The construction of a small communication library

Eindhoven University of Technology

Department of Mathematics and Computing Science

The Construction of a Small Communication Library

by

J.J. Lukkien

95/01

ISSN 0926-4515

editors: prof.dr. J.C.M. Baeten
prof.dr. M. Rem

Computing Science Report 95/01
Eindhoven, January 1995

The Construction of a Small Communication Library

Johan J. Lukkien
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB Eindhoven, The Netherlands
Telephone: +31 40 475147
Telefax: +31 40 436685
Email: johanl@win.tue.nl

January 9, 1995

Abstract

In order to develop portable parallel software a reasonable abstraction from parallel hardware is necessary. Over the last few years, such abstractions have become available in the form of communication libraries. In this paper we first look at the mapping of parallel programs onto networks and we show how this can be generalized into a small but powerful communication library. The focus of the discussion is on the derivation of the processes implementing the library. During the design, efficiency is a key issue. In particular, we pursue a low latency for communications by avoiding buffering as much as possible.

1 Introduction

An important reason to construct parallel programs is that a high performance may be achieved when such a program is executed by a parallel machine. With this prospect of increased performance as a driving force, programs were often designed for a particular target architecture. Since many different architectures have become available this resulted in special-purpose solutions. It has been recognized that this frustrates the development of portable software, i.e., of software that can be used on a new architecture after only minor modifications.

Another approach to parallel programming is to develop programs for some idealized model of a parallel computer, e.g., for a PRAM ([9]). Although this has lead to important results in complexity analysis it has not lead to practical methods to develop parallel programs, partly because these idealized models cannot be implemented realistically.

These developments have lead to the definition of models that are both general and admit an efficient implementation on a variety of architectures. One such model is the *Bulk Synchronous Computer* (BSP, [12]). In this model, the parallel machine consists of a set of processors, each having a local memory. Each processor has access to all non-local memories in a uniform and efficient way. Each processor executes the same program, acting on local data. Such a program consists of a sequence of computation and communication steps. Each communication step is basically a rearrangement of information in the local memories. The importance of such a model is that it admits a reasonably simple performance analysis and that it can be implemented straightforwardly as we show in this paper. Other models are the

ones provided by communication libraries like *Parallel Virtual Machine* (PVM, [14]) or *Message Passing Interface* (MPI, [13]). In these models a parallel program consists of a collection of communicating processes; the program is executed by a collection of processors. The library provides a variety of routines for process creation and for communication and synchronization between the processes. The abstraction from physical topology is most important in all these models. For each particular architecture the problem of implementing the model remains. For a recent overview of communication libraries we refer to [4].

The success of using an abstract model of a machine, like a communication library, is largely determined by the efficiency of the implementation of communication. When we say that a model cannot be implemented realistically, we mean that the overhead in terms of communication time is prohibitively large. The time required for a communication is determined by two parameters: the latency of message transmission (l) and the throughput of the channel that exists between the two partners in the communication (t). The time to transmit a message of length k is given by $l(k) + kt$. It is not hard (though expensive) to design channels with a high throughput; it is the latency that largely determines the usefulness of the parallel machine. This is due to the fact that the latency determines the time that elapses before a process can receive an answer to any message that it sent. The implementation of PVM on a workstation demonstrates this aspect painfully. Therefore, it is important that communication is as efficient as possible.

In this paper we construct a communication library which, though small, resembles the ones mentioned above. It can also be used in an implementation of the BSP model. We first look at how to map a parallel program onto a parallel machine. The program is designed as a collection of communicating processes; the machine consists of a collection of processors, each having a local memory and being connected to a limited number of neighbors; the local memories may also be shared. We look at a limited instance of this problem: we assume that the mapping of processes onto processors is given. The task that remains is the mapping of the communication channels in the program onto the physical network. Since all buffering adds to the latency, we pursue a low latency by minimizing the amount of buffering during the transmission of a message. The solution to this mapping problem is generalized, resulting in a set of communication routines.

The paper is organized as follows. In the next section we introduce our program notation. In order to make this paper self-contained we include a few examples and some discussion. In section 3 we describe the mapping problem mentioned above. If two processes are mapped onto the same processor, a channel between the processes is mapped onto the local memory of that processor; this is described in section 5. General transmission of messages in a network is presented in section 6 which is then used in section 7 to obtain a distributed implementation of the mapping. These results are generalized to the construction of a small library of communication routines in section 8. We end with some conclusions.

2 Program notation and semantics

Our program notation is based on Dijkstra's guarded command language and Hoare's CSP ([3, 7], see also [16]). Instead of **do** $b \rightarrow S$ **od** to denote a repetition, we write $*[b \rightarrow S]$; for the particular case of $b = true$ we write $*[S]$. Instead of **if** $b_0 \rightarrow S_0 \,[]\, b_1 \rightarrow S_1 \ldots$ **fi** to denote a selection, we write $[\, b_0 \rightarrow S_0 \,[]\, b_1 \rightarrow S_1 \ldots]$. Execution of the selection amounts to executing one of the statements S_i for which the corresponding guard b_i holds. We use "|[" and "]|" for opening and closing a context respectively. Procedure and variable declarations are similar to the Pascal conventions. As an example, the following procedure returns in parameter g the greatest common divisor of positive parameters x and y.

```
proc gcd (x,y: int; var g: int)
|[ *[ x ≠ y → [ x < y → y := y − x
              [] x > y → x := x − y
              ]
   ];
   g := x
]|
```

The notation is extended in order to describe concurrency. The concurrent execution of statements $S0$ and $S1$ is denoted by $S0$ || $S1$. Its execution amounts to the simultaneous execution of the actions of $S0$ and $S1$ while the order of the actions as specified in $S0$ and $S1$ is preserved. Operator "||" is called *parallel composition*; it takes precedence over sequential composition. The statements in a parallel composition are called *processes* although we will generally use this term only if they are procedure calls. The declaration of the corresponding procedure is also called a process. A second way to denote parallel execution is to precede a procedure call with the keyword **fork**. The called procedure is then executed concurrently with the caller.

Processes can communicate through directed *channels*. A channel can be regarded as a communication link between two processes. For each channel c there is one sender and one receiver. Sending the value of expression e along the channel is denoted by $c!e$; receiving a value which is then stored in variable x is denoted by $c?x$. These two actions are called an *output* and an *input* action respectively. We require that output and input actions are synchronized. Execution of an input or output statement is suspended until the partner is ready to perform the corresponding output or input statement. The concurrent execution of an output and an input action, therefore, amounts to the assignment $x := e$. As an example we give a process that copies the positive values received along its input channel c to its output channel d.

```
proc filter (var c,d: channel)
|[ var x: int;
   *[ c?x; [ x > 0 → d!x [] x ≤ 0 → skip ]]
]|
```

The above procedure for computing the GCD can be adapted into a process that computes the GCD of all pairs it receives along its input channels c and d and returns the results along its output channel e.

```
proc gcd (var c,d,e: channel)
|[ var x,y: int;
  *[ c?x || d?y;
     *[ x ≠ y → [ x < y → y := y − x
                [] x > y → x := x − y
                ]
      ];
     e!x
   ]
]|
```

Two of the above *filter* processes are combined with a *gcd* as follows.

```
proc gcd_checked (var A,B,C: channel)
|[ var a,b: channel;
  filter (A,a) || filter (B,b) || gcd (a,b,C)
]|
```

The capitalized channels are inputs and outputs to the environment of this process. The internal channels (a and b) are not visible from the environment.

