International Journal of Parallel Programming, Vol. 21, No. 4, 1992

Attempting Guards in Parallel: A Data Flow Approach to Execute Generalized Guarded Commands[1]

R. Govindarajan,[2] S. Yu,[3] and V. S. Lakshmanan[4]

Received March 1991; Revised March 1993

Earlier approaches to execute generalized alternative/repetitive commands of Communicating Sequential Processes (CSP) attempt the selection of guards in a sequential order. Also, these implementations are based on either shared meomry or message passing multiprocessor systems. In contrast, we propose an implementation of generalized guarded commands using the data-driven model of computation. A significant feature of our implementation is that it attempts the selection of the guards of a process in parallel. We prove that our implementation is faithful to the semantics of the generalized guarded commands. Further, we have simulated the implementation using discrete-event simulation and measured various performance parameters. The measured parameters are helpful in establishing the fairness of our implementation and its superiority, in terms of efficiency and the parallelism exploited, over other implementations. The simulation study is also helpful in identifying various issues that affect the performance of our implementation. Based on this study, we have proposed an adaptive algorithm which dynamically tunes the extent of parallelism in the implementation to achieve an optimum level of performance.

KEY WORDS: Communicating sequential processes; data-driven evaluation; fairness; generalized guarded commands; liveness; performance evaluation; safety.

[1] The first author's work was supported by a MICRONET, Network Centers of Excellence, research grant. Support for the second author is from the NSERC (Canada) Grant. The last author's work was supported by grants from NSERC (Canada) and FCAR (Quebec).

[2] Department of Electrical Engineering, McGill University, Montreal, H3A 2A7, Canada., govindr@pike.ee.mcgill.ca.

[3] Department of Computer Science, University of Western Ontario, London, N6A 5B7, Canada., syu@uwocsd.uwo.ca.

[4] Department of Computer Science, Concordia University, Montreal, H3G 1M8, Canada., laks@cs.concordia.ca.

1. INTRODUCTION

Communicating Sequential Processes (CSP), as proposed by Hoare,[1] does not allow output commands to appear in the alternative/repetitive commands. The inability to use output guards in the guarded commands necessitates additional signals and greatly constrains the expressibility of the language.[1-5] This is observed especially in the bounded buffer and the dining philosopher programs. Following these arguments, the guarded commands of CSP have been generalized to allow output commands to appear in them. While such a generalization is easy to perceive, an implementation is quite involved and requires reaching an agreement among the communicating processes. To understand this, consider the following example.

```
Process  P1                Process P2                 Process P3

...   ...   ...            ...   ...   ...            ...   ...   ...
*[ true;P2!x1 --> S1       *[ true;P1?y1 --> S3       *[ true;P1!z1 --> S5
[] true;P3?x2 --> S2       [] true;P3!y2 --> S4       [] true;P2?z2 --> S6
]                          ]                          ]
...   ...   ...            ...   ...   ...            ...   ...   ...
```

In one specific iteration either (i) P1 and P2 execute statements S1 and S3 respectively after agreeing to select their first guards, or (ii) P2 and P3 agree to select their second guards and execute statements S4 and S6 respectively, or (iii) P3 and P1 agree to select, respectively, the first and second guards. Let P1 and P2 make the agreement (or rendezvous). Then it is clear that P3 should not initiate a rendezvous with either P1 or P2. Thus the agreement is not only between the processes that rendezvous, P1 and P2 in this example, but also with other processes, P3 in this example, with which P1 and P2 can potentially communicate in the alternative/repetitive command. Thus a global agreement among mutually communicating processes is required to select a guard in the generalized guarded command.

Proposing an implementation for the generalized guarded command has been of constant interest to the research community. Starting from the restricted implementation of Silberschatz,[6] several solutions have been reported[3,6-8] over a decade, including the recent ones.[5,9,10] The work reported in Refs. 5, 7, 8, and 10 demonstrate a progressive improvement in terms of the number of messages communicated to reach an agreement. Fujimoto's solution[9] is based on a shared memory model unlike the earlier ones which employ a loosely-coupled architecture. A commonality observed in all these solutions is they are based on the conventional von Neumann framework. Also, the selection of guards in a particular process has so far been done sequentially. As a consequence, in reaching an agreement some guards of a process are favored by virtue of the selected order of execution of the guards. The set of favored guards in a process need not be the same over different agreements of the guarded command. Nonetheless, there is a preferred order of execution of the guards in a single agreement, and hence the selection process cannot be called a 'pure' nondeterministic choice.

In contrast to the earlier proposals, we use data-driven evaluation(11) as the basic computation model for implementing generalized guarded commands. The implementation parallelizes the selection of guards in a process which can significantly reduce the execution time of an alternative/repetitive command. Also, as the guards of a process are attempted possibly concurrently without any preferred order of execution, the selection of guards in a process is purely non-deterministic. Furthermore, in our implementation, Processing Elements (PEs) do not busy wait for synchronization of guarded commands.

This paper is organized as follows. In the next section, we briefly introduce data-driven evaluation and discuss the issues related to implementing generalized guarded commands. Section 3 describes the implementation scheme. The subsequent section establishes the correctness of the implementation. A simulation model for our implementation has been developed to obtain certain performance parameters to demonstrate the efficiency and fairness of the implementation in Section 5. Based on the simulation results the implementation is tuned to further improve its performance. A comparison is drawn between our implementation and the existing ones in Section 6.

2. BACKGROUND

Data-driven evaluation has been chosen as the model of computation for our implementation for the following two reasons. Firstly, data flow model does not impose any order (other than what is dictated by data dependency) on the execution. The guards in a process can thus be concurrently attempted in a natural way. Further, fine grain asynchronous concurrency exploited by data flow machines ensures both inter- and intra-process parallelisms in a CSP program. To make this paper self-contained some preliminaries on data flow computation are in order. The reader is referred to Refs. 11 and 12 for more details.

2.1. Data-Driven Evaluation

In data-driven evaluation(11) an instruction may be executed as soon as all its operands are ready. Thus the flow of data determines the execution sequence of a program in this model. Moreover, the exploitation of parallelism is at the instruction or fine-grain level.

The simplest form of a data flow program consists of an acyclic directed graph, called the data flow graph.[13] In a data flow graph, the nodes or actors are the low-level data flow instructions. Each node in the graph has a unique node address. Tokens carrying data values traverse the arcs between the operators. Formally,

Definition 2.1. An actor is any asynchronous computation representing a specified operation on the values carried by the tokens present on the input arcs.

Definition 2.2. An actor is said to be enabled as soon as all its input arcs contain at least one token.

A set of basic data flow actors has been defined in Ref. 13. In addition to arithmetic, boolean, and logical operations, the set includes special actors such as the True Gate, False Gate, Switch and Merge actors to express conditional and iterative program structures. For a description of these actors and their operational semantics the reader is referred to Ref. 13.

An enabled actor is executed by a processing element which performs the operation specified by the actor. When the execution completes, tokens are removed from the input arcs and results are produced on the output arcs. Thus, in the data flow evaluation, control is purely data-driven: the availability of tokens triggers the computation.

Observation 2.1. Any basic data flow actor, when enabled, trivially completes its execution in a finite time.

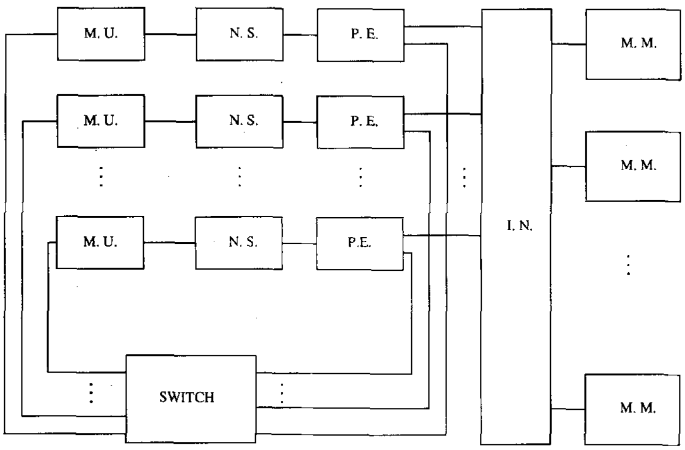
In an acyclic data flow graph each operator is instantiated just once during the execution of a program. However, practical programs require features like iterations and function calls. Such requirements make a data flow graph cyclic. In a cyclic graph, an operator inside a loop or a function body is activated as many times as that body is iterated or called. To ensure the proper execution of reentrant routines, it is necessary to distinguish the various tokens destined to a single node but belonging to different instantiations.

Static data flow systems[14] do not permit concurrent execution of two instantiations of a reentrant code. The triggering rule is augmented to ensure that two tokens are never placed on the same arc at any instant. This rule restricts the parallelism and asynchrony. In dynamic data flow models, also called tagged-data flow model, an environment tag is associated with each token.[12,15] The tag identifies the context. Compared to static systems, dynamic data flow models are more general and can exploit higher parallelism and asynchrony. Further, they do not suffer from control overheads due to token acknowledgment.[14] However, the increased cost in generating and matching tags is the price paid for these advantages.

The execution of a data flow graph on the multi-ring Manchester data flow machine (refer to Ref. 12) is described here.

In Fig. 1, the Switch Unit acts as an interface between the host computer and the data flow system. The Switch Unit routes tokens to the Matching Unit in one of the rings using a simple hash function. The hash function ensures that all tokens destined to a node are routed to the same ring. The function of the Matching Unit is to synchronize tokens arriving on left and right input arcs of a two-input node (known as dyadic tokens). A dyadic token, on entering the Matching Unit, searches for its partner in the Matching Store. The Matching Store stores the dyadic tokens awaiting their partners. A match is successful if the node address and the environment value of the incoming token match with those of a stored token. On a successful match, the partner is removed from the Matching Store and a group-token is formed. If the match is unsuccessful, then the incoming token is stored in the Matching Store, and awaits its partner.

Tokens leading to a single input node (called monadic tokens) perform a dummy operation in the Matching Unit. As no matching is



M. U. -- Matching Unit
N. S. -- Node Store
P. E. -- Processing Element
I. N. -- Interconnection Network
M. M. -- Memory Module

Fig. 1. The Multi-ring manchester machine.

required for a monadic token, the incoming token itself is sent as the group-token. The group-token formed by the Matching Unit is sent to the Node Store Unit. A group-token extracts the node information (such as opcode, destination address, type of result tokens) from the Node Store and forms an executable-token. The processing element executes the operation specified by the executable-token and sends the output token to the Switch Unit. The execution of the data flow graph proceeds this way.

2.2. Implementing CSP

The proposed implementation is based on the multi-ring data flow architecture described in the previous section, though the implementation is suitable for any data flow model. [This includes even the static data flow machines. This is because, the semantics of the repetitive commands of CSP does not allow concurrent execution of loop instantiations, and hence fits well in the static data flow framework.] The underlying architecture that will be assumed in this paper is similar to the Manchester multi-ring data flow machine.(12) The processing elements (or simply the processors) of the data flow machine access and modify certain shared variables used in the implementation which are stored in a shared memory. The processing elements and the shared memory are connected by means of an interconnection network. The shared memory consists of many memory modules which are low-order interleaved to form a contiguous address space. The processors can access the multiple memory modules concurrently, though access to a single module is restricted to one processor at a time. Thus the shared memory system exhibits an EREW (Exclusive Read and Exclusive Write) model in the sense that a read or write access to any memory cell is exclusive. Furthermore, we assume that the interconnection network that arbitrates requests to memory modules is assumed to be *fair* to all requests. That is, it cannot postpone a memory request indefinitely. Such *fair* arbitration can be realized by simply using a *first-in-first-out* order. Besides these two properties, the topology of the network does not play any role in the implementation. Therefore our implementation can be supported with any network topology, as long as the network satisfies these properties, viz. *fair* arbitration and exclusive read/write operation to individual memory blocks. In the simulation experiments, the topology of the network is varied and thus the effect of network on the performance of our implementation will be studied.

