Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

Coordination of massively concurrent activities

F. Arbab

Computer Science/Department of Interactive Systems

CS-R9565 1995

Report CS-R9565
ISSN 0169-118X

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

Coordination of Massively Concurrent Activities

Farhad Arbab

December 4, 1995

Massively parallel and distributed systems open new horizons for large applications and present new challenges for software technology. Many applications already take advantage of the increased raw computational power provided by such parallel systems to yield significantly shorter turn-around times. However, the availability of so many processors to work on a single application presents a new challenge to software technology: coordination of the cooperation of large numbers of concurrent active entities. Classical views of concurrency in programming languages that are based on extensions of the sequential programming paradigm are ill-suited to meet this challenge.

Exploiting the full potential of massively parallel systems requires programming models that explicitly deal with the concurrency of cooperation among very large numbers of active entities that comprise a single application. In practice, the concurrent applications of today essentially use a set of ad hoc templates to coordinate the cooperation of their active components. This shows the lack of proper coordination languages that can be used to explicitly describe complex cooperation protocols in terms of simple primitives and structuring constructs.

In this paper we present a generic model of communication and describe a specific control-oriented coordination language based on this model. The important characteristics of this model include compositionality, which it inherits from the data-flow model, anonymous communication, and separation of computation concerns from communication concerns. These characteristics lead to clear advantages in large concurrent applications.

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: parallel computing, coordination languages, MIMD, models of communication.

Contents

1 Introduction

Recent advances in computer hardware and networking technologies have dramatically changed the reality on which the visions of computing and information processing applications are based. The ever decreasing costs and sizes of processors, their ever increasing speeds, faster and wider-band-width communication links, and global networks have made the potential of applying the computational power of several (even hundreds and thousands) of processors to a single application, a reality.

Many applications can (and some already do) take advantage of the increased raw computational power provided by such parallel systems to yield significantly shorter turn-around times. The fact that using this computational power, some applications can now produce results in near-real-time by itself leads to a qualitative change in the realm of application possibilities and user expectations. However, conceptually, the significance of the availability of so many (tightly or loosely connected) processors to work on an application goes beyond such "performance" issues. The mere idea of allocating more than one *worker* to the same task immediately opens up a new problem solving paradigm, and simultaneously, presents a new challenge. The paradigm is *concurrency*, and the challenge is *coordination*.

In this paper we consider the problem of coordination of very large numbers of concurrent active entities that must cooperate with each other in the context of a single application. In §2 we give a brief introduction to the work on concurrency. We suggest that massively parallel and distributed systems of today add a new twist to the study and use of concurrency which deserves more attention. In §3 we distinguish between *communication* and *cooperation*, and show the need for a coherent model and language to describe the cooperation protocols of active entities in massively concurrent systems. We suggest that many popular communication models of today are not suitable as the basis for such a language, and propose a data-flow-like model of communication to serve this purpose. A simple example in §4 illustrates the advantages of the proposed model in construction and maintenance of modular, reusable components for concurrent systems. The spread of computer networks and faster communication links makes distributed computing more prominent. This by itself makes asynchronous communication models pragmatically more important for concurrent applications of today than synchronous models. However, §5 shows that, irrespective of distribution, there are other software engineering problems caused by using synchronous communication in large concurrent applications. A specific coordination language, called MANIFOLD, that is based on the generic model proposed in §3 is described in §6. Some of the interesting features of MANIFOLD are shown through examples in this section. A summary of related work appears in §7, and the conclusion of the paper is in §8.

2 Concurrency vs. Parallelism

There is often a confusion about the meaning of the terms "parallelism" and "concurrency". Different people mean different things by these terms and often they take it for granted that their intention is clear to others, and do not bother with an explicit definition of the terminology they use. The fact that, regardless of how they are defined, normally the concepts behind these terms are somewhat related to each other, simply adds to the confusion. In this paper, we define *parallelism* as the application of more than one processor to carry out a solution of a problem. On the other hand, *concurrency* refers to devising a solution to a problem in terms of a set of activities that overlap in time.

Concurrency is about the expression of a computation as a set of concurrent activities. As such, it is a problem solving or a programming paradigm. Parallelism is about throwing more resources to carry out a given computation. As such it is a method for realizing a solution, i.e., to carry out a computation. It follows that, strictly speaking, concurrency and parallelism have nothing to do with each other!

Nevertheless, there is a subtle link between concurrency and parallelism: to take full advantage of one requires the other. A solution to a problem expressed in terms of a set of concurrent activities may be elegant and intuitive. However, without the ability to allocate a sufficient number of processors to its concurrent activities, such an elegant solution cannot live up to its potential and deliver the full performance that it can.

On the other hand, given a specific solution to a problem, it is possible to find opportunities to improve its performance by increasing the number of processors allocated to carry out that solution. This discovery of the parallelism that is inherent in a solution to a problem can theoretically be carried out mechanically

and automatically. But, the limits of such automatic parallelization techniques should be obvious from our above definitions: you can expect to discover parallelism in a solution by mechanical means, but you can never mechanically discover the concurrency that is not there to begin with. Automatic parallelization may yield the best possible performance one can get out of a specific solution to a problem. However, a different solution of the same problem that exploits (more) concurrency can yield still better performance, with or without automatic parallelization.

Of course, the study and the application of concurrency in computer science has a long history. The study of deadlocks, the dining philosophers, and the definition of semaphores and monitors were all well established by the early seventies. However, it is illuminating to note that the original context for the interest in concurrency was somewhat different than today in two respects:

- In the early days of computing, hardware resources were prohibitively expensive and had to be shared among several programs that had nothing to do with each other, except for the fact that they were unlucky enough to have to compete with each other for a share of the same resources. This was the *concurrency of competition*. Today, it is quite feasible to allocate tens, hundreds, and thousands of processors to the same task (if only we could do it right). This is the *concurrency of cooperation*. The distinction is that whereas it is sufficient to keep independent competing entities from trampling on each other over shared resources, cooperating entities also depend on the (partial) results they produce for each other. Proper passing and sharing of these results require more complex protocols, which become even more complex as the number of cooperating entities and the degree of their cooperation increase.

- The falling costs of processor and communication hardware only recently dropped below the threshold where having very large numbers of "active entities" in an application makes sense. Massively parallel systems with thousands of processors are a reality today. Current trends in processor hardware and operating system kernel support for threads[1] make it possible to efficiently have in the order of hundreds of active entities running in a process on each processor. Thus, it is not unrealistic to think that a single application can be composed of hundreds of thousands of active entities. Compared to classical uses of concurrency, this is a jump of several orders of magnitude in numbers, and in our view, represents (the need for) a qualitative change.

It is difficult to deal with concurrency, especially when the number of concurrent activities is large, because we do not have proper models to cope with the resulting complexity. This, at least to some extent, is a chicken-and-egg problem. Theoretical work on concurrency, e.g., CSP[1, 2], CCS[3], process algebra[4], and π-calculus[5, 6], has helped to unveil the essence of the problems. However, the models of concurrency proposed and studied in such theoretical work, understandably, are not always directly useful for practical programming. A number of programming languages have used some of these theoretical models as their bases, e.g., Occam[7] uses CSP and LOTOS[8] uses CCS. Nevertheless, the practical use of massively parallel systems is often limited to applications that fall into one of a few simple concurrency structures.

In fact, although MIMD[2] systems have been available for some time, they are hardly ever used as MIMD systems in an application. The basic problem in using the MIMD paradigm in large applications is coordination: how to ensure proper communications among the hundreds and thousands of different pieces of code that comprise the active entities in a single application. A restriction of the MIMD model, called SPMD[3], introduces a *barrier* mechanism as the only coordination construct. This model simplifies the problem of concurrency by allowing several processors, all executing the same program, but on different data, proceed at their own pace up to a common barrier, where they then synchronize.

There are applications that do not fit in the uniformity offered by the SPMD model and require more flexible coordination. Examples include computations involving large dynamic trees, symbolic computation on parallel machines, and dynamic pipelines. Taking full advantage of the potential offered by massively parallel systems in these and other applications requires massive, non-replicated, concurrency. It is impractical to expect such massive levels of concurrency to be hand-crafted for each application in full detail. The bulk of this concurrency must be automatically generated by programs.

[1] Threads are preemptively-scheduled light-weight processes that run within one operating-system level process and share the same address space.

[2] Multiple Instruction, Multiple Data

[3] Single Program, Multiple Data

There are a number of approaches to taking advantage of parallelism, without exposing (the full extent of) the concurrency involved. Parallelizing and vectorizing compilers are well-known classical examples of this approach. Abstract Data Types, categorical data types[9], skeleton functions[10, 11], Functional Programming, and Logic Programming, each has its own inherent properties for hiding a good deal of concurrency from its users, while making effective use of an underlying parallel system. They generate concurrent programs from higher-level user specifications. These specifications are either so abstract that they do not contain any explicit concurrency, or include equational constraints or templates that ensure significant properties and relationships of interest are preserved under alternative concurrency schemes.

We believe there is a clear need for programming models that explicitly deal with concurrency of cooperation among very large numbers of active entities that comprise a single application. Such models cannot be built as extensions of the sequential programming paradigm. Because such applications can be distributed over a network, we believe such models cannot be based on synchronous models of concurrency.

3 Communication Models vs. Cooperation Models

It is important to distinguish between the conceptual model describing the cooperation of a number of concurrent processes in an application, and the underlying model of communication on top of which such cooperation is implemented. Message passing, shared memory, and data-flow are among the most popular communication models used in concurrent applications. Part of their respective popularity is due to the fact that they also happen to be quite appropriate models of cooperation in some of these applications. However, many other applications require more complex protocols to coordinate the activities of their concurrent processes. For them, client-server, master-slave, farms, worker pools, etc., are more appropriate names for the protocols they need to describe their required cooperation, than simple message passing or shared memory – and here the distinction between cooperation models and communication models manifests itself.

The primary concern in the design of a concurrent application must be its model of cooperation: how the various active entities comprising the application are to cooperate with each other. Eventually, a communication model must be used to realize whatever model of cooperation application designers opt for, and the concerns for performance may indirectly affect their design. Nevertheless, it is important to realize that the conceptual gap between the system supported communication primitives and a concurrent application must often be filled with a non-trivial model of cooperation.

