Communicators: Object-Based Multiparty Interactions for Parallel Programming

Dror G. Feitelson
Department of Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, Israel
drorf@cs.huji.ac.il

Technical Report 91-12

November 1991

Abstract

Contemporary parallel programming languages often provide only few low-level primitives for pairwise communication and synchronization. These primitives are not always suitable for the interactions being programmed. Programming would be easier if it was possible to tailor communication and synchronization mechanisms to fit the needs of the application, much as abstract data types are used to create application-specific data structures and operations. This should also include the possibility of expressing interactions among multiple processes at once. *Communicators* support this paradigm by creating abstract communication objects that provide a framework for interprocess multiparty interactions. The behavior of these objects is defined in terms of *interactions*, in which multiple processes can *enrole*. Interactions are *performed* when all the roles are filled by ready processes. Nondeterminism is used when the order of interaction performance is immaterial. Interactions can also be disabled, thereby creating a uniform queueing mechanism where interactions may represent events.

Keywords: Abstraction, Communication patterns, Multiparty interactions, Object-based programming, Parallel programming, Synchronization.

1 Introduction

Object-based programming is an important and growing influence in the software design field. It provides tools for modularity and data encapsulation, and enhances maintainability, portability, and reuse of code. It therefore seems reasonable to apply the concepts of object technology to new fields, such as the programming of communication and synchronization mechanisms in parallel systems.

Applying a new concept also provides an opportunity to rethink old approaches and implementations. Specifically, before we apply object technology to existing communication and synchronization mechanisms, we should rethink the question of what sort of mechanisms we really want. In the context of tightly-coupled parallel systems, it seems that there is considerable space for improvement. Most of the mechanisms now in use were designed for multiprogrammed uniprocessor systems, or for loosely-coupled distributed systems. These mechanisms do not reflect the multiway simultaneous activities that occur in tightly-coupled systems. A new mechanism is required in order to express multiparty interactions.

It should be pointed out from the outset that we are not following the conventional way in which "objects" are combined with "parallelism". Many object-based systems use the separation between objects to define a natural parallel behavior, in which computations relating to distinct objects may be done in parallel. In effect the objects become agents, and messages passed between them drive the program execution forward. This approach is used in the Actors model and in various parallel object-oriented languages (see [42]). Our approach, on the other hand, is to seek the underlying concepts on which object technology is based and apply them to "conventional" parallel programming languages. Specifically, we take the ideas of abstraction and encapsulation, and apply them to the definition of constructs for multiparty interactions.

The language constructs introduced in this paper are not radically new — they are mainly a new combination of proven good ideas with some extensions and enhancements. Well-known results and approaches are brought to bear on the new constructs, including dependence analysis and automatic parallelization, object technology and modularization. All this is done subject to the guideline that the semantics have to be kept crisp and well defined. The final result is quite different from previous proposals.

The motivation for using an object-based approach to implement multiparty interactions is elaborated in the next section. To study the possibility of a general formulation for abstract communication objects, the notion of *communicators* is introduced in section 3. These are abstract communication objects that provide a framework for interprocess multiparty interactions. The behavior of these objects is defined in terms of *interactions*, in which multiple processes can *enrole*. Interactions are *performed* when all the roles are filled by ready processes. Nondeterministic enrolement is used when the order of interaction performance is immaterial. Section 4 then presents the features that must be included in the communicator formalism in order to handle various well known communication and synchronization schemes, assuming they are representative of user needs. For example, interactions can be disabled, thereby creating a uniform queueing mechanism. Section 5 gives an application example. Implementation issues are discussed in section 6; it is shown that communicators can utilize many existing practical results from the field of parallel programming. Section 7 compares communicators with other related proposals, and the conclusions of the study are drawn in section 8.

2 Motivation

2.1 Objects and Abstraction

Object-based programming is an outgrowth of the concept of abstract data types (ADTs). Objects often embody various entities that are used to structure the application. The representation of these entities is encapsulated within the objects, resulting in the creation of new data types that were not provided as primitive data types by the system.

Maybe the most important aspect of object technology is that it allows the user to create new abstractions. Specifically, ADTs allow a programmer to fashion his own data types, complete with the operations that may be performed on them. Porting this idea to parallel environments, it is natural to suggest that the user be allowed to define his own *abstract communication objects*, rather than restricting him to use the primitives provided by the system directly. Such objects would have certain communication and synchronization properties, that are useful for the application in which they are defined. For example, any number of processes could be synchronized and data passed between them in a certain pattern. The implementation of the desired behavior based on the system primitives would be encapsulated within the objects.

The chief virtue of abstractions is that they lead to a separation of concerns. For example, consider the way in which parallelism is expressed. Concurrent systems provide a **fork** primitive, that enables one additional process to be created at a time. Each processor in a parallel machine can also create only one additional process at a time. But parallel languages should provide a **parbegin/parend** construct instead of a **fork**. Such a closed construct is better both because it induces a structured programming style (as opposed to **fork** and **join** which are reminiscent of **goto**) [13], and because it provides a higher level of abstraction, allowing the programmer to express the degree of parallelism in the program *directly*. The underlying implementation might still be based on a serial loop that performs one **fork** per iteration, or else the processes may be spawned in a tree structure with logarithmic delay, or even using fetch-and-add or broadcasts to achieve constant delay. Without the **parbegin/parend** abstraction, the programmer has to contend with these options himself [40]. With it, these are implementation details that are delegated to the system.

Returning to our abstract communication objects, we suggest that they induce a clean separation between the implementation of the interactions on the one hand, and the bodies of the interacting parallel processes on the other. The processes themselves are sequential, and may benefit from the accumulated experience with sequential programming. Much of the implementation of the interactions is passed on to the system. This allows new and improved implementation to be incorporated easily. Portability is also enhanced, because only the communication objects with their well-defined semantics would have to be recoded for the primitives of a new environment; the processes that use these objects would stay the same.

2.2 Multiparty Interactions

A large number of abstractions have been designed to enhance the programming of concurrent (time sharing) systems [2]. As these abstractions were meant to be used on uniprocessor machines, their functionality is largely restricted to regulating the serial order of a number of operations. For example, semaphores can be used to provide mutual exclusion or a producer-consumer rela-

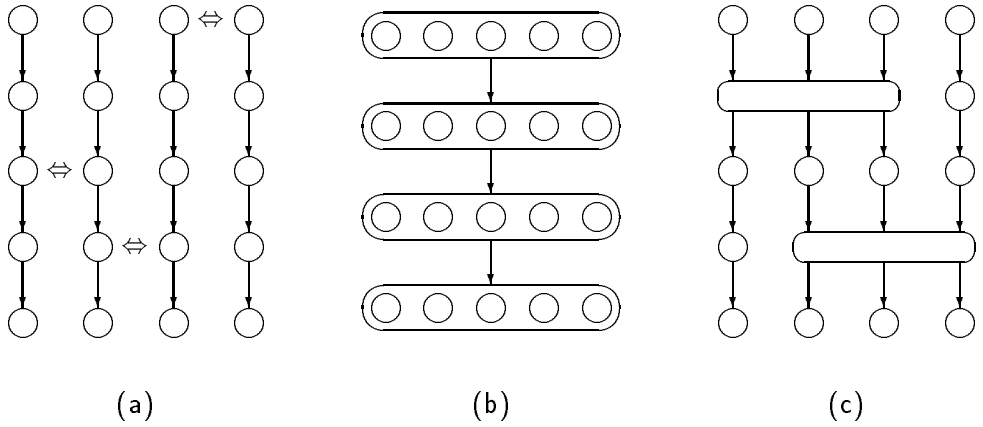

Figure 1: *Three models of parallel programs.*

tionship between a pair of processes [13]. However, semaphores cannot express the semantics of a barrier synchronization[1]. Parallel programming for multiprocessor machines, which employ real parallelism, should use abstractions of *multiparty interactions*.

To understand the role of multiparty interactions, we must first characterize existing models of parallel programming. We restrict the discussion to languages with explicit parallelism. Using terminology introduced by Blelloch [7], such languages can be classified as belonging to either of two types. In *processor-oriented* languages, independent scalar operations are executed in parallel on distinct (virtual) processors (fig. 1 (a)). This is similar to Flynn's MIMD model [15]. Interactions between the instruction streams depend on the exact model of computation. In message passing systems, interactions occur on a pairwise basis (horizontal arrows in the figure). With shared memory, the interactions are implicit.