Viewed from the processing elements, the shared memory organization resembles any shared memory multiprocessor. We will use two synchronization primitives, namely *fetch and increment*(16) and *test and set* in our implementation. It should be noted that two synchronization primitives are used in the description only for reasons of simplicity and they can be replaced by a generalized synchronization primitive such as *fetch and Φ*.[16] The synchronization primitives are performed as atomic operations. The realization of such synchronization primitives is common in shared memory machines, and therefore is realistic to assume in our architecture.

A CSP program consists of a set of communicating processes. We propose to execute a CSP program by converting each process into a data flow graph and executing the resulting graphs on the data flow machine. Since the processes of a CSP program must be executed concurrently, the data flow graphs corresponding to the processes are executed concurrently in our implementation. In this paper we restrict our attention to generalized guarded commands. Implementing simple commands of CSP, namely skip, assignment, and parallel commands are straightforward. Also, work on implementing CSP with constrained guarded commands (i.e., allowing only input commands in the guards) has already been reported.[17] The implementation of the generalized guarded commands will be discussed in the following section. Before proceeding to the implementation we establish certain terminology that will be followed in the rest of this paper.

2.2.1. Terminology

In a distributed program, the processes are given distinct process identifiers, called *process-ids.* Each invocation of an alternative or repetitive command is referred to as a *transaction.* A unique identifier, called *trans-id,* is generated by concatenating the process identifier (of the process which invokes the transaction) and a sequence number. The *trans-ids* are *totally ordered.* A transaction contains a number of guarded commands, each guard having a distinct index. We use the guard index to refer to the guard itself whenever there is no confusion. Consider the guarded command b_i; $c_i \rightarrow S$ in a process P_i, where b_i and c_i are the Boolean and I/O guards respectively. The I/O guard c_i is said to be *matching* with an I/O guard c_j of P_j, if (i) P_j is the process addressed by c_i, and (ii) the type of the variable used in c_i matches with that used in c_j. If c_i and c_j communicate a signal (rather than a value) then the signal name used in both guards should be the same. Further, we say c_i and c_j are *compatible* if one of them is an *input* command and the other is an output command. From the definition of matching and compatible guards, we make the following observation.

Observation 2.2. The relations 'matching' and 'compatible' are symmetric.

If the I/O parts of the guards g_i and g_j in processes P_i and P_j, respectively, are *matching* and *compatible*, then g_i is a potential *communicating complement* (or simply *a complement*) of g_j and *vice-versa.* Also, we can say that P_i is a potential *complement* of P_j and *vice-versa.*

As there can be many guarded commands in a transaction, there are many potential communicating complements. Hence implementing a generalized alternative or repetitive command is equivalent to reaching an agreement among the potential communicating processes and electing one of the many communicating complements. If the processes P_i and P_j agree to communicate, then we say P_i and P_j rendezvous. Equivalently, P_i is *committed* to P_j and *vice-versa.*

2.3. Related Issues

In the first place, a data flow implementation forbids a process from owning a Processing Element (PE). Also, since we allow parallel execution within a single process, various instructions (constructs) belonging to a process can reside at a number of PEs. As a consequence, a single (process) state cannot be assigned to a process. This is in contrast to the existing implementations(3,5,7,9) which rely heavily on the existence of a unique state for each process.

Secondly, the execution of I/O commands in CSP warrants the synchronization of the communicating processes. If a PE executing a communication construct is allowed to 'busy wait,' then this may lead to a deadlock situation: a situation where each PE executing a communication construct is waiting for synchronization, but the corresponding communicating complement is denied a PE due to nonavailability. Thus, to avoid deadlocks, busy waiting in PEs must be prohibited. That is, a PE must be set free while the communication construct executed by it (PE) waits for synchronization. The details of the communication construct need to be stored in the shared memory to enable the execution of the construct later when its complement becomes ready.

3. THE IMPLEMENTATION

First we present an informal description of our implementation of the alternative command. The subsequent subsection presents a precise description of the implementation. In Section 3.4, we extend our implementation to handle repetitive and simple I/O commands.

3.1. Informal Description

Consider the execution of an alternative command in a process P_l. First a unique *trans-id*, t_l, is assigned to the transaction. Then the guards of the transaction are attempted, possibly concurrently, to establish a rendezvous. For any guard g_l in t_l, if P_r is its complement process, attempting the guard g_l results in the following actions.

1. An entry indicating the willingness of g_l to communicate with its complement, is written in a shared data structure for P_l.
2. The data structure for P_r is searched to find out if P_r is willing to communicate with P_l.
3. The absence of a matching and compatible guard indicates the complement process is not ready at this moment to communicate with P_l. Therefore the attempt to establish an agreement fails, terminating the actions of the guard g_l. On the other hand—i.e., if a match is found—then an attempt to make the processes commit to the agreement is undertaken. This is accomplished by acquiring exclusive access to the process variables and modifying the process variables within a mutually exclusive section, thereby guaranteeing that a transaction commits to only one rendezvous. If the attempt to commit the respective processes (or equivalently, transactions) fails, then the guard g_l is unable to establish the agreement and therefore the actions performed by the guard are terminated.

If one of the guards of the transaction t_l is successful in reaching the agreement, then that guard is responsible for triggering the execution of the respective guarded statement (statement following the guard) in the local process P_l as well as its complement guarded statement in process P_r. The implementation guarantees that only one guard of a transaction can succeed as the process variables are modified in a critical section. If none of the guards are successful, then the transaction reaches a state of 'waiting,' and eventually some other transaction will force a rendezvous. In Section 4, we prove that our implementation ensures safety—that exactly one guard in a transaction establishes a rendezvous with exactly one complement transaction—and liveness—whenever there is a possibility of a rendezvous it will eventually take place. [The formal definitions of safety and liveness are presented in Section 4.]

3.2. Alternative Command

The following variables are associated with each process P_l.

Seq-Num (P_l): This integer variable *Seq-Num* (P_l) is used for generating a unique *trans-id* for each transaction in process P_l. *Seq-Num* (P_l) is initialized to 0 at the commencement of the execution of the program and is incremented atomically for each transaction invoked in P_l.

Active (P_l): This variable stores the *trans-id* of the active transaction. The precise definition of an active transaction will be given shortly and it may be seen at that point that having an active transaction in P_l is equivalent to having a nonzero value in *Active* (P_l).

For each transaction t_l, the following variables are stored in the shared memory.

Committed (t_l): A boolean variable indicating whether transaction t_l is committed to some other transaction. Once *Committed* (t_l) is true, no transaction can establish a rendezvous with t_l. To ensure that a transaction commits to exactly one transaction, the *Committed* variable is set to *True* in a critical section.

Excl (t_l): This flag is used to enter the critical section described earlier in a mutually exclusive fashion. The *Excl* flag is set atomically using the synchronization primitive 'test and set.'

G-list (t_l): *G-list* (t_l) is the data structure where an entry is written when each guard attempts to establish an agreement. The data structure is an array of linked lists indexed on the processes. For each process P_r, irrespective of whether P_l wishes to comunicate to P_r or not, there is an entry *G-list* (t_l, P_r). [The arguments for indexing *G-list* (t_l) on the processes rather than on the guard indices are: (i) more than one guard can address the same process; (ii) when the complement process checks the *G-list* (t_l) all the guards (of t_l) which are ready and willing to communicate with the complement process need to be tried; and (iii) the linked list representation helps faster access to the available guards compared to a scheme in which the *G-list* (t_l) is indexed by the guard indices.] We will use *G-list* (t_l) and *G-list* (t_l, P_r), respectively, to refer to the *G-list* of t_l and the list of entries corresponding to process P_r in the *G-list* of t_l. Associated with *G-list* (t_l) is a *reference count* indicating how many guards of t_l have not been attempted. The reference count is used for the purpose of garbage collection which will be explained in Section 3.3.

Initially all entries of *G-list* (t_l) will contain nil pointers, and the reference count will be set to the number of guards in the alternative command. When each guarded command g_l is attempted for execution, a new entry will be linked to *G-list* (t_l, P_r), where P_r is the remote process addressed by g_l. The entry stores a tuple, $\langle g_l, c \rangle$, where g_l is the guard index of the local guard and c stores the remote process name, the signal name, and a flag to identify whether the command is input or output.

Definition 3.1. A guard g_l in a transaction t_l is said to be **ready** if there is an entry $\langle g_l, c \rangle$ in *G-list* (t_l, P_r), for some remote process P_r with which g_l wants to communicate.

We are now ready to define the term *active* transaction.

Definition 3.2. A process P_l is said to have an **active transaction** if Active (P_l) has a nonzero value t_l.

In a CSP program, two guarded commands (where one is not nested in the other) of a process are executed sequentially. Thus any two rendezvouses that take place in a process does not occur concurrently. This is true even for nested guarded commands. [A brief discussion on nested guarded commands is presented in Section 3.5.] In other words, it is not possible to have multiple *simultaneously* active transactions in a process. In the data flow implementation, such a sequentialization of rendezvouses is enforced by data dependency between two transactions. In the absence of such a data dependency, sequentialization can be enforced using a dummy data dependency.

An alternative comand can be executed by converting it into a data flow graph. In the data flow graph, certain abstractions have been followed for the sake of brevity. For example, a set of input arguments is passed to a guarded statement through a single True gate. In practice a number of True gates have to be used for this purpose. Also, we have assumed unlimited fanout for each data flow actor.

A few new data flow actors have been used in our implementation. These data flow actors with their respective input and output arguments are shown in Fig. 2. The operational semantics of the new data flow actors used in our implementation is presented next. The semantics of other data flow actors is same as that presented in Ref. 13.

Get-id: An invocation of an alternative command commences with this actor. When the *Get-id* actor is invoked from a process P_l, the PE executing this actor fetches *Seq-Num* (P_l) and increments its contents by one using the *fetch and increment* primitive. A unique *trans-id*, t_l, is generated by concatenating P_l with the fetched *Seq-Num*. The *Get-id* actor resets the *Excl*(t_l) and *Committed*(t_l) to *False* and then sets *Active*(P_l) to t_l. This order is crucial to the correctness of the implementation.

The **Tryguard** actor uses the following procedures:

Store (P_r, t_l, g_l, c): This procedure links $\langle g_l, c \rangle$ to *G-list* (t_l, P_r).

Fetch (t_r, P_l): The linked list pointed by *G-list* (t_r, P_l) is fetched and returned to the PE executing the Fetch procedure.

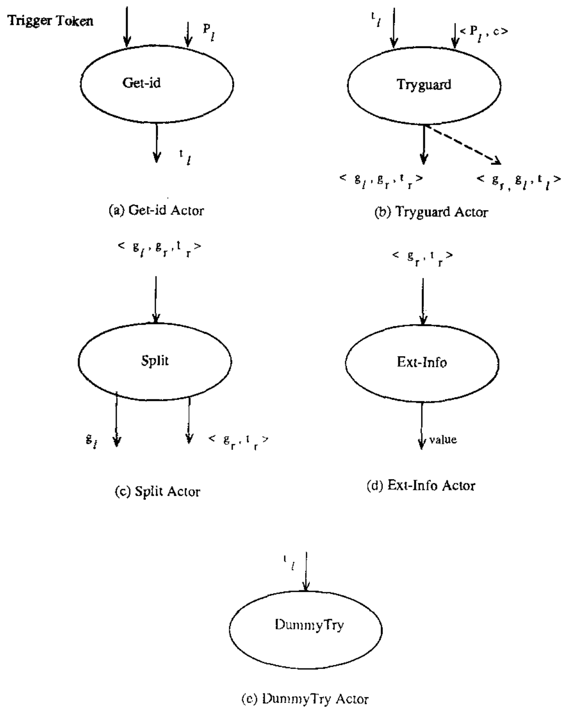


Fig. 2. New data flow actors.