In this paper we are mainly concerned with the implementation of these communication actions. To that end we need a formalization. This formalization is used in two ways. On the one hand, we regard it as a requirement for an implementation, i.e., an implementation satisfying this formalization is called a channel. On the other hand, given an implementation of communication actions we use the formalization to prove facts about programs. Our formalization is based on [11, 15].

From the above description of input and output actions we obtain two requirements, viz. that input and output are synchronized and that together they implement a (distributed) assignment. These two can be regarded as safety requirements. Besides this we have a progress requirement which is necessary because execution of communication actions may become blocked (suspended). This third requirement is that unnecessary blocking does not occur[1]. In order to formalize the requirements, we introduce two notions referring to the state of the program during execution. For A an action in a program, $\#A$ denotes the number of completed executions of A. The first requirement for a channel c is

$$C0: \quad \#c! = \#c?$$

Since processes may become suspended on execution of an action, we introduce $\underline{q}A$ to denote the number of processes suspended on execution of A. No unnecessary blocking is then expressed as follows.

$$C1: \quad \underline{q}c! = 0 \ \vee \ \underline{q}c? = 0$$

Finally, the requirement that input and output implement an assignment is expressed by

$$C2: \quad \{P^x_e\}\ c!e \parallel c?x\ \{P\}$$

A pair of actions satisfying $C0$ through $C2$ is said to implement a channel.

The advantage of using channels for communication between processes is that there is no interference between processes apart from these communications. This supports the modular development of parallel programs. However, there are problems that benefit from other communication mechanisms. In particular, if two processes have access to the same memory they may communicate through this memory. For synchronization between such processes we use semaphores. A semaphore s is an integer with initial value s_0. Operations on s are $P(s)$ and $V(s)$ with the effect of decrementing s by 1 and incrementing s by 1 respectively. The restriction is that s never becomes negative; if execution of a P operation would result in decreasing s beyond 0 the process executing the operation becomes suspended. Together with the requirement that unnecessary blocking is not allowed this results in the following three requirements.

$$S0: \quad s \geq 0$$
$$S1: \quad s = s_0 + \#V(s) - \#P(s)$$
$$S2: \quad (s = 0 \vee \underline{q}P(s) = 0) \ \wedge \ \underline{q}V(s) = 0$$

We do not discuss implementations of semaphores in this paper. Hence, we use $S0$ through $S2$ only to prove facts about programs with semaphores.

Semaphores are often used to provide exclusive access to variables or other resources. As an example, let v be a variable, shared by processes X and Y which increment v every now and then. Semaphore s with initial value 1 is used to provide mutually exclusive access to v.

```
proc X                          proc Y
|[ *[ actions, not using v;     |[ *[ actions, not using v;
     P(s); v := v + 1; V(s)          P(s); v := v + 1; V(s)
   ]                               ]
]|                              ]|
```

[1] In fact this requirement holds for all actions in the program, not only for communication actions. For example, the fact that execution of an assignment never becomes suspended is not mentioned explicitly.

As an illustration, we present a short proof for mutual exclusion on variable v. In order to distinguish the actions of the two processes we use the name of the process as a subscript. From the topology of the program we derive that the state in which variable v is accessed by process X is characterized by

$$\#P_X(s) = \#V_X(s) + 1$$

Similarly, the state in which process Y accesses v is

$$\#P_Y(s) = \#V_Y(s) + 1$$

The conjunction of the two describes a state in which both processes may modify v.

$$\begin{array}{ll} & \#P_X(s) = \#V_X(s) + 1 \ \wedge \ \#P_Y(s) = \#V_Y(s) + 1 \\ \Rightarrow & \quad \{\#P(s) = \#P_X(s) + \#P_Y(s), \#V(s) = \#V_X(s) + \#V_Y(s), \text{ calculus}\} \\ & \#V(s) - \#P(s) = -2 \\ = & \quad \{s_0 = 1, S1\} \\ & s = -1 \\ = & \quad \{S0\} \\ & \mathit{false} \end{array}$$

Hence, exclusive access to variable v is guaranteed. In order to prove progress one has to show that X and Y are never suspended at the same time. This follows from $S2$ using similar reasoning.

3 The mapping problem

A program written in a language like the one described in the previous section consists of a collection of communicating processes. This collection changes dynamically as new processes are created and processes terminate. Some of the communication between these processes goes through shared variables and some of it through message passing. This program has to be mapped onto a machine consisting of a collection of processors together with an interconnection network. In order to simplify matters we assume that this mapping has to be done only once, viz., initially. Hence, only the processes in an outermost parallel construct are mapped.

In order to specify the problem more precisely we introduce some formalization. The parallel machine is represented by a directed graph, $G_I = (V_I, E_I)$ called the *implementation graph*. The vertices of G_I represent the processors and the edges of G_I the directed connections between the processors. Associated with each edge e is a weight, $w_I(e)$, which is either 0 or 1. In case the weight is 0 the processors share their memory (and in that case we have edges both ways); if the weight is 1 communication between the two processors is through message passing[2]. As a result, there is a self loop of weight 0 for each vertex. The program is also represented by a directed graph, $G_C = (V_C, E_C)$ called the *computation graph*. The vertices in the graph represent the processes in the program. For each channel between processes in the program we have an edge in the graph. This edge has weight 1. For every pair of processes communicating through shared memory we have edges in both ways between the processes; these edges have weight 0.

For edge (v, w) we call v the source and w the target. A path in a graph is a non-empty sequence of edges such that, for all but the last edge in this sequence its target equals the source of its successor. The source of a path is the source of the first edge in the path; similarly, for a finite path the target is the target of the last edge in it. For a path p these are denoted by $s.p$ and $t.p$ respectively. We do not distinguish between an edge and a path of length one. The set of finite paths in a graph G is denoted

[2] Of course one could argue that through such a physical connection we can simulate a shared memory but this is one of the problems we are solving.

by $P(G)$. The length of a path is the sum of the weights of the edges on the path. The most general formulation of the mapping problem is as follows.

