Formal Aspects of Computing (2002) 13: 471–492

Formal Aspects of Computing

On Verification of Parallel Message-Passing Processes

Stein Krogdahl[1] and Olav Lysne[2]

[1]Department of Informatics, University of Oslo, Norway
[2]Simula Research Laboratory, Oslo, Norway

Abstract. One way to reason about parallel processes is to assume that the execution of each process is subdivided into 'small enough' steps, and that these are executed in an interleaved fashion, thus obtaining a sequential program. The steps should be so small that for any parallel execution there will, in a suitable sense, exist a corresponding interleaved execution ending in the same state. The usual way to ensure this is to require that each step should contain at most one global access. However, if the global entities are communication channels, then larger steps may in some cases be allowed, and this may make reasoning about the programs easier. This paper explores these cases, and discusses consequences for verification and deadlock avoidance.

Keywords: Interleaved execution; Parallel processes; Program verification

1. Introduction

To reason, intuitively or formally, about parallel programs is usually a demanding task. One approach for obtaining a firmer grasp on this problem is to 'reduce' the parallel execution to a sequential one by subdividing each process of the parallel execution into a sequence of small enough 'elementary steps', and simulate a parallel execution by allowing the steps to be executed in an interleaved fashion. What 'small enough steps' should mean will heavily depend upon how often the processes access global entities. An obviously safe requirement is simply to say that each interleaved step should make at most one global access, and that this access should act as an atomic action relative to the global entity in question.

By this technique, we can to a large extent reason about parallel executions as if they were sequential ones. The underlying assumption is then that for each parallel execution (PE) of the real system, there is a corresponding interleaved execution (IE) where each individual process experiences exactly the same sequence of actions, and where the final state after IE is the same as after PE. There exist several extensions of Hoare logic into parallel settings based on this idea (and on steps with at most one global access), both for shared variables [Lam80, Owe92] and for the message-passing paradigm [StS95].

Reasoning about interleaved, sequential executions can be done in more or less formal ways (and the choice here is not a topic for this paper). However, one natural way to approach such a task is to find an

Correspondence and offprint requests to: Stein Krogdahl, Department of Informatics, University of Oslo, PO Box 1080 Blindern, N-0316 Oslo, Norway, E-mail: steinkr@ifi.uio.no

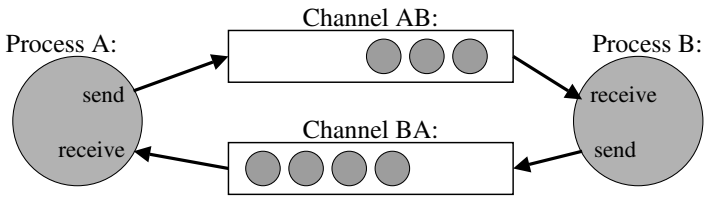

Fig. 1. Illustration of an example discussed in the text. The legal steps of the processes A and B may start with a receive-call and end with a send-call.

invariant that will hold between the steps of the interleaved execution, and from which we can conclude what we want to know about the system. The verification then simply amounts to showing that the invariant is true initially, and that it is preserved by each step (when executed alone).

Obviously, the larger (and thus often 'more natural') the interleaved steps are, the more we can hope that a simple and 'intuitive' invariant will be strong enough to prove what we want. However, making the steps larger very soon gets in conflict with the requirement that each parallel execution should have a corresponding interleaved one. Thus, it is interesting to ask whether there are situations where the interleaved steps may contain more than one global access, without destroying this property. If the global entities are traditional variables, this seems not to be the case. However, if the global entities are communication lines transporting messages, then such cases exist.

One such case can be sketched as follows (see Fig. 1): Assume that two parallel processes A and B are connected by two FIFO-lines AB and BA, carrying messages from A to B and from B to A respectively. Furthermore, assume that process A consists of execution steps that (in the general case) start with one input from BA and end with one output to AB (but where one or both may be missing), and that B is subdivided into similar steps that may input from AB at the start and output to BA at the end. We currently appeal to the reader's intuition when claiming that, in a suitable sense, each parallel execution of such a system will have a corresponding interleaved one.

However, under many other conditions, we can easily construct parallel executions for which no corresponding interleaved execution exists. Assume, e.g., that we allow more than one process to issue messages to the same line, and that we have a set-up with one FIFO-line to which two processes A and B may issue messages, and from which a process C receives messages. Further, assume that both A and B have just one execution step, and that the one of A, *sA*, issues two messages *mA1* and *mA2* to the line, and that the one of B, *sB*, issues one message *mB*. If A and B run in parallel, process C may obviously receive the message sequence *mA1, mB, mA2*. However, no interleaved execution of the steps *sA* and *sB* will make C receive the same sequence.

In reported verification efforts, similar steps to those of the first example above (Fig. 1) are used as units for interleaved executions and thereby for verification, see e.g. [Kro78, KrL97, LyK97, Ver87, GaT90]. However, none of these papers contain a more systematic discussion of what types of steps can be allowed, and how large they may be.

Another area where the size of the interleaved steps has an impact is within debugging of distributed systems. Chandy and Lamport presented a method by which a consistent snapshot of a distributed system can be taken [ChL85]. This method suffers from supporting the test of only a very limited set of global state predicates. In [MaN91] Marzullo and Neiger propose an algorithm that from a terminated (or halted) execution allows a server to reason on whether a given predicate is *definitely* true or *possibly* true at some state in the execution.

