left connected to its input port and it is done printing the units it has received, it terminates.

The second line of code defines a new instance of the manifold `printunits`, calls it `print`, and states (through the keyword `auto`) that this process instance is to be automatically activated upon creation, and deactivated upon departure from the scope wherein it is defined; in this case, this is the end of the application. Because the declaration of the process instance `print` appears outside of any blocks in this source file, it is a global process, known by every instance of every manifold whose body is defined in this source file.

The last lines of this code define a manifold named `Main` that takes no parameters. Every manifold definition (and therefore every process instance) always has at least three default ports: `input`, `output`, and `error`. The definition of these ports are not shown in this example, but the ports are defined for `Main` by default.

The body of this manifold is a block (enclosed in a pair of braces) and contains only a single state. The name `Main` is indeed special in MANIFOLD: there must be a manifold with that name in every MANIFOLD application and an automatically created instance of this manifold, called `main`, is the first process that is started up in an application. Activation of a manifold instance automatically posts an occurrence of the special event `begin` in the event memory of that process instance; in this case, `main`. This makes the initial state transition possible: `main` enters its only state – the `begin` state.

The `begin` state contains only a single primitive action, represented by the stream construction symbol, "→". Entering this state, `main` creates a stream instance (with the default BK-type) and connects the `output` port of the process instance on the left-hand side of the → to the `input` port of the process instance on its right-hand side. The process instance on the right-hand side of the → is, of course, `print`. What appears to be a character string constant on the left-hand side of the → is also a process instance: in MANIFOLD, a constant is just a process that produces its value as a unit on its `output` port and then terminates. [1].

Having made the stream connection between the two processes, `main` now waits for all stream connection made in this state to break up (on at least one of their ends). The stream breaks up, in this case, on its source end as soon as the string constant delivers its unit to the stream and dies. Since there are no other event occurrences in the event memory of `main`, the default transition for a state reaching its end (i.e., falling over its terminator period) now terminates the process `main`.

Meanwhile, `print` reads the unit and prints it. The stream type BK ensures that the connection between the stream and its sink is preserved even after a preemption, or its disconnection from its source. Once the stream is empty and it is disconnected from its source, it automatically disconnects from its sink. Now, `print` senses that it has no more incoming streams and dies. At this point, there are no other process instances left and the application terminates.

Note that our simple example, here, consists of three process instances: two worker processes, a character string constant and `print`, and a coordinator process, `main`. Figure 1 shows the relationship between the constant and `print`, as established by `main`. Note also that the coordinator process `main` only establishes the connection between the two worker processes. It does *not* transfer the units

[1] Conceptually, constants are full-fledged process instances in MANIFOLD. However, in reality, they are implemented as only a block of memory.

Figure 1: The "Hello World" example in Manifold

through the stream(s) it creates, nor does it interfere with the activities of the worker processes in other ways.

5. Bucket Sort

The example in the previous section was simple enough to require only a static pattern of communication. In this section, we illustrate the dynamic capabilities of MANIFOLD through a program for sorting an unspecified number of input units. The particular algorithm used in this example is not necessarily the most effective one. However, it is simple to describe, and serves our purpose of demonstrating the dynamic aspects of the MANIFOLD language well. The sort algorithm is as follows.

There is a sufficiently large (theoretically, infinite) number of *atomic sorters* available, each of which is able to sort a bucket of $n > 0$ units very efficiently. (The number n may even vary from one atomic sorter to the next.) Each atomic sorter receives its input through its `input` port; raises a specific event it receives as a parameter to inform other processes that it has filled up its input bucket; sorts its units; produces the sorted sequence of the units through its `output` port; and terminates.

The parallel bucket sort program is supposed to feed as much of its own input units to an atomic sorter as the latter can take; feed the rest of its own input as the input to another copy of itself; merge the two output sequences (of the atomic sorter and its new copy); and produce the resulting sequence through its own `output` port. Merging of the two sorted sequences can be done by a separate merger process, or by a subprogram (i.e., a manner) called by the sorter.

We assume our application consists of several source files. The first source file contains our `Main` manifold, as shown below. We assume that the merger is a separate process. The merger and the atomic sorter can be written in the MANIFOLD language, but they will be more efficient if they are written in a computation language, such as C. We do not concern ourselves here with the details of the merger and the atomic sorter, and assume that each is defined in a separate source file.