Checkguard (g_l, P_r, c): This procedure is responsible for searching a matching and compatible entry for the guard g_l in *G-list* (t_r), where t_r is the active transaction in P_r. This task is accomplished by searching for a matching and compatible entry in *G-list* (t_r, P_l). When this procedure succeeds (in finding the matching and compatible guard), it returns g_r, the guard index in the remote transaction. A failure is indicated by returning a zero value.

Tryguard: If the boolean component of a guard evaluates to *True*; then the guard is attempted by executing the **Tryguard** actor. The details of the communication guard are specified by the input c as shown in Fig. 2. The execution of the **Tryguard** actor for the guard g_l in transaction t_l results in the following sequence of actions:

1. The reference count associated with *G-list* (t_l) is decremented by 1.
2. An entry $\langle g_l, c \rangle$ is appended to *G-list* (t_l, P_r) using the procedure *Store* described earlier.

3. If there is an active transaction t_r in the complement process P_r and the *Committed* variables of t_l and t_r are *False*, then *G-list* (t_r, P_l) is fetched using the *Fetch* procedure. Otherwise, the execution of the **Tryguard** actor terminates.
4. A matching and compatible guard is searched in *G-list* (t_r, P_l) using the *Checkguard* procedure. Failure to find a match results in the termination of the execution of the **Tryguard** actor.
5. When the complement guards are *ready*, the processor executing the **Tryguard** actor attempts to gain exclusive access to the *Committed* variables of the complement transactions. This is done using the *Excl* flags. Exclusive access to the *Committed* variables of the complement transactions is gained in the strict order of their *trans-id* values to avoid possible deadlocks due to cyclic dependencies.
6. If the processor is successful in gaining exclusive access, then it sets *Committed* (t_l) and *Committed* (t_r) to *True*. It sends the appropriate tokens on the output arcs as shown in Fig. 2. The variables *Active* (P_l) and *Active* (P_r) are reset to zero. If the processor is unsuccessful in gaining exclusive access, then the processor relinquishes the exclusive access by setting the corresponding *Excl* flags to *False*.

A complete description of the operational semantics of the **Tryguard** actor is presented in Fig. 3.

DummyTry: Whenever the boolean component of a guard fails, the **DummyTry**(guard) actor is executed. The execution of this actor results in decrementing the reference count associated with *G-list* (t_l). The reference count is used for the purpose of garbage collection which will be explained in Section 3.3.

Definition 3.3. A guard g_l in a transaction t_l is said to have **failed** if its boolean component evaluates to False.

Definition 3.4. A process P_r or a transaction t_r is said to be **captured** by a guard g_l if the processor executing the **Tryguard** actor for g_l is successful in gaining exclusive access to Committed (t_r).

Note that in our implementation the attempt to capture a transaction t_r takes place only if *Committed* (t_r) is *False*. Also, in this definition t_r could be t_l. This is because for establishing a rendezvous the guard g_l must capture both t_l and t_r.

```
PROCEDURE Tryguard ($t_l, g_l, c$);
(* $P_l$, $t_l$, $g_l$ : local process id, trans-id, and guard index;
(* c: communication details of the IO guard; *)
(* $P_r$, $t_r$, $g_r$ refer to remote process-id, trans-id, and guard index *)
begin
   Store ($P_r$, $t_l$, $g_l$, c); (* store an entry in G-list ($t_l$) *)
   $t_r$ := Active ($P_r$);
   if $t_r$ = 0 then skip;
   else
   begin
      fetch ($t_r$, $P_l$);
      $g_r$ := Checkguard ($g_l$, c);
      if $g_r \neq 0$ then
      begin
         $t_{min}$ := min ($t_l$, $t_r$); $t_{max}$ := max ($t_l$, $t_r$);
         flag := TRUE; release ($t_{min}$) := FALSE; release ($t_{max}$) := FALSE;
         while ( flag AND (NOT Committed ($t_{min}$)) AND (NOT Committed ($t_{max}$))) do
            flag := test_and_set (Excl ($t_{min}$))
            (* capture the transaction with a lower trans-id *)
         if (not flag) then
         begin release ($t_{min}$) := TRUE;
            if (NOT Committed ($t_{max}$)) then
            begin
               flag := true;
               while (flag AND (NOT Committed ($t_{max}$))) do
                  flag := test_and_set (Excl ($t_{max}$)); (* capture the other transaction *)
               if (not flag) then
               begin release ($t_{max}$) := TRUE;
                  Committed ($t_{min}$) := TRUE; Active ($t_{min}$) := 0;
                  Committed ($t_{max}$) := TRUE; Active ($t_{max}$) := 0;
                  output-token ($\langle g_l, g_r, t_r \rangle$, Split-Node ($t_l$)); (* output to the Split actor of $t_l$ *)
                  output-token ($\langle g_r, g_l, t_l \rangle$, Split-Node ($t_r$)); (* output to the Split actor of $t_r$ *)
               end
            end
         end
      if release ($t_{min}$) then Excl ($t_{min}$):= FALSE; (* release captured transactions, if any *)
      if release ($t_{max}$) then Excl ($t_{max}$):= FALSE;
      end
   end
end ;
```

Fig. 3. Operational semantics of **Tryguard** Actor.

Definition 3.5. A captured process or transaction is **released** by resetting the *Excl* flag of the transaction to False.

Definition 3.6. A transaction is said to be **committed** if its Commited flag is True.

Split: The **Split** actor receives a triple $\langle g_l, g_r, t_r \rangle$ as its input from a **Tryguard** actor. [Or from a *False gate* when all boolean guards of a transaction are *False.*] This actor splits the triple and outputs the value g_l on one output arc and the pair $\langle g_r, t_r \rangle$ on the other output arc. The value g_l is used to enable the appropriate guarded statement. The pair $\langle g_r, t_r \rangle$ provides the necessary information for the extraction of input data in the I/O guards whenever the guard is an input command.

Ext-Info: When the I/O guard is an input command, the corresponding **Tryguard** actor only achieves the synchronization. The actual communication (reception of data) has to be performed with the help of an **Ext-Info** (meaning, extract information) actor. As the necessary synchronization has already been achieved, the data can be input without further synchronization delay.

Lemma 3.1. Any of the newly introduced data flow actors, namely **Split**, **Get-id**, **Tryguard**, **DummyTry**, and **Ext-Info**, when enabled, will complete their execution in a finite time.

Proof. We will prove this lemma for each of these actors.

Get-id Actor: In order to prove that the **Get-id** actor completes its execution in a finite time, we just need to show that the processor executing the **Get-id** actor eventually gets access to the shared variable *Seq-Num*. This is guaranteed by the fact that the network arbitration is *fair*.

Tryguard Actor: As mentioned earlier, clearly any access to shared memory can only take a finite time. Let g_l in transaction t_l be the guard corresponding to the **Tryguard** actor. Further, let P_r be the remote process addressed by g_l. The execution of the **Tryguard** actor terminates if at least one of the following is true.

1. P_r does not have an active transaction.
2. P_r has an active transaction t_r and, for every guard g_r in t_r, either g_r is not matching and compatible with g_l or g_r is not ready.
3. P_r has an active transaction t_r, $Committed(t_r)$ is *True* and, further, g_l has not captured t_r.
4. $Committed(t_l)$ is *True* and g_l has not captured t_l.

In these cases the attempt to establish a rendezvous fails and the execution of the **Tryguard** actor terminates.

Now, suppose that none of the conditions (1)–(4) are true. That is, each of the following conditions is true.

1. P_r has an active transaction, say t_r.
2. There exists at least one guard g_r in t_r such that (a) g_r is matching and compatible with g_l, and (b) g_r is ready.
3. $Committed(t_r)$ is *False* or g_l has captured t_r.
4. $Committed(t_l)$ is *False* or g_l has captured t_l.

The logical disjunction in conditions (3) and (4) leads to four different nontrivial cases; in all four cases (1) and (2) are also true.

Case 1. *Committed*(t_l) is *False*, g_l has not captured t_l, *Committed*(t_r) is *False*, and g_l has not captured t_r. In this case, the execution of the **Tryguard** actor terminates, resulting in a rendezvous between t_l and t_r. This readily follows from the following observations.

(a) Any guard g, other than g_l, that captures t_l must eventually release it.

(b) Any guard g, other than g_l, that captures t_r must eventually release it.

(c) The number of guards in each process and the number of processes in a program are finite.

(d) The implementation ensures that any pair of complement transactions (t_1, t_2) are captured by a guard in the strict order of the *trans-ids* of t_1 and t_2.

(e) The *trans-ids* are totally ordered.

Case 2. *Committed*(t_l) is *False* and g_l has not captured t_l but g_l has captured t_r. Subsumed by Case 1.

Case 3. g_l has captured t_l and *Committed*(t_r) is *False* but g_l has not captured t_r. Subsumed by Case 1.

Case 4. g_l has captured both t_l and t_r. The proof for this case is trivial.

Split Actor: Follows from the semantics of the **Split** Actor.

Ext-Info Actor: As the **Ext-info** actor does not need to synchronize (the synchronization has already been achieved by the corresponding **Tryguard** actor), its execution will complete in a finite time.

DummyTry Actor: The action performed by this actor is decrementing the reference count of *G-list* (t_l). Since access to shared memory is guaranteed by *fair* network arbitration, the execution of a **DummyTry** actor completes in a finite time. ▌

We now describe our implementation in the data flow framework. The execution of an alternative command is started by sending a trigger token (refer to Fig. 4) to the **Get-id** node in the data flow graph. The execution of the **Get-id** actor assigns a unique *trans-id* for the alternative command. It also resets the *Committed* and *Excl* flags. The boolean guards b_1 and b_2 are evaluated, possibly concurrently. The boolean values b_1 and b_2 control the two **Switch** actors. The value arriving at the input of the **Switch** actor is sent either to a **Tryguard** actor or to a **DummyTry** actor depending

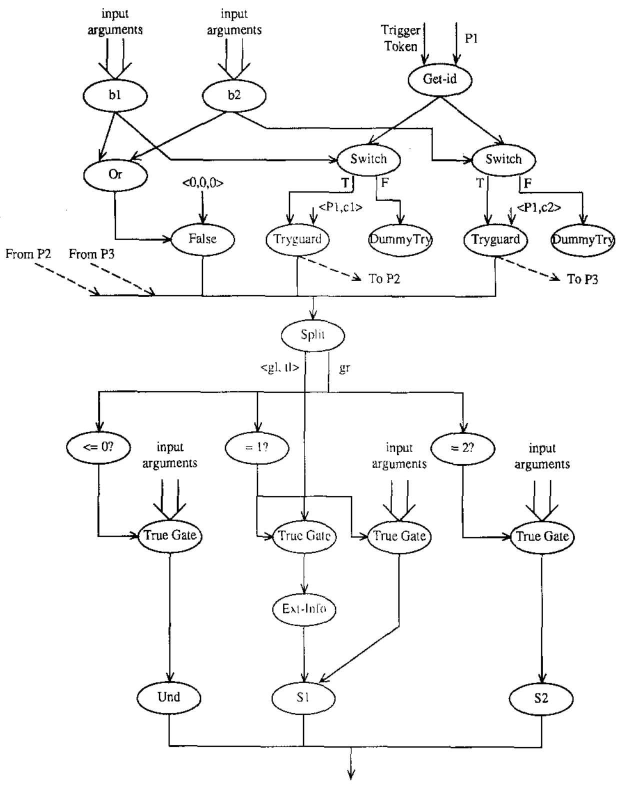


Fig. 4. Data flow graph for $[b1;P2?x \rightarrow S1 \square b2;P3!y \rightarrow S2]$.

on whether the boolean value is *True* or *False*. Thus, if the boolean component of a guard evaluates to *True*, then the corresponding **Tryguard** actor receives the *trans-id* value sent by the **Get-id** node. Otherwise, i.e., when the boolean guard evaluates to *False*, the **DummyTry** actor gets the *trans-id* value and decrements the reference count associated with the corresponding *G-list*. As there is no data dependency between the guards of a transaction (or the respective **Tryguard** guard actors), they could be executed in any order, possibly concurrently. Though more than one **Tryguard** actor of a transaction can attempt to establish a rendezvous, only one of them can succeed, as the **Tryguard** actors must capture the respective transactions before setting their *Committed* variables to *True*. The successful **Tryguard** actor sends a token containing $\langle g_l, g_r, t_r \rangle$ to the **Split** node of the local transaction and $\langle g_r, g_l, t_l \rangle$ to the **Split** node of the remote transaction. The local guard name in the token is useful to identify which guard was successful in establishing the rendezvous and enable the corresponding guarded statement. The guard index and the *trans-id* of the remote process are used by input guards to perform the communication (by the **Ext-Info** actor).