Although both of the above results have been refined and improved since they were first published (see e.g. [GaW96, VeD95, MaI92, Lee99]), they are still severely limited by the number of states they need to consider. The reason is that they implicitly rely on a notion of atomic steps, where each step is allowed to perform at most one input or output of a message from/to a line.

The theme of this paper is to start with the simple case described in Fig. 1, and try to generalise it as far as possible. We will look at questions like: What will happen if the lines are not FIFO? What if more than one process can send to or receive from one given line (as in the second example above)? What if the capacity of the lines to store messages is limited? What about deadlock, which can easily occur in such systems? What type of properties can be proven by this type of reasoning?

In the discussion of these questions we shall refer to the communication lines as 'channels', and to the

portions or steps used for the interleaved execution as 'actions'. We start by making the whole setting more precise.

2. The General Setting

We first define a 'parallel executing system' or 'PE-system' as follows:

1. A PE-system has the ability to execute in parallel a number of sequential processes, each controlled by a suitable program. Each process can access its own local variables and may have local input/output, but can communicate with other processes only through 'channels' (see below).
2. For each sequential process, the elementary steps of the execution are gathered into larger portions called 'actions' (e.g. by indicating the start of each new action by special statements in the program text).
3. The only allowed communication between the processes is through a number of 'channels' (modelling some type of message transportation medium). Each channel will at any time contain a number of 'messages', and can only be accessed through the operations 'send' and 'receive'. A call on 'send' to a given channel should be accompanied by a message, and should have the effect that the message is stored in that channel.
   A call on 'receive' will pick and remove one of the messages stored in the channel, and deliver it as result. Calls on 'receive' or 'send' may sometimes have to wait to be serviced, e.g. because the channel is empty. In that case the calls are not necessarily serviced by the channel in the sequence they arrived, but when serviced the service operation should be an atomic action.

The details about how 'receive' works (e.g. concerning which message is picked if there is more than one, or what happens if there are no messages in the channel) will vary in the different cases discussed below. We shall first assume that the channels have unlimited capacity to store messages, but we shall later consider channels with limited capacity.

An obvious fact about this model is that a 'receive'-call picking a certain message from a channel is always serviced after the 'send'-call that inserted that message into the channel. This relation corresponds to the 'happened-before' relation used in [ChL85, GaW96, Lee99, MaI92, MaN91, StS95].

A program controlling a given execution of a PE-system is called a PE-program, and such programs will become important in later sections. However, for the time being we shall only discuss executions of PE-systems (seen as sequences of state-changes and operation calls) without reference to an underlying program.

2.1. Parallel, Interleaved and Complete Executions

The different sequential processes in a PE-system will normally execute in full parallel, so that actions in different processes can freely overlap. However, we shall also talk about 'interleaved executions' for such systems, by which we shall mean executions where only one action from one process is allowed to run at a time, and where an action that is started must run to full completion before another action (maybe in another process) is started.

To distinguish interleaved executions from normal executions we shall usually refer to the latter as 'parallel executions' (even though an interleaved execution is also a normal execution). For short we shall refer to them as I-executions and P-executions. The theme of this paper is, roughly speaking, to describe under what conditions the set of I-executions for a PE-system is representative for the larger and less tractable set of P-executions for that system.

By a 'complete execution' we shall mean an (I- or P-) execution where all the processes of the PE-system end with a completed action. Thus, after a complete execution, we know, for example, that no 'send' or 'receive' calls are waiting to be serviced at any of the channels.

2.2. Observationally Equivalent Executions

Two executions of a PE-system may often look the same from within each sequential process, even if they can differ in the way the processes are scheduled with respect to each other. More formally, we shall say that two (P-)executions E1 and E2 are 'observationally equivalent' if the following is true:

(i) The sequence of state-changes and operation calls performed by each sequential process is exactly the same in E1 and in E2, also concerning data involved, and to which channels the 'send'- and 'receive'-calls are made. We may therefore, for each call on 'send' or 'receive' in E1, talk about 'the corresponding call' in E2, and vice versa.

(ii) Messages that are read from a channel by corresponding 'receive'-calls in E1 and E2 are also inserted into that channel by corresponding 'send'-calls in the two executions.

(iii) The remaining content in the channels after E1 and after E2 should be such that additional 'receive'-calls to a channel should yield the same sequence of messages after E1 and after E2. In some cases below 'receive' itself will be non-deterministic, and in that case this requirement should instead say that the *set of possible message sequences obtainable by a sequence of 'receive'-calls should be the same.*

The reason for requiring (iii) is, roughly speaking, that we want two observationally equivalent executions still to be observationally equivalent if they are extended by identical actions.

For short we shall write 'obs-equivalent' instead of 'observationally equivalent'. Obviously, if two executions are obs-equivalent, they will end up with exactly the same values in the local variables of each sequential process.

3. The Basic Proposition

Below we will, in four different ways, specify how the 'send'- and 'receive'-operations of the channels should work and how these operations may be called with respect to the processes and the actions. Our first aim is to prove the following statement for these four cases (whose names will be explained later):

Proposition 1. If a PE-system conforms to one of the cases F11, F*1, F1* and U** described below, then, for each complete P-execution of the system, we can find an obs-equivalent (and thereby complete) I-execution.