When we consider the models of cooperation used in concurrent applications of today, we note that they are essentially a set of ad hoc templates that have been found to be useful in practice. There is no paradigm wherein we can systematically talk about cooperation of active entities, and wherein we can compose cooperation scenarios such as (and as alternatives to) models like client-server, workers pool, etc., out of a set of primitives and structuring constructs. Consequently, programmers must directly deal with the lower-level communication primitives that comprise the realization of the cooperation model of a concurrent application. Because these primitives are generally scattered throughout the source code of the application and are typically intermixed with non-communication application code, the cooperation model of an application generally never manifests itself in a tangible form – i.e., it is not an identifiable piece of source code that can be designed, developed, debugged, maintained, and reused, in isolation from the rest of the application code.

The inability to deal with the cooperation model of a concurrent application in an explicit form contributes to the difficulty of developing working concurrent applications that contain large numbers of active entities with non-trivial cooperation protocols. In spite of the fact that the implementation of complex protocols are often the most difficult and error prone part of an application development effort, the end result is typically not recognized as a "commodity" in its own right, because the protocols are only implicit in the behavior of the rest of the concurrent software. This makes maintenance and modification of the cooperation protocols of concurrent applications much more difficult than necessary, and their reuse next to impossible.

There are several different flavors to each of the message passing and the shared memory models of communication. Typically, any of these models is capable of emulating all others and thus, in a sense, they are all "equivalent" to one another. In the shared memory model, inter-process communication is only an implicit side-effect of the delay patterns imposed by the synchronization primitives on the

processes that take turn to access and update certain common storage areas. Given the primary concern for communication among a set of cooperating processes, this implicit approach to information exchange is not conducive to explicit coordination. In contrast to the shared memory model, the message passing model uses primitives for explicit information exchange, which implicitly impose the required synchronization on the communicating parties.

Subordinating synchronization to information exchange makes the message passing model somewhat more flexible than the shared memory model and, therefore, it is the dominant model used in concurrent applications. However, both shared memory and message passing are too low-level to serve as a proper foundation for systematic construction of cooperation protocols as explicit, tangible pieces of software. In §3.1, we show some of the shortcomings of a typical message passing model in this context. In §3.2, we present a communication model that avoids these shortcomings and can be used in a paradigm to construct complex cooperation protocols.

3.1 The TSR Model of Communication

A common characteristic of most flavors of the message passing model of communication is the distinction between the roles they assign to the two active entities involved in a communication: the sender and the receiver. A sender s typically sends a message m to a receiver r. The identity of r is either statically known to s or it is dynamically evaluated at execution time. Sometimes, there is more than one receiver, i.e., the message m is multi-cast to a number of receivers, or it may even be broadcast to all active entities running in an application. One way or the other, the send operation is generally targeted to a specific (set of) receiver(s). A receiver r, on the other hand, typically waits to receive a message m from any sender, as it normally has no prior knowledge of the origin of the message(s) it may receive. We use the term *Targeted-Send/Receive*, or TSR, to refer to the communication models that share this characteristic. This encompasses theoretical models such as CSP[1, 2], CCS[3], and the Actor model[12][4]; programming languages such as Occam[7], LOTOS[8], and various flavors of concurrent Object-Oriented languages; and concurrent programming tools such as PVM[13], MPI[14, 15], P4[16], PARMACS[17], etc.

Consider the following simple example of a concurrent application where the two active entities (i.e., processes) p and q must cooperate with each other. The process p at some point produces two values which it must pass on to q. The process q, in turn, must perform some additional computation using the input it receives from p, and then pass on the result of this computation back to p. Note that it is perfectly meaningful to talk about the cooperation model of this application, independent of the actual processes involved, i.e., p and q, or the computation they perform – as a matter of fact, this is exactly what we just did. The source code for this concurrent application looks something 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
```

The first thing to notice in the above listing is that it is simultaneously both a description of what computation is performed by p and q, and a description of how they cooperate with each other. The communication concerns are mixed and interspersed with computation. Thus, in the final source code of the application, there will be no isolated piece of code that can be considered as the realization of its cooperation model, such that, e.g., we can use it as a module in another application where two other processes are to cooperate with each other in a similar fashion.

[4]Note that π-calculus is not mentioned here. In our view, π-calculus is a theoretical model for the specification of communication that, in terms of its emphasis and fundamental concerns, has more in common with the IWIM model presented in §3.2 in this paper, than with the TSR model.

The second significant point to note in the above listing is the asymmetry between send and receive operations. Every send must specify a target for its message, whereas a receive can receive a message from any anonymous source.[5] In our example, p must know q, otherwise, it cannot send a message to it. The proper functioning of p depends on the availability of another process in its environment that (1) must behave as p expects (i.e., be prepared to receive $m1$ and $m2$), and (2) must be accessible to p through the name q. On the other hand, p does not (need to) know the source of the message it receives as m. And this ignorance is a blessing. If after receiving $m1$ and $m2$, q decides that the final result it must send back to p is to be produced by yet another process, x, p need not be bothered by this "delegation" of responsibility from q to x.

We can better appreciate the significance of the asymmetry between send and receive in a tangible form when we compare the processes p and q with each other. The assumptions hard-wired into q about its environment (i.e., availability and accessibility of other processes in the concurrent application) are weaker than those in p. The process q waits to receive a message $m1$ from any source, which it will subsequently refer to as z; expects a second message $m2$ (which it can verify to be from the same source, z, if necessary); computes some result, m; and sends it to z. The behavior of the process p, on the other hand, cannot be described without reference to q. The weaker dependence of q on its environment, as compared with p, makes it a more reusable process that can perform its service for other processes in the same or other applications.

Note, however, that q is not as flexible as we may want it to be: the fact that the result of its computation is sent back to the source of its input messages is something that is hard-wired in its source code, due to its final targeted send. If, perhaps in a different application environment, we decide that the result produced by q is needed by another process, y, instead of the same process, z, that provides it with $m1$ and $m2$, we have no choice but to modify the source code for q. This is a change only to the cooperation model in the application, not a change to the substance of what q does. The unfortunate necessity of modification to the source code of q, in this case, is only a consequence of its targeted send.

3.2 The IWIM Model of Communication

In §3.1 we made two observations about the direct use of the TSR model of communication in concurrent applications. The first observation was that intermixing communication concerns with computation concerns makes the cooperation model of the application implicit in the send and receive primitives that are scattered throughout the source code. The second observation was that targeted send strengthens the dependence of individual processes on their environment. Of course, parameterization or evaluation can be used to avoid hard-coding the names of send targets in the source code. This, however, simply camouflages the dependency on the environment under more computation.

In order to appreciate the combined effect of the above two observations, it is illuminating to look at concurrent applications in general, using an anthropomorphic view. We may regard each process as an individual worker. When the TSR model is used directly, a worker must, naturally, know how to produce the (partial) results expected of him, *and* he also must either (1) know by name the workers he must deliver his results to, or (2) know how to find out the identities of those workers, e.g., know someone who knows them.

This dual concern by each worker leads to a tight coupling of the activities of a team of workers and intermixes production responsibilities of individual workers with the organizational/managerial responsibilities for the cooperation of the team as a whole. Furthermore, how such a team cooperates is only an implicit image induced by the the rigid communication links, the knowledge of which is scattered among its members. This scenario works, and may even be very effective, for small teams of workers. However, the individual attention necessary to hand-craft the composite responsibilities of each worker, makes it very difficult to use this approach in larger teams. The hand-crafted composite responsibilities of each worker also makes it less likely that such a specialized worker can be used, with no change, as a member

[5] In some message passing models, an optional source can be specified in a receive. Although this makes receive look symmetric to send in its appearance, semantically, they are still very different. A send is semantically meaningless without a target. On the other hand, a receive without a source is always meaningful. The function of the optional source specified in a receive is to filter incoming messages based on their sources. This is only a convenience feature – the same effect can also be achieved using an unrestricted receive followed by an explicit filtering.

of another team. Effective use of the resources of hundreds and thousands of workers requires a cleaner separation between organizational and production responsibilities, and a weaker dependence of individual workers on their environment.

It is worth mentioning that there is more to our analogy of processes with workers than a pedagogical anthropomorphic metaphor. Malone and Crowston present a survey of what they call *coordination theory*[18]. They define coordination as "managing dependencies between activities" and characterize its study as an emerging research area with an interdisciplinary focus. They posit that research in this area uses and extends ideas about coordination from disciplines as diverse as computer science, organization theory, operations research, economics, linguistics, and psychology. They expect further progress in this area by extracting the commonalities in the work of the researchers in these various fields on coordination, many of whom are not yet aware of each others' activities.

In the following, we consider an alternative generic model of communication that, unlike the TSR model, supports the separation of responsibilities and encourages a weak dependence of workers on their environment. We refer to this generic model as the Idealized Worker Idealized Manager (IWIM) model. Like the TSR model, the IWIM model is described only in terms of its most significant characteristics. As such, like the TSR model, it indeed defines not a specific model of communication, but a family of such models. Various members in this family can have different significant characteristics, e.g., with regards to synchronous vs. asynchronous communication.

3.2.1 Basic Concepts

The basic concepts in the IWIM model are *processes*, *events*, *ports*, and *channels*. 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. A port is a named opening in the bounding walls of a process through which units of information are exchanged using standard I/O type primitives analogous to read and write. Without loss of generality, we assume that each port is used for the exchange of information in only one direction: either into (input port) or out of (output port) a process. We use the notation $p.i$ to refer to the port i of the process instance p.

The interconnections between the ports of processes are made through channels. A channel connects a (port of a) producer (process) to a (port of a) consumer (process). We write $p.o \rightarrow q.i$ to denote a channel connecting the port o of the producer process p to the port i of the consumer process q.

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 principle, any process in an environment can pick up a broadcast event occurrence. In practice, usually only a few processes pick up occurrences of each event, because only they are *tuned in* to their sources.

Note that although processes, events, ports, and channels are generic basic *concepts* in the IWIM model, they need not correspond to explicit constructs in every incarnation of the IWIM model. The specific incarnation of the IWIM model that is used as the basis for the specific coordination language presented in §6 in this paper, happens to expose each of these concepts as an explicit construct. However, it is perfectly conceivable for a different communication model in the IWIM family to make only an implicit use of some of these concepts (e.g., hide ports within higher-level explicit constructs).

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.

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 (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. Aside from the atomic workers, the categorization of a process as a worker or a manager process is subjective: a manager process m that coordinates the communication among a number of worker processes, may itself be considered as a worker process by another manager process responsible

for coordinating the communication of m with other processes.

3.2.2 Communication Channel

A channel is a communication link that carries a sequence of bits, grouped into (variable length) *units*. A channel represents a reliable, directed, and perhaps buffered, flow of information in time. Reliable means that the bits placed into a channel are guaranteed to flow through without loss, error, or duplication, with their order preserved. Directed means that there are always two identifiable ends in a channel: a *source* and a *sink*. Once a channel is established between a producer process and a consumer process, it operates autonomously and transfers the units from its source to its sink.

If we make no assumptions about the internal operation of the producer and the consumer of a channel C, we must consider the possibility that C may contain some pending units. The *pending units* of a channel C are the units that have already been delivered to C by its producer, but not yet delivered by C to its consumer.

The possibility of the existence of pending units in a channel gives it an identity of its own, independent of its producer and consumer. It makes it meaningful for a channel to remain connected at one of its ends, after it is disconnected from the other.

In general, there are five different alternatives for a channel C in the IWIM model:

1. S channel: In this situation, we have the guarantee that there are never any pending units in C. This implies synchronous communication between the producer and the consumer of C through their respective ports. In this case it is meaningless to talk about a channel without a complete producer-consumer pair.

2. BB channel: In this situation, the channel is disconnected from either of its processes automatically, as soon as it is disconnected from the other.

3. BK channel: In this situation, the channel is disconnected from its producer automatically, as soon as it is disconnected from its consumer, but disconnection from its producer does not disconnect the channel from its consumer.

4. KB channel: In this situation, the channel is disconnected from its consumer automatically, as soon as it is disconnected from its producer, but disconnection from its consumer does not disconnect the channel from its producer.

5. KK channel: In this situation, the channel is not disconnected from either of its processes automatically, if it is disconnected from the other.

The last four types of channels are useful for asynchronous communication. Furthermore, given that the last four types of channels may contain pending units, it is meaningful to reuse a channel of any one of these types for another communication after the breakup of the first.

3.2.3 Primitives for a Worker

There are two means of communication available to a worker process: via its ports, and via events. The communication primitives that allow a process to exchange units through its ports are analogous to the traditional read and write I/O primitives. A process can attempt to read a unit from one of its input ports. It hangs if no unit is presently available through that port, and continues once the unit is made available. Similarly, a process can attempt to write a unit to one of its output ports. Again, it hangs if the port is presently not connected to any channel, and continues once a channel connection is made to accept the unit.

A process p can broadcast an event e to all other processes in its environment by *raising* that event. The identity of the event e together with the identity of the process p comprise the broadcast *event occurrence*. A process can also pick up event occurrences broadcast by other processes and react on them. Certain events are guaranteed to be broadcast in special circumstances; for example, termination of a process instance always raises a special event to indicate its death.

3.2.4 *Primitives for a Manager*

A manager process can create new instances of processes (including itself) and broadcast and react on event occurrences. It can also create and destroy channel (re)connections between various ports of the process instances it knows, including its own. Creation of new process instances, as well as installation and dismantling of communication channels are done dynamically. Specifically, these actions may be prompted by event occurrences it detects.

Each manager process typically controls the communications among a (dynamic) number of process instances in a data-flow like network. The processes themselves are generally unaware of their patterns of communication, which may change in time, by the decisions of a coordinator process.

4 Communication vs. Computation

Let us reconsider the example in §3.1, and see how it can be done in the IWIM model. Our example now consists of three processes: revised p, revised q, and a coordinator process c which is responsible to facilitate their communication. The source code for this version of the application looks something like the following:

```
process p:

compute m1
write m1 to output port o1
compute m2
write m2 to output port o2
do other things
read m from input port i1
do other computation using m
```

```
process q:

read m1 from input port i1
read m2 from input port i2
compute m using m1 and m2
write m to output port o1
```

```
process c:

...
create the channel p.o1 →q.i1
create the channel p.o2 →q.i2
create the channel q.o1 →p.i1
...
```

In this example, the pattern of cooperation between the processes p and q is simple and static. Therefore, the responsibility of the coordinator process c is, indeed, very simple: perhaps, it first creates the processes p and q, establishes the communication channels defined above, and then may wait for the proper condition (e.g., termination of p and/or q) to dismantle these channels and terminate itself. Nevertheless, moving the communication concerns out of p and q and into c already shows some of the advantages of the IWIM model.

The processes p and q are now "ideal" workers. They do not know and do not care where their input comes from, nor where their output goes to. They know nothing about the pattern of cooperation in this application; they can just as easily be incorporated in any other application, and will do their job provided that they receive "the right" input at the right time. The cooperation model of this application is now explicit: it is embedded in the coordinator process c. If we wish to have the output of q delivered to another process, or to have yet another process deliver the input of p, neither p nor q, but only c is to be modified.

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 also knows when additional new process instances are supposed to be created, how the network of communication channels among processes must change in reaction to significant event occurrences, etc. (none of which is actually a concern in this simple example).

It is very likely that such ideal worker processes developed for one application can be used in other concurrent applications, with very different cooperation patterns. Removing the communication concerns out of worker processes enhances the modularity and the re-usability of the resulting software. Furthermore, the fact that such ideal manager processes know nothing about the tasks performed by the workers they coordinate, makes them generic and reusable too. The cooperation protocols for a concurrent application can be developed modularly as a set of coordinator processes. It is likely that some of such ideal managers, individually or collectively, 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 used. Modularity and re-usability of the coordinator processes also enhances the re-usability of the resulting software.

5 Synchronous vs. Asynchronous Communication

The communication between a producer and a consumer of a message can be synchronous or asynchronous. By *synchronous communication* we mean that the producer and the consumer of the message synchronize with each other through a rendezvous mechanism which succeeds only after the message is received by the consumer. The rendezvous point is explicitly identified (e.g., by send and receive primitives) within the source code of both parties. They synchronize with each other in the sense that the first one that reaches its rendezvous point, waits for its mate to perform its matching primitive.

By *asynchronous communication* we mean that the producer of a message does not perform a rendezvous with its consumer. It continues with its processing immediately after the message is safely placed into a *message buffer* for the consumer. When the consumer is ready to pick up a message, it consults this message buffer. It can scan all messages available in the message buffer to pick up the message with the right *message type* and/or from a specific producer. The consumer can use a non-blocking primitive to poll for the availability of a message in the message buffer, or use a blocking primitive to pick it up (in which case, it will suspend until a suitable message arrives in the message buffer, if none is available).

The synchronous and asynchronous models of communication are equally expressive: each can be used to model the other. Synchronous communication models are somewhat simpler to describe: they need fewer (forms of) primitives and also fewer concepts (e.g., there is no need to talk about a message buffer). This simplicity makes synchronous communication models attractive; they are certainly the favorites in theoretical work on concurrency. However, simplicity of concept does not always imply simplicity of use. Using synchronous communication introduces a form of complexity in a system that tends to increase as the number of its components and the degree of their inter-communication increase. To illustrate the problem consider the example of the three processes p, q, and r, as shown below, using a synchronous TSR model of communication:

```
process p:                 process q:      process r:

receive m1                 compute m1      compute m2
receive m2                 send m1 to p    send m2 to p
compute using m1 and m2
```

Note that in this example, q and r are independent processes that have nothing to do with each other. Nevertheless, the order of the two (synchronous) receive primitives in p imposes a certain ordering on the execution of q and r: the send in r cannot complete before the send in q returns. In principle, the ordering of the two receive primitives in p are immaterial and the actual order in which they appear must be an internal detail in p. But, synchronous communication primitives expose this detail to the rest of the system: other processes that communicate with p, and transitively, the ones that communicate with them, cannot remain oblivious to this internal detail inside p. This may not seem very important at first, but if we change our example slightly, its detrimental implications become more clear.

Suppose that q requires another value, $m3$, which is also produced by r. We keep the same p and modify the code for q and r as shown below:

```
process p:                 process q:      process r:

receive m1                 receive m3      compute m2
receive m2                 compute m1      send m2 to p
compute using m1 and m2    send m1 to p    compute m3
                                           send m3 to q
```

Now we have a deadlock: q waits to receive $m3$, r waits to deliver $m2$ to p, and p waits to receive $m1$ before it can accept what r has to offer. But, $m1$ will never be sent by q because $m3$ will never be sent by r. Of course, the fact that deadlocks are possible with synchronous communication is not an

exclusive property of this model: deadlocks are possible with asynchronous communication as well. What is special about this particular form of deadlock is that it does not represent a "real" case of deadlock; it is an artifact caused by the synchronous communication primitives, not by any real circular dependency among the partial results produced by these processes. If, for example, the computation of $m1$ required $m3$, then there would be a circular dependency and a "logical" reason for a deadlock. We could discover and understand the problem by a data dependency analysis. As it stands, the computation of $m1$ does not depend on $m3$ and there is no real reason for a deadlock. This deadlock is a side-effect of the "overkill" of the synchronous communication.

The process r computes two independent values and sends them to two independent processes. The process q receives a value, and independently of that, computes and sends another value to another process. There is no dependency between the two activities in q, nor between the two in r. There is no real reason why the two receives in p should come in any particular order either. The ordering of these activities within each of these processes should be an internal detail that is irrelevant from the outside. But, the synchronous communication primitives break open the modularity of these processes and make their ordering of their activities everyone's business. A system containing these three processes can be properly designed only if we consider the composite effect of this aspect of their internal detail in the system.

This problem can only get worse as the number of entities in the system and the number of their communication primitives increase. The effect is that often unnecessary timing dependencies permeate throughout the whole system through synchronous communication primitives, violating an aspect of the modularity of its components, and make it more difficult to design, debug, modify, and maintain large systems. This is a recognized problem in chip design. Most of the processor chips of today consist of a large number of components that work together synchronously by the beat of the same clock. There is a new trend in chip design that goes back to an asynchronous model with no clock[19]. The revival of asynchronous hardware design is triggered by the increased complexity of contemporary chips and is still somewhat controversial. But, researchers at several commercial laboratories and universities have already produced working chips with asynchronous components. They demonstrate that these chips can be somewhat faster, due to the increased concurrency on the chip. More importantly, asynchronous chips require significantly less power. But, many consider the biggest advantage of asynchronous chips in the long run to be in their ease of design. The asynchronous model allows the design of each module to evolve independently to improve its performance. It also makes it possible to verify the correctness of each module without simultaneously considering every other component in the system. Considering the life cycle of a complex concurrent software system and its components, as opposed to those of a chip, these engineering considerations become even more important in software design than in hardware design.

6 Manifold

In this section, we introduce MANIFOLD: a coordination language for managing complex, dynamically changing interconnections among sets of independent, concurrent, cooperating processes[20, 21]. The conceptual model behind MANIFOLD is based on the IWIM model, described in §3.2. To our knowledge, presently, MANIFOLD is the only language or system based on the IWIM model. Specifically, the MANIFOLD model is a concrete version of the IWIM model where:

1. Each of the basic concepts of process, event, port, and channel in IWIM corresponds to an explicit language construct.

2. All communication is asynchronous. Thus, there is no synchronous communication channel (type S in §3.2.2), and raising and reacting to events do not synchronize the processes involved.

3. The separation of computation and communication concerns, i.e., the distinction between workers and managers, is more strongly enforced.

The MANIFOLD system consists of a compiler, a run-time system library, a number of utility programs, and libraries of builtin and predefined processes of general interest. A MANIFOLD application consists of a (potentially very large) number of 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 and 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. Some of the processes will run as independent operating-system-level processes, and some will run together as light-weight processes (preemptively scheduled threads) inside an operating-system-level process. None of this detail is relevant at the level of the MANIFOLD source code, and the programmer need not know anything about the eventual configuration of his or her application in order to write a MANIFOLD program. The utility programs in the MANIFOLD system work on the object files produced by the (MANIFOLD and other language) compilers and take care of the proper composition of the executable files in an application; and of their mapping onto the proper actual hosts at run time (see §6.9).

The library routines that comprise the interface between MANIFOLD and processes written in other languages (e.g., C), automatically take care of data format conversions, only when such conversions are necessary. This means that in a heterogeneous environment, as a data item passes through several hosts with different data formats it is *not* converted every time a boundary is crossed. The conversion is done at most once, only if the data format of the producer of the item differs from that of its final consumer. All such conversions are done transparently, and the programmer need not be concerned with them[6].

MANIFOLD is a strongly-typed, block-structured, declarative, event driven language. The primary entities created and manipulated in a MANIFOLD program are processes, ports, events, and streams. The MANIFOLD system supports separate compilation. A MANIFOLD source file constitutes a program *module*, encapsulating all that is declared locally within its scope. A module can access entities defined in other modules by `import`ing a definition that is `export`ed by them, or share the ones declared as `extern`.

The MANIFOLD system runs on multiple platforms. Presently, it runs on IBM RS6000, IBM SP1/2, HP, SUNOS, Solaris, and SGI IRIX. Linux and Cray Unicos ports are under way. The system was developed with emphasis on portability and support for heterogeneity of the execution environment. It can be ported with little or no effort to any platform that supports a thread facility functionally equivalent to a small subset of the Posix threads, plus an inter-process communication facility roughly equivalent to a small subset of PVM.

6.1 Events

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. 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 that process at its own leisure and according to its own sense of priorities. The event memory of a process behaves as a set: there can be at most one copy of the occurrence of the same event raised by the same source in the event memory. If an event source repeatedly raises an event faster than an observer reacts on that event occurrence, the event memory of the observer induces an automatic sampling effect: the observer detects only one such event occurrence.

6.2 Streams

The asynchronous communication channels in MANIFOLD are called *streams*. A stream has an infinite capacity that is used as a FIFO queue, enabling asynchronous production and consumption of units by the processes connected to the stream as its source and sink. When the sink process of a stream requires a unit, it is suspended only if no units are available in the stream. The suspended sink process resumes as soon as the next unit becomes available for its consumption. The source process of a stream is never suspended because the infinite buffer capacity of a stream is never filled.

There are four primary types of streams in MANIFOLD, corresponding to the BB, BK, KB, and KK type channels in §3.2.2. Only KB and BK type streams can be reconnectable in MANIFOLD: once such a stream is disconnected from the process on its B-side, it can be reconnected to another source/sink on its dangling side.

Note that as in the IWIM model, the constructor of a stream between two processes is, in general,

[6]Except that certain decisions on the part of the programmer regarding the composition of the application and the options used on the producer side can affect the performance of a distributed application.

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. The flows of units of information in streams are automatically replicated and merged at outgoing and incoming port junctions, as necessary. Thus, a unit placed into a port that is connected to more than one outgoing streams is automatically duplicated, with a separate copy placed into each outgoing stream. Analogously, when a process attempts to fetch a unit from a port that is connected to several incoming streams, it obtains the first unit available in a non-empty incoming stream, selected non-deterministically.

6.3 Processes

The atomic workers of IWIM are called atomic processes in MANIFOLD. 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 program running as an atomic process, to support a richer interface between the atomic process and the MANIFOLD world. An atomic process that takes advantage of this interface library is called a *compliant* atomic process.

Strong separation of computation from communication concerns in MANIFOLD is achieved by not (directly) providing the usual computational capabilities of other programming languages in the MANIFOLD language. Furthermore, the interface library for the compliant atomic processes does not provide any means for an atomic process to perform coordination functions (i.e., the library does not contain functions that correspond to the primitives described in §3.2.4). Thus, atomic processes (compliant or not) can only produce and consume units through their ports, raise and receive events, and compute.

Manager/coordinator processes are written in the MANIFOLD language and are called manifolds. A manifold definition (i.e., the definition of a coordinator process in this language) consists of a header and a body. The header of a manifold (or atomic process definition) gives its name, the number and types of its parameters, and the names of its input and output ports. Parameters of a manifold can be processes, events, and 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, in terms of a conjunction of patterns that can match with 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.

6.4 Manners

A *manner* is a parameterized subprogram that can be called by manifolds. 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. Invocation of a manner never creates a new process: the processor executing the manner call enters the manner.

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 implementation of 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.

Parameters to a manner can be processes, events, ports, and other manners. Manners embody coordination sub-protocols that can be used in various places within a MANIFOLD application. Thus, general and application-specific higher-level abstract coordination protocols can be built out of the basic primitives in MANIFOLD as manners and stored in libraries.

6.5 State Transitions

The actions in a simple state body essentially correspond to the primitives described in §3.2.4: create and activate instances of (atomic and/or manifold) processes, raise (broadcast) events, post events (place an occurrence of the event in the event memory of the running manifold instance only), and construct and/or (re)connect streams between ports of various process instances. Upon transition to a state, the actions specified in its body are performed in a non-deterministic order. Conceptually, this is an atomic

action and takes no time. Then, the state becomes *preemptable*: i.e., if the conditions for transition to another state are satisfied, the current state is preempted and a transition to a new state takes place. Preemption of a state preempts all streams constructed and/or (re)connected in that state.

Preemption of a stream breaks off the connection between the stream and its source and/or sink process at the B-end(s) of the stream. Thus, preempting a BB-type stream breaks the connections at both ends of the stream; preempting a BK-type stream breaks its connection with its source; preempting a KB-type stream breaks its connection with its sink; and a KK-type stream is not affected by preemption.

The event-driven state transition mechanism described above is the only control mechanism in the MANIFOLD language. 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 a convenience.

6.6 Values

In a MANIFOLD application, values are produced and consumed by processes and flow through streams. There are only four types of values in MANIFOLD: *process references*, *port references*, *event references*, and *bit strings*. Event occurrences themselves are *not* considered as values in MANIFOLD, because they do not flow through streams. Process, port, and event references have implementation dependent fixed formats. They are internal identifiers for specific events, ports, and process instances in a running MANIFOLD application. Special facilities are provided in the MANIFOLD language to produce and dereference event, process, and port reference values.

All "user data" produced and consumed by various atomic processes in a MANIFOLD application (e.g., integers, floating point numbers, arrays, structures, etc.) are regarded as bit strings. MANIFOLD itself imposes no interpretation, nor any size and format restrictions, on what it considers to be a bit string. However, the atomic processes and atomic manners (including some built-in ones) that are the ultimate producers and consumers of the values in a MANIFOLD application sometimes require certain bit strings that represent specific values in the data format of their underlying platform. In order to make it more convenient to use such atomic manners and processes, the MANIFOLD language supports special syntax to refer to specific processes that produce certain implementation dependent bit strings; these are called *constants*.

A constant in MANIFOLD is a process instance that is activated at the start up of a MANIFOLD application. Just as any other process, a constant also has the three standard ports `input`, `output`, and `error`. Constants never read anything from their `input` ports and never write anything to their `error` ports. Every time a constant detects a new stream connection on its `output` port, it produces a new unit containing a specific bit string on its `output`.

The exact contents of this bit string depends on the name of the constant (which also implies its type), as well as the data format of the hardware platform wherein the constant runs. For instance, `26`, `6.4`, `12e-6`, and `"I am a character string constant"`, are examples of MANIFOLD constants that produce the appropriate (platform-dependent) bit strings which represent an integer, two floating point numbers, and a character string.

6.7 Coordination in Manifold

Gelernter and Carriero elaborate the distinction between *computational models and languages* as against *coordination models and languages*[22]. 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 in computing. MANIFOLD is a language whose sole purpose is to manage complex, dynamically-changing interconnections among independent concurrent processes. As such like Linda[23, 24, 25], 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.

Figure 1 shows an abstract representation of a process instance in MANIFOLD. The crux of the coordination paradigm in MANIFOLD is to dynamically orchestrate the communications among sets of such processes *from the outside*, by *third party* specialized coordinator processes that are written in the MANIFOLD language. Furthermore, the coordinator processes must do their job with no knowledge of

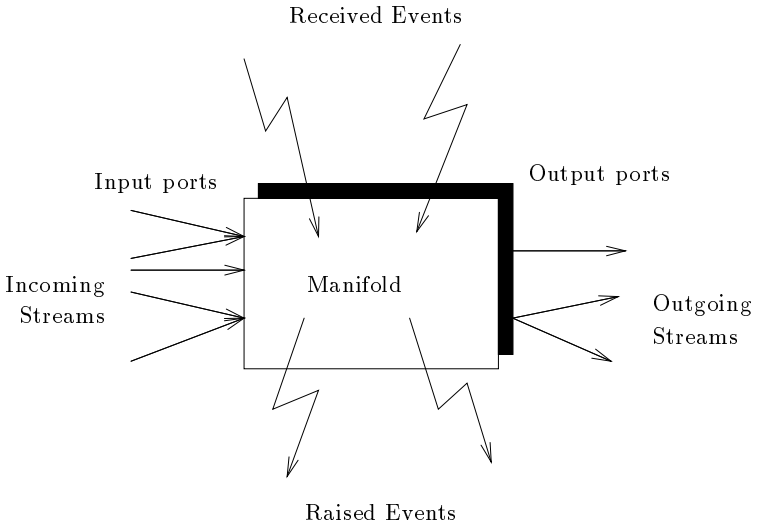

Figure 1: The model of a process in Manifold

the internal details of the computations carried out by the processes whose cooperation they coordinate. The only information available to a coordinator process is the external behavior of the processes it coordinates, which contains the specification of their input and output sequences and the relative timing of the events they raise and/or expect. Coordinator processes not only can react to events of interest from other processes, they can also initiate actions on their own. Thus, in contrast to *reactive* coordination systems, (e.g., through the enforcement of constraints), coordination in MANIFOLD is *pro-active*.

We regard the clear separation (indeed, isolation) of computation and communication in MANIFOLD and its control-oriented approach to coordination of cooperation, essential and novel. The dynamic data-flow like network inherent in the IWIM model is an important component of MANIFOLD's control-oriented approach to concurrency. The compositional property of data-flow networks makes it possible to decompose the coordination of a complex, dynamic application into separate, simpler sub-coordination problems, and then combine them together. This is a significant issue in building massively concurrent applications.

The particular style of event-driven programming with state transitions advocated by the MANIFOLD language, although effective, is less essential. It is perfectly possible to devise alternative languages based on the IWIM model to describe coordinator processes that embody control-oriented protocols for the management of the cooperation among sets of concurrent processes. Indeed, we are presently developing and experimenting with one such alternative language based on the same principles as MANIFOLD, using a visual programming paradigm.

6.8 Examples

It is beyond the scope of this paper to present the details of the syntax and semantics of the MANIFOLD language. However, because MANIFOLD is not very similar to any other well-known language, in order to appreciate the applicability of its underlying concepts to practical concurrent programming, it is essential to grasp the utility and the expressibility of some of its basic constructs. In this section we present a number of examples to illustrate the features and the capabilities of the MANIFOLD language and its underlying model.

6.8.1 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:

```
manifold PrintUnits() import.
```

```
auto process print is PrintUnits.


manifold Main()
{
  begin: "Hello World!"  →  print.
}
```

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: a constant in MANIFOLD is a special process instance that produces its value as a unit on its `output` port and then dies[7].

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 2 shows the relationship between the constant and `print`, as established by `main`. Note also that the coordinator process `main` only establishes

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

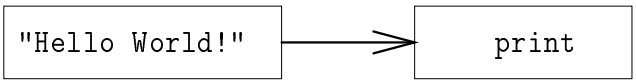

Figure 2: The "Hello World" exmple in Manifold

the connection between the two worker processes. It does *not* transfer the units through the stream(s) it creates, nor does it interfere with the activities of the worker processes in other ways.

6.8.2 Variables

We saw in §6.8.1 that constants are processes in MANIFOLD. In fact, MANIFOLD knows only processes; there are no data structures in MANIFOLD, not even the simplest kind, a variable. What corresponds to a variable in MANIFOLD is also a process. The fact that this process has a special predefined behavior is irrelevant – MANIFOLD treats it the same way as it does any other process. In this section we define the behavior of variable processes and use them to illustrate some of the features of the MANIFOLD language.

An instance of the manifold `variable()` reads the units it receives on its `input` port. Each time, it remembers the unit it reads, and if the departure side of its `output` is connected, it passes the unit on through its `output` port. A remembered unit is repeatedly copied to `output` every time the departure side of `output` is connected to a stream. Instances of `variable` never terminate, and they never raise any events.

The manifold `variable(port in` *i*`)` provides initialized variables. An instance of this manifold behaves as an instance of `variable()` does, except that it first obtains a single unit from its parameter *i*, and uses it as its initial value.

We see here that MANIFOLD allows functors, e.g., `variable`, to be overloaded. A manifold or manner definition is fully identified not by its functor name alone, but by a combination of the its functor name and its parameter types. Thus, `variable()` and `variable(port in)` designate different manifolds. An instance of the manifold `variable(port in)` can be created, for example, in a construct such as `process q is variable(2.3).` In this example, 2.3 is a constant, which is a process, and its `output` port is passed as the actual parameter to a newly created instance of `variable(port in)`, named `q`. As far as `q` is concerned, its parameter is a port from which it can read: i.e., an `in`-port. By its definition, `q` will use the unit it obtains from its parameter as its initial value. Similarly, a construct like `process q is variable(proc.out1)` causes `q` to use a unit produced by the process `proc` on its output port `out1` as its initial value.

Now consider the MANIFOLD program below. It contains a new construct, called a group, in the `begin` state of the manifold `Main`. All streams defined in a group are constructed simultaneously. Thus, in our example, once `main` enters its `begin` state, three BK-type (by default) streams are constructed among the three process instances `v`, `21`, and `print`. Figure 3.a shows these connections.

```
manifold PrintUnits() import.
manifold variable() import.

auto process v is variable.
auto process print is PrintUnits.

manifold Main()
{
  begin: (21 → v, v → v, v → print).
}
```

The constant `21` dies immediately after it delivers its value into the stream connecting its `output` port to the `input` port of `v`. This stream therefore breaks at its source. Note that at this point, the stream connecting the `output` port of `v` to the `input` port of `v` contains no units. Thus, when `v` attempts to read from its `input` port, it is bound to read the unit produced by `21`. As soon as `v` reads in this unit, the stream that contained it becomes empty, and because it is also disconnected from its source, it dies and disappears. Figure 3.b shows the remaining connections at this point.

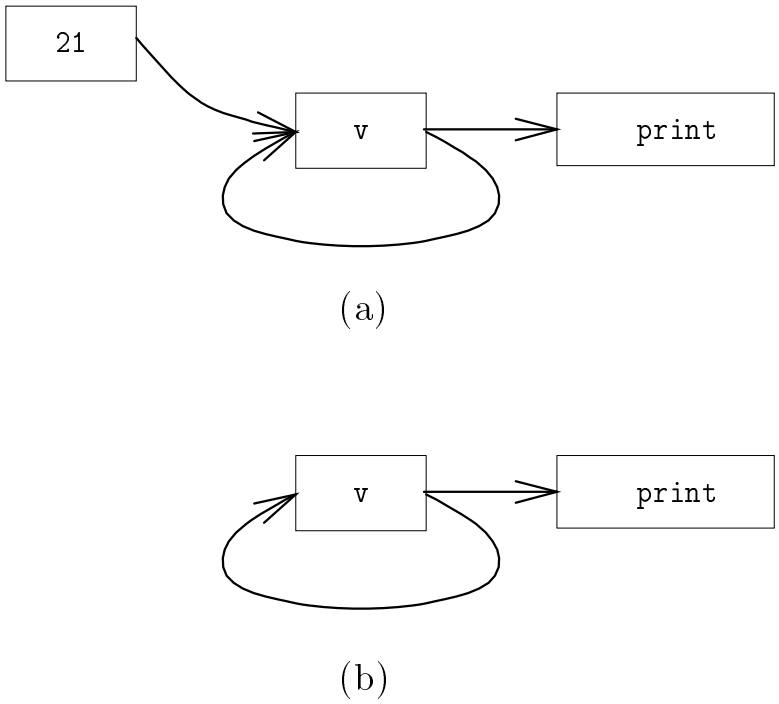

Figure 3: An infinite loop using a variable

The definition of v states that it will produce a copy of the unit it has read on its `output` port. As soon as this happens, the unit is duplicated and a copy of it is placed into each of the two outgoing streams connected to the `output` port of v. One of these will end up in the process `print`, which will print it. The other one finds its way back at the `input` port of v. Once again, v reads in this unit and copies it to its `output` port, and the same sequence is repeated indefinitely.

This example shows the "plumbing" aspect of MANIFOLD programming: no explicit action is necessary to "move" information around in MANIFOLD – provide the pipes, and the units will flow. The process `main` in this example will never terminate. A unit containing the value `21` loops indefinitely from the `output` port of v to its `input` port, and each time a copy of it is printed by `print`. Although there is no explicit loop construct in this program, it loops for ever and produces an output that consists of the value `21` printed out an infinite number of times.

The following MANIFOLD program uses the initialized variable to yield the same behavior.

```
manifold PrintUnits() import.
manifold variable(port in) import.

auto process v is variable(21).
auto process print is PrintUnits.

manifold Main()
{
  begin: (v → v, v → print).
}
```

6.8.3 Fibonacci Series

The looping effect explained in the examples in §6.8.2 can be used to construct useful programs. In the example shown below, we use the initialized variable of §6.8.2 and a new atomic process to compute the Fibonacci series. There are two new linguistic element in this program. The first new element is in the

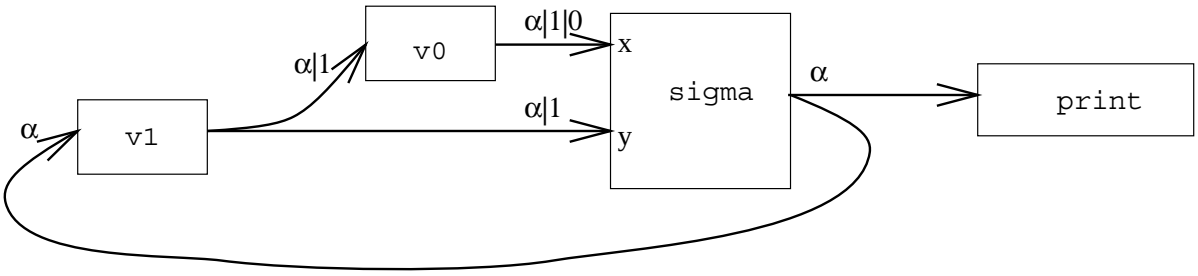

Figure 4: The Fibonacci series

declaration of the manifold `sum`. In addition to the default ports that all manifolds have, this manifold has two input ports named `x` and `y`. The interface declaration of `sum`, thus, contains the declarations for these ports. The real body of `sum` is contained in another source file. The second new element is the declaration of `overflow` as an event.