The `main` manifold in this application creates `read`, `sort`, and `print` as instances of manifold definitions `ReadFile`, `Sorter`, and `printunits`, respectively. It then connects the `output` port of `read` to the `input` port of `sort`, and the `output` port of `sort` to the `input` port of `print`. The process `main` terminates when both of these connections are broken.

The process `read` is expected to read the contents of the file `unsorted` and produce a unit for every sort item in this file through its `output` port. When it is through with producing its units, `read` simply terminates. The process `sort` is an instance of the manifold definition `Sorter`, which is expected to sort the units it receives through its `input` port. This process terminates when its `input` is disconnected and all of its output units are delivered through its `output` port.

The manifold definition `Sorter`, shown below, is our main interest. In its `begin` state, an instance of `Sorter` connects its own `input` to an instance of the `AtomicSorter`, it calls `atomsort`. It also installs two *guards*, one on each of its `input` and `output` ports. The guard on the `input` port posts the event `finished` if it has an empty stream connected to its departure side, after the arrival side of this port has no more stream connections, following a first connection. This means that the event `finished` is posted in an instance of `Sorter` after a first connection to the arrival side of its `input` is made, then all connections to the arrival side of its `input` are severed, and all units passed through this port are consumed. The guard on the `output` port posts the event `flushed` after there is no stream connected to the arrival side of this port following its first connection. This means that the

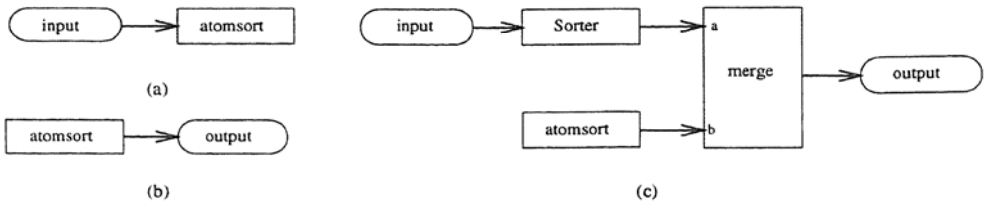

Figure 2: Bucket sort

event `flushed` is posted in an instance of `Sorter` after a connection is made to its arrival side, and all units arriving at this port have passed through. The connections in this state are shown in Figure 2.a.

```
1 export manifold Sorter()
2 {
3   event filled, flushed, finished.
4   process atomsort is AtomicSorter(filled).
5   stream reconnect KB input -> *.
6   priority filled < finished.
7
8   begin: (
9     activate(atomsort), input -> atomsort,
10    guard(input, a_everdisconnected!empty, finished) // no more input
11  ).
12
13  finished: (
14    ignore filled. //possible event form atomsort
15
16    begin: atomsort -> output  //your output is only that of atomsort
17  ).
18
19  filled: (
20    process merge<a, b | output> is AtomicIntMerger.
21    stream KK * -> (merge.a, merge.b).
22    stream KK merge -> output.
23
24    begin: (
25      activate(merge),
26      input -> Sorter -> merge.a,
27      atomsort -> merge.b,
28      merge -> output
29    ).
30
31    end | finished:.
32  ).
33
34  end: (
35    begin: (
36      guard(output, a_disconnected, flushed), // ensure flushing
37      terminated(void) //wait for units to flush through output
38    ).
39
40    flushed: halt.
41  ).
42 }
```

Two events can preempt the `begin` state of an instance of `Sorter`: (1) if the incoming stream connected to `input` is disconnected (no more incoming units) and `atomsort` reads all units available in its incoming stream, the guard on `input` posts the event `finished`; and (2) the process `atomsort` can read its fill and raise the event `filled`. Normally, only one of these events occurs; however, when the number of input units is exactly equal to the bucket size, n, of `atomsort`, both `finished` and `filled` can occur simultaneously. In this case, the priority statement makes sure that the handling of `finished` takes precedence over `filled`.

Assume that the number of units in the input supplied to an instance of `Sorter` is indeed less than

or equal to the bucket size n of an atomic sorter. In this case, the event `finished` will preempt the `begin` state and cause a transition to its corresponding state in `Sorter`. In this state, we ignore the occurrence of `filled` that may have been raised by `atomsort` (if the number of input units is equal to the bucket size n); and deliver the output of `atomsort` as the output of the `Sorter`. The connections in this state are shown in Figure 2.b.