Note that for *any* PE-system the opposite is always true, as any complete I-execution is also a complete P-execution.

3.1. The Four Cases of Proposition 1

Below we shall look at the four cases of Proposition 1. Each case will be described by specifying the following points:

A: Which message a 'receive'-call to the channels will choose, if more than one is present.

B: A set of restrictions concerning how 'send' and 'receive' for the different channels can be called from the processes and their actions.

For all cases we assume that if 'receive' is called for an empty channel, then it will wait until a message arrives.

Case F11

A: The operation 'receive' will always pick the oldest message contained in the given channel. Thus the channels are of the FIFO-type.

B1: For each channel, only one given process is allowed to call 'send' and only one given process is allowed to call 'receive' for that channel. (Thus we may say that each channel 'leads from one given process to another').

B2: Each action may make any number of 'send'-calls and 'receive'-calls to different channels (within the restrictions of B1), but in each action all 'receive'-calls must occur before all 'send'-calls.

The name 'F11' should be understood as follows: The 'F' indicates that we have FIFO-channels and '11' indicates that for each channel there is only one sending and one receiving process. The later variations over this scheme are, we hope, obvious.

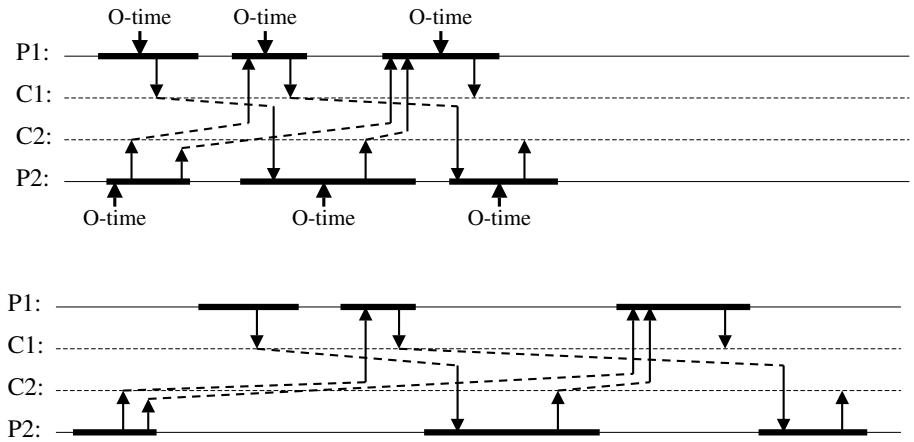

Fig. 2. Illustration to the proof of case F11. Top: A P-execution in a system with two processes P1 and P2 and two channels C1 (leading from P1 to P2) and C2 (leading from P2 to P1). Bottom: The corresponding obs-equivalent I-execution constructed in the proof.

Proof of Proposition 1.

Case F11

We look at an arbitrary complete P-execution PE of the system (see top execution in Fig. 2), and our obligation is to come up with an obs-equivalent I-execution IE. To obtain this we first, for each action A performed (by some process) in PE, choose some point of time called the 'ordering-time of A', or 'O-time of A' for short. For each action A, this O-time should be chosen within the time-span of the action, and so that there is no 'send'-call before it and no 'receive'-call after it (see Fig. 2). From the definition of case F11 such an O-time can obviously be found for each action.

The I-execution IE will now be formed by performing the actions of PE one by one in order of increasing O-times (see the bottom part of Fig. 2). For this execution to be consistent we must show that the 'receive'-calls in IE will never be to empty channels, and that the message received in each such call is 'the same' as in the corresponding 'receive'-calls in PE (where 'the same message' means that it is sent by corresponding 'send'-calls in IE and in PE). If IE is consistent in this way, it will obviously also be obs-equivalent to PE.

To show this consistency, we first prove the following lemma:

Send/Receive Lemma. Assume that R is a 'receive'-call in PE, and that S is the 'send'-call that issued the message received by R. Then, in IE, the 'send'-call corresponding to S will be scheduled before the 'receive'-call corresponding to R.

To prove this, we first observe that S obviously occurs before R in PE. By construction, the O-time of the action in which S occurs, say action AS, will be before the 'send'-call S, and likewise the O-time of the action in which R occurs, say action AR, will be after R. Together this shows that the O-time of AS will be before that of AR, and therefore AS will be scheduled before AR in IE. From this the lemma above follows directly.

We then prove the the following lemma (which is rather obvious for the current case F11, but slightly more involved for cases F*1 and F1*):

Channel Lemma. Assume that C is a channel, that $S_1, S_2, ..., S_m$ is the sequence of 'send'-calls done to C in PE, and that $R_1, R_2, ..., R_k$ is the sequence of 'receive'-calls done to C in PE. Denote by S_i' and R_j' the calls in IE that corresponds to S_i and R_j respectively. Then $S_1', ..., S_m'$ will be scheduled in that order in IE, and likewise for $R_1', ..., R_k'$. We also know that $m \geqslant k$.

The reason why $m \geqslant k$ is simply that each 'receive'-call R_j in PE obtains a message issued from one of the 'send'-calls S_i. The reason why the first part is true is that all 'send'-calls S_i are done from the same sequential process (see B1 above) and likewise for all R_j. Thus, as the ordering of calls within each process is the same in PE and IE, the lemma follows directly.