The second type is *collection-oriented* languages. Here the control is serial, meaning that instructions are executed one after the other. Each instruction, however, operates on a collection of data elements in parallel (fig. 1 (b)). This approach includes Flynn's SIMD model [15], data-parallel programming [7, 22], and action systems [3]. The parallel instructions that are available depend on the model. Some might operate on each element in the collection independently, while others include some interaction, e.g. performing a permutation or calculating a parallel prefix. The chief virtue of this approach is that the control is strictly serial, matching it to the mental capabilities of human programmers who find it hard to "think in parallel".

Multiparty interactions combine these two approaches. Part of the computation is carried out in the processor-oriented mode, with independent parallel streams of scalar operations. When necessary, any subset of these parallel streams can come together and interact in some way (fig. 1 (c)). This approach has a number of advantages. First, whenever there is no need for interaction the parallel streams are completely independent of each other. Second, all interactions are explicit and encapsulated in some way, making the semantics easier to follow. Finally, multiparty interactions provide a higher level of abstraction than pairwise interaction. This makes programming easier, and allows many low-level implementation details to be delegated to the system.

Consider the dining philosophers problem as an example. The crux of the problem is to convey

[1] Of course, a barrier can be implemented using semaphores, but this is an implementation based on certain primitives, not an abstraction.

the message that a philosopher needs two forks simultaneously in order to eat. However, most parallel programming languages can only express pairwise interactions. The programmer must therefore contend with the difficulty of expressing a three-way interaction (philosopher and two forks) with primitives for two-way interactions. Languages like CSP may help by providing the means to model the two-way interactions and check if the resulting behavior is correct and deadlock free [23], but they do not help very much in *deriving* the solution[2]. Parallel languages with multiparty interactions, on the other hand, provide a more suitable abstraction: the three-way interaction can be expressed directly [17]. The implementation details are thereby delegated to the system, together with the problem of preventing deadlock.

It is interesting to note that multiparty interactions create a generalization of the processor-oriented and collection-oriented approaches, and include them as special cases. The processor-oriented approach is obtained by only using pairwise interactions. The collection-oriented approach results when all interactions involve all the elements, and there are no scalar operations between successive interactions. Using other types of multiparty interactions exposes a full spectrum of possibilities between these two extremes.

2.3 User Interface

The main idea proposed in this paper is to combine object technology with multiparty interactions, and specifically to use abstract communication objects to implement multiparty interactions. Thus the user is given the opportunity to create communication and synchronization mechanisms with any desirable properties. The implementation is encapsulated within the objects, and only a procedure-like interface is accessible from other parts of the program.

The availability of such abstraction mechanisms can be expected to simplify parallel programming to a great extent. Contemporary languages typically provide a single, low-level, general primitive, which forces the programmer to find ingenious ways in which it can be used to implement various types of interactions. An abstraction mechanism, on the other hand, provides the user with the means to tailor the synchronization or interconnection scheme so as to best fit his needs. Therefore the programmer can concentrate on the solution of the problem at hand, rather than having to modify the algorithms to fit a certain primitive.

It should be noted that the concept of multiparty interactions by itself is not new at all. Examples of multiparty interactions that are commonly used include the multicast and broadcast communication primitives [11, 6], the barrier synchronization [19], permutations of data elements [38], the **scan** operation [7], and paralation operations [36]. However, each is a specific primitive multiparty interaction. If it fits the programmer's needs, all is well. But if it does not, the programmer is again required to change the algorithm so that it fits the available primitive. More general mechanisms are reviewed in section 7. However, none of them were designed in the context of general purpose parallel languages for tightly coupled machines.

As the system designer can never anticipate all what the users might want, it is necessary to supply tools that allow the users to express their needs in a high level of abstraction. This paper is a first step towards the design and implementation of such a tool, which we call *communicators*. It is expected that when programmers have the power to express multiparty interactions, this will also

[2]Common solutions are either to break the symmetry by having one philosopher pick up the forks in the opposite order, or to add a footman that does not allow more than four philosophers to the table at once.

lead to new algorithmic solutions for various problems. For example, Forman shows how multiparty interactions were material in developing a new solution for the lift problem [16]. In his solution, lifts cooperate to find the one that can service a new request with minimal cost, but do so without sacrificing the natural distributed control scheme.

It should be noted that this study is practical in nature. We do not search for "sufficient" or "minimal" notations, but rather for language constructs that will be convenient, useful, and also amenable to efficient implementation. Hopefully, this approach will help parallel programming advance towards the maturity of sequential programming.

3 Communicators

Before delving into the details, let us be specific about the context of the discussion. Communicators are suggested as an extension to the expressiveness and structuring capabilities of imperative programming languages with explicit parallelism, used on a tightly-coupled architecture. Such a system would typically be used for *transformational* computations, i.e. terminating computations of some input/output function, as opposed to *reactive* computations, such as embedded applications that continuously react to stimuli from their environment [31]. The distinction is important because practically all the work on multiparty interactions to date was done in the context of the design of reactive systems, e.g. a system of multiple lifts or a network of point-of-sale outlets (and see section 7).

The assumptions about the language and programming model are as follows. It is assumed that independent processes may be created. In case processes are created dynamically, this is done using a **parbegin/parend** construct, and the parent process is suspended until all its descendants terminate. Such constructs can be nested in each other and combined with conventional constructs for control-flow, e.g. loops and conditionals. A process has read-only access to the states of its ancestors. All processes have read-only access to the application's global state. This allows information that existed before the processes were created to be shared asynchronously without explicit interactions. Communicators are instantiated like other objects, and observe similar scoping rules. In particular, all the descendants of a process that created a communicator may share its use. All interactions between processes are mediated by communicators. The exact details of the language are immaterial. For example, the whole issue of variable typing is orthogonal to the possible use of communicators, and is therefore ignored in this presentation. There are no assumptions about the notation used for expressions, assignments, control flow, function calls, etc. It can be the same as in Pascal, C, or any other imperative language. Hence the following description of communicators is generic rather than specific.

In the description of how communicators are used, we shall focus on two main issues: the entry into multiparty interactions, and the specification of the semantics of such interactions.

3.1 Definitions

Communicators provide a framework in which *multiple* processes can communicate and synchronize. **Interactions**[3] occur only when all of the processes come together and possibly share some

[3] This is similar but not identical to the interactions proposed by Evangelist et. al. [14]. The differences are discussed below.

data and computation — hence they are *synchronous*. The interactions are formally defined by the sequence of actions that describe how data is manipulated when all the processes come together. The participating processes need not be known in advance (hence communicators provide *first-order* multiparty interactions in the terminology of [25]). An interaction includes certain **roles**[4] which are assumed by the participating processes. A process may **enrole**[4] by specifying the communicator, interaction, and role that it wishes to enrole in. The process is queued until all the required processes have enroled and the interaction can commence. Interactions are executed in a mutually exclusive manner; each execution is called a **performance**[4]. If a number of processes enrole for the same role, they will have to wait for subsequent performances of the interaction.

The syntax of a communicator definition is basically similar to that of objects in many modular languages:

communicator *name*
  var: *identifiers-list*
  initialization
  {
    initialization code
  }
  interaction *name*
  {
    role: *role-identifier-list*
    var: *identifier-list*
    interaction body
  }
endcomm

A generic **var** is used to avoid the issue of typing when defining variables. The communicator's global variables and the initialization are optional. there may be any number of interactions, and each must have at least one role. Role identifiers are identifiers followed by a list of parameters between parentheses. Arrays of communicators, interactions, and roles are allowed. The dimensions of the array must be known when the communicator is created, and we again avoid the issue of whether this is static or dynamic. When an array of interactions is specified, an index variable should be given if the different interactions exhibit different behaviors.

Communicators extend the object-based concept of ADTs in a straightforward manner. A communicator may have encapsulated internal state, just like an ADT. Interactions are a multiparty generalization of methods — the procedures used to access the ADT. Roles are introduced simply to cope with the fact that each interaction has multiple entry points, for the different participating processes. Due to the different environment, however, communicators have functionality that does not exist in ADTs. For example, in many cases the dynamics of interaction execution are all that is needed, and the communicator does not need to save any internal state; an ADT without internal state, on the other hand, would be meaningless. In addition there is the ability to disable interactions temporarily, which is reminiscent of guarded commands in other languages. ADTs usually do not provide this capability, although it may be possible to block processes.

[4] A conscientious effort was made to use existing terminology when possible, rather than introducing new terms. This terminology was introduced by [18, 17]. In particular, the new verb *enrole* means "to assume a role".