Now suppose the number of units in the input supplied to an instance of `Sorter` is greater than the bucket size n of an atomic sorter. In this case, the event `filled` will preempt the `begin` state and cause a transition to its corresponding state in `Sorter`. In this state we create an instance of the merger process, called `merge`. A new instance of the `Sorter` is created in the `begin` state of the nested block. The rest of the input is passed on as the input to this new `Sorter`, and its output is merged with the output of the atomic sorter and the result is passed as the output of the `Sorter` itself. The connections in this state are shown in Figure 2.c. An occurrence of `finished` in this state preempts the connected streams and causes a transition to the local `finished` state in this block. This preemption is necessary to inform the new instance of `Sorter` (by breaking the stream that connects `input` to it) that it has no more input to receive, so that it can terminate. The empty body of the `finished` state means that it causes an exit from its containing block.

The purpose of the `end` state in `Sorter` is to make sure it stays alive until all units in the incoming streams connected to its `output` are transferred out to some outgoing stream. To see why this is necessary, consider an extreme case where there is no outgoing stream connected (from the outside) to the `output` port of an instance of `Sorter`. The streams set up in either of the states depicted in Figures2.b and 2.c can break up, signaling the end of their respective states; i.e., the manifold instance can "fall off the edge" over the terminator period of either of these two states. If there is no `end` state in the manifold definition, this results in termination of the manifold instance. Should this happen, (part of) the output of the `Sorter` instance will be lost, since it remains in the incoming stream connected to its `output` port as it dies.

In the `end` state, a `Sorter` instance waits for the termination of the special predefined process `void`, which will never happen (the special process `void` never terminates). This effectively causes the `Sorter` instance to hang indefinitely. The only event that can terminate this indefinite wait is an occurrence of `flushed` which indicates there are no more units pending to go through the `output` port of the `Sorter` instance.

An interesting aspect of the `Sorter` manifold is the dynamic way in which it switches connections among the process instances it creates. Perhaps more interesting, is the fact that, in spite of its name, `Sorter` knows nothing about sorting! If we change its name to X, and systematically change the names of the identifiers it uses to Y_1 through Y_k, we realize that all it knows is to divert its own input to an instance of some process it creates; when this instance raises a certain event, it is to divert the rest of its input to a new instance of itself; and to divert the output of these two processes to a third process, whose output is to be passed out as its own output.

What `Sorter` embodies is a protocol that describes how instances of two process definitions (e.g., `AtomicSorter` and `AtomicIntMerger` in our case) should communicate with each other. Our `Sorter` manifold can just as happily orchestrate the cooperation of any pair of processes that have the same input/output and event behavior as `AtomicSorter` and `AtomicIntMerger` do, regardless of what computation they perform. The cooperation protocol defined by `Sorter` simply doles out chunks of its input stream to instances of what it knows as `AtomicSorter` and diverts their output streams to instances of what it knows as `AtomicIntMerger`. What is called `AtomicSorter` needs not really sort its input units, the process called `AtomicIntMerger` needs not really merge them, and neither has to produce as many units through its `output` as it receives through its `input` port. They can do any computation they want.

By parameterizing the names of the manifolds used in `Sorter` and changing its name to `ProtocolX`,

we obtain a more general program:

```
1 export manifold ProtocolX(manifold M1(event), manifold M2<a, b | output>)
2 {
3   event filled, flushed, finished.
4   process m1 is M1(filled).
5   stream reconnect KB input -> *.
6   priority filled < finished.
7
8   begin: (
9     activate(m1), input -> m1,
10    guard(input, a_everdisconnected!empty, finished) // no more input
11  ).
12
13  finished: (
14    ignore filled. //possible event from m1
15
16    begin: m1 -> output  //your output is only that of m1
17  ).
18
19  filled: (
20    process m2<a, b | output> is M2.
21    stream KK * -> (m2.a, m2.b).
22    stream KK m2 -> output.
23
24    begin: (
25      activate(m2),
26      input -> ProtocolX(M1, M2) -> m2.a,
27      m1 -> m2.b,
28      m2 -> output
29    ).
30
31    end | finished:.
32  ).
33
34  end: (
35    begin: (
36      guard(output, a_disconnected, flushed), // ensure flushing
37      terminated(void) //wait for units to flush through output
38    ).
39
40    flushed: halt.
41  ).
42 }
```