Having established these two lemmas, we can establish the consistency of IE as follows: Choose a channel C, and assume as in the Channel Lemma that $S_1, S_2, ..., S_m$ and $R_1, R_2, ..., R_k$ are the sequences of 'send'-calls

and 'receive'-calls done to C in PE. Then, because of the FIFO-nature of C, we know that R_i receives the message sent by S_i for all $i = 1, ..., k$, and that the messages left in C after termination of PE will be those sent by $S_{k+1}, ..., S_m$ respectively.

Then, again, denote by S'_i and R'_j the calls to C in IE that corresponds to S_i and R_j respectively. From the two lemmas above, we know:

1. In IE the call S'_i will be scheduled before R'_i, for all $i = 1, ..., k$.
2. The calls $S'_1, ..., S'_m$ will be scheduled in that order in IE, and likewise for $R'_1, ..., R'_k$.

Thus, if we let IE really unfold as an execution we will, again because of the FIFO-nature of C, experience that R'_i reads the message sent by S'_i for all $i = 1, ..., k$, and that the remaining messages in C after termination of IE are those sent by $S'_{k+1}, ..., S'_m$ respectively. Thus, IE is consistent in the way we want.

*Case F*1*

The change here compared to case F11 is that in B1 we now allow any number of processes to call 'send' for each channel (but still only one can call 'receive'). On the other hand, we restrict B2 so that each action can do at most one 'send'-call (which must still be done after all 'receive'-calls). Notice that we restrict B2 so that the (failing) second example in the introduction is excluded.

The main change in the proof compared to that of case F11 is that the O-time of an action A should now be chosen immediately before the 'send'-call of A is serviced by the channel (that is, after the previous call to the same channel is serviced). If there is no 'send'-call in A we choose any time in A after its last 'receive'-call.

With this choice we can easily prove the Send/Receive Lemma as for case F11. To see that also the Channel Lemma is valid, observe that in IE we schedule the actions in the order they do their only 'send'-call (if any). Thus, for a given channel C, even if more than one process does 'send'-calls to it, the sequence of these send-calls will be the same in IE and in PE. The 'receive'-calls can be treated as in case F11.

Thus, we are in the same situation as in case F11, and the obs-equivalence of PE and IE follows as for that case.

*Case F1**

In this case we restrict things in the opposite way as for case F*1. Compared to case F11, B1 now allows only one process to call 'send' for a given channel, while multiple processes are allowed to call 'receive'. B2 is restricted so that each action can make at most one 'receive'-call, and this must be made before any 'send'-call.

This case is symmetrical (with respect to send and receive) to case F*1, and by choosing the O-time of each action immediately after the 'receive'-call of the action is serviced (if any), the obs-equivalence of PE and IE follows in the same way.

*Case U***

This case models a system where the channels are not of the FIFO-type. We therefore give fully new versions of points A and B:

A: If the channel is not empty, the procedure 'receive' will non-deterministically choose a message among those currently in the channel (while if the channel is empty, it will have to wait).

B: An action can do any number of 'receive'-calls and 'send'-calls on different channels, but in each action all 'receive'-calls must be made before any 'send'-call. There are no restrictions concerning what processes can make 'send' or 'receive' calls to what channels.

For the proof we again assume a given PE and we choose the O-time of each action to be somewhere after all 'send'-calls and before all 'receive'-calls of that action (exactly as for case F11), and we schedule the actions in IE according to this.

Here, we can easily prove the *Send/Receive Lemma* in the same way as in the three previous cases. However, concerning the *Channel Lemma*, one can easily check that *it is not valid for this case*. However, because of the non-determinism of the channels it turns out that we can manage without it.

This is because we know, from the Send/Receive Lemma, that when we arrive at a 'receive'-call in IE, the message we should receive to make IE obs-equivalent to PE, is already in the channel. As our only obligation

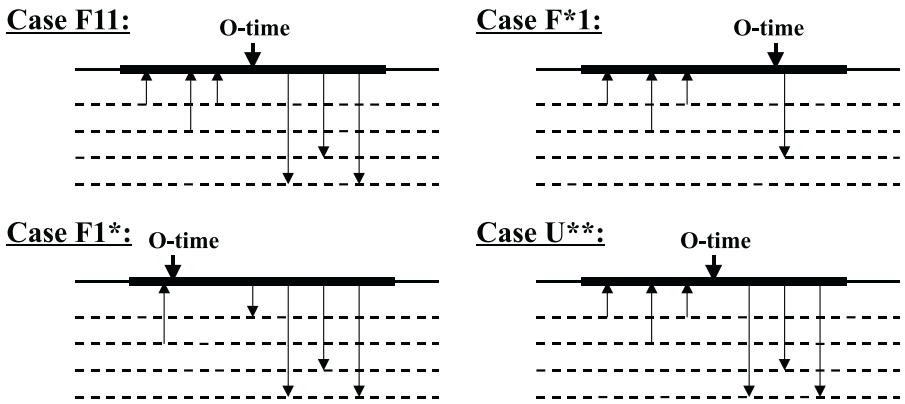

Fig. 3. Illustration of how the actions may access channels and how the O-time should be chosen for the four cases of Proposition 1.