Mapping problem *Given a computation graph G_C and an implementation graph G_I, find a function $m = (m_V, m_E) : (V_C, E_C) \rightarrow (V_I, P(G_I))$ that satisfies*

$$s.m_E(e) = m_V(s.e)$$
$$t.m_E(e) = m_V(t.e)$$

A discussion of this problem can be found in [6]. In this paper we assume that function m_V is given. The problem that remains is the construction of m_E and programs to implement it. This amounts to the problem of using the shared memory for channel communications (when a channel is mapped onto an edge of length 0) and the problem of using the physical connections for many of these channel communications. We adopt one restriction on m_V: processes using shared memory are mapped either onto the same processor or onto processors with shared memory. This implies that edges of weight 0 can be mapped onto edges of length 0.

4 Action refinement

The requirements for semaphores and channels as listed in section 2 define the execution of the corresponding actions as indivisible. For example, requirement $C0$ does not allow intermediate states in which one of the communication actions has been completed and the other one has not. In an implementation of these actions this atomicity cannot always be guaranteed: as soon as several simple actions are used to implement a complex action it is possible that intermediate states occur, possibly violating the requirements. Implementing an action by simpler ones is called *action refinement*. Formal discussions on the subject can be found in [1, 17]. We illustrate this by an example of an implementation.

If two processes communicating along a channel are mapped onto the same processor or onto processors with shared memory, we map the channel onto an edge of length 0 using shared memory. The implementation has to satisfy $C0$ through $C2$. We focus on $C0$, the synchronization requirement. Synchronization via a shared memory can be achieved through semaphores. We recall from section 2 that semaphores provide a means of "one-way synchronization": for semaphore s with initial value 0 and actions X and Y defined by

$$X: \quad V(s)$$
$$Y: \quad P(s)$$

we have that $\#X \geq \#Y$. This condition realizes already a part of $C0$. A second semaphore, t, also with initial value 0 is needed to establish the synchronization in the other direction. We choose instead of the above

$$X: \quad V(s); P(t)$$
$$Y: \quad P(s); V(t)$$

Now, X and Y appear to satisfy $C0$. However, because X and Y have become compound actions we have introduced two problems. First, we have to be more explicit about the meaning of $\#A$ and $\underline{\mathrm{q}}A$ for a compound action A. We extend this as follows. The number of completed executions of a compound action is the number of times the *last* action of this compound action is executed. A process is suspended on a compound action if it is suspended on *any* of the constituent actions. Second, we have introduced the non-atomicity mentioned above. For example, a state exists in which Y has been completed but X not yet. As a result the implementation does not satisfy the strict synchronization requirement $C0$.

In order to allow action refinement, we have to allow that requirements are violated temporarily. We introduce the following rules.

1. The requirements hold whenever all participating processes are at a point outside the implementation.

2. A state in which the requirements are violated does not persist.

A persisting state is also called a stable state. It is a state in which all participating processes are suspended or have terminated. We usually prove the correctness by showing that a state for which a requirement does not hold is one which is unstable and for which at least one process is inside the implementation.

With this relaxed formulation of the rules, we can show that X and Y satisfy $C0$. The proof proceeds by using invariants, derived from the program text, and properties of semaphores ($S0$ through $S2$). From the program text we derive

$$\#V(s) \geq \#P(t) \tag{1}$$
$$\#P(s) \geq \#V(t) \tag{2}$$

We consider the states in which $C0$ does not hold.

$$\begin{array}{ll} & \#X \neq \#Y \\ = & \quad \{\#X = \#P(t),\ \#Y = \#V(t),\ \text{calculus}\} \\ & \#V(t) > \#P(t) \ \vee \ \#V(t) < \#P(t) \\ = & \quad \{t_0 = 0, S0, S1 : \#V(t) \geq \#P(t)\} \\ & \#V(t) > \#P(t) \\ = & \quad \{(2)\} \\ & \#P(s) > \#P(t) \ \wedge \ \#V(t) > \#P(t) \\ \Rightarrow & \quad \{s_0 = 0, S0, S1 : \#V(s) \geq \#P(s)\} \\ & \#V(s) > \#P(t) \ \wedge \ \#V(t) > \#P(t) \end{array}$$

From the first conjunct we conclude that a state in which $C0$ does not hold is one in which the process executing the implementation of X is at the semicolon immediately preceding $P(t)$. But it is not blocked in this state as follows from the second conjunct and $t_0 = 0$, together implying $t > 0$. Hence, the only state in which $C0$ does not hold is one inside the implementation and it is unstable.

5 A shared-memory implementation

We continue the example of the previous section. Stated more precisely, for a channel c we want refinements of actions $c!e$ and $c?x$ that satisfy $C0, C1$ and $C2$. It was already demonstrated that two semaphores can be used to implement synchronization on a shared memory. Hence, for a channel c we introduce a data structure $(c.s, c.v, c.t)$ where $c.s$ and $c.t$ are semaphores with initial value 0 and $c.v$ is a variable capable of storing one element of the type of c. Synchronization is achieved through $c.s$ and $c.t$; $c.v$ is used to pass a value along the channel. We have to see to it that an assignment to $c.v$ and inspection of $c.v$ happen in the right order. The refinement is as follows.

$$\begin{array}{ll} c?x: & V(c.s); P(c.t); x := c.v \\ c!e: & P(c.s); c.v := e; V(c.t) \end{array}$$

We have to prove that this refinement satisfies both $C1$ and $C2$; the correctness of $C0$ was already shown in the previous section. That same proof is still valid as we only introduced some extra unstable states within the implementation. Requirement $C1$ follows in a similar way as $C0$: just inspect the states in which it might not hold and show that these states, if they exist, are unstable and inside the

implementation. This is left to the reader. The proof of correctness of $C2$ is more involved. We have to show that the assignments are executed in the right order. This follows if as a precondition for "$c.v := e$" we have $\#(c.v := e) = \#(x := c.v)$ and as a precondition for "$x := c.v$", $\#(c.v := e) = \#(x := c.v) + 1$. We prove the former. As a precondition for "$c.v := e$" we have

$$\#P(c.s) = \#(c.v := e) + 1 = \#V(c.t) + 1$$

Using $S2$ for both $c.s$ and $c.t$ we obtain

$$\#V(c.s) \geq \#P(c.s) = \#(c.v := e) + 1 = \#V(c.t) + 1 \geq \#P(c.t) + 1$$

Since the difference between $\#V(c.s)$ and $\#P(c.t)$ is at most 1, equality holds. We conclude that the process executing $c?x$ is suspended on execution of $P(c.t)$. Hence,

$$\#P(c.t) = \#(x := c.v)$$

From this we conclude that the required precondition holds. The proof of the validity of the second precondition is left to the reader.

Remark. Instead of using a variable to record a message passed along the channel, one may pass the address of the variable in which the message has to be stored.

$$c?x: \quad c.v := address(x); V(c.s); P(c.t) \tag{3}$$

$$c!e: \quad P(c.s); c.v\uparrow := e; V(c.t) \tag{4}$$

In this way a fixed amount of memory is used for the implementation of a channel. More importantly, the message is not copied an extra time thus limiting the latency.