The keyword `export` on line 1 allows other separately compiled MANIFOLD source files to `import` and use this coordinator manifold, e.g., from a protocol library. The new version of the bucket sort program using `ProtocolX` is:

```
1 manifold printunits import.
2 manifold ProtocolX(manifold M1(event), manifold M2) import.
3 manifold ReadFile(process filename) atomic {internal.}.
4 manifold AtomicSorter(event) atomic {internal.}.
5 manifold AtomicIntMerger port in a, b. atomic {internal.}.
6
7 /*************************************************************/
8 manifold Main
9 {
10  auto process read is ReadFile("unsorted").
11  auto process sort is ProtocolX(AtomicSorter, AtomicIntMerger).
12  auto process print is printunits.
13
14  begin:   read -> sort -> print.
15 }
```

As a concrete demonstration of the reusability of coordinator modules, in the next section, we present an example that uses the coordinator `ProtocolX` in a numerical optimization problem.

6. Domain Decomposition

Consider the following optimization problem:

$$\max z = x^2 + y^2 - 0.5 * \cos(18 * x) - 0.5 * \cos(18 * y) \text{ with (x, y) } \in \text{ [-1.0, 1.0]} \quad (6.1)$$

Figure 3 shows the landscape formed by this function on its domain.

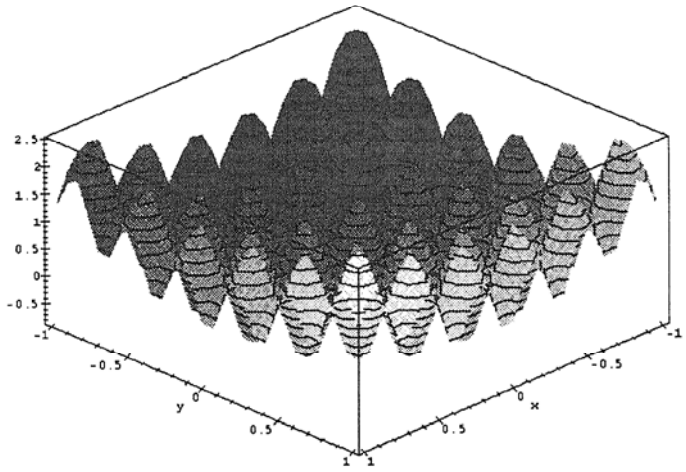

Figure 3: The function $z = x^2 + y^2 - 0.5 * \cos(18 * x) - 0.5 * \cos(18 * y)$

Analytical solutions to such problems are, in general, non-existent and domain decomposition is a common numerical search technique used to solve them. Domain decomposition imposes a grid on the domain of the function, splitting it into a number of sub-domains, as determined by the size of the grid. Next, we obtain a (number of) good rough estimate(s) for the highest value of z in each sub-domain. Then, we select the sub-domains with the most promising z values and decompose them into smaller sub-domains. New estimates for the highest value of z in each of these sub-domains, recursively narrow this search process further and further into smaller and smaller regions that (hopefully) tend towards the area with the real maximum z, while the estimates for the obtained maximum z values become more and more accurate. In single grid domain decomposition, the same grid is imposed on all successive sub-domains. Multi-grid domain decomposition techniques allow a different grid for each sub-domain, whose granularity and other properties may depend on the attributes of the sub-domain and those of the function within that region.

For our example, we consider a single grid method. We need four computation modules for this example: `ap_printobjects`, `Split`, `AtomicEval`, and `AtomicObjMerger`. An instance of `ap_printobjects` simply prints the units it reads from its `input`, each of which describes a (sub-)domain and the x, y, and z values for the estimated maximum z value in that (sub-)domain. An instance of `Split` receives as its parameters the specification of a grid (in our case, 6×6). Next, it reads from its `input` port a unit that describes a (sub-)domain, produces units on its `output` port that describe the sub-domains obtained by imposing the grid on this input domain, and terminates. An instance of *AtomicEval* reads a bucket of $n > 0$ sub-domains (for simplicity, let $n = 1$ once and for all) from its `input` port and raises a specific event, which it receives as a parameter, to inform other processes that it has filled up its input bucket with some sub-domains descriptions. It then finds the best estimate for the optimum z value in each of its sub-domains, producing an ordered sequence of units describing the best solutions it has found through its `output` port, and terminates. In our example, we use sampling: we simply evaluate z for a number of (say 1000) sample points in each sub-domain and consider the sample point