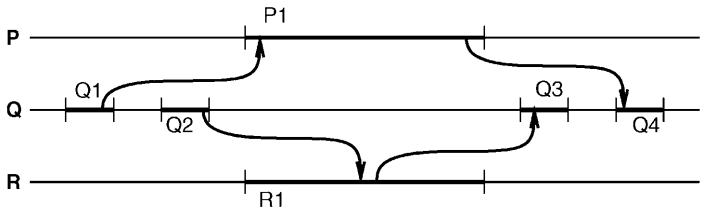

Fig. 4. A P-execution for which no obs-equivalent I-execution exists. All messages are sent on the same channel. See text for further explanation.

is to come up with *one legal obs-equivalent* I-execution IE, we can utilise the non-determinism of the channels and for each 'receive'-call pick exactly the message that makes IE obs-equivalent to PE. This concludes the proof for case U**. □

Figure 3 sums up the number of 'send'- and 'receive'-calls allowed, and how the O-time should be chosen, for the four cases of Proposition 1.

A case that does not work

Looking at the cases F11, F*1 and F1* using FIFO channels, one might suspect that there is a fourth case with FIFO channels that could also work, which is characterised by 'one "receive" and one "send" in each action and all processes accessing all channels'. However, the example in Fig. 4 shows that this is not possible. There, all messages are sent on one common FIFO channel, and because of the messages from Q1 and Q2, P1 has to be done before R1 in an I-execution. However, because of the messages to Q3 and Q4, R1 has to be done before P1. Thus, no obs-equivalent I-execution is possible.

3.2. Using Non-Blocking Versions of 'Receive'

In the four cases above, we have assumed that a 'receive'-call will not return until it can really deliver a message. However, in many situations it is convenient to use a version of 'receive' that always returns, and that delivers a special marker, called e.g. 'nomessage', if no message can be delivered. We shall refer to the original version of 'receive' as 'blocking' and to the new version as 'non-blocking'. We shall say that a 'receive'-call (blocking or non-blocking) is 'successful' in a certain execution if it returns and delivers a genuine message.

Then, to what extent can we use non-blocking instead of blocking 'receive'-calls in Proposition 1 above? We could try the following: When we shall decide, for one of the four cases above and for a given execution, whether the requirements of that case is fulfilled, we simply say that only the successful 'receive'-calls in the execution count. However, if we try to repeat the proof of Proposition 1 under these conditions, we get

trouble. This is because, for a given P-execution, the I-execution produced in the proof of Proposition 1 will often have more messages in the channels at the corresponding 'receive'-calls. Thus, for a given P-execution we can in general not find an I-execution where the channels are empty all the places where 'receive'-calls delivered 'nomessage' in the P-execution.

A reasonable way to get around this is to say that the non-blocking version of 'receive' may deliver 'nomessage' also when there *are* messages in the channel, and that this choice is made non-deterministically. This will naturally model a situation where it takes some ('non-deterministic') time from when a message is sent into the channel until it can be fetched by a 'receive'-call.

Thus, we adapt this version of 'receive' and we say that a P-execution PE fits into one of the four cases if it does so in the 'traditional way' when we disregard those calls that resulted in 'nomessage'. With this setting it is easy to see that what we tried above can be proven. During the construction of an I-execution IE in the proofs of the four cases we simply disregard the unsuccessful 'receive'-calls, and can thus do the constructions as before. Finally, as our aim is to show that there *exists* a 'legal' I-execution (conforming to the same case) which is obs-equivalent to PE, we can simply choose that 'receive' returns 'nomessage' in IE whenever it did in PE, regardless of the content of the channel at that point in IE.

In the above discussion our view has been to study ('already performed') executions as such. The more usual situation is that we are writing a program in which, for example, we represent each action by a separate section of the program, and that we want to do this so that each such section, *when executed as an action*, always conforms to the restrictions of the chosen case. If, for example, we use a non-blocking 'receive' and the chosen case allows only one 'receive'-call in each action, then the program may call 'receive' repeatedly within each such section (maybe even to different channels), but it has to stop doing this as soon as a genuine message is received.

Similar considerations to the above can be made also for most of the discussions in the rest of this paper. However, to keep things simple, we shall along the way assume that only blocking 'receive'-calls (and later also 'send'-calls) are used. It is left to the interested reader to work out in detail how non-blocking versions will affect the conclusions.

4. PE-Programs and Verification

Recall that a 'PE-program' is some type of description that can control executions in a PE-system. By definition, a PE-program should consist of a description of each of the sequential processes of the PE-system, e.g. in some programming language. We also require a PE-program to say nothing about the scheduling and relative speed of the processes (except what is implied by the 'send'- and 'receive'-calls).

Finally we require that a PE-program is described so that we get a well-defined subdivision of each process into a sequence of actions (e.g. by indicating the boundary between the actions by special statements in the program text). Thus, with the actions of any execution defined, we can talk about 'a PE-program of type X', in the sense that all executions of the program will conform to Case X, where X is F11, F*1, F1* or U** of Proposition 1.

Each sequential process of a PE-program may take local input, and may themselves have local non-determinism. Thus, with the added non-determinism from the scheduling of the parallel processes, a PE-program will represent a large (usually unlimited) set of executions for a PE-system. The aim of a verification effort is usually to find, and give convincing arguments for, properties about the set of executions spanned by a certain PE-program.