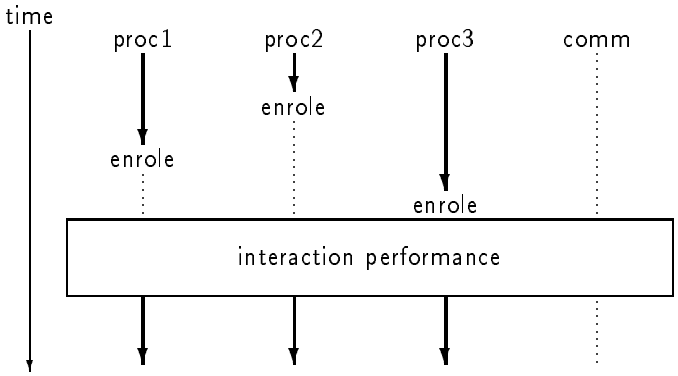

Figure 2: *Schematic representation of the dynamics of interaction performance. The manner in which data passes between the processes and the communicator depends on the definition of the interaction.*

3.2 Model of Computation

A parallel program with communicators is composed of two types of entities: processes and communicators. Processes are active entities. Each process executes a sequential block of code. The code may, however, include closed constructs (e.g. **parbegin/parend**) to spawn additional processes. When such a construct is encountered, the process is suspended until all of its descendants terminate. The code may also include instructions to enrole in the interactions of various communicators.

The communicators are passive entities, not schedulable processes. They only provide the *framework* for interactions between processes. The performances of distinct interactions in the same communicator are mutually exclusive. The code in the body of an interaction is executed by one or more of the participating processes, depending on the nature of the interaction and on the implementation. In any case, the programmer has no control over such details. As far as each process is concerned, data is passed to and received from the interaction in the form of enrolement parameters. It is up to the system to implement the interaction itself in the most efficient manner. The dynamics of an interaction are shown in fig. 2.

3.3 Enrolement

Recall that communicators embody abstract communication objects. Just like ADTs, they can be declared in various places in the program, and used by processes within their scope. Depending on the system environment, it might also be possible to pass capabilities for access to communicators from one process to another.

A process enroles by specifying the communicator, interaction, and role, and supplying the required arguments to match the formal role parameters:

enrole@*communicator.interaction.role*(*arg 1*, $\cdots$, *arg k*)

If the communicator, interaction, and/or role were declared to be an array, the desired element should be specified. The parameters are passed either by value or by reference. This distinction is

orthogonal to the issue of using communicators, being the same as passing arguments to procedures and functions.

A process may propose to enrole in a number of different roles, which might belong to different interactions and different communicators. This is expressed by the **select_enrole** and **multi_enrole** instructions:

```
select_enrole@···          multi_enrole@···
    or_enrole@···            and_enrole@···
    or_enrole@···            and_enrole@···
endselect                  endmulti
```

When the system finds that any one of these interactions may commence, the relevant enrolement is consummated. If more than one is possible, one is chosen nondeterministically. The others are either aborted (in a **select** enrolement), or just delayed until the current performance terminates (in a **multi** enrolement). This behavior is similar to that of nondeterministic constructs in CSP, Occam, and Ada, and expresses a deliberate decision to ignore details such as the order of execution at this level of abstraction. It is necessary in order to allow the system sufficient freedom in scheduling the various interactions, thus relieving the programmer of the need to define the sequence of interactions in advance [23].

A similar construct providing nondeterministic choice has been included in practically all of the languages supporting multiparty interactions (section 7). However, there is an important difference. The other proposals are for a construct that combines nondeterministic choice with iteration: one of the ready interactions is executed each time, and the construct terminates only when none are ready. Thus there is no control over the number of times each interaction will be executed. While this is probably the correct construct for reactive systems, where the external stimuli are unknown in advance, it seems ill-suited for transformational languages, which are used to implement a terminating algorithm. We therefore favor the two constructs suggested above: one specifies that exactly one interaction will be performed, and the other specifies that each interaction will be performed exactly once. An example of the usefulness of this construct is given in section 5.

3.4 Interactions

The behavior of a communicator is defined by the interactions which it provides. The interactions are described in a sequential language. Like other issues, the exact syntax is immaterial and is not discussed. Data transfer is simply handled by assignments between role parameters. This allows the following patterns to be expressed:

- Data transfer from one role to another.
- Data divergence, where a datum from one role is routed to a number of other roles.
- Data reduction, where several data items (possibly form different sources) are combined to create a single new value.
- Data buffering, by assigning to the communicator's global variables. This provides the possibility of storing data in one interaction and retrieving it in another.

Special instructions need be added to deal with synchronization. In ADTs, this is done by explicit waiting on an event queue. Such queueing can be interpreted either as blocking the process, or as blocking the execution of the operation it is performing on the ADT. The two interpretations are equivalent in the context of ADTs, because there is exactly one process involved in the operation. But when the concept is generalized to multiparty interactions on communicators, the two interpretations differ. Blocking and unblocking of individual enroled processes is problematic, as it causes semantic difficulties. For example, what happens if only part of the enroled processes block? What happens when they are subsequently resumed? It therefore seems that the correct interpretation is that blocking and unblocking should apply to an interaction performance. Blocking blocks all the enroled processes, but leaves the interaction enabled. Thus another group of processes can engage in another performance of the same interaction.

Another possible synchronization mechanism is the disabling and enabling of interactions. Disabling prevents new performances of a interaction. If an interaction disables itself, there is no effect on the current performance. Interactions can disable each other, as opposed to blocking where an interaction can only block a performance of itself.

The possibility of disabling interactions is a new synchronization mechanism. It is interesting because it has the potential for increased concurrency and improved performance. If an explicit blocking command is used, the interaction must first be performed. During the performance, the processes will find that actually they cannot proceed, so the performance would block itself. This costs extra overhead, as the processes are enqueued twice: first waiting for the performance to commence, and then waiting for the event. It also causes an unnecessary delay for other processes that could perform other interactions at the same time.

By disabling interactions, processes that cannot perform useful work are blocked from entering the communicator in the first place. In effect, events are represented by interactions. This results in a uniform queuing mechanism, where The same queue is used for processes waiting for a interaction to commence as well as processes waiting for an event to occur. It is therefore suggested that only the disabling of interactions be used as a synchronization mechanism in the communicator formalism, and that the option to block interactions not be provided. The use of these options and the resulting expressive power are further discussed in section 4.

Finally, we relate our proposal with the interactions proposed by Evangelist et. al. [14]. The first and most obvious difference is that ours are first-order interactions in which processes must enrole; they come in the context of communicators, and may include synchronization operations as described above. Those proposed by Evangelist et. al. are independent zero-order primitives, and their performance is controlled by Boolean guards. As for the five required properties that were proposed,

1. *Synchronization upon entry:* this is the same in both proposals.

2. *Split bodies:* we feel this is a drawback rather than a virtue. This important point is elaborated in section 3.5 below.

3. *Interprocess access only within interactions:* the same. Processes are completely decoupled when not engaged in an interaction.

4. *Frozen state:* this is a direct consequence of enrolement using a procedure-call interface. The outcome of the interaction depends only on the values passed into it as parameters.

5. *Bounded duration:* this is another difference. Our interactions may include any block of code, even if this means that they cannot be called "primitive". Thus we allow the user more freedom in structuring the parallel application.

3.5 Syntax and Semantics

As stated before, the exact syntactical details so not concern us at the moment. However, some of the more salient features deserve to be highlighted. The main one is the way in which the code that describes interactions is encapsulated, and the clean interface with the code that describes the parallel processes themselves. Actually this is nothing new: it is just a straightforward generalization to multiple participants of the well known and widely accepted interface for calling subroutines. Similar syntax has been used in other proposals for first-order multiparty interactions, e.g. scripts [18]. It deserves mention mainly to contrast it with proposals for zero-order multiparty interactions, which reject the advantages of this interface (e.g. interactions in [14]). Instead, such proposals specify multiparty interactions by placing the following clause in each of the interacting processes:

```
name [ loc_var = exp ]
```

The interaction name serves to identify corresponding clauses in the different processes. The body of the clause contains any number of assignments to local variables, where the expressions may use variables that are local to the other participating processes; this is where the interaction comes in. However, when looking at the code of a specific process, one has no indication of where the other variables are defined, or what their types are. The subroutine-like interface used for communicators solves this problem: it improves the structure of the code and makes it more readable. As shown in section 6, it may also make implementation easier by allowing the use of automatic parallelization techniques.

As for semantics, communicators are designed to keep their semantics crisp and free of side effects. Despite the fact that programs with communicators use explicit parallelism, the parallel processes are strictly separated and cannot interfere with each other. In this sense, communicators follow the lead of languages such as CSP, Occam, and Ada. As in those languages, this feature allows formal techniques to be used to reason about the parallel programs. The nondeterminism involved in **select** or **multi** enrolement is explicit, and can also be included in the formalism (see, e.g., [23]).