```
manifold PrintUnits() import.
manifold variable(port in) import.
manifold sum(event)
  port in x.
  port in y.
  import.
event overflow.

auto process v0 is variable(0).
auto process v1 is variable(1).
auto process print is PrintUnits.
auto process sigma is sum(overflow).

manifold Main()
{
  begin: (v0 → sigma.x, v1 → sigma.y, v1 → v0, sigma → v1, sigma → print).
  overflow.sigma: halt.
}
```

The manifold `sum` can be defined in the MANIFOLD language, or as an atomic process whose body is a C function. Either way, an instance of `sum` reads a pair of units, one from each of its input ports `x` and `y`, verifies that they contain numeric values, adds them together, and produces the result in a unit on its `output` port. It then tries to obtain a new pair of input units to produce their sum, and continues to do so indefinitely, as long as its input ports are still connected to incoming streams. If a pair of input values are so large that their addition causes an overflow, `sum` produces a special error unit on its `output` and raises the event it receives as its actual parameter. In our case, the process `sigma`, an instance of `sum` defined a few lines later, specifies this event as `overflow`.

Figure 4 shows the connections made among various processes in the `begin` state of `main`. In order to understand how this set of connections produces the Fibonacci series, we consider the sequence of units that flow through each stream. Let α be the sequence of units produced through the `output` port of the process `sigma`. Clearly, this is the sequence of units printed by `print`, and we want to show that it is indeed the Fibonacci series.

The sequence of units that show up at the `input` port of `v1` is, obviously, α. This means that the sequence of units produced through the `output` port of `v1` consists of 1 (the initial value of `v1`) followed by α. This same sequence shows up at the `input` port of `v0` and at the port `y` of `sigma`. It follows that the sequence of units produced through the `output` port of `v0` (which shows up at the `x` port of `sigma`) consists of 0 (the initial value of `v0`), followed by 1, followed by α.

Now, observe that the first pair of units that arrive at the ports `x` and `y` of `sigma` contain, respectively, 0 and 1. Thus, by the definition of `sum` (of which `sigma` is an instance), the first unit in α contains $0+1$,

i.e., 1. Therefore, the second pair of units that arrive at the ports `x` and `y` of `sigma` contain, respectively, 1 and 1 (the first unit in α). Hence, the second unit in α contains $1 + 1$, i.e., 2. The third pair of units that arrive at the ports `x` and `y` of `sigma` contain, respectively, 1 (the first unit in α) and 2 (the second unit in α), which produces a 3 for the third unit in α.

This configuration of processes will continue to produce the Fibonacci numbers 1, 2, 3, 5, 8, 13, 21, 34, 55, etc., until `sigma` encounters an overflow. In reaction to the occurrence of the event `overflow` raised by `sigma`, `main` makes a transition to its corresponding state. The transition out of the `begin` state preempts (i.e., breaks up) the streams connected therein. Both `sigma` and `print` terminate as soon as they detect they have no incoming streams. The transition into the new state executes the action `halt`, which terminates the `main` process.

6.8.4 Bucket Sort

The examples in the previous sections were simple enough to require only an essentially 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.

```
manifold PrintUnits() import.
manifold ReadFile(port in) import.
manifold Sorter import.

manifold Main()
{
  auto process read is ReadFile("UnsortedFile").
  auto process sort is Sorter.
  auto process print is PrintUnits.

  begin: read → sort → print.
}
```

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 `UnsortedFile` 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.

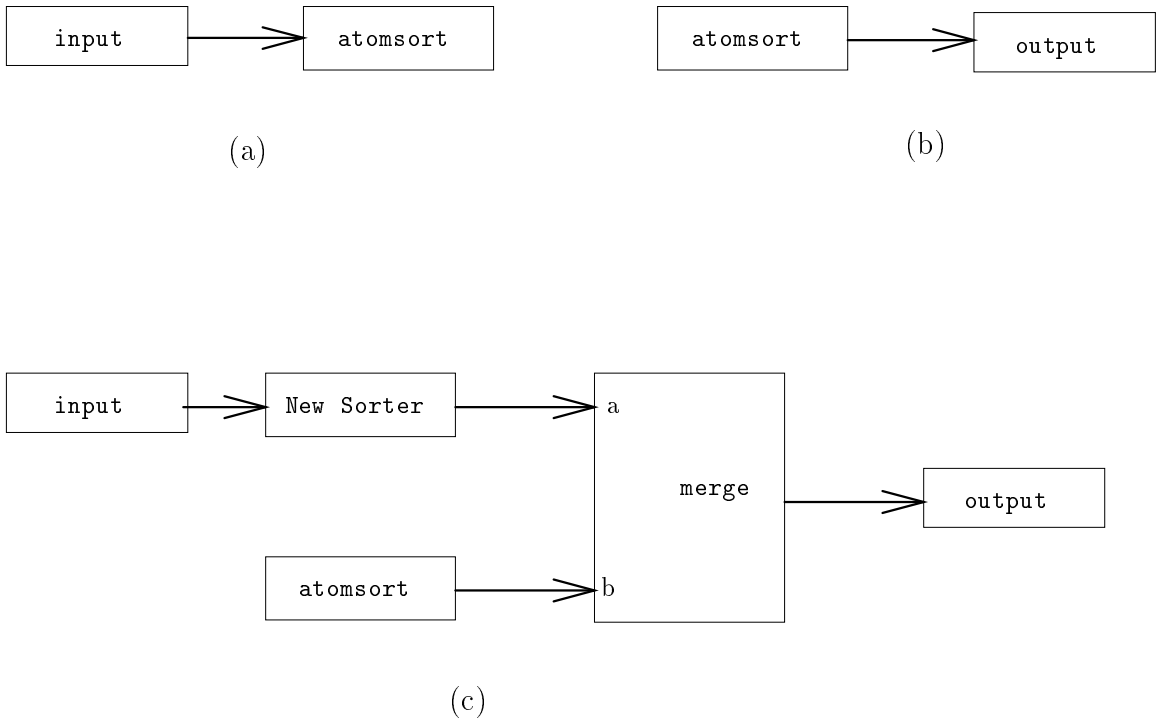

Figure 5: Bucket sort

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 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 5.a.

```
manifold AtomicSorter(event) import.
manifold Merger() import.

manifold Sorter()
{
  event filled, flushed, finished.
  process atomsort is AtomicSorter(filled).
  stream reconnect KB input → *.
  priority filled < finished.

  begin: (activate(atomsort),
         input → atomsort,
         guard(input, a_everdisconnected!empty, finished),
         guard(output, a_everdisconnected, flushed)).

  finished: {
     ignore filled.  // possible event from atomsort

     begin: atomsort → output.  // your output is only that of atomsort
   }.
```

```
  filled: {
      process merge<a,b|output> is Merger.
      stream KK * → (merge.a, merge.b).
      stream KK merge → output.

      begin: (activate(merge),
            input → Sorter → merge.a,
            atomsort → merge.b,
            merge → output).

      finished:. // do nothing and leave this block
    }.