In this setting there is a certain class of properties that can naturally be handled. Formally they are predicates defined over the set of all complete executions of a PE-system, and we shall call them 'scheduling independent predicates', or only 'SI-predicates':

Definition. A predicate defined over all complete executions in a PE-system is called an SI-predicate if, whenever it is true for one (P-)execution E, it is also true for all (P-)executions that are obs-equivalent to E.

An SI-predicate could typically say something about the content of the local variables and the channels after the execution, or say something about the sequence of value-changes occurring for the local variables in one of the sequential processes.

Predicates that will typically *not* be SI-predicates are, for example, statements about which of two given actions in different processes will terminate first, or about the number of messages in a channel when a certain 'send' or 'receive' operation is performed.

Now, assume that we have a PE-program of type F11, F*1, F1* or U**, and want to verify that a certain SI-predicate SIP is true for all complete (P-)executions for that PE-program. We can then try to prove that SIP is valid for all I-executions of that PE-program. If we manage, Proposition 1 and the definition of SI-predicates will tell us that *all* complete (P-)executions of the PE-program will in fact satisfy SIP.

As discussed in the Introduction, a useful technique for reasoning about I-executions is to give a global invariant (which may then talk about values of all local variables and the contents of all the channels) and verify that all actions of the PE-program maintain this invariant when executed alone. However, any type of formal or informal reasoning may be used to verify that a certain SI-predicate of interest is valid for all I-executions, and thus also for all P-executions.

4.1. On Deadlock

In the discussions above, we have only considered complete executions in which all actions (and thus also all 'send'- and 'receive'-calls) that are started are also fully finished. However, it would be valuable for verification purposes if we could also prove similar statements concerning absence or presence of deadlock. For the discussion of this we shall say that a PE-program is 'executed in I-mode' if only one action is executed at a time in the whole system, so that an I-execution results. Otherwise it is said to 'execute in P-mode'.

By definition, we say that a deadlock situation for a PE-program executing in P-mode has occurred if some of the sequential processes end up waiting in 'receive'-calls to empty channels, and that the rest of the processes have finished their statements. Note here that some of the processes may be non-terminating (e.g. an infinite loop), and to obtain a deadlock situation all these processes must have stopped in a 'receive'-call to an empty channel.

For the discussion below one should note that even if an I-execution is also a P-execution, it is not at all obvious that a PE-program that may deadlock when run in I-mode will also deadlock when run in P-mode. A program run in I-mode may in fact very easily deadlock, simply by issuing a 'receive'-call to an empty channel. However, if we then could shift to P-mode we could often remedy the situation by starting a parallel action with a 'send'-call that could satisfy the waiting 'receive'-call. In I-mode this is by definition not possible.

To take care of this, we make the following definition:

Receive-safe I-mode. A PE-program is said to execute in 'receive-safe I-mode' if it executes in I-mode with the additional restriction that an action can only be started if the contents of the channels will allow all 'receive'-calls of that action to be satisfied.

From the above definition of receive-safe, we see that when a PE-program executed in receive-safe I-mode deadlocks, it is not because 'receive'-calls to empty channels are never finished, but because no action can legally be started.

For proving the proposition below, it turns out that we have to restrict case U** a little. The reason for this is intuitively that, as long as an action has not successfully completed all its 'receive'-calls, it should not in any way have influenced what messages other processes may read. Thus, we make the following definition:

Case UA (Case U** with restriction A).** A PE-system is said to conform to case U**A if it conforms to case U**, and in addition either all actions make at most one 'receive'-call or each channel is read (by 'receive'-calls) by only one process. Likewise, a PE-program whose executions conform to this is said to be of type U**A.

We can now prove the following proposition:

Proposition 2. A PE-program of type F11, F*1, F1* or U**A is deadlock free when executed in P-mode if and only if it is deadlock free when executed in receive-safe I-mode.

Proof. We first observe that for the cases allowed by this proposition, one of the following two conditions (a) and (b) will be true. Below we will often treat these separately:

(a) Each action makes at most one call on 'receive'. This is true for case F1* and 'part of' case U**A.
(b) Each channel is read (by 'receive') by only one process. This is true for cases F11, F*1 and the rest of U**A.

We do the proof by showing that there is a deadlocking P-execution if and only if there is a deadlocking I-execution.

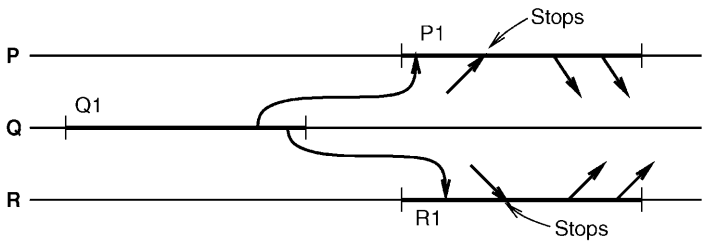

Fig. 5. A deadlocking P-execution. See text for further explanation.

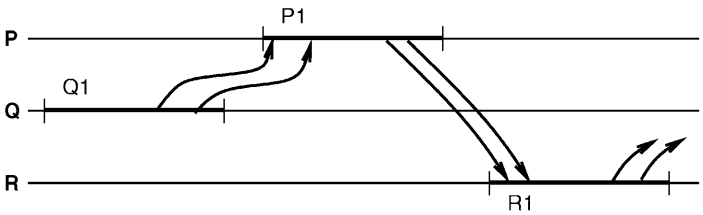

Fig. 6. An I-execution corresponding to the P-execution in Fig. 5. See text for further explanation.