The multiparty interactions are also well defined. Interactions are performed in a mutually exclusive manner, one after the other. This is similar to the invocation of operations on monitors or abstract data types. Enroling processes are synchronized from the moment they enrole until the interaction terminates. During the performance, the processes share some of their local states. However, the description of how this sharing takes place leaves nothing undefined, as it is based on a sequential block of code. This enables the programmer to provide the desired operational semantics, without having to "think in parallel". There is no indeterministic parallel code involved.

4 Features and Expressive Power

In this section we review several features that are included in the definition of interactions, and show how these features are used to express various well known synchronization and communication

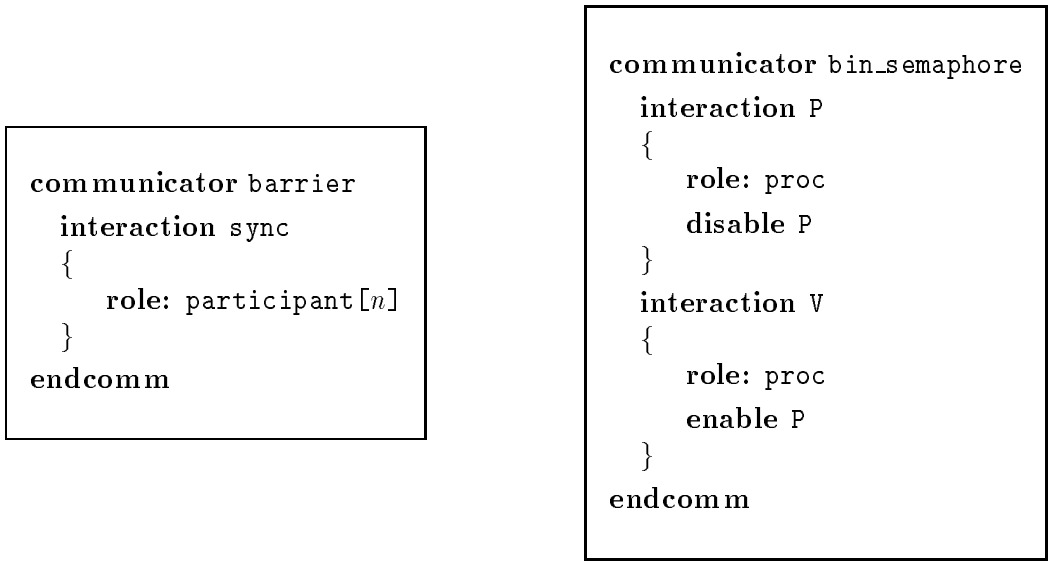

Figure 3: *Communicators that implement a barrier synchronization and a binary semaphore.*

schemes. It is assumed that these schemes are representative of what users might want, and therefore the ability to express them in the communicator formalism indicates that this formalism has sufficient expressive power to be useful. The implementation with communicators is compared with other notations.

Some syntactic shortcuts are taken in the examples, to avoid cluttering them with immaterial details. For example, "$\forall i$" is used rather than an explicit loop that iterates over the range of values that i may assume. As a side note, however, it is worth mentioning that such notational shorthands may ultimately be used to define a language that allows communication patterns to be expressed in a very high level of abstraction. This would have an advantage over languages where the communication pattern must be expressed in terms of pairwise transfers (i.e. assignments), which might lead to unnecessary serialization.

4.1 Synchronization

Several synchronization effects can be achieved without any data transfer, and also without any saved state. The simplest and most obvious is the empty interaction, which induces a *barrier synchronization* on the participating processes (fig. 3 left). If multiparty interactions are not available, the user would have to program barrier synchronizations explicitly[5]. As a barrier synchronization is required at the beginning of every interaction, support for communicators motivates the use of hardware support for this operation. Other languages would have to include barriers as language primitives in order to benefit from such support.

A *binary semaphore* may be created by a communicator with two interactions: `P` and `V`. The `P` interaction disables itself, and the `V` interaction enables it (fig. 3 right). Recall that the semantics of disabling are that a new performance of the interaction will not be started as long as it is disabled;

[5]In systems that support both MIMD and SIMD modes of computation, a barrier may be induced by an empty block of SIMD [8].

it has no effect on the current performance. Thus processes that try to enter a closed semaphore are blocked by the disabled `P` interaction. As explained above, such an implementation holds promise for improved efficiency and concurrency. The way to implement a counting semaphore is mentioned below.

The option to cause processes to wait for an interaction to be enabled also improves program structure and readability. Systems where the only event a process can wait for is communication, such as CSP or Occam, require the program to use communication in unnatural ways. For example, a semaphore can be implemented in CSP by a special process that always performs pairs of communications with the same partner: the first represents a `P` and the second a `V`. This restricts the semantics of semaphores by requiring that the process that gains access to the semaphore be the one that releases it. In addition, it incurs the overhead of an additional process, and also has the disadvantage that all the potential users of the semaphore must be named in advance.

4.2 Data Transfer

While synchronization is certainly an important aspect of parallel programming, interactions between cooperating parallel processes nearly always involve transfer of data as well as transfer of control. Numerous parallel algorithms are designed as phases of local computation interspersed by phases of data transfer in certain patterns. This is especially common in algorithms designed for special architectures that support specific patterns in hardware, such as the perfect shuffle permutation [38] or transfer along a certain dimension of a hypercube [35, 5]. Communicators are well suited to express such structures. Any pattern of communication can be expressed in a high level of abstraction, and later translated into primitives provided by the hardware.

Examples of how to express a broadcast, a perfect shuffle, or an exchange along a dimension of a hypercube are given in fig. 4. In these examples, each participant enters the interaction with one variable and receives one value; this is expressed by the role name with an extension of the variable name. In general, roles may be associated with any number of variables. The use of an array of roles allows for a simple expression of a generic behavior. For example, in the broadcast communicator all the receiving roles are assigned the value of the sending role. This is similar to repeated processes in Occam or other languages. The same goes for an array of interactions, except that here we must define the index variable in advance (this is done by the notation `[index:bottom..top]`, as shown in the hypercube example). The operator $\ll$ signifies a left shift, and $\oplus$ is exclusive or.

Pairwise synchronous message passing, as in CSP, is a special case of multiparty data transfer. It can be expressed by a communicator like that used for the broadcast, except that there is only one receiving role. The same communicator may be used by many processes, depending on its scope, thus alleviating the need for explicit naming of communication partners. This is useful for the implementation of servers that do not know the identities of all their potential clients in advance, and can therefore support processes that are created dynamically. The same communicator can also be used to distribute work among a number of servers, thus effectively providing the semantics of a mailbox.

A simple communicator with one message-passing interaction, which is used by many sending processes and one receiving process, implements the CSP choice between different inputs (and its derivatives, the Ada **select** and Occam **ALT** statements), but without Boolean expressions in the guards. This is actually the server mentioned above. A **select** *with* Boolean guards can also be implemented, but this requires an independent interaction for each sending process. As the guards

```
communicator broadcast
  interaction transfer
  {
     role: send(in), rec[n](out)
     ∀i rec[i].out := send.in
  }
endcomm
```

```
communicator shuffle
  interaction transfer
  {
     role: proc[n](in,out)
     ∀i proc[i ≪ 1].out := proc[i].in
  }
endcomm
```

```
communicator hypercube
  interaction dim[j :0.. lg n − 1]
  {
     role: proc[n](in,out)
     ∀i proc[i ⊕ (1 ≪ j)].out := proc[i].in
  }
endcomm
```

Figure 4: *Examples of communicators that implement various communication patterns.*

depend only on the internal state of the receiving process, it can calculate them by itself and use the results to disable or enable various input interactions.

In some cases the processes participating in a data transfer do not actually require the raw data values, but rather some functions thereof. It is then useful to combine the data transfer with some operations on the data. An example is given in fig. 5. This is a communicator that expresses the **scan** operation with addition, which is similar to a parallel-prefix sums [7]. Each participant receives the sum of the inputs from lower-indexed participants. Likewise, it is easy to write a communicator that implements operations that are typical in image processing or numerical solutions to partial differential equations. In this case, the participating processes are logically arranged in a two-

```
communicator scan
  interaction sum
  {
     role: proc[n](in,out)
     ∀i proc[i].out := Σ_{j=0}^{i-1} proc[j].in
  }
endcomm
```

Figure 5: *Example of a communicator that combines data transfer with operations on the data values.*

```
communicator paralation
  var: map[n]
  interaction match
  {
     role: f1[n](key), f2[n](key)
     1.  match key values to derive
         mapping from f1 to f2
     2.  store the mapping
  }
  interaction move
  {
     role: f1[n](out), f2[n](in)
     1.  retrieve the mapping
     2.  for each case where many
         inputs map to the same
         output, apply the reduction
         function
     3.  assign output values or
         default values
  }
endcomm
```

Figure 6: *A communicator with data-dependent communication patterns.*

dimensional grid, and each receives some function of the inputs from its immediate neighbors. The example in section 5 is of this type.