with the maximum z as the best estimate for that sub-domain. An instance of `AtomicObjMerger` reads from its ports `a` and `b` two ordered sequences of units describing sub-domains and their best estimates, and produces a sequence of one or more of its best sub-domains on its `output` port.

We need a MANIFOLD program, say *Eval*, to coordinate the cooperation of the instances of `AtomicEval` and `AtomicObjMerger` to solve our optimization problem in a parallel/distributed fashion. *Eval* receives through its `input` port units describing (sub-)domains. It is supposed to feed as much of its own input units to an atomic evaluator as the latter can take; feeds the rest of its own input as the input to another copy of itself; merge the two output sequences (of the atomic evaluator and its new copy); and produce the resulting sequence through its own `output` port. The similarity between the description of *Eval* and that of `Sort` in §5 suggests that we can use the same coordination module for our optimization problem. Indeed, *Eval* is merely a version of `ProtocolX` with `AtomicEval` and `AtomicObjMerger` as its parameters. The following MANIFOLD program shows a single iteration of our domain decomposition application using the separately compiled `ProtocolX` of §5.

```
 1 manifold ap_printobjects atomic {internal.}.
 2 manifold ProtocolX(manifold M1(event), manifold M2) import.
 3 manifold Split(port in, port in) atomic {internal.}.
 4 manifold AtomicEval(event) atomic {internal.}.
 5 manifold AtomicObjMerger port in a, b. atomic {internal.}.
 6
 7 /*********************************************************/
 8 manifold Main
 9 {
10   auto process split is Split(6, 6).
11   auto process eval is ProtocolX(AtomicEval, AtomicObjMerger).
12   auto process print is ap_printobjects.
13
14   begin:  <<1, -1.0, -1.0, 1.0, 1.0>> -> split -> eval -> print.
15 }
```