We first assume that a deadlocking P-execution PE for the PE-program is given, and we will construct an I-execution that deadlocks when executed in receive-safe I-mode. In PE, at least one of the local processes ends in a 'receive'-call to an empty channel. The 'half-executed' actions containing such calls will be referred to as "blocked actions". These obviously occur as the last action of a sequential process, and they have not executed any 'send'-call at all (as they stopped in a 'receive'-call).

We now observe that we can obtain a consistent P-execution PE', by executing as in PE from the start, but stopping those processes that end with blocked actions immediately before that action starts, and let the rest run to completion. We may say that we obtain PE' from PE by 'removing' the blocked actions. The execution PE' is obviously complete, and it is consistent in the sense that a 'receive'-call in PE' will receive the 'same' message as the corresponding 'receive'-call in PE. The reason is as follows: In case (a) the blocked actions will not contain any (successful) 'receive'-call at all, so removing them will make no difference. In case (b) we may remove 'receive'-calls, but other 'receive'-calls to the same channels will not be disturbed, as they are done by the same process and thus earlier than the removed ones.

From the complete P-execution PE' we then construct an obs-equivalent I-execution IE' as in the proofs for the four cases of Proposition 1. IE' can obviously be seen as executed in receive-safe I-mode, and the situation after IE' is exactly as after PE'.

We shall now show that none of the blocked actions can be started in a receive-safe I-execution after IE' (and as all other processes are terminated, this will complete the proof). We again look separately at the cases (a) and (b):

(a) We here know that none of the blocked actions we removed did any successful 'receive'-calls before they stopped. Therefore the channels in question were all empty after PE', and thus after IE'. Thus, none of the blocked actions can be started after IE' in receive-safe I-mode.

(b) Since each channel is read by only one process, what is left in each channel is exactly as after PE', and what was read by a blocked action is not disturbed by whether the other blocked actions are executed in parallel or not. Thus, as all 'receive'-calls could not be satisfied for these actions in P-mode, there are not enough messages to start them in receive-safe I-mode.

For the proof in the other direction, we assume that we have a receive-safe I-execution IE that has deadlocked. This means that IE ended with a completed action, and that there are processes whose next action is allowed to start for internal reasons (they have not finished), but where none of these actions can have their 'receive'-calls satisfied. To obtain a P-execution that deadlocks we first take IE itself as the first part of a P-execution, and then let all the processes that have actions of the above type proceed in parallel.

Now, it is rather obvious that, for exactly the same reason that they could not be started in receive-safe I-mode, they will all stop in a 'receive'-call when executed in P-mode. Roughly, this is because the processes will not interfere with each other, and because no additional 'send'-calls will ever be executed. Again, one has

to look at the cases (a) and (b) separately, but this is rather straightforward and the details are left to the reader. □

It follows from this proposition that questions concerning deadlock for a PE-program of type F11, F*1, F1* or U**A can be answered instead by studying the same questions for receive-safe I-executions for that PE-program.

One may wonder whether restricting case U** to U**A was really necessary to make Proposition 2 go through. The example in Figs 5 and 6 shows that allowing case U** in full will at least not work. In that example all messages are sent along the same channel. As shown in Fig. 5, a P-execution may easily deadlock. However, all I-executions will execute nicely, e.g. as shown in Fig. 6.

5. Introducing Finite Channel Capacities

Until now we have assumed that the capacity of the channels to hold messages has been unlimited, but we shall now turn to the case where the channels may have a finite capacity, given as an upper limit on the number of messages a channel can hold. One obvious new complication then is that also 'send' may have to wait if the channel is 'full'. Thus, systems may also deadlock in new ways, as some of the processes may wait in 'send'-calls and others in 'receive'-calls.

We shall, in the discussion that follows, also allow the channel-capacities to be zero. This should mean that messages cannot be stored in the channel at all, and message transfers therefore have to be made directly from one process to the other, in a sort of rendezvous. We shall see in the following that this case nicely fits in with the others in propositions etc.

5.1. Some Immediate Results

We may first of all observe that any execution in a PE-system with limited channel capacities is also a legal execution in the PE-system formed by removing the capacity restrictions. Thus, if we have a PE-program and (by any type of reasoning) have established that some predicate P is valid for all complete executions of this program in the unrestricted system, then P will also be valid for all complete executions (if any) in the restricted system. However, there may obviously be properties of the restricted system that cannot be caught in this way, e.g. properties that stem from the special scheduling enforced by the limited capacities.

Note that the predicate P above should be defined from the properties of an execution E as such, and not, for example, depend upon how these relate to the 'set of possible executions', as this set may change when we restrict the system.

In the above, it is rather obvious that the case with zero capacity causes no problems. This is also true for Propositions 3 and 4 below, but we leave it to the reader to check the details.

Next, the following can easily be proven:

Proposition 3. Assume that PE is a complete P-execution conforming to case F11, F*1, F1* or U** of Proposition 1, and that IE is an obs-equivalent I-execution scheduled according to legal O-time choices in the proof of Proposition 1 for that case. Then the maximum number of messages in a channel during PE will not exceed the maximum number of messages in that channel during IE. Also, this maximum will in IE be reached between actions, or at the end.