6 Routing messages

Now we consider the problem of mapping a channel onto a path of positive length. The result of this mapping is that messages communicated along the channel somehow have to be transported along the physical connections represented by the path. This is a motivation to study the transportation or the routing of messages first. In the next section we focus on how this message routing may be used to implement the required functionality.

First we give a more detailed description of the implementation graph. We assume that it consists of P processors, numbered $0, \ldots, P-1$. Each processor has some local memory that may be shared with other processors. Each processor is capable of executing multiple processes in a time-sliced fashion. These processes may communicate through the shared memory using semaphores as described before. As a notational convention we use the number of a processor as a subscript for variables if this is relevant.

The physical connections to other processors are modeled in the language as arrays of channels: processor k has outgoing channels $C_k(i : 0 \leq i < m_k)$ and incoming channels $D_k(i : 0 \leq i < n_k)$, hence, we implicitly assume that communication between connected processors has been implemented. How this can be done can be found, for instance, in [8]. Notice that, if two processors communicate through shared memory only, we may use the implementation of the previous section to model a channel between the processors.

We develop additional processes for routing messages from a source processor to a target processor. In such a process it is necessary to decide to which outgoing channel a message has to be forwarded. On each processor k we introduce a routing function, RF_k, specified as follows.

$$RF_k(d) = i \;\equiv\; C_k(i) \text{ is the first step on a path from } k \text{ to } d$$

Hence, if a message has to be sent from k to d it is transmitted by k along $C_k(RF_k(d))$[3]. By applying this repeatedly, the message finally arrives at its destination. As a result we require that each message is accompanied by the identification of its destination. A message m is a pair, $(m.h, m.c)$. We call $m.h$ the header and $m.c$ the contents of the message. The destination of the message is given by $m.h.d$.

On each processor we introduce for each incoming channel $D(i)$ a process that accepts messages from $D(i)$ and forwards them, if they are destined for another processor or handles them if they are destined for this processor. Since in this way all these processes use the outgoing channels we need exclusive access to each outgoing channel. This is done by using semaphore $cs(l)$ for each channel $C(l)$. Given the routing function, the forwarding of messages becomes trivial. This results in the following process definition.

```
proc Router (var D: channel)
|[ var m: message; l: int;
  *[ D?m.h;
     [ m.h.d = k → D?m.c; "handle m"
     [] m.h.d ≠ k → l := RF(m.h.d);
                    P(cs(l)); C(l)!m.h || D?m.c; C(l)!m.c; V(cs(l))
     ]
   ]
]|
```

A process on processor k sending a message to a processor $d \neq k$ executes a similar program as is used for forwarding a message. We use the following procedure.

```
proc Send (d: int; z: messsage_body)
|[ var h: message_header; l: int;
  h.d := d; l := RF(d);
  P(cs(l)); C(l)!h; C(l)!z; V(cs(l))
]|
```

With respect to the correctness of this program we have to prove two facts. First, each message is transported to the correct destination, and second, no deadlock occurs. The first follows simply by induction on the length of the path generated by RF. In order to analyze the possibility of deadlock we introduce some notation. Let $Q = (V_Q, E_Q)$ be the graph defined as follows.

$$x \in V_Q \;\equiv\; x \in E_I \wedge w_I(x) = 1$$
$$(x, y) \in E_Q \;\equiv\; (\exists d :: RF_{s.x}(d) = x \wedge RF_{t.x}(d) = y)$$

In words, V_Q consists of the physical connections in the implementation graph and E_Q contains a pair of physical connections (x, y) if a message may be routed from x onto y. The graph Q is (statically) determined by both G_I and the routing function RF.

For each element x of V_Q we have a router process, *Router* (x). A deadlock is a stable state in which a number of these router processes is suspended on actions other than input actions along their incoming channels. By inspection of the text we learn that router processes may become suspended in one of three ways:

1. on inputs from D,
2. on a P operation on $cs(t)$ or on a communication along $C(t)$,
3. on handling a message locally.

[3]Each channel $C_k(i)$ corresponds to an element v of the set E_I as well. With a slight abuse of notation we also say $RF_k(d) = v$.

The first case does not contribute to a deadlocked state. Let $R = (V_R, E_R)$ be the graph defined by

$$V_R = V_Q$$
$$(x, y) \in E_R \equiv \textit{Router}\,(x) \text{ is waiting for } y \text{ (the second case above)}$$

This graph is determined by RF and G_I as well, but it dynamically changes. From the program text of *Router* we observe that R is a subgraph of Q. Consider a non-empty set of suspended routers, corresponding to a subset of V_R. We can choose a path in this set for which we have two possibilities: the path is infinite or it is finite. In the first case the path must contain a cycle since V_R is finite. In the second case we have that for the last element of this path it must be suspended on handling a message. We conclude that deadlock can be avoided and, hence, messages are delivered eventually when the following two rules are obeyed.

1. The graph Q is acyclic (hence, R is acyclic as well).
2. Handling a message eventually terminates.

The first rule actually restricts RF. We may ask whether we can find such an RF for each strongly connected network such that still messages can be transmitted between each pair of processors. The answer is twofold: indeed, it is possible to find such an RF by restricting the routes that messages may take. If this is considered to be too expensive the technique of introducing virtual channels ([2, 5]) may be applied. It goes beyond the scope of this paper to discuss the latter in detail. However, using virtual channels boils down to multiplexing several of these virtual channels onto one physical channel. Multiplexing a number of channels onto one physical channel is in fact a special case of the mapping problem and, hence, our solutions can be used for multiplexing as well.

The second rule is realized in most message passing systems by claiming a buffer for each message that arrives. Since we pursue a low latency, we develop different solutions.

In the above algorithms we have used a type *message* consisting of a header and a contents. In process *Router* the entire contents is read before it is forwarded. This method is known as *store-and-forward* routing (see, for instance, [5, 10]). A much lower latency is obtained using *cut-through* or *wormhole* routing in which case a message is split into packets of some fixed size. A second advantage of using cut-through routing is that the message buffers may be of a fixed, limited size. An algorithm for cut-through routing can be used instead of the above store-and-forward routing. This does not affect the correctness of the algorithm, only the efficiency.

7 A distributed implementation

Using the routing processes of the previous section, we complete the implementation of channel communication actions for a channel c which is mapped onto a path of positive length. Let the source and the target of the path be denoted by A and B respectively. Hence, the problem is to find implementations for $c!e$ on A and $c?x$ on B that satisfy $C0$ through $C2$. From section 5 we recall the shared-memory implementation consisting of (3) and (4).

$$c?x: \quad c.v := \textit{address}(x); V(c.s); P(c.t)$$
$$c!e: \quad P(c.s); c.v\uparrow := e; V(c.t)$$

We distribute these actions across A and B by distributing the data structure associated with c. If an action cannot be performed because the variable is not local, a message is sent. Handling the message then results in the required action. Not all actions can be dealt with in this way and this guides the distribution. We analyze for each action which processor has to perform it.