The output of this program, below, shows the result produced by 36 instances of `AtomicEval`, each taking in the description of a single sub-domain. The top four lines show the best estimates to be in the neighborhoods of the four corners of the domain for our symmetric function in Figure 3.

```
domain = (-1.000, -1.000) (-0.667, -0.667) point = (-0.883, -0.880), z =  2.541
domain = ( 0.667,  0.667) ( 1.000,  1.000) point = ( 0.889,  0.884), z =  2.540
domain = ( 0.667, -1.000) ( 1.000, -0.667) point = ( 0.881, -0.889), z =  2.539
domain = (-1.000,  0.667) (-0.667,  1.000) point = (-0.878,  0.881), z =  2.539
domain = (-0.667,  0.667) (-0.333,  1.000) point = (-0.528,  0.884), z =  2.048
domain = ( 0.333, -1.000) ( 0.667, -0.667) point = ( 0.533, -0.882), z =  2.048
domain = ( 0.667, -0.667) ( 1.000, -0.333) point = ( 0.885, -0.527), z =  2.048
domain = (-0.667, -1.000) (-0.333, -0.667) point = (-0.534, -0.885), z =  2.047
domain = (-1.000, -0.667) (-0.667, -0.333) point = (-0.881, -0.535), z =  2.047
domain = (-1.000,  0.333) (-0.667,  0.667) point = (-0.883,  0.536), z =  2.046
domain = ( 0.667,  0.333) ( 1.000,  0.667) point = ( 0.877,  0.535), z =  2.043
domain = ( 0.333,  0.667) ( 0.667,  1.000) point = ( 0.537,  0.878), z =  2.043
domain = ( 0.000, -1.000) ( 0.333, -0.667) point = ( 0.177, -0.885), z =  1.802
domain = (-1.000,  0.000) (-0.667,  0.333) point = (-0.885,  0.173), z =  1.801
domain = ( 0.667,  0.000) ( 1.000,  0.333) point = ( 0.881,  0.181), z =  1.800
domain = ( 0.000,  0.667) ( 0.333,  1.000) point = ( 0.171,  0.883), z =  1.799
domain = (-0.333,  0.667) ( 0.000,  1.000) point = (-0.183,  0.884), z =  1.798
domain = (-1.000, -0.333) (-0.667,  0.000) point = (-0.876, -0.175), z =  1.798
domain = (-0.333, -1.000) ( 0.000, -0.667) point = (-0.169, -0.885), z =  1.797
domain = ( 0.667, -0.333) ( 1.000,  0.000) point = ( 0.875, -0.174), z =  1.796
domain = ( 0.333,  0.333) ( 0.667,  0.667) point = ( 0.530,  0.531), z =  1.555
domain = ( 0.333, -0.667) ( 0.667, -0.333) point = ( 0.528, -0.529), z =  1.555
domain = (-0.667,  0.333) (-0.333,  0.667) point = (-0.532,  0.531), z =  1.555
domain = (-0.667, -0.667) (-0.333, -0.333) point = (-0.521, -0.534), z =  1.548
domain = (-0.667, -0.333) (-0.333,  0.000) point = (-0.533, -0.179), z =  1.307
domain = ( 0.333, -0.333) ( 0.667,  0.000) point = ( 0.527, -0.178), z =  1.307
domain = (-0.333, -0.667) ( 0.000, -0.333) point = (-0.172, -0.531), z =  1.307
domain = (-0.667,  0.000) (-0.333,  0.333) point = (-0.532,  0.181), z =  1.307
domain = (-0.333,  0.333) ( 0.000,  0.667) point = (-0.180,  0.534), z =  1.306
domain = ( 0.000, -0.667) ( 0.333, -0.333) point = ( 0.177, -0.524), z =  1.305
domain = ( 0.333,  0.000) ( 0.667,  0.333) point = ( 0.537,  0.176), z =  1.304
domain = ( 0.000,  0.333) ( 0.333,  0.667) point = ( 0.164,  0.528), z =  1.295
```

```
domain = ( 0.000, -0.333) ( 0.333,  0.000) point = ( 0.175, -0.174), z =  1.061
domain = (-0.333, -0.333) ( 0.000,  0.000) point = (-0.179, -0.172), z =  1.059
domain = (-0.333,  0.000) ( 0.000,  0.333) point = (-0.177,  0.183), z =  1.058
domain = ( 0.000,  0.000) ( 0.333,  0.333) point = ( 0.171,  0.182), z =  1.057
```

A straight-forward generalization of this program repeats this single step until a termination criterion (such as a maximum number of iterations, or the diminishing of improvements below a threshold) is reached. Each iteration selects a (few of the) best sub-domain(s) found so far as input to another instance of `Split` and *Eval*. This would be yet another MANIFOLD program that coordinates the cooperation of different instances of *Eval* and `Split`. The following output is produced by such a program using a 2×2 grid. The first line in this output is our initial input unit representing the whole domain. Each succeeding group of four lines then represents one iteration. The best sub-domain found in each iteration is fed as input to the next iteration. The first line of the last group (representing the third iteration) shows the best solution found ($z = 2.542$) which is slightly better than the best solution we found using our single step 6×6 grid ($z = 2.541$).

```
domain = (-1.000, -1.000) ( 1.000,  1.000)

domain = ( 0.000,  0.000) ( 1.000,  1.000) point = ( 0.885,  0.879), z =  2.540
domain = (-1.000, -1.000) ( 0.000,  0.000) point = (-0.884, -0.890), z =  2.539
domain = ( 0.000, -1.000) ( 1.000,  0.000) point = ( 0.890, -0.893), z =  2.532
domain = (-1.000,  0.000) ( 0.000,  1.000) point = (-0.880,  0.911), z =  2.484

domain = ( 0.500,  0.500) ( 1.000,  1.000) point = ( 0.879,  0.892), z =  2.536
domain = ( 0.000,  0.500) ( 0.500,  1.000) point = ( 0.498,  0.866), z =  1.941
domain = ( 0.500,  0.000) ( 1.000,  0.500) point = ( 0.880,  0.490), z =  1.920
domain = ( 0.000,  0.000) ( 0.500,  0.500) point = ( 0.498,  0.499), z =  1.400

domain = ( 0.750,  0.750) ( 1.000,  1.000) point = ( 0.883,  0.883), z =  2.542
domain = ( 0.500,  0.750) ( 0.750,  1.000) point = ( 0.530,  0.883), z =  2.049
domain = ( 0.750,  0.500) ( 1.000,  0.750) point = ( 0.886,  0.529), z =  2.048
domain = ( 0.500,  0.500) ( 0.750,  0.750) point = ( 0.531,  0.530), z =  1.555
```