  end: {
      begin: terminated(void).  // wait for units to flush through output
      flushed: halt.
    }.
}
```

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 5.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 5.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 Figures 5.b and 5.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 `Merger` 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 `Merger` 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 `Merger`. `AtomicSorter` need not really sort its input units, `Merger` need 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.

As a concrete example of this notion of reusable coordinator modules, it is worth mentioning that one of our colleagues is using the exact same sort program described above as the coordination module for the parallel/distributed version of a dynamic domain decompostion algorithm. The original version of this algorithm is part of a package of numerical algorithms under development by another group at CWI[26]. No change to the coordination scheme described above is necessary to handle the single-grid version of the dynamic domain decomposition algorithm. Only a small change is necessary to the coordination modules to handle the multi-grid version of this algorithm.

6.9 Execution of Manifold Applications

A MANIFOLD application consists of one or more executable files, called *tasks*, each of which can run on a number of hosts, with the same or different hardware/software architectures. Conceptually, none of this information is relevant to the programmer at the level of the MANIFOLD language: all that a programmer sees are manifold definitions and process instances. However, the placement of the executable code for these manifold definitions in executable files (tasks) and the hosts on which these tasks can run, are details that must be specified at link and execution time, before a MANIFOLD application can run.

The MANIFOLD system includes a utility program called the MANIFOLD linker. The MANIFOLD linker expects an input file describing the composition of the executable files of an application and produces a set of C programs that contain the necessary link information for the tasks of the application, plus a set of specifications for how the final executable files should be linked. Each such executable file is created on a host with proper architecture by linking together the object files resulting from the output of the MANIFOLD compiler; the object files resulting from compiling the additional user C programs such as atomic processes and atomic manners; the object files resulting from the link programs produced by the MANIFOLD linker; and the proper libraries.

The execution of a MANIFOLD application starts by running any one of the executable files of the application on an appropriate host. This creates an instance of the task and passes it the command line parameters used in its invocation. It is immaterial (except, perhaps, for performance reasons) which one of the tasks that constitute a MANIFOLD application is invoked first to run the application. The part of the MANIFOLD run-time system and the link information produced by the MANIFOLD linker that are incorporated in all tasks comprising a single application, ensure that the same logical sequence of actions described below will take place.

Once running, the first invoked task instance in the application searches to locate a run-time configuration file for the application. (Normally, this configuration file is in the same directory where the executable file for the invoked task resides.) The contents of this configuration file tell the MANIFOLD run-time system what hosts are to be used for this run of the application, and which tasks can run on which of these hosts.

Next, the running task instance locates a task that contains the `Main` manifold of the application

(§6.9.2). There may be more than one task that contains the `Main` manifold, in which case one is selected, according to some implementation dependent criteria. The candidate task may or may not be the same as the one that the running task instance is an instance of. An instance of this candidate task is started up on an appropriate host, and the first instance of the `Main` manifold of the application, known as the process `main`, is created in this task instance. Finally, the command line parameters used to invoke the application are passed on to `main`, and it is activated.

There is nothing inherently special about the first task instance used to start up a MANIFOLD application, nor about `main`: both of them can terminate as soon as they are no longer needed, and the MANIFOLD application can still go on unaffected. Termination of the `Main` manifold, the `main` process, or the task instance that contains it, does not force the termination of the application. Thus, there is no need in MANIFOLD to keep any process alive beyond the point that is required by the logic of the program.

Other instances of manifolds and atomic processes are dynamically created as required during the life time of the application, and instances of tasks to contain them are also created as necessary on their appropriate hosts (see §6.9.1). A MANIFOLD application terminates when all processes created therein, except constants and a number of predefined special system processes have terminated.

6.9.1 Task Instances

The code for a manifold definition can be compiled on one or more platforms. The resulting object code can be combined with the object code from zero or more other manifold definitions into any number of executable files for one or more architectures. The input file to the MANIFOLD linker can specify a weight for each manifold and each atomic process, and can also specify a maximum load for each task in the application. This information is used by the MANIFOLD run-time system to decide when and where a new instance of a process is to be created.

Any time during the execution of an application when a new process instance must be created, the MANIFOLD run-time system first finds an appropriate task instance to contain it. It tries to avoid creating a new task instance, if the new process instance can be housed in one of the existing task instances. However, a new task instance may have to be created if either the executable code for the process instance is not included in any of the tasks whose instances are currently running, or the sum of the weights of the process instances running in a task instance that could otherwise house the new process instance exceeds its specified maximum load.

If a new task instance must be created to house the new process instance, the MANIFOLD run-time system selects one of the tasks that contain the executable code for this process, selects a host that can run this task, and starts an instance of this task on the selected host. A task instance dies when all process instances it houses are terminated.

6.9.2 The Main Manifold

Manifold definitions with the name `Main()` and `Main(process)` are special. The MANIFOLD linker verifies that there is at least one `Main()` or `Main(process)` manifold defined in every MANIFOLD application. It prefers a `Main(process)` manifold definition over a `Main()`, if both exist in an application. The `Main` manifold selected by the MANIFOLD linker becomes the designated `Main` manifold for the application.

At the start-up of the application, the MANIFOLD run-time system creates and activates an instance of its designated `Main` manifold as its first process instance. This process instance is the external process `main`. An actual parameter is passed to this process instance only if the designated `Main` expects it (i.e., its signature is `Main(process)`). Analogous to a MANIFOLD constant, the actual parameter supplied by the run-time system is a MANIFOLD process that produces a unit containing the same tuple every time a connection is made to its output port. This tuple contains all the command line parameters used to invoke the running MANIFOLD application.

7 Related Work

As mentioned earlier, the survey by Malone and Crowston[18] characterizes coordination as an emerging research area with an interdisciplinary focus. They observe that coordination has been and is a key

issue in many diverse disciplines other than computer science. Although tackling coordination problems in operating systems, parallel programming, and databases (to name but a few) has a long history in computer science, the notion of coordination as a research area and coordination languages as a serious topic are rather recent developments. Nevertheless, a number of models and systems have already appeared for coordination. Many of them deal with some limited aspect of coordination, or coordination in a specific and somewhat limited context.

HOP is a model for describing object composition patterns [27]. The main concepts in HOP are objects and ports. A HOP object is closer to the concept of an object in an object oriented language such as Smalltalk or C++, than to a process in IWIM or MANIFOLD. HOP objects are different than objects in typical object oriented languages primarily because they have ports. The underlying rationale for the concept of port in HOP is very similar to that of ports in IWIM and MANIFOLD: disallow access to foreign entities except through clearly designated boundary points (i.e., ports). This enhances reusability and leads to locality of reasoning: the ability to understand how an entity works by considering it locally, without its whole context. In HOP, an object that has no port bound to something outside of itself is called a value. Values (e.g., integer constants) can easily be replicated, such that other objects that use them can have and bind to their own local copies. HOP objects can directly bind only within their local context; bindings to remote objects must be done through ports located at object boundaries.

HOP views objects as having characteristics similar to records: they have named entries akin to record fields that, analogous to object methods or public instance variables, are accessed individually. HOP also views objects as having characteristics similar to functions: they have common templates (e.g., function body or object class) that are bound to parameters upon activation or instantiation. Based on these views, HOP provides a single composition construct that combines function application and field access. Some of the common composition techniques used in object oriented systems can be modeled using this single composition construct: e.g., message passing, recursion, classes, inheritance, etc.

To the extent that inter-object dependencies represent communication, IWIM (and MANIFOLD) share some of the same underlying observations and concerns with HOP (e.g., reusability, locality of reasoning, graphical representation). However, the purpose of HOP is to directly model dependencies among objects and their composition. Although message passing can be *modeled* in HOP as a dependency pattern through an intermediary object, it is not the purpose of HOP to explicitly model and manage inter-object communication per se. Thus, we do not consider HOP as a coordination model or language, in the sense of IWIM or MANIFOLD.

Language support for the expression of certain kinds of multi-object coordination is presented in[28]. The main construct offered is a synchronizer, which is to be integrated into an object oriented concurrent language that adheres to the Actor model[12] of computation. Coordination patterns can be expressed as constraints that restrict invocation of a group of objects. These constraints are defined in terms of the interface of the objects being invoked, rather than their internal representation. Such invocation constraints enforce properties, such as temporal ordering and atomicity, that must hold when objects are invoked in a group. Through invocation constraints, coordination patterns can be specified abstractly, independent of the protocols needed to implement them.

Enforcement of constraints is done by synchronizers, which are special objects that observe and limit the invocations accepted by a set of ordinary objects which are being constrained. This is somewhat similar to the idea of workers and managers in the IWIM model. As with managers in IWIM and manifolds in MANIFOLD, synchronizers can overlap: multiple separately specified synchronizers can constrain the same objects. Synchronizers themselves are not "real objects" in the sense that it is not possible to, recursively, constrain their behavior using other synchronizers, and ordinary objects cannot send messages to them as they do to other ordinary objects. In IWIM (and MANIFOLD) a manager (manifold instance) is externally indistinguishable from an ordinary worker (atomic process).

Synchronizers enforce constraints that can express certain abstract high-level coordination concerns only. As the authors emphasize, the concept of synchronizers is not the complete answer to the challenge of coordination[28]. Among other shortcomings, only "reactive" behavior can be described through constraints. In IWIM, coordination can be expressed explicitly (in MANIFOLD, this is done as state diagrams), which allows for "pro-active" coordination behavior.

In an Actor-based language extended with synchronizers, as proposed in[28], the basic model of communication is a variant of the (asynchronous) TSR model, wherein the computation and communication concerns are mixed together in the same modules. In contrast, there is a clear separation of computation

and communication concerns in IWIM (and MANIFOLD).

Dragoon[29] is an object oriented programming language that allows specification of synchronization constraints externally to an an object. Unlike the synchronizers in[28], Dragoon allows specification of coordination of single objects only. Procol[30] augments the notions of constraints and events to an object oriented language. Events can trigger constraints which can reactively observe invocations (message passing), but, unlike the synchronizers in[28], they cannot inhibit or limit them. The communication models of Procol and Dragoon fit in the TSR family, rather than IWIM.

One of the best known coordination languages is Linda[24, 25]. Linda uses a so called generative communication model, based on a shared tuple space[23]. The tuple space of Linda is a centrally managed space which contains all pieces of information that processes want to communicate. A process in Linda is a black box. The tuple space exists outside of these black boxes which, effectively, do the real computing. Linda processes can be written in any language. The semantics of the tuples is independent of the underlying programming language used. There are only four communication primitives provided by Linda, each of which associatively operates on a single tuple in the tuple space. The primitive **in** searches the tuple space for a matching tuple and deletes it; **out** adds a tuple to the tuple space; **read** searches for a matching tuple in the tuple space; and **eval** starts an active tuple (i.e., a process).

Linda is meant to augment regular programming languages which are to be used to express normal computation. Linda extensions to many languages exist, e.g., C-Linda, Pascal-Linda, Fortran-Linda, etc. In contrast, MANIFOLD is a "complete" language. The Linda model addresses only part of the underlying concerns of the IWIM model. There is a symmetry between the communication primitives in Linda, and the communication between processes is accomplished anonymously through the tuple space. However, there is nothing to prevent complete mixing of communication concerns with computation. There is no clear separation of workers and managers, as in IWIM. Unlike Linda, MANIFOLD encourages programmers to develop "pure coordination modules" separately and independently of the "pure computation modules" in their applications. This manifests the result of the substantial effort invested in the coordination component of an application in a tangible form as modular "pure coordinators" which can be reused in other applications. A significant difference between the underlying models of Linda and MANIFOLD can be characterized as a more data-oriented (Linda) versus a more control-oriented (MANIFOLD) approach to coordination of the cooperation among concurrent processes.