The routing of data does not have to follow a predefined pattern. Data-dependent data transfers are also possible. As an example, consider the paralation model introduced in [36]. This communication mechanism is captured by the communicator that is schematically described in fig. 6. No claim is made that it would be easy to implement such a communicator; it is just as easy (or hard) as the implementation of paralations directly in any other system. The point is that this abstraction can be expressed using communicators.

4.3 Asynchronous Interactions

Programs written for MIMD parallel computers allow the processes to execute with different rates. In some cases, the different rates have to be counteracted by explicit synchronization; this can be expressed by the synchronous interactions of communicators as shown in the preceding subsections. But in other cases full synchronization would cause a performance degradation, and correctness can be achieved with a lower level of coordination. This is done by asynchronous interactions, in which processes may deposit data which is subsequently used by other processes.

In order to support asynchronous interactions, communicators must have the capacity to save internal state from one interaction to another. This leads directly to a generalization of ADTs and monitors, where encapsulated internal state is accessed through mutually exclusive procedures. In fact, monitors and ADTs are similar to communicators with the restriction that each interaction only have one role. This capability exists in most object-based languages.

The difference between communicators and ADTs is in the mechanism for blocking processes. Communicators do so by disabling interactions, whereas ADTs typically have the capacity to block processes explicitly, using mechanisms similar to event variables in monitors [24]. While disabling interactions is less general, it seems to be sufficient for most practical purposes. For example, fig. 7 shows how a bounded buffer may be expressed using communicators. This description is exceptionally transparent and readable. Note that a buffer of size 1 is equivalent to a shared memory location which is accessed subject to the state of a full/empty bit, so this too may be expressed with communicators. Other asynchronous objects with internal state, such as counting semaphores, can be expressed with equal ease. As explained above, the ability to block processes before they enter the communicator also has certain performance benefits.

The advantages of expressing asynchronous interactions with communicators are even more apparent when compared with other languages such as CSP [23] or Raddle87 [16]. Using the bounded buffer example for concreteness, we find that in these languages the buffer must be implemented by an additional active process. Thus each insert and delete operation requires the run time system to schedule the buffer process to perform it. As a result the completion of the operation is delayed and the overhead associated with scheduling and context switching is paid.

As another example, let us consider the fuzzy barrier [21]. This synchronization mechanism is an extension of the barrier synchronization described above. The difference is that instead of a barrier point, the programmer defines a barrier *region*. The semantics are that no process may leave the region before all processes have entered it. A communicator that implements this mechanism is shown in fig. 8. In order to decouple the processes, arrays of interactions are used with one role each. The `end_region` interactions are enabled only after all the processes performed their respective `begin_region` interactions.

```
communicator bounded_buffer
  var: buffer[n]
  initialization
  {
     initialize buffer
     disable delete
  }
  interaction insert
  {
     role: producer(in)
     insert producer.in into buffer
     if buffer is full
        disable insert
     enable delete
  }
  interaction delete
  {
     role: consumer(out)
     assign consumer.out from buffer
     if buffer is empty
        disable delete
     enable insert
  }
endcomm
```

Figure 7: *A communicator that implements a bounded buffer.*

4.4 Things that Communicators Cannot Express

It should be noted, however, that disabling interactions is not powerful enough to express all possible asynchronous interactions. Specifically, it cannot express situations in which the decision whether or not to block the process is data dependent. Such a situation exists in the primitives provided by the Linda notation, where data transfer is based on pattern matching between tuples [1]. If a matching tuple is not found in the global tuple space, the process is blocked. In order to express this in the communicator notation, the capability to block an instance of an interaction performance is needed. The performance would subsequently be unblocked when another interaction outputs the desired value.

While it is straightforward to add this capability to the communicator notation, we refrain from doing so at present. It is felt that it would be better to first implement the more limited interface described above, and based on experience with it, to consider extensions.

Other interactions that cannot be expressed in the communicator framework include various

```
communicator fuzzy
  var: count
  initialization
  {
     ∀i disable end_region[i]
     count := 0
  }
  interaction begin_region[n]
  {
     role: proc
     count := count + 1
     if count = n
        ∀i enable end_region[i]
  }
  interaction end_region[n]
  {
     role: proc
  }
endcomm
```

Figure 8: *A communicator that implements a fuzzy barrier.*

forms of control over the participating processes. Two types of control that have been mentioned in the literature are "chosen partners" and "required sets" [18, 25]. With the chosen partners option, a process that enroles for a certain role may dictate which processes it wants as partners for the interaction. If the desired processes do not enrole, the interaction will not be performed, even if other processes do enrole. While this option is not provided by communicators, the same effect can be achieved by declaring distinct communicators for use by different sets of processes.

The option of a required set allows the programmer to define interactions that can be performed even if all the roles have not been filled. There is a subset of roles that are required, and the rest are optional. While this option might be important to cope with the highly asynchronous nature of reactive systems, which are the target environment for other proposals of languages with multiparty interactions, it seems to be unnecessary in communicators, which are targeted at transformational systems.

5 An Application Example

To illustrate the use of communicators, we now give a short application example. The application is a PDE solver using the Jacobi method on the Laplace equation. The domain is an $n \times n$ grid. Each gridpoint is allocated to a distinct process. In each iteration of the computation, the gridpoint value is replaced by the average of its four neighbors. The iterations continue until the maximum

```
communicator neighborhood[n, n]
  interaction update
  {
     role: C(new_val), N(val), S(val), E(val), W(val)
     C.new_val := (N.val + S.val + E.val + W.val)/4
  }
endcomm

communicator terminate
  interaction check
  {
     role: element[n, n](delta,flag)
     if (max_{i,j} { element[i, j].delta } < ACCURACY)
        ∀i, j element[i, j].flag := TRUE
     else
        ∀i, j element[i, j].flag := FALSE
  }
endcomm

parfor i := 0 to n − 1
   parfor j := 0 to n − 1
   {
      var: old, new, change, flag
      old := initial_value( i, j )
      repeat {
         multi_enrole@neighborhood[i, j].update.C( new )
           and_enrole@neighborhood[i − 1, j].update.N( old )
           and_enrole@neighborhood[i + 1, j].update.S( old )
           and_enrole@neighborhood[i, j − 1].update.E( old )
           and_enrole@neighborhood[i, j + 1].update.W( old )
         endmulti
         change := | new − old |
         old := new
         enrole@terminate.check.element[i, j]( change, flag )
      } until (flag = TRUE)
      record_final_value( i, j, new )
   }
   parend
parend
```

Figure 9: *Example of a PDE solver using communicators.*

change over all gridpoints falls below a predefined accuracy. The example is written in pseudocode in a style similar to that conventionally used to describe algorithms (fig. 9).

We start by defining the communicator that is used to calculate the average of the neighbors. Note that this is a square array of n^2 communicators, because we need a distinct one for each neighborhood. As in previous examples, the variables belonging to each role are given in parentheses after the role declaration. The special cases along the boundaries are omitted for brevity. Next comes the communicator used for the termination detection. In this case, there is one communicator and a square array of $n \times n$ roles.

The program itself creates the n^2 processes, and starts the iterations. The communicators were declared before the processes are spawned, so they are shared by all. The variables declared within the process blocks (`old`, `new`, etc.) are local and private. In each iteration, every process engages in five interactions with its neighbors in an unspecified order; this is expressed by the **multi_enrole** construct. The system is free to perform these interactions in any way it finds convenient. Then the processes compute their change and engage in the global interaction to check the termination condition. The **enrole** (and **multi_enrole**) directives accept a variable number of parameters. The first, after the `@`, identifies the communicator, interaction, and role. The rest (between parentheses) are substituted for the formal parameters.

This example is very fine-grained in order to show a lot of communicators in a short piece of code. Such fine-grain interactions place a heavy burden in the compiler and run-time system that have to implement them efficiently. Programs with less frequent interactions are obviously simpler. Implementation issues are discussed in the next section.

6 Implementation Strategy

No communicator language has been implemented yet, but the implementation strategy is quite clear. Many parts of the implementation can be based on previous research in various areas of parallel computing. This section reviews known algorithms and results that can be put to use, and identifies those aspects of the implementation that require further research.

An implementation of communicators has to contend with two major issues: the coordination of interaction performance, and the implementation of the interaction bodies. We review both in turn.