The data flow graphs corresponding to other processes are also executed concurrently on the data flow machine. Even if all the **Tryguard** actors of a transaction fail to establish a rendezvous, one of its many potential complements will eventually force a rendezvous and send the necessary token to the **Split** node of the transaction.

From this discussion we can define the establishment of a rendezvous as:

Definition 3.7. A guard g_l of a transaction t_l selects (rendezvouses with) g_r of t_r if the **Split** node of t_l receives a non-zero tuple $\langle g_l, g_r, t_r \rangle$ on its input arc.

In this definition, we exclude the case where the **Split** actor received the tuple $\langle 0, 0, 0 \rangle$. This corresponds to the case where all boolean guards in the alternative command have failed, and no rendezvous takes place.

In a successful rendezvous between a pair guards (or transactions) g_l and g_r (or between t_l and t_r), either g_l or g_r (but not both) must have been successful in capturing (see Definition 3.4) the transactions t_l and t_r. Suppose that g_l was successful in capturing t_l and t_r. In this case, we say that g_l is the *declaring guard* and g_r is the *accepting guard* of this rendezvous. Also. we can say that g_l initiates the rendezvous. Extending this terminology to transactions, t_l is the *declaring transaction* and t_r is the *accepting transaction.*

If all the boolean guards of an alternative command fail, then a run-time error results. The data flow actor *UND* is used for this purpose. It may be observed that in our implementation more than one guard of a process can be executing the respective **Tryguard** actors in parallel; some of them may even succeed in finding their respective partners. The order in which the **Tryguard** actors of a process are executed is purely data-driven. Even though multiple guards of a transaction may concurrently attempt for the rendezvous, only one guard will succeed. This will be established in Section 4.

3.3. Garbage Collection

In our implementation, the memory space associated with the variables for each transaction is reclaimed after the transaction commits to a rendezvous. In our implementation it is possible for a transaction to commit even before some of its guards become *ready*. If the process variables are reclaimed immediately after the transaction commits, then some guard of the transaction may become *ready* at a later point in time. This, in turn, would cause an error if an attempt is made to access the process variables that are already reclaimed. To avoid this problem, the reclamation is deferred until (i) every guard in the transaction has become *ready* or *failed* and (ii) the *Committed* variable is *True*. The reference count associated with the corresponding *G-list* can be used to check condition (i). Thus, if the reference count is zero and the *Committed* variable is *True*, the process variables can be reclaimed.

3.4. Repetitive and Simple I/O Commands

The scheme for alternative command can be extended to implement repetitive and simple I/O commands. The data flow graph for the repetitive command is shown in Fig. 5. The repetitive command is executed loop sequentially, circulating the input arguments and the trigger token at the

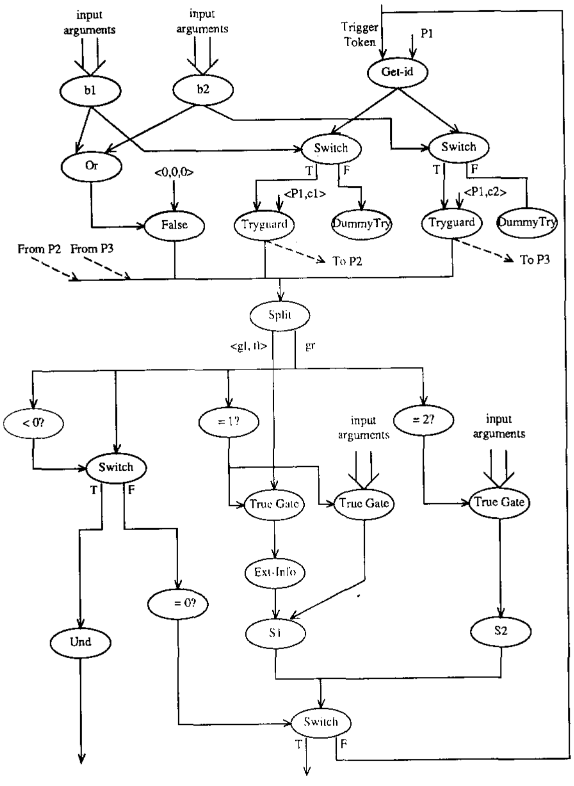


Fig. 5. Data flow graph for $*[b1;P2?x \rightarrow S1 \square b2;P3!y \rightarrow S2]$.

end of execution of each repetition. [The recirculation of input arguments is not shown in Fig. 5 for the sake of clarity.] Such a sequentialization is necessitated by the semantics of the repetitive command. The execution of a repetitive command terminates when all the guards in that command fails. This is achieved by sending a tuple $\langle 0, 0, 0 \rangle$ to the **Split** node as shown in Fig. 5.

Finally, an input command $P_r?x$ generated by a process P_l is implemented by considering it as an alternative command

$$\text{True}; P_r?x \rightarrow \text{skip}.$$

Similarly, an output command $P_r!\,y$ is translated to:

$$\text{True}; P_r!\,y \rightarrow \text{skip}.$$

The reason for considering a simple I/O command as an alternative command in the implementation is as follows. Consider the situation in which a guard of some transaction has as its complement a simple I/O command. This guard will never succeed as the proposed implementation searches the matching compatible guards only in other alternative/repetitive commands. To take care of this situation, either the search (for matching and compatible guard) should be extended to simple I/O commands as well or the simple I/O commands be made as alternative/repetitive commands. We chose the latter as it makes our implementation simple and uniform.

3.5. Remarks

In this subsection we discuss how our implementation handles certain special cases.

3.5.1. Nested Guarded Commands

Consider a program in which a process has a nested guarded command as:

```
Process P1
...   ...   ...
*[ true;P2!x1 -->
         [ true;P2?x3 --> S3;
         [] true;P3?x4 --> S4;
         ]
 ]
...   ...   ...
```

In this case there is only one guard in the repetitive command. Whenever P1 rendezvous (with P2), it has to execute an alternative command, which itself is a transaction. However, the transaction (due to the outer repetitive command) has already reached an agreement, and hence is no longer *active*. Thus even in the case of nested guarded commands, the transactions in a process take place sequentially. Further, as mentioned earlier, the transactions corresponding to a repetitive guarded command also take place sequentially. Thus it is never the case that a process has multiple *simultaneously* active transactions.

3.5.2. Self Communication

Consider, the case in which a process tries to communicate to itself in a guarded command. For example,

```
Process P1
  ...   ...   ...
  [ true;P1!x1 --> S1;
  [] true;P1?x2 --> S2;
   ]
  ...   ...   ...
```

This program is erroneous since process P1 must agree to select both guards of the alternative command for the communication to happen. Such an error could easily be detected by a compiler. However, it is still possible to write such self-communicating processes when a family of processes are defined, as in:

```
Process P[1..n]
  ...   ...   ...
  [  (i:=1 .. n) true;P[i]!x1 --> S1;
  [] (i:=1 .. n) true;P[i]?x2 --> S2;
   ]
  ...   ...   ...
```

In this case, each process $P[i]$ tries to communicate with every other process $P[j]$, for $j = 1, \ldots, n$, including $j = i$. To the best of our knowledge, there does not exist an accepted semantics, and several possible interpretations could be given for this program. For example, this process can *always* result in an error. Alternatively it may be allowed to execute as long as $P[i]$ rendezvouses with $P[j]$, for $j \neq i$. We believe the latter is more appropriate particularly since the guards are chosen nondeterministically.

Our implementation handles one part of this interpretation, viz., when $P[i]$ communicates with a different $P[j]$. To abort the rendezvous of a process with itself, the semantics of the **Tryguard** actor need to be modified as follows. After a guard has captured the transaction which corresponds to the smaller *trans-id*, it should check whether the *trans-ids* of the local and the remote transactions are equal. If not, the execution of the **Tryguard** actor proceeds as before. When the *trans-ids* are equal, a transaction is trying to communicate to itself. In this case, the **Tryguard** actor sets the *Committed* flag to *True*, resets the *Active* variable to zero, and sends a tuple $\langle -1, -1, -1 \rangle$ to its **Split** node along the solid arc. No token is sent along the dotted arc. The *Excl* flag is also set to *True*. An input tuple of $\langle -1, -1, -1 \rangle$ at the **Split** node corresponds to an error and the program can be aborted using the **Und** actor as shown in Fig. 4. With this modification, our implementation can now handle self-communicating processes. Further, for preciseness, Definition 3.7 is modified to exclude the case where the **Split** actor receives tuples with negative values.

In the following section, we establish the correctness of our implementation. Even though in the proof we do not explicitly consider various special cases, such as nested guarded commands, and self-communicating processes, it can be seen that the arguments given in the proof hold for these special cases as well.

4. PROOF OF CORRECTNESS

The correctness of our implementation can be established by proving that during the potentially infinite execution, all processes (of the application program) and their interplay maintain 'safety' and 'liveness.'(18,19) The first property, safety, means any rendezvous that occurs is correct. Liveness ensures two processes which should rendezvous eventually will, provided either of them does not rendezvous with any other process. In the following two subsections we define the safety and liveness properties and establish that our implementation satisfies both of them.

To prove safety and liveness properties, we make the following assumptions.

(i) The functional units of the data flow system are free from failures.

(ii) An enabled data flow actor will eventually be scheduled for execution in one of the processing elements.

(iii) No message communicated between any two functional units of the data flow machine is lost.

(iv) A processing element of the data flow machine cannot be denied access to the shared memory for an indefinitely long period.

(v) The memory system behaves as an EREW model. Whenever there are simultaneous access requests to a single memory cell, the network resolves the conflict by means of some arbitration logic and allows the accesses to proceed in some fair order.

Lastly, we use the notation $g_l(t_l)$ to represent the guard g_l of a transaction t_l.

4.1. Safety

The safety property requires every rendezvous that takes place must obey the semantics of the generalized guarded commands. By this we mean that each instance of an alternative/repetitive command must select exactly one guarded statement in the alternative/repetitive command. This means that every transaction rendezvouses with exactly one transaction. The selected guarded statements in the respective transactions must have guards that are both matching and compatible, so that communication can take place between the processes. Finally, the agreement between the two transactions must be mutual. This can be formalized as:

Definition 4.1. The rendezvous between the guard g_l in transaction t_l and the guard g_r in transaction t_r is **safe** if whenever g_l selects g_r, the following conditions are satisfied:

1. g_r selects g_l,
2. g_l and g_r are matching and compatible, and
3. no transaction, other than t_r, can select t_l.

An implementation is safe if all the rendezvouses between guards that it allows are safe in this sense.

Theorem 4.1. The implementation described in the previous section is safe.

Proof. Suppose that a guard $g_l(t_l)$ selects $g_r(t_r)$. Then we will prove the conditions (1)–(3) in Definition 4.1 hold.

Part 1: Here we need to prove the rendezvous is mutual. There are two cases to consider.

Case 1. If g_l is the declarer, then by definition g_l must have successfully captured t_l and t_r. Then the **Tryguard** actor for g_l sends the tuple $\langle g_r, g_l, t_l \rangle$ to the **Split** node of t_r.