Linda is so simple, it is viewed as an "assembly level coordination language" on a shared data space. A number of coordination languages are based on the Linda model but go beyond it.

Gamma[31, 32] is a programming model based on non-deterministic rewriting of multi-sets. It provides a framework in which programs can be expressed with a minimum of explicit control. Ideally, efficient execution schedules for such high-level program specifications can be found automatically. Alternatively, a coordination language can be used to explicitly add the necessary scheduling information to tame the otherwise highly non-deterministic execution of a multi-set transformer program to a desired level of determinism[33].

A mix of data-flow, control flow, and path-expressions is used in [34] to describe the coordination of objects comprising a subsystem. The model used here is meant to *describe*, rather than *prescribe* a coordination protocol. In contrast, IWIM (and MANIFOLD), as well as most other systems mentioned in this section, are prescriptive.

The concept of *contracts*, introduced in[35, 36] extends the notion of type to capture the message passing obligations of objects. Contracts are intuitively similar to constraints imposed on object behavior as in[28], except that contracts are *descriptive* as opposed to the prescriptive nature of the synchronizers of [28].

Some coarse-grain parallel logic programming languages have also been viewed as coordination languages. PMS-Prolog[37] is an example. Multi-Prolog[38] and Prolog-D-Linda[39] are Prolog extensions based on the Linda model. Shared Prolog[40] is based on an extended Linda model that uses multiple data spaces.

Strand[41] is a parallel logic programming language with an emphasis on coordination. It offers a set of parallel programming mechanisms independent of the mechanisms for controlling sequential computations. It uses fine-grain AND-/OR-parallelism to evaluate implicitly parallel atomic goals (i.e., sequential computations).

A number of other parallel logic programming languages also exist, e.g., Constraint Logic Programming languages[42] and Oz[43], that can, in principle, be used for coordination, in as much as logic clauses can

represent the constraints and the protocols for concurrent execution of atomic goals (i.e., sequential computation fragments). However, Strand is different than these other languages because of its emphasis on coordination constructs. Indeed, Strand is offered as a coordination language and, like Linda, has been used to augment imperative sequential languages such as C and Fortran, yielding Strand-C and Strand-Fortran.

The metaphor of Interaction Abstract Machines (IAMs)[44] and its underlying formal computational model, Linear Objects[45], present a paradigm for abstract modeling of concurrent agent-oriented computation. The operational semantics of the agents and their interactions is given in terms of the proof theory of Linear Logic. The notion of "property-driven communication" in IAMs is analogous to the concept of anonymous communication through ports and events in IWIM (and MANIFOLD). Furthermore, The fan-in and fan-out of the streams at ports in MANIFOLD can be seen as the equivalent of the notion of broadcasting on specific channels in IAMs, making the flow of units in the IWIM (and MANIFOLD) streams analogous to waves in IAMs.

IAMs, Linear Objects, tuple space in Linda, Shared Prolog, and Blackboards[46] all propose a shared, open unrestricted data-structure (formally modeled as a multi-set) as an appropriate medium for communication in a concurrent or distributed data-driven system. The IWIM model, on the other hand, does not contain the notion of any central or shared entity, and is inherently a distributed model. Furthermore, IWIM supports a more control driven specification of coordination than these models.

8 Conclusion

Programming of parallel systems is often considerably more difficult than (what intuitively seems to be) necessary. A good deal of promising work is carried out to investigate how to take advantage of the parallelism provided by a hardware platform, without exposing the programmers to concurrency. This approach is effective for certain types of applications. An alternative approach is to regard concurrency not as a necessary evil, but as a powerful paradigm that programmers can take advantage of. This approach is more general and more challenging.

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

In this paper, we illustrate the shortcomings of the direct use of communication models that are based on targeted-send and receive primitives in large concurrent applications. We propose that the gap between the requirements of concurrent applications and the communication primitives supported by the platform on which they are implemented must be filled with an explicit model of cooperation. More importantly, we argue that there is an urgent need for practical models and languages wherein various models of cooperation can be built out of simple primitives and structuring constructs. Theoretical work in this area, e.g., π-calculus or process algebra, is still too fundamental to be used directly in large concurrent applications, where the practical programming concerns for efficiency, modularity, maintainability, and re-usability predominate theoretical concerns for elegance, minimality, efficacy, and expressibility. On the other hand, the tried and true models of cooperation, such as client-server, barrier synchronization, etc., that are used in practical applications of today are simply a set of ad-hoc, special-case templates; they do not constitute a coherent paradigm for the definition of cooperation protocols.

In our view, massively parallel and distributed systems open new horizons for large applications and present new challenges for software technology. Classical views of concurrency 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 distributed and massively parallel systems solely on strictly synchronous communication.

We present the IWIM model as a more suitable basis for control-oriented coordination languages. The significant characteristics of the IWIM model include compositionality, which it inherits from data-flow,

anonymous communication, and separation of computation concerns from communication concerns. These characteristics lead to clear advantages in large concurrent applications.

MANIFOLD is a specific coordination language based on the IWIM model. Some of the interesting properties of MANIFOLD were illustrated through simple examples in this paper. More experience is still necessary to thoroughly evaluate the practical usefulness of MANIFOLD. However, our experience so far indicates that MANIFOLD is well suited for describing complex systems of cooperating concurrent processes.

MANIFOLD uses the concepts of modern programming languages to describe and manage connections among a set of independent processes. The unique blend of event driven and data driven styles of programming, together with the dynamic connection graph of streams seem to provide a promising paradigm for concurrent systems. The emphasis of MANIFOLD is on orchestration of the interactions among a set of autonomous *agents*, each providing a well-defined segregated piece of computation, into an integrated concurrent system for accomplishing a larger task.

We believe it is possible and useful to go beyond MANIFOLD and develop languages and support environments that provide higher level abstractions, constructs, and tools for the development and debugging of the coordination components of massively concurrent applications. The ongoing work in our group on the visual programming language and environment based on MANIFOLD is one direction in which we pursue this goal.

9 Acknowledgment

I would like to acknowledge the direct and indirect contributions of all past and present members of the MANIFOLD group at CWI to the work reported in this paper. In particular, Paul ten Hagen inspired some of the original concerns and the motivation for MANIFOLD. Many of the original concepts in MANIFOLD were fleshed out, revised and shaped in discussions with Ivan Herman. It is a pleasure to acknowledge his contributions as a co-designer/developer of the first version of the MANIFOLD language and its implementation.

The design and implementation of the current version of the MANIFOLD language and its run-time system is primarily the work of the author, Freek Burger, and Kees Blom, with additional support and contributions by Michael Guravage and Ivan Herman. Kees Everaars has worked his way through our still inadequate user-level documentation and has developed a number of working examples in MANIFOLD. His efforts have helped us to debug our system and our documents. The on-going work of Pascal Bouvry on the visual programming environment for MANIFOLD, and Eric Rutten's work on the formal semantics of the first version of MANIFOLD are also acknowledged. Many past and present members of the MANIFOLD group worked on various examples, a modified version of some of which is presented in this paper. I am thankful for all of their contributions, and also, for the helpful comments of Krzysztof Apt and George Papadopoulos on earlier versions of this paper.

10 References

[1] C. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, August 1978.

[2] C. Hoare, *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science, New Jersey: Prentice-Hall, 1985.

[3] R. Milner, *Communication and Concurrency.* Prentice Hall International Series in Computer Science, New Jersey: Prentice Hall, 1989.

[4] J. A. Bergstra and J. W. Klop, "Process algebra for synchronous communication," *Information and Control*, vol. 60, pp. 109–137, 1984.

[5] R. Milner, "The polyadic π-calculus: A tutorial," Tech. Rep. Res. Report LFCS-91-180, Lab. for Foundations of Computer Science, Edinburgh University, 1991.

[6] R. Milner, "Elements of interaction," *Communications of the ACM*, vol. 36, pp. 78–89, January 1993.

[7] INMOS Ltd., *OCCAM 2, Reference Manual*. Series in Computer Science, London — Sydney — Toronto — New Delhi — Tokyo: Prentice-Hall, 1988.

[8] T. Bolognesi and E. Brinksma, "Introduction to the ISO specification language LOTOS," *Computer Networds and ISDN Systems*, vol. 14, pp. 25–59, 1986.

[9] D. B. Skillicorn, "Structuring data parallelism using categorical data types," in *Proceedings of PMMP93: Programming Models for Massively Parallel Computers*, IEEE Computer Society, September 1993.

[10] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman, 1989.

[11] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, Q. Wu, and R. L. While, "Parallel programming using skeleton functions," in *Proceedings of PARLE93, Parallel Architectures and Languages*, June 1993.

[12] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[13] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, "PVM 3 user's guide and reference manual," Tech. Rep. ORNL/TM-12187, Oak Ridge National Laboratory, September 1994.

[14] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker, "An introduction to the MPI standard," Tech. Rep. CS-95-274, University of Tennessee, Jan. 1995.

[15] J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker, "A proposal for a user-level, message passing interface in a distributed memory environment," Tech. Rep. ORNL/TM-12231, Engineering Physics and Mathematics Division, Mathematical Sciences Section - Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, June 1993.

[16] R. Butler and E. Lusk, "User's guide to the p4 parallel programming system," Tech. Rep. ANL-92/17, Argonne National Laboratory, Oct. 1992. Version 1.4.

[17] R. Hempel, H. Hoppe, U. Keller, and W. Krotz, "PARMACS v6.1 specification," tech. rep., PALLAS GmbH, Hermulheimer Strasse 10, D-50321, August 1995.

[18] T. Malone and K. Crowston, "The interdisciplinary study of coordination," *ACM Computing Surveys*, vol. 26, pp. 87–119, March 1994.

[19] W. W. Gibbs, "Turning back the clock," *Scientific American*, p. 22, June 1995.

[20] 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.

[21] F. Arbab, "Manifold version 2: Language reference manual," Tech. Rep. preliminary version, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, 1995.

[22] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Communication of the ACM*, vol. 35, pp. 97–107, February 1992.

[23] D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, 1985.

[24] N. Carriero and D. Gelernter, "LINDA in context," *Communications of the ACM*, vol. 32, pp. 444–458, 1989.

[25] W. Leler, "LINDA meets UNIX," *IEEE Computer*, vol. 23, pp. 43–54, February 1990.

[26] C. T. H. Everaars, P. W. Hemker, and W. Stortelder, "Manual of splds, a software package for parameter identification in dynamic systems," Tech. Rep. preliminary version, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, September 1995.

[27] L. Dami, "HOP: Hierarchical objects with ports," in *Object Frameworks* (D. Tsichritzis, ed.), Centre Universataire d'Informatique, University of Geneva, 1992.

[28] S. Frolund and G. Agha, "A language framework for multi-object coordination," in *Proc. ECOOP '93*, vol. 707 of *Lecture Notes in Computer Science*, pp. 346–360, Springer-Verlag, 1993.

[29] C. Atkinson, S. Goldsack, A. D. Maio, and R. Bayan, "Object-oriented concurrency and distribution in DRAGOON," *Journal of Object-Oriented Programming*, March/April 1991.

[30] J. van den Bos and C. Laffra, "PROCOL: A concurrent object-oriented language with protocols delegation and constraints," *Acta Informatica*, vol. 28, pp. 511–538, 1991.

[31] J. Banatre and D. L. Metayer, "The GAMMA model and its discipline of programming," *Science of Computer Programming*, vol. 15, pp. 55–77, November 1990.

[32] J. Banatre and D. L. Metayer, "Programming by multiset transformations," *Communications of the ACM*, vol. 36, pp. 98–111, January 1993.

[33] M. Chaudron, "Schedules for multiset transormer programs," Tech. Rep. Technical Report 94-36, Rijksuniversiteit Leiden, PO Box 9512, 2300 RA Leiden, The Netherlands, December 1994.

[34] R. H. Campbell and N. Islam, "A technique for documenting the framework of an object-oriented system," in *Proc. Second Int. Workshop on Object Orientation in Operating Systems*, September 1992.

[35] R. Helm, I. Holland, and D. Gangopadhyay, "Contracts: Specifying behavioral compositions in object-oriented systems," *SIGPLAN Notices*, vol. 25, pp. 169–180, October 1990.

[36] I. Holland, "Specifying reusable components using contracts," in *Proc. ECOOP '92* (O. L. Madsen, ed.), vol. 615 of *Lecture Notes in Computer Science*, pp. 287–308, Springer-Verlag, July 1992.

[37] M. Wise, D. Jones, and T. Hintz, "PMS-Prolog: A distributed Prolog with processes, modules and streams," in *Implementations of Distributed Prolog*, Series in Parallel Computing, pp. 379–404, Wiley, 1992.

[38] K. DeBosschere, "Blackboard communication in Prolog," in *Parallel Execution of Logic Programs*, vol. 569 of *Lecture Notes in Computer Science*, pp. 159–172, Springer-Verlag, 1991.

[39] G. Sutcliffe, "Prolog-D-Linda v2: A new embedding of Linda in SICStus Prolog," in *Proc. Workshop on Blackboard-based Logic Programming*, pp. 105–117, June 1993.

[40] A. Borgi and P. Ciancarini, "The concurrent language Shared Prolog," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 1, pp. 99–123, 1991.

[41] I. Foster and S. Taylor, *Strand: New Concepts in Parallel Programming.* Prentice-Hall, 1990.

[42] E. Shapiro, "The family of Concurrent Logic Languages," *ACM Computing Surveys*, vol. 21, pp. 412–510, September 1989.

[43] M. Henz, G. Smolka, and J. Wurts, "Oz: A programming language for multiagent systems," in *Proc. Int. Joint Conf. on Artificial Intelligence (IJCAI 93)*, pp. 404–409, August 1993.

[44] J. Andreoli, P. Ciancarini, and R. Pareschi, "Interaction Abstract Machines," in *Trends in Object-Based Concurrent Computing*, pp. 257–280, MIT Press, 1993.

[45] J. Andreoli and R. Pareschi, "Linear Objects: Logical processes with built-in inheritance," *New Generation Computing*, vol. 9, no. 3-4, pp. 445–473, 1991.

[46] H. P. Nii, "Blackboard systems," in *The Handbook of Artificial Intelligence* (A. Barr, P. Cohen, and E. Feigenbaum, eds.), vol. 4, pp. 1–82, Addison-Wesley, 1989.