6.1 Coordinating Interaction Performances

Interactions can be performed when processes enrole in all their roles. However, a process may enrole in a number of roles, possibly belonging to different communicators and interactions. The problem is therefore one of instantiating a certain enrolement. This must be coordinated with other processes, so that indeed all the roles of one specific interaction are instantiated.

The difficulty of doing this is illustrated by fig. 10. This is a bipartite graph, with processes on one side and roles on the other. Arcs denote the enrolement of processes in roles. A process with two or more outgoing arcs is using a **multi_** or **select_enrole**. Take process #1 for example. It can immediately cause interaction `I1` to be performed, because this interaction only has one role. But that would prevent interaction `I2` from being performed, frustrating processes 2, 4, and 5. Interaction `I3`, on the other hand, cannot be performed in any case, even though all of its roles

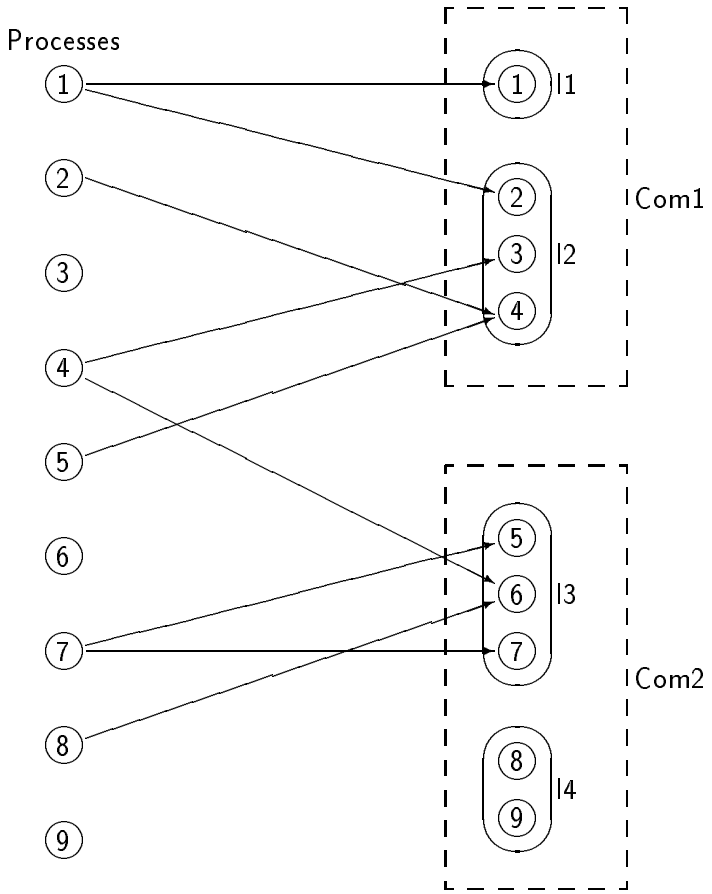

Figure 10: *Bipartite graph representing enrolement of processes in roles.*

have ready processes. the reason is that roles 5 and 7 have the same process enroled, but a process can only play one role when the interaction is performed.

Simple Interaction Patterns

In some cases there is actually no need to coordinate between enrolements from distinct processes. These cases can be recognized at compile time. The compiler can then generate simple code to handle the performance of interactions, rather than generating code for the general case. For example, if the processes only use the **enrole** instruction, and do not use **multi_** or **select_enrole**, then each process is only willing to participate in one specific interaction. Starting the performance of an interaction is then reduced to a simple barrier synchronization of the enroling processes.

Simple interaction patterns that can be handled at compile time can also be found when non-deterministic enrolement is used. For example, the **multi_enrole** used in the PDE solver of fig. 9 is repeated by all the processes in the system. Thus it allows the square array of processes to by tiled by +-shaped tiles of five adjacent processes, representing a set of `update` interactions from distinct `neighborhood` communicators that can be performed simultaneously. Using five shifted versions of this tiling pattern, all the interactions are performed. If the compiler recognizes this

possibility, it can determine the order in which each process should enrole to the five interactions it participates in, and again the actual initiation of the performances of these interactions is reduced to a simple barrier synchronization. Note that the compiler can use a brute-force algorithm to create the schedule, as the overhead is paid only at compile time and does not influence the run time.

The General Case

In the general case, the coordination of interaction performances is an instance of the "committee coordination" problem [9, chap. 14]. This archetypical problem involves a number of professors, each of whom is a member in a number of committees. A professor is occasionally willing to participate in a meeting of any committee in which he is a member[6]. The problem is to schedule committee meetings when all their members are willing, under the obvious restriction that no two committees with a common member may convene at the same time. In our case, each process is a professor, and each interaction is a committee. The problem is to know when all the roles of a interaction have been filled by ready processes, subject to the complication that processes may enrole to a number of interactions simultaneously, but participate only in one performance. Therefore the performance of one interaction may invalidate the readiness of another.

The committee coordination problem has received only scant attention in the literature. Chandy and Misra, who formally introduced it, show how committee coordination can be reduced to an instance of the dining philosophers problem [9, chap. 14]. Each committee is represented by a philosopher, and each professor by a fork. The fork is shared by all the philosophers that represent committees in which the professor is a member. A philosopher needs all his forks to eat, just as a committee needs all its members to convene; doing so prevents neighboring philosophers from eating, just as committees with common members are prevented from convening.

Bagrodia presents a family of algorithms for committee coordination [4]. The first is a centralized algorithm, where a designated manager maintains the information about the whole system. The others distribute the solution, by defining a set of managers that are responsible for different committees. The managers coordinate their activities by passing a token, or by a passing messages that implement a drinking philosophers protocol. In the context of communicators, it would be natural to associate a manager with each communicator, and let it deal with all the interactions defined in that communicator. All these algorithms (both by Chandy and Misra and by Bagrodia) are designed for the zero-order case.

Joung and Smolka have designed an algorithm for the first-order case [25]. This is a symmetrical algorithm, in which any process that enroles to a certain role becomes the manager for that interaction. It then tries to secure a quorum of other processes to fill the other roles. If it fails, it becomes idle and waits for some other process to initiate the performance. The overhead of the coordination is linear in the number of processes that may potentially enrole. While this is a promising result, the overhead is still large enough to motivate research into the automatic identification of simpler special cases, like those mentioned above.

[6] As pointed out by Joung and Smolka, the membership of professors in committees is known in advance in the original formulation. This represents zero-order multiparty interactions, but not first-order ones [25]. To capture the complications of first-order multiparty interactions (including communicators), professors must be allowed to join various committees on a whim.

It should be noted that the above algorithms do not cover all the aspects of communicator implementation. Specifically, with communicators there is the added difficulty of ensuring mutual exclusion between interactions from the same communicator, and knowing about disabled interactions. However, this only requires a small amount of additional information relative to the information required to deal with enrolement.

Mutual Exclusion Between Interactions

An additional restriction on the performance of interactions is that interactions belonging to the same communicator must be performed with mutual exclusion. This does not necessarily mean that they must actually be performed one after the other. It is enough to coordinate the performances so that the final effect is as if they were serialized. Thus performances of communicator interactions can use the same concurrency control techniques that were developed to guarantee the serializability of transactions on databases [30].

In some cases, simple or no concurrency control is sufficient. Communicators may be classified into two types: those with internal state and those that are stateless. Internal state consists of communicator global variables and disabled interactions, so it is easy to identify stateless communicators at compile time. An interesting point about stateless communicators is that the operations in one interaction have absolutely no effect on other interactions or on subsequent performances of the same interaction. Therefore the interactions do not have to be executed in a mutually exclusive manner. The communicator can be shared by a large number of processes, and service disjoint subsets of them concurrently.

The communicators with internal state can be further classified into two types, based on their interactions. The simple case involves communicators with interactions that may be categorized as "reading" or "writing". Reading interaction are those that do not assign to communicator global variables, and do not disable any interaction. However, they may enable interactions. Writing interactions are those that do assign to global variables and do disable interactions. For example, The `match` interaction in the `paralation` communicator of fig. 6 is writing, because it saves the found mapping in a global array. the `move` interaction, on the other hand, is reading. Many `move` interactions may be performed simultaneously, using the same stored mapping. The `end_region` interaction in the fuzzy barrier example of fig. 8 is also a reading interaction. Thus processes do not need to be serialized when they exit the barrier region.

The importance of this distinction is that the interactions can then be performed according to a readers-writers locking scheme [20, 27]: any number of reading interactions may be performed simultaneously, while only writing interactions require mutual exclusion. Note that the identification of the interactions as reading or writing is part of the implementation, and so is the readers-writers protocol. The programmer is shielded by the communicator abstraction from having to deal with these issues himself.