1. $c.v\uparrow := e$ has to be performed on B since it directly refers to writing a value into the memory. Hence, $c.v$ is local to B and $c.v := address(x)$ has to be performed on B as well.

2. Execution of a P operation can result in suspension. Therefore, execution of a P operation cannot be implemented by sending a message. Hence, processor A stores $c.s$ and B stores $c.t$.

We obtain the following distributed implementation.

(B) $c?x$: $c.v := address(x)$; Send $(A,$ "$V(c.s)$"$)$; $P(c.t)$
(A) $c!e$: $P(c.s)$; Send $(B,$ "$c.v\uparrow := e$"$)$; Send $(B,$ "$V(c.t)$"$)$

We use process *Router*, developed in the previous section for forwarding the messages. The only modification made in *Router* is in handling the message at the destination; the part referring to forwarding messages remains the same.

The two *Send* actions in $c!e$ can become one action. It is not necessary to include anything referring to the V operations in a message since processes *Router* on A and B can execute the appropriate actions based just on the receipt of a message. This means that sometimes and empty message is sent, denoted by "–".

(B) $c?x$: $c.v := address(x)$; Send$(A, -)$; $P(c.t)$
(A) $c!e$: $P(c.s)$; Send(B, e);

Process *Router* on A becomes

```
proc Router (var D: channel)
|[ var m: message; l: int;
  *[ D?m.h;
     [ m.h.d = A → V(c.s)
     [] m.h.d ≠ A → l := RF(m.h.d);
                    P(cs(l)); C(l)!m.h || D?m.c; C(l)!m.c; V(cs(l))
     ]
   ]
]|
```

and on B

```
proc Router (var D: channel)
|[ var m: message; l: int;
  *[ D?m.h;
     [ m.h.d = B → D?(c.v↑); V(c.t)
     [] m.h.d ≠ B → l := RF(m.h.d);
                    P(cs(l)); C(l)!m.h || D?m.c; C(l)!m.c; V(cs(l))
     ]
   ]
]|
```

thus executing the actions that cannot be performed by the sender of the messages. The correctness follows from the discussions in the previous sections. Notice that handling of messages in both *Routers* always terminates since only non-blocking operations are executed.

We have now mapped exactly one channel, viz., c. For a process on A it is an outgoing channel and for a process on B it is an incoming channel. This is reflected in the program text of the routers for these processors. In the general case, each processor has both incoming and outgoing channels and it has more than one of them. We observe that we need to make a distinction between incoming and outgoing

channels, i.e., between actions that are part of $c!e$ and those that are part of $c?x$. Therefore, we introduce a message type in the header, $m.h.tp$ that can be either *Shriek* or *Query* corresponding to these two cases respectively. We also make the channel a parameter by introducing it as part of the message header, $m.h.c$. This results in the following implementation.

```
proc Send (d: int; c: channel_id; tp: message_type; z: messsage_body)
|[ var h: message_header; l: int;
  h.d := d; h.c := c; h.tp := tp; l := RF(d);
  P(cs(l)); C(l)!h; C(l)!z; V(cs(l))
]|

proc Router (var D: channel)
|[ var m: message; l: int;
  *[ D?m.h;
     [ m.h.d = k → [ m.h.tp = Shriek → D?(m.h.c.v↑); V(m.h.c.t)
                   [] m.h.tp = Query  → V(m.h.c.s)
                   ]
     [] m.h.d ≠ k → l := RF(m.h.d);
                    P(cs(l)); C(l)!m.h || D?m.c; C(l)!m.c; V(cs(l))
     ]
   ]
]|
```

(B) $c?x$: $c.v := address(x)$; Send$(A, c, Query, -)$; $P(c.t)$

(A) $c!e$: $P(c.s)$; Send$(B, c, Shriek, e)$;

Finally, we observe that A, B, and c are parameters of this implementation. Together with the results of the previous section we conclude that this implementation is correct for arbitrary channels and pairs of processors.

This completes our solution to the problem of constructing m_E. These programs can be generated automatically by a compiler, based on descriptions of the computation graph and the implementation graph.

8 A communication library

Until now we have looked at the problem as a compilation problem, i.e., as a problem that has to be solved statically, by a compiler. In this section we develop a different point of view. We first generalize the programs such that channels can be defined dynamically, under control of the program. Then we introduce the concept of *Remote Process Calls* which supports an asynchronous form of message passing. This results in a small collection of procedures comprising a library of communication routines. We rely heavily on the implementations given in the previous sections.

8.1 Dynamic configuration

In the mapping of a channel onto a path, the name of a channel was used to send information between the two partners in a communication. On processor A, c was used to send information and on processor B it was used to receive information. In fact, the pairs (A, c) and (B, c) played the role of two "sides" of a channel. From now on we call these sides *ports*, denoted by c_A and c_B respectively.

In the previous section two ports were connected automatically through the fact that c had to be mapped onto a path from A to B. The data structure $(c.s, c.v, c.t)$ was distributed. Hence, port c_A was

actually identified by $c_A.s$ and c_B by $(c_B.v, c_B.t)$. This asymmetry reflects the fact that the first port is used for sending values while the other one is used for receiving values. We restore the symmetry by associating a port p with $(p.s, p.v, p.t)$. In this way, two ports make up a *pair* of channels and a port can be used both for sending and for receiving values. On each processor k we introduce an array of ports, $p_k(i : 0 \leq i < N)$. Two ports $p_A(i)$ and $p_B(j)$ for $0 \leq A, B < P$ and $0 \leq i, j < N$ can be connected to form a channel. For each port p the partner port has to be recorded, identified by a pair consisting of a processor and a port index. We record this as part of the data structure associated with a port p: $(p.proc, p.port)$ and we leave it to the program to connect two ports. Notice that for consistency reasons we need

$$p_A(i).proc = B \wedge p_A(i).port = j \quad \equiv \quad p_B(j).proc = A \wedge p_B(j).port = i$$

For the implementation we can use basically the same programs as in the previous section. The only distinction is that we now describe the sending and receiving of messages along a *port*. Instead of using the channel notation $p(i)?x$ and $p(i)!e$, we use routines *PortReceive* and *PortSend* respectively.

```
proc Send (proc, port: int; tp: message_type; z: messsage_body)
|[ var h: message_header; l: int;
   h.d := proc; h.c := port; h.tp := tp; l := RF(d);
   P(cs(l)); C(l)!h; C(l)!z; V(cs(l))
]|

proc PortReceive (i: int; var x: message_body)
|[ p(i).v := address(x); Send(p(i).proc, p(i).port, Query, −); P(p(i).t) ]|

proc PortSend (i: int; e: message_body)
|[ P(p(i).s); Send(p(i).proc, p(i).port, Shriek, e) ]|

proc Router (var D: channel)
|[ var m: message; l: int;
   *[ D?m.h;
      [ m.h.d = k → [ m.h.tp = Shriek → D?(p(m.h.c).v↑); V(p(m.h.c).t)
                    ▯ m.h.tp = Query  → V(p(m.h.c).s)
                    ]
      ▯ m.h.d ≠ k → l := RF(m.h.d);
                    P(cs(l)); C(l)!m.h ‖ D?m.c; C(l)!m.c; V(cs(l))
      ]
   ]
]|
```

We use the following procedure for connecting two ports.

```
proc Connect (i, pr, j: int)
|[ p(i).proc := pr; p(i).port := j ]|
```

Notice that *Connect* has to be called twice, once for both ports.