The highly modular structure of this application is remarkable. Its computation modules (C functions) are simple and have no idea of how they relate to or cooperate with one another. The coordination module *Eval* knows nothing about what these computation modules actually do; it is just as happy coordinating the sorter workers in §5 as it is managing these numerical optimization workers. The various processes comprising this application can run on parallel or distributed platforms without any change to their source code.

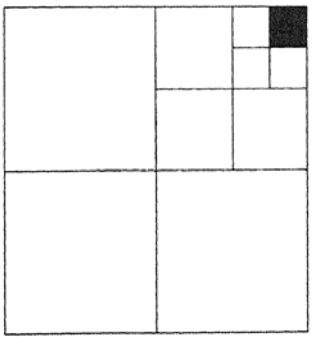

Figure 4: Visualizer snap-shot of 2×2 distributed domain decomposition

The plumbing paradigm of MANIFOLD makes it easy to divert the flows of units, change the coordination structures, and dynamically modify the topology of communication links among (computation

as well as coordination) modules to adapt an application to new requirements. We can plug in graphics modules to display an on-going computation. Indeed, we have a small computational steering environment built around this example, using MANIFOLD coordinators and a few generic graphics interaction modules. The user interface for this program graphically shows the on-going activity of various atomic evaluators (that may be running on different hosts) and allows the user to interactively direct the focus of the attention of the program onto one or more areas of interest, simply by drawing a box to designate a sub-domain. Figure 4 shows a snap-shot of this user interface as our 2×2 distributed domain decomposition optimization application moved on (in its third iteration) to the top-right corner sub-domain in the run that produced the above output.

7. Conclusion

IWIM is a model of communication that supports anonymous communication and separation of computation responsibilities from communication and coordination concerns. MANIFOLD is a coordination language that takes full advantage of these two key concepts of IWIM. Unlike other coordination languages, MANIFOLD encourages decomposition of a parallel and/or distributed application into a hierarchy of pure computation and pure coordination modules, none of which contain hard-coded dependencies on their environment. This leads to highly reusable computation modules, and more interestingly, also to highly reusable coordination modules.

The examples in this paper show a single coordination module used in two very different applications. We have also used MANIFOLD to reorganize existing Fortran 77 sequential code into a parallel and distributed application[6]. The usefulness of the IWIM model and, in particular, the MANIFOLD language in these and other applications has been very encouraging. The plumbing paradigm inherent in IWIM makes it easy to compose and recompose a MANIFOLD application and adapt it to new requirements. To enhance the effectiveness of this coordination language, we are presently developing a visual programming environment around MANIFOLD which takes advantage of its underlying plumbing paradigm.

References

1. D. Gelernter and N. Carriero, "Coordination languages and their significance," *Communication of the ACM*, vol. 35, pp. 97–107, February 1992.
2. F. Arbab, I. Herman, and P. Spilling, "An overview of Manifold and its implementation," *Concurrency: Practice and Experience*, vol. 5, pp. 23–70, February 1993.
3. F. Arbab, "Coordination of massively concurrent activities," Tech. Rep. CS–R9565, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, 1995. Available on-line: `http://www.cwi.nl/ftp/CWIreports/IS/CS-R9565.ps.Z`
4. F. Arbab, "Manifold version 2: Language reference manual," Tech. Rep. in preparation, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, 1995.
5. F. Arbab, "The IWIM model for coordination of concurrent activities," in: P. Ciancarini and C. Hankin (eds.), *Coordination '96*, Lecture Notes in Computer Science #1061, Springer-Verlag, April 1996.
6. C.T.H. Everaars, F. Arbab, and F.J. Burger, "Restructuring Sequential Fortran Code into a Parallel/Distributed Application," submitted to: *International Conference of Software Maintenance '96*, Monterey, California, November 1996.