The other type of communicators with internal state have general interactions (alternatively, all interactions are writing). In this case, database concurrency control techniques may be used. This approach has been proposed in the past for the implementation of shared memory [39] and for the implementation of shared abstract data types [41].

6.2 Implementing Interactions

Once the required processes have enroled and committed themselves to the performance of a certain interaction, the body of the interaction has to be executed. This can be done serially by one of the processes, or in parallel by some or all of them. A methodology for choosing a single process to execute the common code was presented by Ramesh [32].

Using only one process might seem wasteful at first glance. After all, we know that all the participating processes are available when the interaction commences. However, some or even most of them may not be executing. It is certainly plausible that there would be more processes than processors in the system. In this case, processes that arrive early would probably be suspended while waiting for the performance of an interaction. Causing them to be scheduled simultaneously to execute the interaction might be more trouble than it is worth. However, the possibility of executing the interaction by a set of parallel processes should not be ruled out.

Recall that the body of interactions is specified in a conventional serial programming language. Therefore a parallel execution must be based on the automatic parallelization of this code. Luckily, a large body of research has been done on this problem. The basic approach is as follows. First, a dataflow graph of the required computation is generated. This graph is then partitioned into modules that may execute in parallel. Arcs contained within such a module dictate the order of the operations in it, whereas arcs between modules indicate that interprocessor communication is needed. Finally, the modules are mapped to processors, or in our case, to processes [29, 12, 37].

While significant progress has been made in the field of automatic parallelization in recent years, not all codes can be parallelized efficiently. However, it is felt that this approach may be quite useful in the context of communicators. The reason is that communicators foster a programming style in which communication between processes is localized in the bodies of interactions. Assuming that the program and algorithm are well structured, this should lead to structured and modular communication patterns. Such code is easier to parallelize than the general code that is found in the implementation of numerical algorithms.

Finally, it is not reasonable to expect an automatic parallelization tool to be perfect in every case. The programmer has to be reasonable, and express the interaction body in the most easily parallelizable manner. For example, the `sum` interaction from the `scan` communicator in fig 5 may be written in two equivalent ways. The first is

```
proc[1].out = 0
for i := 2 to n
   proc[i]out := proc[i-1].out + proc[i-1].in
```

This is very concise, but gives the impression of being inherently serial. An alternative form is

```
for i := 1 to n
   proc[i].out = 0
   for j := 1 to i-1
      proc[i]out := proc[i].out + proc[j].in
```

While this is slightly more cumbersome, it makes the available parallelism completely transparent.

7 Comparison with Related Work

The preceding section showed that certain parts in the implementation of communicators can be identified with previous research. This section reviews previous work that is related to the concept of communicators as a whole.

The concept of sequential processes that interact through synchronous communication was introduced by Hoare in his work on CSP [23]. However, CSP only allows pairwise interactions. Several extensions to multiparty interactions have been proposed over the years. The most direct is *CSPS*, proposed by Roman and Day [34], which introduces a notation that allows subsets of processes to synchronize; in effect, this is simply a barrier synchronization. However, there is no multiparty data transfer.

Francez et. al. introduced *scripts* as a method for abstracting patterns of communication with more than two participants [18]. However, this was not considered an addition to the language but rather a high level description that would be translated into the base language. Many properties of scripts were therefore left to be defined by the language being used. The *shared actions* of Ramesh and Mehndiratta represent a similar approach; their emphasis is on the transformation into a distributed implementation [33]. Charlesworth introduced the *compact*, a language constructs that is used to establish a communication pact between any number of processes; the resulting interaction is called a *multiway rendezvous* [10]. He shows that the support provided by CSP and other languages is inadequate for the implementation of multiway rendezvous, and suggests extensions. All of these proposals are close to the style and spirit of CSP, whereas communicators depart from that style. For example, the notion of different interactions that are associated with the same communicator is absent from the other proposals. Thus there is no option of saving internal state from one interaction to another (unless this is done externally by the participating processes). Naturally, the option to disable an interaction also does not exist.

The language *Raddle*, presented by Forman, also provides a capability for design with multiparty interactions, which should later be refined and decomposed manually to use the pairwise interactions provided by the hardware [16]. This is combined with another level of modularization, namely *teams* of roles. Only roles in the same team can participate in a multiparty interaction; interactions among teams are essentially mediated by remote procedure calls. While this turns the original zero-order multiparty interactions into more abstract first-order interactions, it also adds the restriction of making it necessary to define an active process in a team that is only used to abstract a pattern of communication. Adding unnecessary processes like this complicates the program and places a heavier load on the run-time system. The *IP* language, designed by Francez and Forman [17], suffers from the same drawback. On the other hand, it adds the possibility for superimposition of different activities among the same set of processes, e.g. termination detection superimposed on a distributed computation.

Back and Kurki-Suonio take the concept of multiparty interactions to the extreme. In their *action systems*, the whole program is expressed in terms of possible interactions [3]. The model of execution is that interactions are performed continuously in sequence until none are enabled any more. This is implemented in the *DisCo* language [26], which also includes object-oriented features. The *data-parallel vector model* of Blelloch is similar, except for not using nondeterminism [7]. Communicators take a much more conservative approach, based on sequential processes that interact.

Many of the abovementioned proposals for multiparty interactions are intended for use in the

design of distributed reactive systems[7]; much research is devoted to procedures for the subsequent refinement of the multiparty interactions, replacing them with pairwise interactions. The processes represent real-world entities, and the program implements their interactions and reactions to outside stimuli. Nondeterminism is used to cope with the unpredictable nature of the environment. Communicators, on the other hand, are a structuring mechanism for regular (transformational) parallel programs. Their expressive power is intentionally limited, so as to enable automatic compilation and efficient run-time support. The intended environment is tightly-coupled parallel computers, not distributed systems. Therefore transformations into pairwise interactions may be unnecessary. Nondeterminism is used only as part of the abstraction, when the exact sequencing is not important and may be left to the system.

It should be noted that communicators are not completely new with respect to using multiparty interactions in transformational parallel programs. For one thing, some systems support primitives such as multicast [11, 6]. In addition, there are initial attempts at more expressive constructs incorporated into a language. Examples include the CAB construct [28] and paralations [36]. However, these are much more restricted than communicators in the sense of limiting the pattern of data transfer. As shown above, communicators can be used to express any pattern. On the other hand, there are some patterns of interaction between processes that cannot be expressed by the communicator notation.

Another important distinction differentiates between communicators and other language extensions that introduce parallelism into an existing base language. For example, the Linda notation is an add-on that is independent and orthogonal to the base language. All it needs is a run-time library that is linked with the source code. With communicators, however, there are certain requirements on the language. The use of communicators is blended with the mechanism for expressing the parallelism, and static scoping rules are used to determine the accessibility of processes to communicators. Thus communicators are integrated into the language, and their support requires a specialized compiler.

8 Conclusions

Communicators may be viewed as a study of how to apply the principles of ADTs to parallel programming in new ways. The conventional object-based model was augmented with two main features to create them. The first was interactions, which are actually procedures with multiple entry points that serve to synchronize the participating processes. The second was the ability to disable and enable interactions, so as to control the execution of other processes. It seems that most useful interaction patterns may be expressed by this augmented model. The model opens the way for abstraction, improved portability, and easier programming.

Communicators also provide a framework in which different interactions can be described and compared. The diversity of interactions that can be expressed testifies to the richness of the design space. The features that are needed in the interactions also serve to expose the inherent properties of various interaction patterns and to define a certain hierarchy among them.

Programming with communicators exposes a whole spectrum of possible program structures, from sequential processes with simple and primitive interactions, to a sequence of complex interactions with nothing in between. This wide range of possibilities indicates that communicators have

[7] The distinction between reactive an transformational computations is due to Pnueli [31].

a high expressive power, and can be used for different programming styles.

The detailed design and implementation of communicators are yet to be done. However, it is already possible to envision certain benefits of using them. These benefits include:

- *Easier programming.* Communicators provide a higher level of abstraction than most contemporary languages, by not limiting the programmer to pairwise interactions between processes.
- *Better program structure.* Using communicators, there is a crisp separation between independent serial blocks of code and parallel interactions. The localization of the parallel code improves the possibility of using automatic parallelization and verification techniques.
- *Efficient execution.* The simple features incorporated in communicators alleviate the need for additional processes that are necessary for the interactions but were not part of the original problem solution; such processes are needed in most other proposals for multiparty interactions. The possibility of disabling interactions provides a uniform blocking mechanism.
- *Improved portability and reusability.* A library of common communicators could be defined and implemented on various systems.