8.2 Remote Process Calls

We recall from section 6 one of the requirements for deadlock-avoidance: handling a message must always terminate. In the above this has been realized by making communication synchronous. A different solution is obtained when a message is accompanied by the process that handles the message. If such a

message arrives at its destination the process that handles it is started as a new process. Its first task then is to retrieve the message.

This idea is incorporated as follows. We recall that starting a new process is denoted by the keyword **fork**. The new process has to be started by process *Router* on receipt of a header. Since both this new process and *Router* use the same incoming channel for a while, exclusion is required through a semaphore s. Both the incoming channel and this semaphore are parameters to the new process. When the process has retrieved the message from the channel it performs a V operation on the semaphore. We assume that a process is identified by an index in a table, $PROC(i : 0 \leq i < M)$. Hence, starting process i with the above parameters amounts to execution of **fork** $PROC(i)(D, s)$.

On the sending side, sending such a message is actually similar to starting a new process though not locally, through **fork**, but remotely on an arbitrary processor. Therefore, we call sending such a message a *remote process call.* Its implementation is simpler than the implementation of channels, since no synchronization is required. For the implementation of channels we already used indices $0, \ldots, N-1$. For these remote process calls we use indices $N, \ldots, N+M-1$.

proc Rcall (d, i: int; m: message_body)
|[$Send(d, i+N, Shriek, m)$]|

Process *Router* changes accordingly by adding the distinction between channel communications and Remote Process Calls.

proc Router (**var** D: channel)
|[**var** m: message; l: int; s: semaphore;
  $s := 0$;
  *[$D?m.h$;
    [$m.h.d = k \rightarrow$ [$m.h.tp = Shriek \wedge m.h.c < N \rightarrow D?(p(m.h.c).v \uparrow); V(p(m.h.c).t)$
                  [] $m.h.tp = Shriek \wedge m.h.c \geq N \rightarrow$ **fork** $PROC(m.h.c - N)(D, s); P(s)$
                  [] $m.h.tp = Query \rightarrow V(p(m.h.c).s)$
                  ]
    [] $m.h.d \neq k \rightarrow l := RF(m.h.d);$
                  $P(cs(l)); C(l)!m.h \parallel D?m.c; C(l)!m.c; V(cs(l))$
    ]
  ]
]|

The table, $PROC$, can be filled in during execution of the program. A procedure that is used as such a remote process has to satisfy the restrictions mentioned above. This implies that it is of the following shape.

proc rpc (**var** D: channel; **var** s: semaphore)
|["declaration of local variables";
  $D?\ldots; V(s)$;
  "other actions"
]|

Some form of synchronization is required while the table is being filled in, since otherwise processors may call a process while it has not yet been defined. This may be implemented, for example, by some form of global synchronization mechanism. It goes beyond the scope of this paper to discuss this in detail.

For some applications it may be too expensive (in time and/or memory) to start a new process for each call. The reason to start it as a new process is that in this way deadlock is avoided, as it is guaranteed that the message is accepted and that process *Router* regains control. If the new process contains no communication or synchronization actions by which it becomes suspended and it is also guaranteed to

terminate, it is not necessary to start it as a process; it can be called simply as a procedure. This can be recorded in table *PROC* and process *Router* can be adapted accordingly. An example of such an operation is *remote write* through which a processor can write a value in the memory of another processor. Operation *remote read* however, requires the value that is read to be returned to the caller. Since this requires communication, *remote read* has to be started as a new process. Both operations are easily written using this mechanism of Remote Process Calls.

An abstraction mechanism that resembles our Remote Process Calls is the Remote *Procedure* Call. Such a procedure call is semantically equivalent to a regular procedure call; however, it is executed by another processor. As such it is synchronous: execution of the process on the caller is delayed until the procedure call returns. We can implement each of the two mechanisms in terms of the other.

9 Conclusion

We have developed a small communication library based on a reasonable abstraction of a parallel machine. Although the individual steps were not very complicated, the resulting program is quite involved. Through a precise specification and a careful separation of concerns we were able to maintain a clear picture. In this way we could also show the correctness of the program. In the implementation we have avoided to use extra buffering. In this way the latency of communication is reduced.

The library can be extended with other communication primitives. Examples are the multicast (one process sends a message to a number of other processes), the barrier synchronization (synchronization of a number of processes) and global aggregation of data. These operations can be implemented through the use of Remote Procedure Calls or, more efficiently, by incorporating them in in the routing system described in section 6.

The mechanism of Remote Process Calls is also known as *Asynchronous Remote Procedure Calls*. Sometimes the messages corresponding to such a process call are called *active messages* ([4], pp.481-496).

10 Acknowledgements

I want to thank Anne Kaldewaij and Peter Hilbers for detailed comments on an earlier version of this paper. I want to thank Anne Kaldewaij and Tom Verhoeff for helpful comments on the current version.

References

[1] Back, R.J.R., On the correctness of refinement steps in program development, report A-1978-4, Åbo Akademo, Department of computer science, Finland.

[2] Dally, W.J., Seitz, C.L., Deadlock free message routing in multiprocessor interconnections networks, Dept. Comp. Science, California Institute of Technology, Tech. Rep. 5206:TR:86, 1986.

[3] Dijkstra, E.W., A discipline of programming, Prentice Hall, Englewood Cliffs, NJ, 1976.

[4] Hempel, R., et. al. (eds.), Parallal Computing, Special issue on message passing, 20 (1994).

[5] Hilbers, P.A.J., Lukkien, J.J.: Deadlock-free message routing in multicomputer networks. In: Distributed Computing, Vol. 3, Nr. 4, 1989.

[6] Hilbers, P.A.J., Processor networks and aspects of the mapping problem, Cambridge University Press, Cambridge, 1991.

[7] Hoare, C.A.R., Communicating Sequential Processes, CACM, Vol.21(8), pp.666-677, August 1978.

[8] Inmos ltd, Transputer Reference Manual, Prentice Hall, London, 1988.