Case 2. If g_r is the declarer, then its **Tryguard** actor must have sent the tuple $\langle g_l, g_r, t_r \rangle$ to the **Split** node of t_l. It can be seen from Fig. 3 that the same **Tryguard** actor would also send the tuple $\langle g_r, g_l, t_l \rangle$ to the **Split** node of t_r.

Thus in either case, the **Split** node of t_r receives the tuple $\langle g_r, g_l, t_l \rangle$. From Definition 3.7, it follows that $g_r(t_r)$ selects $g_l(t_l)$.

Part 2. Here also there are two cases to consider.

Case 1. If g_l is the declarer of the rendezvous, then the execution of its **Tryguard** actor ensures g_l and g_r are matching and compatible (see Fig. 3).

Case 2. If g_l is the acceptor of the rendezvous, then g_r must be the declarer. From Part 1 of the theorem, we know that g_r selects g_l. Now using Case 1 of Part 2, g_r and g_l are matching and compatible. By Observation 2.2 matching and compatible are symmetric relations.

Part 3. We have to prove that there exists no other transaction t_k where $k \neq l$ and $k \neq r$ such that t_k rendezvouses with t_l. This reduces to showing that the **Split** node of t_l never receives two tuples. From Part 1 of the theorem, any rendezvous that takes place is mutual and the **Split** nodes of the declarer and the acceptor receive one tuple each. Therefore to prove the theorem it remains to show that a **Split** node receives at most one tuple.

From the semantics of the **Tryguard** actor (refer to Fig. 3), a **Split** node cannot receive more than one tuple on any given arc. Notice that the **Split** node of a transaction t_l can receive tuples via two arcs only when two different guards of any two transactions, not necessarily distinct, have captured the transaction t_l. This is impossible since capturing the transactions takes place in a mutual exclusion region. Mutual exclusion is guaranteed as the *Excl* flag is set by the *test and set* synchronization primitive and, further, read/write access to memory is exclusive. Once a rendezvous takes place, the *Committed* flags of the respective transactions t_l and t_r are set to *True*. After this, no other guard can capture t_l. Therefore the **Split** node of t_l receives exactly one tuple. ▌

In the following subsection we establish that our implementation is live.

4.2. Liveness

Under liveness, we need to ensure the following two things. First every rendezvous that takes place leads to the completion of the respective transactions which participated in the rendezvous. Also, liveness must capture the notion that the implementation does not prevent any rendezvous that is allowed by the semantics of the guarded commands. More precisely, if two transactions can possibly rendezvous, and neither of these two transactions rendezvouses with a third transaction, then the implementation must ensure a rendezvous between the first two transactions takes place in a finite time. The condition, 'neither of these two transactions rendezvouses with a third transaction,' carefully avoids 'fairness' issues by not requiring that every possible rendezvous should take place. The discussion on fairness is deferred to Section 4.3.

This intuition on liveness can be formalized as:

Definition 4.2. An implementation of generalized guarded commands is said to be **live** if

1. for every rendezvous between $g_l(t_l)$ with $g_r(t_r)$, the transactions t_l and t_r complete their execution and terminate, provided the statements following the guards g_l and g_r terminate.
2. if t_l and t_r are two transactions with guards g_l and g_r, respectively, such that g_l and g_r are matching and compatible guards, the boolean components of g_l and g_r evaluate to True, and neither t_l nor t_r rendezvouses with a third transaction, then eventually t_l rendezvouses with t_r.

Unlike the safety property, liveness is inherently temporal in nature. Therefore one needs to include 'time' in the analysis and proofs of liveness properties. We do this by explicitly including time as a parameter in the definition of various events in our implementation.

4.2.1. Notation

We say that an event *has occurred by time T* to mean that the event occurred at some time $T' \leqslant T$.

enabled (g, T): g has been enabled by time T.

ready (g, T): By time T, the guard g has become ready.

accessed (g, t_r, T): By time T, *G-list* (t_r) has been accessed by the **Tryguard** actor for g.

captured (g, t_l, T): By time T, the guard g has captured the transaction t_l.

selected (g_1, g_2, T): By time T, guard g_1 has selected guard g_2.

selected (t_1, t_2, T): By time T, the transaction t_1 has selected t_2. (Recall that by Part 1 of Theorem 4.1, *selected* $(t_1, t_2, T) \Leftrightarrow$ *selected* (t_2, t_1, T).)

It may be observed that all predicates defined here, including *captured*, are *monotonic* in the sense that if an atom is true at time T, then it is true at all time $T' \geqslant T$.

Theorem 4.2. The implementation discussed in the previous section is live.

Proof. We need to prove that conditions 1 and 2 in Definition 4.2 hold.

Part 1: If $g_l(t_l)$ selects $g_r(t_r)$, then we will show that the transactions t_l and t_r complete their execution and terminate. Since g_l selects g_r, by Theorem 4.1 (1), g_r must select g_l. This means that the **Split** nodes of t_l and t_r must respectively receive the tuples $\langle g_l, g_r, t_r\rangle$ and $\langle g_r, g_l, t_l\rangle$. From Observation 2.1 and Lemma 3.1 the execution of all data flow actors used in our implementation terminates in a finite time. From the data flow graph shown in Fig. 4, it is easy to see that once the **Split** actor of t_l receives the tuple $\langle g_l, g_r, t_r\rangle$, the data flow graph for the guarded statement corresponding to g_l will receive its necessary input tokens and therefore will get executed in a finite time, and by Lemma 3.1 its execution will terminate. Finally, it is given, in Definition 4.2, that the guarded statement following g_l also terminates. Thus the execution of t_l itself terminates in a finite time. A symmetric argument holds for the remote transaction t_r.

Part 2: We first prove the following claim.

Claim 4.1. Suppose that $g_l(t_l)$ and $g_r(t_r)$ are matching and compatible guards, and their boolean components evaluate to True. Then there exists a (suitably large) time T such that selected (t_l, t_k, T) and selected (t_r, t_k, T) are false, $\forall k$, and

$$\exists(T_l', T_l'', T_r', T_r'')[(T_l' < T_l'') \wedge (T_l'' \leqslant T) \wedge (T_r' < T_r'') \wedge (T_r'' \leqslant T)] \wedge$$
$$[(\neg accessed(g_l, t_r, T_l') \wedge accessed(g_l, t_r, T_l'') \wedge ready(g_r, T_l'))$$
$$\vee (\neg accessed(g_r, t_l, T_r') \wedge accessed(g_r, t_l, T_r'') \wedge ready(g_l, T_r'))]$$

Proof. Since the boolean components of g_l and g_r evaluate to *True*, it can be seen from the data flow graph shown in Fig. 4 that the **Tryguard** actors corresponding to g_l and g_r receive the necessary input tokens on their input arcs. Therefore there exists some time, say T_l^1 such that the **Tryguard** actor for g_l has been enabled by time T_l^1. Similarly, the **Tryguard** actor for g_r has been enabled by time T_r^1. Let us choose T_l^1 and T_r^1 to be the minimum values such that *enabled* (g_l, T_l^1) and *enabled*(g_r, T_r^1) hold. By the underlying assumption of the data flow model (see Section 4), any enabled data flow actor will eventually be executed on one of the processors. From the semantics of the **Tryguard** actor, it follows that first g_l becomes *ready* by some time $T_l^2 \geqslant T_l^1$. Formalizing these arguments, we have

$$(\exists T_l^1)\, enabled(g_l, T_l^1) \wedge (\exists T_l^2)\, ready(g_l, T_l^2) \wedge (T_l^2 \geqslant T_l^1)$$

Let T_l^2 be the minimum value such that this condition holds. Also, from Fig. 3,

$$ready(g_l, T_l^2) \Rightarrow (\exists T_l^3)\, accessed(g_l, t_r, T_l^3) \wedge (T_l^3 \geqslant T_l^2)$$

Without loss of generality, we choose T_l^3 to be the smallest such time. Similarly, for g_r

$$(\exists T_r^1)\, enabled(g_r, T_r^1) \wedge (\exists T_r^2)\, ready(g_r, T_r^2) \wedge (T_r^2 \geqslant T_r^1) \text{ and}$$

$$ready(g_r, T_r^2) \Rightarrow (\exists T_r^3)\, accessed(g_r, t_l, T_r^3) \wedge (T_r^3 \geqslant T_r^2)$$

Once again, T_r^2 and T_r^3 are chosen to be the minimal values required to satisfy these conditions.

Figure 6 represents the partial order in the chronology of various events discussed here. Since access to *G*-list (t_l) is required for both the events *ready* (g_l, T_l^2) and *accessed*(g_r, t_l, T_r^3), and by the EREW nature of the memory system, we have

$$T_l^2 \neq T_r^3$$

Using a symmetric argument,

$$T_r^2 \neq T_l^3$$

Now, to prove the claim, it is sufficient to show that

$$(T_l^2 < T_r^3) \vee (T_r^2 < T_l^3)$$

is true. To show this, notice that

$$(T_l^2 > T_r^3) \wedge (T_r^2 > T_l^3)$$

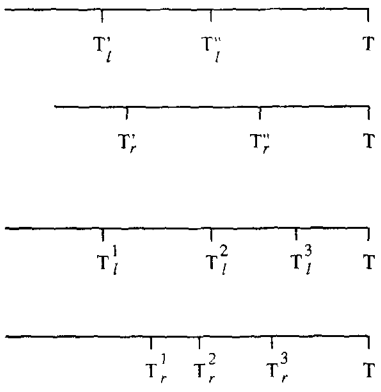


Fig. 6. Chronology of events.

is impossible, since

$$(T_l^2 > T_r^3) \wedge (T_r^2 > T_l^3) \Rightarrow [(T_l^3 \geqslant T_l^2 > T_r^3) \wedge (T_r^3 \geqslant T_r^2 > T_l^3)]$$

Hence, either $(T_l^3 > T_r^2)$ or $(T_r^3 > T_l^2)$. Suppose $T_l^3 > T_r^2$. Since T_l^3 is the minimum for which $accessed(g_l, t_r, T_l^3)$ is true, we can say, $(\neg accessed(g_l, t_r, T_r^2) \wedge accessed(g_l, t_r, T_l^3))$ holds. So choose, $T_l' = T_r^2$ and $T_l'' = T_l^3$. Clearly, we have,

$$\neg accessed(g_l, t_r, T_l') \wedge accessed(g_l, t_r, T_l'') \wedge ready(g_r, T_l')$$

The argument for the case $T_r^3 > T_l^2$ is symmetric. To complete the proof, notice that we can choose T to be any time greater than $max(T_l'', T_r'')$. Thus we have,

$$\exists(T_l', T_l'', T_r', T_r'')[(T_l' < T_l'') \wedge (T_l'' \leqslant T) \wedge (T_r' < T_r'') \wedge (T_r'' \leqslant T)] \wedge$$
$$[(\neg accessed(g_l, t_r, T_l') \wedge accessed(g_l, t_r, T_l'') \wedge ready(g_r, T_l'))$$
$$\vee (\neg accessed(g_r, t_l, T_r') \wedge accessed(g_r, t_l, T_r'') \wedge ready(g_l, T_r'))] \blacksquare$$

Proof (of Part 2). First we will prove the theorem for the special case when there is exactly one pair of matching and compatible guards in the transactions t_l and t_r. It is given that neither t_l nor t_r rendezvouses with another transaction t_k. Therefore we have,

$$(\forall T)(\forall t_k)[(t_k \neq t_l) \wedge (t_k \neq t_r)$$
$$\rightarrow (\neg selected(t_l, t_k, T) \wedge (\neg selected(t_r, t_k, T))] \qquad \text{(i)}$$

Also we know that g_l and g_r are matching and compatible. Now, suppose *selected* (t_l, t_r, T), and hence *selected* (t_r, t_l, T), is true for some time T, then we are done. So assume that there is a (suitably large) time T' such that *selected* (t_l, t_r, T') is false. Now, using (i) we have,

$$(\forall T')(\forall t_k)[\neg selected(t_l, t_k, T') \wedge \neg selected(t_r, t_k, T')]$$

By Claim 4.1, we have

$$(\exists T_l^1)[accessed(g_l, t_r, T_l^1) \wedge ready(g_r, T_l^1)]$$
$$\vee (\exists T_r^1)[accessed(g_r, t_l, T_r^1) \wedge ready(g_l, T_r^1)] \qquad \text{(ii)}$$

for some $T_l^1, T_r^1 \leqslant T'$.