Much of the promise of communicators can be traced to the fact that they allow users to delegate various responsibilities to the system. Such delegation of authority requires two issues to be resolved: first, one must define an interface by which the user can express what he wants the system to do. Second, system algorithms that implement this interface must be designed and implemented. For example, semaphores were introduced as a means to tell the system that a process is waiting for an abstract event, and they were easily implemented by maintaining a queue of waiting processes. Communicators provide a much richer interface, and allow the user to tell the system about more involved patterns of interprocess interaction. While the study of implementation is still in its infancy, there are indications that efficient implementation would be possible by bringing together previous research on many diverse issues in parallel programming.

References

[1] S. Ahuja, N. Carriero, and D. Gelernter, "*Linda and friends*". *Computer* **19(8)**, pp. 26–34, Aug 1986.

[2] G. R. Andrews and F. B. Schneider, "*Concepts and notations for concurrent programming*". *ACM Comput. Surv.* **15(1)**, pp. 3–43, Mar 1983.

[3] R. J. R. Back and R. Kurki-Suonio, "*Distributed cooperation with action systems*". *ACM Trans. Prog. Lang. & Syst.* **10(4)**, pp. 513–554, Oct 1988.

[4] R. Bagrodia, "*Process synchronization: design and performance evaluation of distributed algorithms*". *IEEE Trans. Softw. Eng.* **15(9)**, pp. 1053–1065, Sep 1989.

[5] P. Banerjee, M. H. Jones, and J. S. Sargent, "*Parallel simulated annealing algorithms for cell placement on hypercube multiprocessors*". *IEEE Trans. Parallel & Distributed Syst.* **1(1)**, pp. 91–106, Jan 1990.

[6] K. P. Birman and T. A. Joseph, "*Reliable communication in the presence of failures*". *ACM Trans. Comput. Syst.* **5(1)**, pp. 47–76, Feb 1987.

[7] G. E. Blelloch, *Vector Models for Data-Parallel Computing.* MIT Press, 1990.

[8] E. C. Bronson, T. L. Casavant, and L. H. Jamieson, "*Experimental application-driven architecture analysis of an SIMD/MIMD parallel processing system*". *IEEE Trans. Parallel & Distributed Syst.* **1(2)**, pp. 195–205, Apr 1990.

[9] M. K. Chandy and J. Misra, *Parallel Program Design: A Foundation.* Addison-Wesley, 1987.

[10] A. Charlesworth, "*The multiway rendezvous*". *ACM Trans. Prog. Lang. & Syst.* **9(3)**, pp. 350–366, Jul 1987.

[11] D. R. Cheriton and W. Zwaenepoel, "*Distributed process groups in the V kernel*". *ACM Trans. Comput. Syst.* **3(2)**, pp. 77–107, May 1985.

[12] R. Cytron, M. Hind, and W. Hsieh, "*Automatic generation of DAG parallelism*". In *Proc. Conf. Prog. Lang. Design & Implementation*, pp. 54–68, SIGPLAN, Jun 1989.

[13] E. W. Dijkstra, "*Co-operating sequential processes*". In *Programming Languages*, F. Genuys (ed.), pp. 43–112, Academic Press, 1968.

[14] M. Evangelist, N. Francez, and S. Katz, "*Multiparty interactions for interprocess communication and synchronization*". *IEEE Trans. Softw. Eng.* **15(11)**, pp. 1417–1426, Nov 1989.

[15] M. J. Flynn, "*Very high-speed computing systems*". *Proceedings of the IEEE* **54(12)**, pp. 1901–1909, Dec 1966.

[16] I. R. Forman, "*Design by decomposition of multiparty interactions in Raddle87*". In 5th *Intl. Workshop Software Specification & Design*, pp. 2–10, May 1989.

[17] N. Francez and I. R. Forman, "*Interacting Processes: a language for coordinated distributed programming*". In 5th *Jerusalem Conf. Information Technology*, pp. 146–161, IEEE Computer Society Press, Oct 1990.

[18] N. Francez, B. Hailpern, and G. Taubenfeld, "*Script:* a *communication abstraction mechanism*". *Sci. Comput. Programming* **6(1)**, pp. 35–88, Jan 1986.

[19] P. O. Frederickson, R. E. Jones, and B. T. Smith, "*Synchronization and control of parallel algorithms*". *Parallel Comput.* **2(3)**, pp. 255–264, 1985.

[20] A. Gottlieb, B. Lubachevsky, and L. Rudolph, "*Basic techniques for the efficient coordination of very large numbers of cooperating sequential processes*". *ACM Trans. Prog. Lang. & Syst.* **5(2)**, pp. 164–189, Apr 1983.

[21] R. Gupta, "*The fuzzy barrier: a mechanism for high speed synchronization of processors*". In 3rd *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 54–63, Apr 1989.

[22] W. D. Hillis and G. L. Steele, Jr., "*Data parallel algorithms*". *Comm. ACM* **29(12)**, pp. 1170–1183, Dec 1986.

[23] C. A. R. Hoare, *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[24] C. A. R. Hoare, "*Monitors: an operating system structuring concept*". *Comm. ACM* **17(10)**, pp. 549–557, Oct 1974.

[25] Y-J. Joung and S. A. Smolka, "*Coordinating first-order multiparty interactions*". In 18th *Ann. Symp. Principles of Programming Languages*, pp. 209–220, Jan 1991.

[26] R. Kurki-Suonio and H-M. Järvinen, "*Action system approach to the specification and design of distributed systems*". In 5th *Intl. Workshop Software Specification & Design*, pp. 34–40, May 1989.

[27] J. M. Mellor-Crummey and M. L. Scott, "*Synchronization without contention*". In 4th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 269–278, Apr 1991.

[28] P. A. Nelson and L. Snyder, "*Programming paradigms for nonshared memory parallel computers*". In *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon, and R. J. Douglass (eds.), pp. 3–20, MIT Press, 1987.

[29] D. A. Padua, D. J. Kuck, and D. H. Lawrie, "*High speed multiprocessors and compilation techniques*". *IEEE Trans. Comput.* **C-29**, pp. 763–776, Sep 1980.

[30] C. H. Papadimitriou, "*The serializability of concurrent database updates*". *J. ACM* **26(4)**, pp. 631–653, Oct 1979.

[31] A. Pnueli, "*Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends*". In *Current Trends in Concurrency*, J. W. de Bakker, W-P. de Roever, and G. Rozenberg (eds.), pp. 510–584, Springer-Verlag, 1986. Lecture Notes in computer Science, Vol. 224.

[32] S. Ramesh, "*A new and efficient implementation of multiprocess synchronization*". In *Parallel Arch. & Lang. Europe*, vol. II, pp. 387–401, Springer-Verlag, 1987. Lecture Notes in Computer Science Vol. 259.

[33] S. Ramesh and S. L. Mehndiratta, "*A methodology for developing distributed programs*". *IEEE Trans. Softw. Eng.* **SE-13(8)**, pp. 967–976, Aug 1987.

[34] G-C. Roman and M. S. Day, "*Multifaceted distributed systems specification using processes and event synchronization*". In 7th *Intl. Conf. Softw. Eng.*, pp. 44–55, Mar 1984.

[35] K. W. Ryu and J. JáJá, "*Efficient algorithms for list ranking and for solving graph problems on the hypercube*". *IEEE Trans. Parallel & Distributed Syst.* **1(1)**, pp. 83–90, Jan 1990.

[36] G. W. Sabot, *The Paralation Model: Architecture-Independent Parallel Programming.* MIT Press, 1988.

[37] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, 1989.

[38] J. T. Schwartz, "*Ultracomputers*". *ACM Trans. Prog. Lang. & Syst.* **2(4)**, pp. 484–521, Oct 1980.

[39] D. Shasha and M. Snir, "*Efficient and correct execution of parallel programs that share memory*". *ACM Trans. Prog. Lang. & Syst.* **10(2)**, pp. 282–312, Apr 1988.

[40] D. Vrsalovic, Z. Segall, D. Siewiorek, F. Gregoretti, E. Caplan, C. Fineman, S. Kravitz, T. Lehr, and M. Russinovich, "*MPC - multiprocessor C language for consistent abstract shared data type paradigms*". In 22nd *Ann. Hawaii Intl. Conf. System Sciences*, vol. I, pp. 171–180, Jan 1989.

[41] W. E. Weihl, "*Commutativity-based concurrency control for abstract data types*". *IEEE Trans. Comput.* **37(12)**, pp. 1488–1505, Dec 1988.

[42] A. Yonezawa and M. Tokoro (eds.), *Object-Oriented Concurrent Programming.* MIT Press, 1987.