[9] JaJa, J., An introduction to parallel algorithms, Addison Wesley, Amsterdam. 1992.

[10] Kumar, V., et al., Introduction to parallel computing, Benjamin/Cummings publishing Company, Redwood City CA, 1994.

[11] Martin, A.J. An axiomatic definition of synchronization primitives, Acta Informatica, 16 (1981) 219-235.

[12] McColl, W.F., General purpose parallel computing, Programming Research Group, Oford University, april 1992.

[13] MPI: a message passing interface standard, University of Tennessee, Knoxville, 1994.

[14] Geist, A., et al., PVM 3 user's guide and reference manual, Oak Ridge National Laboratory, 1994.

[15] van de Snepscheut, J.L.A., Martin, A.J., Design of synchronization algorithms, In: M. Broy (ed.), Constructive methods in computing science, NATO ASI series, Vol. F55, Springer-Verlag, Berlin, 1989.

[16] van de Snepscheut, J.L.A., What computing is all about, Springer-Verlag, New York, 1993.

[17] van de Snepscheut, J.L.A (Editor), Mathematics of program construction, conference proceedings, LNCS 375, Heidelberg, 1989.

Computing Science Reports

Department of Mathematics and Computing Science
Eindhoven University of Technology

In this series appeared:

91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.
91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.

91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.
91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.
91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.
91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.

91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.
92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.
92/04	J.P.H.W.v.d.Eijnde	Program derivation in acyclic graphs and related problems, p. 90.
92/05	J.P.H.W.v.d.Eijnde	Conservative fixpoint functions on a graph, p. 25.
92/06	J.C.M. Baeten J.A. Bergstra	Discrete time process algebra, p.45.
92/07	R.P. Nederpelt	The fine-structure of lambda calculus, p. 110.
92/08	R.P. Nederpelt F. Kamareddine	On stepwise explicit substitution, p. 30.
92/09	R.C. Backhouse	Calculating the Warshall/Floyd path algorithm, p. 14.
92/10	P.M.P. Rambags	Composition and decomposition in a CPN model, p. 55.
92/11	R.C. Backhouse J.S.C.P.v.d.Woude	Demonic operators and monotype factors, p. 29.
92/12	F. Kamareddine	Set theory and nominalisation, Part I, p.26.
92/13	F. Kamareddine	Set theory and nominalisation, Part II, p.22.
92/14	J.C.M. Baeten	The total order assumption, p. 10.
92/15	F. Kamareddine	A system at the cross-roads of functional and logic programming, p.36.
92/16	R.R. Seljée	Integrity checking in deductive databases; an exposition, p.32.
92/17	W.M.P. van der Aalst	Interval timed coloured Petri nets and their analysis, p. 20.
92/18	R.Nederpelt F. Kamareddine	A unified approach to Type Theory through a refined lambda-calculus, p. 30.
92/19	J.C.M.Baeten J.A.Bergstra S.A.Smolka	Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.
92/20	F.Kamareddine	Are Types for Natural Language? P. 32.

92/21	F.Kamareddine	Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.
92/22	R. Nederpelt F.Kamareddine	A useful lambda notation, p. 17.
92/23	F.Kamareddine E.Klein	Nominalization, Predication and Type Containment, p. 40.
92/24	M.Codish D.Dams Eyal Yardeni	Bottum-up Abstract Interpretation of Logic Programs, p. 33.
92/25	E.Poll	A Programming Logic for Fω, p. 15.
92/26	T.H.W.Beelen W.J.J.Stut P.A.C.Verkoulen	A modelling method using MOVIE and SimCon/ExSpect, p. 15.
92/27	B. Watson G. Zwaan	A taxonomy of keyword pattern matching algorithms, p. 50.
93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Baeten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.
93/12	K.M. van Hee	Systems Engineering: a Formal Approach Part IV: Analysis Methods, p. 63.
93/13	K.M. van Hee	Systems Engineering: a Formal Approach Part V: Specification Language, p. 89.

93/14	J.C.M. Baeten J.A. Bergstra	On Sequential Composition, Action Prefixes and Process Prefix, p. 21.
93/15	J.C.M. Baeten J.A. Bergstra R.N. Bol	A Real-Time Process Logic, p. 31.
93/16	H. Schepers J. Hooman	A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27
93/17	D. Alstein P. van der Stok	Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.
93/18	C. Verhoef	A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.
93/19	G-J. Houben	The Design of an Online Help Facility for ExSpect, p.21.
93/20	F.S. de Boer	A Process Algebra of Concurrent Constraint Programming, p. 15.
93/21	M. Codish D. Dams G. Filé M. Bruynooghe	Freeness Analysis for Logic Programs - And Correctness?, p. 24.
93/22	E. Poll	A Typechecker for Bijective Pure Type Systems, p. 28.
93/23	E. de Kogel	Relational Algebra and Equational Proofs, p. 23.
93/24	E. Poll and Paula Severi	Pure Type Systems with Definitions, p. 38.
93/25	H. Schepers and R. Gerth	A Compositional Proof Theory for Fault Tolerant Real-Time Distributed Systems, p. 31.
93/26	W.M.P. van der Aalst	Multi-dimensional Petri nets, p. 25.
93/27	T. Kloks and D. Kratsch	Finding all minimal separators of a graph, p. 11.
93/28	F. Kamareddine and R. Nederpelt	A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.
93/29	R. Post and P. De Bra	GOLD, a Graph Oriented Language for Databases, p. 42.
93/30	J. Deogun T. Kloks D. Kratsch H. Müller	On Vertex Ranking for Permutation and Other Graphs, p. 11.
93/31	W. Körver	Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.
93/32	H. ten Eikelder and H. van Geldrop	On the Correctness of some Algorithms to generate Finite Automata for Regular Expressions, p. 17.
93/33	L. Loyens and J. Moonen	ILIAS, a sequential language for parallel matrix computations, p. 20.

93/34	J.C.M. Baeten and J.A. Bergstra	Real Time Process Algebra with Infinitesimals, p.39.
93/35	W. Ferrer and P. Severi	Abstract Reduction and Topology, p. 28.
93/36	J.C.M. Baeten and J.A. Bergstra	Non Interleaving Process Algebra, p. 17.
93/37	J. Brunekreef J-P. Katoen R. Koymans S. Mauw	Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.
93/38	C. Verhoef	A general conservative extension theorem in process algebra, p. 17.
93/39	W.P.M. Nuijten E.H.L. Aarts D.A.A. van Erp Taalman Kip K.M. van Hee	Job Shop Scheduling by Constraint Satisfaction, p. 22.
93/40	P.D.V. van der Stok M.M.M.P.J. Claessen D. Alstein	A Hierarchical Membership Protocol for Synchronous Distributed Systems, p. 43.
93/41	A. Bijlsma	Temporal operators viewed as predicate transformers, p. 11.
93/42	P.M.P. Rambags	Automatic Verification of Regular Protocols in P/T Nets, p. 23.
93/43	B.W. Watson	A taxomomy of finite automata construction algorithms, p. 87.
93/44	B.W. Watson	A taxonomy of finite automata minimization algorithms, p. 23.
93/45	E.J. Luit J.M.M. Martin	A precise clock synchronization protocol,p.
93/46	T. Kloks D. Kratsch J. Spinrad	Treewidth and Patwidth of Cocomparability graphs of Bounded Dimension, p. 14.
93/47	W. v.d. Aalst P. De Bra G.J. Houben Y. Kornatzky	Browsing Semantics in the "Tower" Model, p. 19.
93/48	R. Gerth	Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.
94/01	P. America M. van der Kammen R.P. Nederpelt O.S. van Roosmalen H.C.M. de Swart	The object-oriented paradigm, p. 28.