We will prove the theorem for each of these two cases.

Case 1. Supposed $accessed(g_l, t_r, T_l^1) \wedge ready(g_r, T_l^1)$. There are two cases to consider here. Either $t_l < t_r$ or $t_l > t_r$.

Case 1.1. Let $t_l < t_r$. By (i) we know that any transaction t_k, $(t_k \neq t_l \wedge t_k \neq t_r)$ that captures t_l must eventually release t_l. Further, there are only a finite number of guards and a finite number of processes in a program. From these arguments, and from $(accessed(g_l, t_r, T_l^1) \wedge ready(g_r, T_l^1))$, we have

$$(\exists T_r^2)[captured(g_r, t_l, T_r^2) \wedge (T_r^2 \geqslant T_r^1) \wedge (T_r^2 \leqslant T')]$$
$$\vee\ (\exists T_l^2)[captured(g_l, t_l, T_l^2) \wedge (T_l^2 \geqslant T_l^1) \wedge (T_l^2 \leqslant T')]$$

The two subcases to consider here are:

Case 1.1.1. Let $captured(g_r, t_l, T_r^2)$ hold. Now, t_r has captured t_l and will proceed to capture t_r. Any other transaction t_k $(t_k \neq t_l)$ that captures t_r must eventually release it, since *selected* (t_r, t_k, T) is false for all T. Also, since $t_l < t_r$ and t_l has been captured by g_r, t_l cannot capture t_r. Therefore, $(\exists T_r^3)(T_r^3 \geqslant T_r^2) \wedge (T_r^3 < T')$ such that $captured(g_r, t_r, T_r^3)$ is also true. From Fig. 3 and by the monotonicity of events,

$$captured(g_r, t_l, T_r^2) \wedge captured(g_r, t_r, T_r^3) \Rightarrow selected(t_r, t_l, T_r^4)$$
$$\text{for some } (T_r^4 \geqslant T_r^3 \geqslant T_r^2) \text{ and } (T_r^4 \geqslant T')$$

Case 1.1.2. Suppose *captured* (g_l, t_l, T_l^2). Using arguments similar to those used in Case 1.1.1., we can say $captured(g_l, t_r, T_l^3)$ holds for some $T_l^3 > T_l^2$. Also,

$$captured(g_l, t_l, T_l^2) \wedge captured(g_l, t_r, T_l^3) \Rightarrow selected(t_l, t_r, T_l^4)$$
$$\text{for some } (T_l^4 \geqslant T_l^3) \wedge (T_l^4 \geqslant T')$$

Note that in capturing transactions, a deadlock due to cyclic dependencies is avoided as the transactions are captured in the strict order of the *trans-ids*. By Part 1 of Theorem 4.1,

$$selected(t_l, t_r, T_l^4) \Leftrightarrow selected(t_r, t_l, T_l^4)$$

Case 1.2. Suppose $t_l > t_r$. The proof for this case is similar to Case 1.1 except that the order in which the transactions are captured by the guards is reversed.

Case 2. Let $accessed(g_r, t_l, T_r^1) \wedge ready(g_l, T_r^1)$. A symmetric argument to that used for Case 1 can be used to prove this case.

Now, for the case when there is more than one pair of matching and compatible guards between t_l and t_r, condition (ii) has many possibilities, two for each pair of matching and compatible guards. Using arguments similar to those used in Case 1.1, it can be easily proved that *selected* (t_l, t_r, T_l^4) is true in each of these cases. ▮

4.3. Fairness

Besides safety and liveness, in general, it is desirable to support 'fairness' in implementations involving nondeterministic choices. In particular two kinds of fairness, *weak* and *strong* fairness, have been defined in literature.[9,20]

Definition 4.3. An implementation of guarded commands is **weakly fair** if it can be guaranteed that during an infinitely repetitive execution, a guard that remains continuously available (i.e., its boolean guard evaluates to True and its complement process is ready to communicate) will eventually rendezvous.

Definition 4.4. If it can be guaranteed that a guard which is available infinitely often (though not necessarily continuously) will eventually rendezvous, then the implementation is **strongly fair**.

In our implementation, the guards of a single process can be executed in any order, possibly concurrently. In the absence of a *total* order of execution of the guards, analytically proving fairness (either *weak* or *strong* fairness) is a difficult task. However, using the simulation approach, we obtain certain performance parameters which are indicative of the extent of fairness of our implementation. The details are discussed in Section 5.2.

5. PERFORMANCE EVALUATION USING SIMULATION

In this section, we evaluate the performance of our implementation using discrete-event simulation.

5.1. Simulation Details

In order to obtain empirical performance metrics of our implementation, we developed a simulator for the underlying architecture. Various functional units of the architecture, such as the Switch Unit, the Matching Unit, the Node Store Unit, the Processing Elements, the Interconnection Network, and the Shared Memory system, have been simulated at a functional level. In the simulation, each of these units take a specified time to process a token. The execution time associated with each functional unit is based on Refs. 21 and 22. In Table I, we list these execution times expressed in time units. All functional units, except the processing elements, take a constant time for processing. For the processing element, the execution time varies depending on the operation performed. This is simulated by assuming an exponential execution time with a mean value of 20 time units. Complex data flow actors, namely **Tryguard, Get-id, Ext-Info** data flow actors, require additional execution time, over and above the normal execution time, for accessing shared memory. This has also been faithfuly simulated in the simulator.

The topology of the interconnection network is left as a parameter in the simulation. We have experimented with a bus, a multistage network, and a crossbar network. Buffering of messages is assumed whenever there is a contention in the network path. The multistage network causes logarithmic delay, while the bus and crossbar networks cause a constant delay. The simulator is written in Pascal and run on a Unix platform. Application programs represented as data flow graphs are input to the simulator through a simple coding scheme. The simulator executes the application program, with each functional unit performing the token processing as would be done in an actual machine. As mentioned earlier, each functional unit takes a specified processing time, and remains *busy* during this period in the simulation. Tokens arriving at the functional unit during this period are queued in the input queue associated with that functional

Table I. Execution Time of Functional Units

Functional Unit	Execution Time
Switch Unit	2
Matching Unit	
Successful Match	8
Unsuccessful Match	16
Node Store Unit	2
Processing Element	20
Shared Memory Unit	
Memory Access Time	2
Interconnection Network	
Communication Delay	
Bus	2
Crossbar Network	2
Multistage Network[a]	$2 * \log(n)$

[a] The number of stages in the Multistage Network is $\log(n)$.

unit. Queued tokens are processed subsequently, one after another, in the order of their arrival.

5.1.1. Input Parameters

To evaluate the performance of our implementation, a synthetic work load program was designed. The synthetic program consists of *m* processes, each process executing a repetitive command having *n* guards. The repetitive commands belonging to the *m* processes are such that the I/O guards in one have matching and compatible guards in the complement processes. Without loss of generality, all the guards (in all the processes) are enabled; that is, their boolean components always evaluate to *True*. The number of rings in the multi-ring data flow machine can be varied and is an input parameter for the simulator. The number of (interleaved) memory blocks and the type of network used are other input parameters. All these parameters can be varied independently.

5.1.2. Output Parameters

The following performance parameters are measures of efficiency and fairness of the implementation.

(i) *Average Tries*: The number of guards that have been tried before the rendezvous is evaluated for each transaction. The mean value of this figure is the average tries. It can be evaluated as the ratio of the sum of the number of guards tried (before a rendezvous) for each transaction to the total number of transactions. The parameter is of interest from the efficiency viewpoint. The value of average tries for our implementation could be between 2 (corresponding to one guard in each of the complement processes) and $2 * n$ (where n is the number of guards per transaction). The value of average tries can be normalized by dividing it by $2 * n$.

(ii) *Average Rendezvous Time*: For each rendezvous that took place, the time elapsed between the initiation (**Get-id** node) and commitment, called the rendezvous time, is measured. The ratio of the sum of the rendezvous time for all the transactions to the total number of transactions gives the average time for a rendezvous.

(iii) *Guard Bias Figure*: If the implementation is fair, then every guard is equally likely to participate in a rendezvous. Hence, the number of times a guard has participated in a rendezvous, called the success count, should provide a measure of fairness. Ideally, the success count for each guard should be equal to

$$\frac{2 * \text{total number of transactions}}{\text{number of guards}}$$

(The value 2 in the numerator is to account for the fact that two guards participate in every rendezvous.) Therefore the standard deviation of the success counts of the guards is a measure of biasing. This figure, referred to as the guard bias figure, indicates the extent to which a guard is biased/favored. For a fair implementation, this value should be very low.

(iv) *Process Bias Figure*: This figure is similar to the guard bias and can be obtained by computing the standard deviation of the success counts of the various processes.

Since all the guards are enabled continuously in the simulation study, the process and guard bias figures are measures of *weak* fairness.

(v) *Average Concurrency*: Our implementation is a parallel implementation for the generalized guarded commands. It would therefore be interesting to measure the amount of concurrency exploited in our implementation. In measuring the concurrency in our implementation, we separate the concurrency in executing communication-related operations from that in executing other supporting actors. We refer to these concurrencies as *communication-actor concurrency* and *support-actor concurrency* respectively. As the data flow actors, **Get-id, Tryguard, Split, Ext-Info,** and **DummyTry** are directly useful in establishing the rendezvous, we call these actors as communication-related operations. Other data flow actors used in our implementation are support actors. The average concurrency in executing each of these kinds of actors is defined as the ratio of the total time spent by all the processors in executing the actors of this class to the total simulation period.

5.2. Performance Results

Three different synthetic programs have been used in our simulation experiments. In the first program, there are five processes, each communicating with every other process twice, once through an input command and once through an output command. The second program is similar to the first one, except that nine processes interact with each other. There are sixteen guarded commands and five processes in the third program, where the communication between any two processes can take place using any of the four guarded commands (two input and two output guards).

The input programs were run on the simulator for different configurations of the data flow machine. The number of rings and the number of memory modules were varied from 1 to 32. In all configurations the three network topologies, namely a single bus, multi-stage, or crossbar network, were tried. The simulation period was kept high enough to accommodate at least 1000 transactions in each process, a total of 5000 to 9000 rendezvous taking place in the entire simulation period. It is observed that the simulation results settle down fairly quickly, even with a total of 100 transactions.

Average Tries: Table II lists the normalized value of average tries (for a rendezvous) for various simulation runs. We make the following two observations:

1. The average tries value varies from 0.6 to 0.97 for the different runs. In particular, when the number of guards per process is smaller and the number of rings in the data flow machine is larger, the value of average tries is high. This can be reasoned as follows. When there is a low parallelism, i.e., fewer number of guards to try, and a high resource availability, the chances are high for more guards of a process to be tried before a rendezvous. Further, since all boolean guards in the application program trivially evaluate to *True*, it is possible that all guards of a transaction are attempted (i.e. the respective **Tryguard** actors are executed) more or less at the same time, accounting for the large value of average tries. This further adds to the possibility that the **Tryguard** actors of a process are executed simultaneously, accounting for the large number of tries before a rendezvous.
2. There is only a minor variation in the average tries as the network topology is changed. This is attributed to the low network-memory traffic. The ratio of memory access time to rendezvous

Table II. Average Tries

Input Programs with Processes/Guards		Network Topology	Normalized Average Tries No. of Rings								
			1	2	4	6	8	12	16	24	32
5	8	Bus	0.73	0.70	0.71	0.62	0.84	0.84	0.87	0.88	0.90
5	8	Multistage	0.73	0.70	0.73	0.65	0.91	0.93	0.95	0.97	0.97
5	8	Crossbar	0.73	0.70	0.73	0.61	0.86	0.87	0.87	0.92	0.95
5	16	Bus	0.62	0.61	0.63	0.61	0.70	0.76	0.75	0.84	0.80
5	16	Multistage	0.62	0.61	0.67	0.63	0.73	0.81	0.90	0.88	0.90
5	16	Crossbar	0.62	0.61	0.64	0.62	0.69	0.76	0.76	0.78	0.76
9	16	Bus	0.70	0.66	0.67	0.65	0.72	0.79	0.86	0.85	0.86
9	16	Multistage	0.70	0.67	0.68	0.66	0.75	0.83	0.95	0.96	0.97
9	16	Crossbar	0.70	0.67	0.67	0.65	0.73	0.79	0.82	0.82	0.82

time is found to be low, less than 12%, in a typical run. From this, we can say that the network/memory traffic does not dominate the implementation.

Average Rendezvous Time: The average rendezvous time as the architectural configuration is changed are tabulated in Table III. The average time decreases continuously as the number of PEs is increased up to 6 or 8, and then the value saturates. As in the case of average tries, a change in the number of memory units or network topology has little effect on average rendezvous time. Again, this is due to the low network/memory traffic.

Fairness Measures: Table IV summarizes the guard bias and process bias figures for the benchmark program with 9 processes. The bias figures for other benchmark programs exhibit a similar trend and hence are not shown here. We notice these values are very low, at most 2 transactions out of nearly 1000 transactions. Such a low degree of biasing is acceptable to an implementation involving nondeterminism.

Average Concurrency: The average communication-actor concurrency and the support-actor concurrency for various architectural configurations using the shared bus interconnection are shown in Table V. The performance results exhibit a similar trend when other network topologies are used, and hence are not included here. As can be observed from Table V,

Table III. Average Rendezvous Time

Input Programs with Processes/Guards		Network Topology	Average Rendezvous Time (in time units) No. of Rings								
			1	2	4	6	8	12	16	24	32
5	8	Bus	2174	910	467	304	336	308	313	318	320
5	8	Multistage	1972	912	489	322	346	321	318	310	322
5	8	Crossbar	2038	912	479	300	328	309	303	314	300
5	16	Bus	3615	1555	840	535	571	532	512	548	536
5	16	Multistage	3500	1555	865	571	585	530	519	534	527
5	16	Crossbar	3615	1555	841	550	560	509	539	517	511
9	16	Bus	5954	2455	1206	890	773	770	769	760	770
9	16	Multistage	5721	2478	1246	930	780	765	753	739	748
9	16	Crossbar	5954	2469	1202	898	748	768	748	749	653

Table IV. Fairness Measures

Input Programs with Processes/Guards			Network Topology	Bias Figures (in number of transactions) No. of Rings								
				1	2	4	6	8	12	16	24	32
			Bus	3.0	3.2	1.0	1.2	1.6	0.8	1.3	1.2	0.8
9	16	Guard	Multistage	1.6	1.0	1.2	1.2	1.2	1.0	1.1	1.0	0.7
		Bias	Crossbar	1.6	0.7	1.4	0.7	1.2	1.2	0.9	0.9	0.9
			Bus	1.2	1.2	1.0	1.4	1.4	1.4	1.4	1.6	1.4
9	16	Process	Multistage	1.2	1.2	1.2	1.3	1.3	1.5	1.6	1.5	1.5
		Bias	Crossbar	1.2	1.1	1.0	1.2	1.4	1.5	1.4	1.7	1.4

the communication-actor concurrency increases continuously with an increase in the number of processors, while the support-actor concurrency saturates beyond 16 processors.

5.3. Throttling Parallelism

The simulation results show that average tries for a rendezvous is quite high in our implementation. This is so especially when there is a high resource availability. The reason for the high value of this performance figure is, partly, our greedy approach—attempting all the guards of a process simultaneously. Though such an approach increases the parallelism it can result in poor resource utilization and even poor performance in cer-

Table V. Concurrency Figures

Input Programs with Processes/Guards			Concurrency Figures No. of Rings								
			1	2	4	6	8	12	16	24	32
5	8	Commn.-Actor	0.29	0.55	1.09	1.41	1.88	2.08	2.35	2.42	2.50
		Support-Actor	0.71	1.44	2.77	3.56	3.54	4.21	4.96	5.05	5.06
5	16	Commn.-Actor	0.29	0.57	1.09	1.58	1.94	2.38	2.74	2.89	3.02
		Support-Actor	0.70	1.43	2.65	3.82	4.57	4.81	4.98	5.05	5.09
9	16	Commn.-Actor	0.29	0.59	1.15	1.67	2.07	2.48	2.93	3.19	3.39
		Support-Actor	0.71	1.41	2.83	4.05	4.75	5.00	5.19	5.19	5.19

tain cases. Similar studies have been reported in Ref. 23, where unbounded unraveling of loops causes a poor utilization and a performance penalty. Restricting the amount of parallelism (or unraveling) has been proposed as a remedy.[23] We adopt a similar method to reduce the average tries for a rendezvous. Instead of allowing all the (enabled) guards of a process to attempt a rendezvous, we modify our implementation to attempt at most k guards simultaneously. That is, in each process only k **Tryguard** actors can be active at a time. On completion of a **Tryguard** actor for a guard i in process P, the $(k+i)$th guard (in P) can be initiated. This scheme facilitates tuning the amount of parallelism in our implementation.

5.3.1. Simulation Results

The simulator has been modified to implement the new scheme. The value of k, called the *parallelism bound*, can be specified at the runtime of the simulator. The simulation experiments were conducted for the values of $k = 16$, 12, 8, 4, 2 and 1. The results of the simulation experiments are plotted in Fig. 7 for the benchmark program with 9 processes and 16 guards. These experiments were conducted for three different architecture

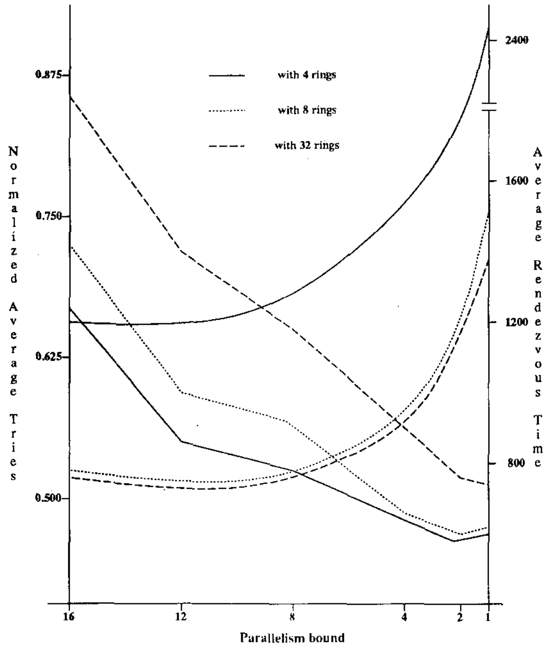


Fig. 7. Effect of parallelism bound.

configurations with 4, 8, or 32 rings and with a bus interconnection. Other benchmark programs and architectural configurations have similar performance curves and hence are not shown here.

From Fig. 7, we observe that the normalized average tries for a rendezvous decreases as the parallelism bound k is decreased. The normalized average tries drops to 0.42 when the parallelism bound is 2. The average rendezvous time, on the other hand, increases with decreasing values of k. This is not unexpected, because when the parallelism is bounded, it is natural for the rendezvous time to go up. Both the average tries and average rendezvous time are important parameters and need to be as low as possible for a good implementation. Hence in tuning the parallelism we must strike a balance between the average rendezvous time and the average tries. In order to achieve this we propose an adaptive algorithm in the following subsection.

5.4. An Adaptive Algorithm for Tuning Parallelism Bound

In the adaptive algorithm, the implementation starts with a parallelism bound equal to the number of guards in any process. After every x rendezvous, the adaptive scheme tries to increase or decrease the parallelism bound (by 1) depending on the performance parameters. This is done by calculating the current values of the normalized average tries and the average rendezvous time and comparing them with those calculated previously (x rendezvous before). If the normalized average tries has decreased and the average rendezvous time has not increased by, say 5% then the previous change in the parallelism bound is accepted. Also, the parallelism bound is further decreased (by 1), hoping for further improvement in the performance parameters. On the other hand, if the average rendezvous times has increased significantly (by more than, say, 5%) then the parallelism is increased by 1. The current value of parallelism bound is used for the next x rendezvous. The value of x is critical in the adaptive scheme. If x is chosen to be too large, then changes in parallelism bound occur once after a long interval and therefore it may take a long time for the parallelism bound to settle. On the other hand, if the value of x is too low, there is more fluctuation in the parallelism bound as the performance parameters are computed sooner, and therefore may not fully reflect the changes made in the parallelism bound during the last adaptive step.

The simulator has been modified to implement the adaptive scheme that tunes the parallelism bound. The results of various run have been tabulated in Table VI. A bus interconnection network was used throughout this experiment. Using other interconnection networks does not seem to

Table VI. Performance of the Adaptive Scheme

Input Programs with Processes/Guards		Performance Parameter	Number of Rings 4	8	16	24	32
5	16	Average Parallelism Bound	4.52	5.44	5.34	5.60	5.50
		Normalized Average Tries	0.52	0.55	0.58	0.60	0.60
		Average Rendezvous Time	933	561	559	549	558
		Commn.-Actor Concurrency	1.06	1.91	2.24	2.31	2.36
		Support-Actor Concurrency	2.67	4.88	4.98	5.02	5.05
9	16	Average Parallelism Bound	5.03	5.52	6.10	5.80	6.00
		Normalized Average Tries	0.51	0.52	0.58	0.57	0.58
		Average Rendezvous Time	1349	858	833	867	858
		Commn.-Actor Concurrency	1.12	2.03	2.34	2.52	2.56
		Support-Actor Concurrency	2.86	5.03	5.19	5.20	5.20

alter the performance parameters significantly and hence their results are not reported here.

It can be observed that the adaptive scheme tunes the parallelism bound depending on the available resources (in this case, the number of rings in the multi-ring architecture) to strike a balance between the normalized average tries and the average rendezvous time. The effectiveness of the adaptive scheme can be appreciated by comparing the average tries and the average rendezvous time for the adaptive scheme with those for an implementation where the parallelism bound is fixed. The comparison is done for the benchmark program with 9 processes on an architecture with 32 rings for various values of parallelism bound, namely 2, 4, 8, 12, and 16. Figure 8 shows how the adaptive scheme achieves a tradeoff between average tries and average rendezvous time.

Table VI also summarizes the concurrency figures for the adaptive scheme. Even in the presence of adaptive throttling, our implementation exploits concurrency in executing both communication and support actors. However the values for the communication-actor concurrency in Table VI are lower compared to the corresponding values in Table V. The difference is significant for larger number of rings, e.g., with 16, 24, or 32 rings. This is quite intuitive as the (adaptive) throttling scheme essentially controls the parallelism bound, especially in executing communication actors. Further, the throttling scheme is more effective for larger number of processors. The values for support-actor concurrency figure in Table VI are more or less equal to those in Table V.