94/02	F. Kamareddine R.P. Nederpelt	Canonical typing and Π-conversion, p. 51.
94/03	L.B. Hartman K.M. van Hee	Application of Marcov Decision Processe to Search Problems, p. 21.
94/04	J.C.M. Baeten J.A. Bergstra	Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.
94/05	P. Zhou J. Hooman	Formal Specification and Compositional Verification of an Atomic Broadcast Protocol, p. 22.
94/06	T. Basten T. Kunz J. Black M. Coffin D. Taylor	Time and the Order of Abstract Events in Distributed Computations, p. 29.
94/07	K.R. Apt R. Bol	Logic Programming and Negation: A Survey, p. 62.
94/08	O.S. van Roosmalen	A Hierarchical Diagrammatic Representation of Class Structure, p. 22.
94/09	J.C.M. Baeten J.A. Bergstra	Process Algebra with Partial Choice, p. 16.
94/10	T. verhoeff	The testing Paradigm Applied to Network Structure. p. 31.
94/11	J. Peleska C. Huizing C. Petersohn	A Comparison of Ward & Mellor's Transformation Schema with State- & Activitycharts, p. 30.
94/12	T. Kloks D. Kratsch H. Müller	Dominoes, p. 14.
94/13	R. Seljée	A New Method for Integrity Constraint checking in Deductive Databases, p. 34.
94/14	W. Peremans	Ups and Downs of Type Theory, p. 9.
94/15	R.J.M. Vaessens E.H.L. Aarts J.K. Lenstra	Job Shop Scheduling by Local Search, p. 21.
94/16	R.C. Backhouse H. Doombos	Mathematical Induction Made Calculational, p. 36.
94/17	S. Mauw M.A. Reniers	An Algebraic Semantics of Basic Message Sequence Charts, p. 9.
94/18	F. Kamareddine R. Nederpelt	Refining Reduction in the Lambda Calculus, p. 15.
94/19	B.W. Watson	The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46.

94/20	R. Bloo F. Kamareddine R. Nederpelt	Beyond β-Reduction in Church's λ→, p. 22.
94/21	B.W. Watson	An introduction to the Fire engine: A C++ toolkit for Finite automata and Regular Expressions.
94/22	B.W. Watson	The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.
94/23	S. Mauw and M.A. Reniers	An algebraic semantics of Message Sequence Charts, p. 43.
94/24	D. Dams O. Grumberg R. Gerth	Abstract Interpretation of Reactive Systems: Abstractions Preserving ∀CTL*, ∃CTL* and CTL*, p. 28.
94/25	T. Kloks	$K_{1,3}$-free and W_4-free graphs, p. 10.
94/26	R.R. Hoogerwoord	On the foundations of functional programming: a programmer's point of view, p. 54.
94/27	S. Mauw and H. Mulder	Regularity of BPA-Systems is Decidable, p. 14.
94/28	C.W.A.M. van Overveld M. Verhoeven	Stars or Stripes: a comparative study of finite and transfinite techniques for surface modelling, p. 20.
94/29	J. Hooman	Correctness of Real Time Systems by Construction, p. 22.
94/30	J.C.M. Baeten J.A. Bergstra Gh. Ştefanescu	Process Algebra with Feedback, p. 22.
94/31	B.W. Watson R.E. Watson	A Boyer-Moore type algorithm for regular expression pattern matching, p. 22.
94/32	J.J. Vereijken	Fischer's Protocol in Timed Process Algebra, p. 38.
94/33	T. Laan	A formalization of the Ramified Type Theory, p.40.
94/34	R. Bloo F. Kamareddine R. Nederpelt	The Barendregt Cube with Definitions and Generalised Reduction, p. 37.
94/35	J.C.M. Baeten S. Mauw	Delayed choice: an operator for joining Message Sequence Charts, p. 15.
94/36	F. Kamareddine R. Nederpelt	Canonical typing and Π-conversion in the Barendregt Cube, p. 19.
94/37	T. Basten R. Bol M. Voorhoeve	Simulating and Analyzing Railway Interlockings in ExSpect, p. 30.
94/38	A. Bijlsma C.S. Scholten	Point-free substitution, p. 10.

94/39	A. Blokhuis T. Kloks	On the equivalence covering number of splitgraphs, p. 4.
94/40	D. Alstein	Distributed Consensus and Hard Real-Time Systems, p. 34.
94/41	T. Kloks D. Kratsch	Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.
94/42	J. Engelfriet J.J. Vereijken	Concatenation of Graphs, p. 7.
94/43	R.C. Backhouse M. Bijsterveld	Category Theory as Coherently Constructive Lattice M . Theory: An Illustration, p. 35.
94/44	E. Brinksma J. Davies R. Gerth S. Graf W. Janssen B. Jonsson S. Katz G. Lowe M. Poel A. Pnueli C. Rump J. Zwiers	Verifying Sequentially Consistent Memory, p. 160
94/45	G.J. Houben	Tutorial voor de ExSpect-bibliotheek voor "Administratieve Logistiek", p. 43.
94/46	R. Bloo F. Kamareddine R. Nederpelt	The λ-cube with classes of terms modulo conversion, p. 16.
94/47	R. Bloo F. Kamareddine R. Nederpelt	On Π-conversion in Type Theory, p. 12.
94/48	Mathematics of Program Construction Group	Fixed-Point Calculus, p. 11.
94/49	J.C.M. Baeten J.A. Bergstra	Process Algebra with Propositional Signals, p. 25.
94/50	H. Geuvers	A short and flexible proof of Strong Normalazation for the Calculus of Constructions, p. 27.
94/51	T. Kloks D. Kratsch H. Müller	Listing simplicial vertices and recognizing diamond-free graphs, p. 4.
94/52	W. Penczek R. Kuiper	Traces and Logic, p. 81
94/53	R. Gerth R. Kuiper D. Peled W. Penczek	A Partial Order Approach to Branching Time Logic Model Checking, p. 20.