The results shown in Fig. 8 and Table VI were obtained by performing the adaptive scheme once every 40 rendezvouses (i.e. $x = 40$). Figure 9

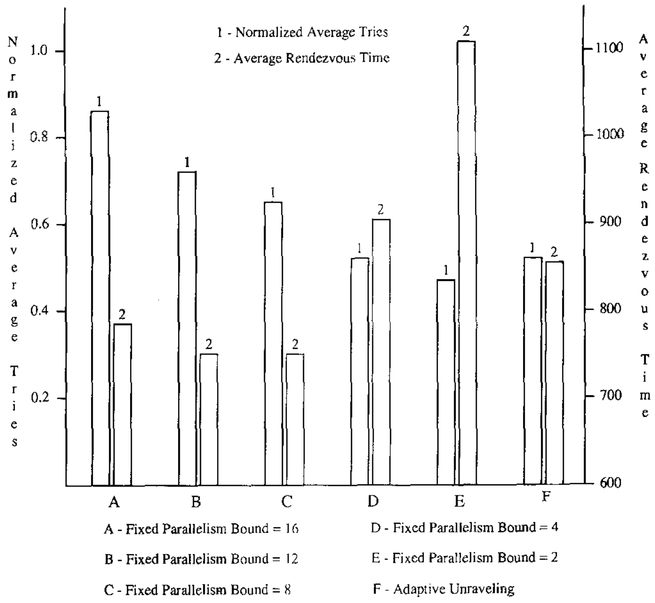


Fig. 8. Comparison of adaptive scheme with fixed parallelism bound.

shows how the adaptive scheme constantly changes the value of the parallelism bound during the entire simulation period. The results are specific to the architecture configuration with 32 rings connected by a shared bus topology and for the benchmark program with 9 processes. A similar trend is observed for other configurations and benchmark programs. To show the effect of x on the adaptive scheme, we plot a similar curve for the value of $x = 5$ in Fig. 10. We observe that when the value of

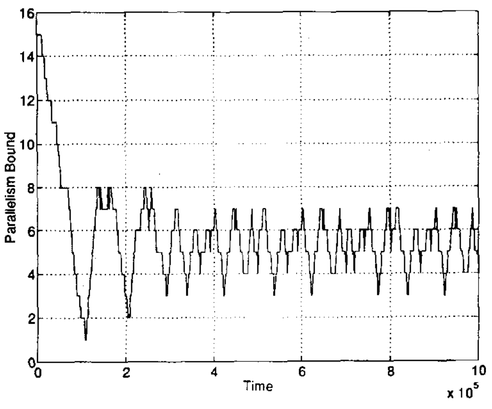


Fig. 9. Adaptive unraveling: change after 40 rendezvouses.

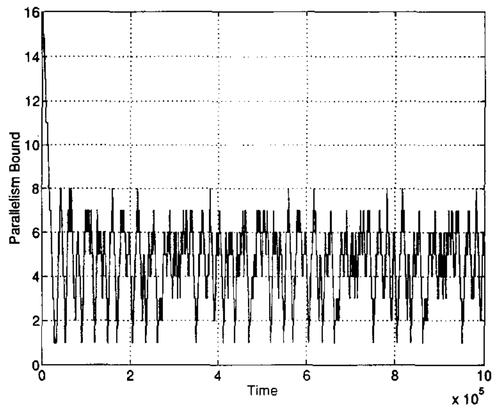


Fig. 10. Adaptive unraveling: change after 5 rendezvouses.

x is too low the parallelism bound fluctuates to a large extent, but nonetheless, does improve the normalized average tries and the average rendezvous time.

6. RELATED WORK AND COMPARISON

Our implementation uses the data flow model of computation in contrast to the earlier von Neumann multiprocessing models. As mentioned earlier, the data-driven computation facilitates the exploitation of fine grain asynchronous parallelism and concurrency not only among processes but also within a process. A marked difference from other models is that in our implementation the guards are not attempted in any prespecified sequential order; rather they are tried concurrently. Yet our implementation ensures the correct semantics of the construct. Further, the communication constructs which need to wait for synchronization with their partners relinquish the PEs. This allows other data flow actors to acquire the PEs and do userful work, which otherwise might have been wasted in busy waiting.

6.1. Shared Memory Systems

The work reported in Ref. 9 is based on shared memory model and resembles our implementation in some aspects. In their shared memory implementation,(9) a rendezvous between two transactions takes place only after one of them enters the 'waiting' state. This means that all guards of one transaction have to be attempted before the rendezvous. That is, for the rendezvous to occur, in the best case $(n+1)$ guards are attempted. In the worst case, $2 * n$ tries have to be carried out. The average case results are not reported in Ref. 9. Theoretically, the best and worst case figures for our implementation are 2 and $2 * n$ respectively. Also, the simulation results indicate that the normalized average tries is (0.6) when the adaptive throttling scheme is followed. Fujimoto *et al.*(24) have evaluated the performance of their implementation on a BBN Butterfly Parallel Processor. The average rendezvous time for their implementation is 17.2 milliseconds for a program with 16 processes and 16 guards per process. However, since these results were obtained from an actual implementation while ours are based on simulation, it is unfair to make a direct comparison of these empirical results.

6.2. Message Passing Systems

The implementation presented in Refs. 3, 5, 7, 8, and 10 use a distributed memory architecture. Some of them(3,7) are based on a two-phase algorithm. Our implementation, like the ones presented in Refs. 5, 9, and 10, involves only one phase. There is no need for a second phase as every **Tryguard** actor terminates with a definite answer. That is, noncommittal replies and retrying, as in Refs. 3 and 7, do not occur in our implementation. A quantitative comparison of any of the implementations based on message passing system could not be made due to the nonavailability of quantitative results for these implementation.

Our implementation retains the spirit of the criteria listed in Refs. 5 and 7 for a loosely-coupled multiprocessor system. For instance, five of the six criteria [Criterion (i) of Ref. 5 is not relevant to the discussion.] can be appropriately redefined as: (i) the amount of system information stored (in the shared memory) should be minimal; (ii) when both t_l and t_r are ready to select the guards g_l and g_r, respectively, then at least one of the transactions will select a guard (not necessarily g_l or g_r) within a finite time; (iii) the number of attempts made in selecting a guard of a transaction should be bounded; (iv) the time taken by a **Tryguard** actor to determine whether it can establish a rendezvous must be finite; and (v) if a process has a guarded command that is enabled continuously for an infinite time, then it should eventually succeed. It can be easily proved, with the help of the theorems stated in Section 4, that our implementation satisfies these criteria. In fact it performs better than others(3,5,7–10) with respect to (ii) and (iii), as parallelizing the selection of guards significantly reduces the execution time.

7. CONCLUSIONS

In this paper, we have presented a decentralized parallel implementation of generalized guarded commands of CSP using the data flow model of computation. The implementation is guaranteed to be safe and live. Performance parameters for efficiency and fairness of the implementation have been obtained using discrete-event simulation. The performance results and comparison reveal the superiority of our implementation over other existing ones. Various performance parameters have been measured for different values of parallelism bound. The simulation results reveal a tradeoff between average tries and average rendezvous time with respect to the parallelism bound. In order to strike a balance between the two performance parameters, we propose an adaptive algorithm which dynamically tunes the parallelism bound to achieve an optimum level of performance. The simulation results of the adaptive algorithm establish a marked improvement in performance, in terms of both average tries and average rendezvous time.

It can be easily proved that our implementation does not leave lingering tokens in the data flow graph. In the implementation of generalized guarded commands we have not addressed the issue of process termination. However, our implementation can be easily extended to take care of process termination.

The implementation proposed in this paper suits any data flow machine although we have used Manchester multi-ring data flow computer in the simulation experiment. One criticism against data flow machines is the cost involved in explicit synchronization and instruction scheduling. Multi-threaded architectures(25) overcome this problem by following a hybrid approach, combining both explicit and implicit instruction scheduling. It is possible to extend our implementation to multi-threaded architectures(25) as well.

ACKNOWLEDGMENTS

The authors are thankful to the anonymous referees whose comments have substantially improved the presentation and organization of this paper. The authors acknowledge Bhama Sridharan for her many constructive criticisms on the initial drafts of this paper.

REFERENCES

1. C. A. R. Hoare, Communicating Sequential Processes, *Commun. of the ACM* **21**(8):666–677 (August 1978).
2. A. J. Bernstein, Output Guards and Nondeterminism in Communicating Sequential Processes, *ACM Trans. on Programming Lang. and Syst.* **2**(2):234–238 (April 1980).
3. G. N. Buckley and A. Silberschatz, An Effective Implementation for the Generalized Input Output Construct of CSP, *ACM Trans. on Programming Lang. and Syst.* **5**(2):233–235 (1983).

4. R. B. Kieburtz and A. Silberschatz, Comments on Communicating Sequential Processes, *ACM Trans. on Programming Lang. and Syst.* **1**(2):218–225 (October 1979).
5. S. Ramesh, A New Implementation of CSP with Ourput Guards, *Proc. of the 7th Int'l. Conf. on Distr. Comput. Syst.*, pp. 266–273 (1987).
6. A. Silberschatz, Communication and Synchronization in Distributed Systems, *IEEE Trans. on Software Engineering* **SE-5**(6):542–546 (November 1979).
7. R. J. R. Back, P. Ekslund, and R. Kurki-Suonia, A Fair and Efficient Implementation of CSP with Output Guards, Technical Report, Ser. A, No. 38, Abo Akademic, Finland (1984).
8. R. Bagrodia, A Distributed Algorithm to Implement the Generalized Alternative Command in CSP, *Proc. of the 6th Int'l. Conf. on Distr. Comput. Syst.*, pp. 422–427 (1986).
9. R. N. Fujimoto and Hwa-chung Feng, A Shared Memory Algorithm and Proof for the Generalized Alternative Construct in CSP, *IJPP* **16**(3):215–241 (1987).
10. R. Bagrodia, Synchronization of Asynchronous Processes in CSP, *ACM Trans. on Programming Lang. and Syst.* **11**(4):585–597 (October 1989).
11. P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins, Data-Driven and Demand-Driven Architecture, *Computing Surveys* **14**(1):93–143 (March 1982).
12. J. R. Gurd, I. Watson, and C. C. Kirkham, The Manchester Prototype Data Flow Computer, *Comm. of the ACM* **28**(1):34–52 (January 1985).
13. A. L. Davis and R. M. Keller, Data Flow Program Graphs, *IEEE Computer* **15**(2):26–41 (February 1982).
14. J. B. Dennis, Data Flow Supercomputers, *IEEE Computer* **13**(11):48–56 (November 1980).
15. Arvind and K. P. Gostelow, The U Interpreter, *IEEE Computer* **15**(2):42–49 (February 1982).
16. A. Gottlieb, R. Grishman, C. P. Kruskal, L. Rudolph, and M. Snir, The NYU Ultracomputer—Designing an MIMD Shared Memory Parallel Computer, *IEEE Trans. on Computers* **C-32**(2):175–189 (February 1983).
17. L. M. Patnaik and J. Basu, Two Tools for Interprocess Communication in Distributed Data Flow Systems, *The Computer Journal* **29**(6):506–521 (December 1986).
18. S. Owicki and L. Lamport, Proving Liveness Properties of Concurrent Programs, *ACM Trans. on Programming Lang. and Syst.* **6**(2):455–495 (July 1982).
19. D. A. Reed, A. D. Malony, and B. D. McCredie, Parallel Discrete Event Simulation: A Shared Memory Approach, *Proc. of the ACM Sigmetrics Conf. on Measuring and Modeling Computer Systems* **15**(1):36–38 (May 1987).
20. N. Francez, *Fairness*, Springer-Verlag, New York (1986).
21. J. R. Gurd and I. Watson, Data-Driven Systems for High Speed Parallel Computing: Part 2: Hardware Design, *Computer Design*, pp. 97–106 (July 1980).
22. J. G. D. de Silva and J. V. Woods, Design of a Processing System for the Manchester Data Flow Computer, *IEEE Proceedings* **128**(5):218–224 (September 1981).
23. D. E. Culler and Arvind, Resource Requirements of Data Flow Programs, *15th Ann. Int'l. Symp. on Computer Architecture*, pp. 141–150 (1988).
24. H. Feng and R. M. Fujimoto, A Shared Memory Algorithm and Performance Evaluation of the Generalized Alternative Construct in CSP, *Int'l. Conf. on Parallel Processing*, pp. 176–180 (1988).
25. R. A. Iannucci, Toward a Data Flow/von Neumann Hybrid Architecture, *15th Ann. Int'l. Symp. on Computer Architecture*, pp. 131–140 (1988).