Proof. This proposition simply relies on the fact that in the proof of all the cases F11, F*1, F1*, and U** the O-times (used to define the order of the actions in IE) are always chosen *after* all 'receive'-calls in the action and *before* all 'send'-calls in the action. So, let T be any point of time during PE, and assume that the number of messages in a channel C at T is N.

To obtain a point of time in IE when there are at least N messages in C, we look at exactly the point T' in IE when all actions A with O-times smaller than T in PE are (fully) performed. From the way the O-times are chosen, we then know that all 'receive'-calls before T' in IE occur before T in PE and all 'send'-calls before T in PE occur before T' in IE. Thus, simply by counting calls we know that the number of messages in C at T' in IE is at least N. And also, this happens between actions, or at the end, of IE. □

We can use this proposition for verification purposes as follows: Assume you are given a PE-program of type F11, F*1, F1* or U** executing in an environment with channels of limited capacity. Then, just disregard the

capacity limits and try to prove that for any complete I-execution of the PE-program the capacity limits will never be exceeded (and it is here enough to consider the content of the channels between the actions, and at the end).

If this can be proven, we know from Proposition 3 that all *complete* P-executions of the PE-program will also stay within these limits. Thus any complete execution of this program will be totally unaffected by the capacity constraints, and we can therefore disregard the capacity limits altogether and do all further reasoning concerning complete executions as in the previous sections.

Deadlock

In the above we have only discussed complete executions, and therefore said nothing about deadlock. However, if anything can be said about this along the same lines, it would obviously be of interest. Concerning the first observation above there is in fact very little we can say. It is rather straightforward to see that even if a given PE-system without capacity restrictions will never deadlock, it may easily deadlock if we impose capacity restrictions.

It turns out that the opposite is also true. That is, we can construct examples where the unrestricted system may deadlock, while the same system with certain capacity restrictions will not. One such example can be constructed as follows: We first observe that with a channel C of capacity one, which can be accessed freely by all processes, we can obtain mutual exclusion zones in the sequential processes simply by starting each such zone with 'C.send("some message")' and by ending it with 'M := C.receive' (and use C only in this way). Through this mechanism we can, for example, force an otherwise parallel execution to behave as an I-execution. Thus if the mutual exclusion zones are similar to the actions in Figs 5 and 6 (where C is different from the channel used there), then we see that the system will not deadlock. However, if we remove the capacity restriction on C, then we no longer get an I-execution, and deadlock may occur as in Fig. 6.

We now turn to the situation around Proposition 3. Here, it would be nice if we could prove that if all complete I-executions of a PE-program in an unrestricted system never fill the channels above certain limits, then the PE-program will never deadlock in the unrestricted system *if and only if* it will never deadlock in a system where these limits are enforced as capacities. This may seem 'almost obvious', but it remains to be shown that executions that cannot be prolonged to a complete execution (because of a deadlock) will not make trouble. To show that this is the case, we prove the following proposition:

Proposition 4. Assume that, for a given PE-program of type F11, F*1, F1* or U**, executing in an unrestricted system, it holds that no complete I-execution will ever exceed some given limits for the number of messages in the channels. Then no P-execution (complete or not) of this PE-program executing in an unrestricted system will ever exceed the given limits.

Proof. We choose some point of time T during a P-execution PE of the PE-program (executed in an unrestricted system). We concentrate on the part PE' of this execution that is performed before T, and assume that PE' ends in state S. Now, from state S we produce a (possibly new) continuation of PE' by letting all processes that at T were inside an action proceed, and run that action to completion if possible, but not starting another action. We call the resulting execution PE″. If all processes are able to complete their action, PE″ is complete, and we can use Proposition 3 to conclude that the limits were not exceeded at T. Otherwise, we can reason as follows:

When producing PE″, we know that all actions that at T have completed all their 'receive'-calls are always able to complete the full action (as the channels have unlimited capacity). However, there may be other processes that cannot complete their action, and these must have stopped in a 'receive'-call to an empty channel. As in the proof of Proposition 2, we call these 'blocked actions'.

Also as in Proposition 2, we observe that we obtain a consistent P-execution, by executing from the start as in PE″, but stopping those processes that ended with blocked actions immediately before that action starts (and let the rest run to completion). That is, we have 'removed' the blocked actions from PE″. To see that this is a consistent execution, we look at the different cases:

*Cases F11 and F*1:* In these cases only one process can receive messages from each channel, and removing a number of 'receive'-call at the end of one process will therefore not disturb the other processes (as the capacity of the channels is unlimited).

Case F1:* In this case each blocked action does at most one 'receive'-call, and each blocked action therefore did not make any successful 'receive'-call at all. Removing such an unsuccessful 'receive'-call will obviously not disturb the other processes.

*Case U**:* In this case we are not restricted by any FIFO-discipline of the channels. Therefore, removing a few 'receive'-calls will not disturb the legality of the remaining calls to 'pick' exactly the same messages as they did before.

Thus, we can remove the blocked actions without disturbing the rest of the execution, and thereby end up with a complete execution PE‴, where some processes complete their final action before T and some do it after T.

Let S‴ be the situation at T in P‴. Then, as P‴ is complete we can conclude from Proposition 3 that in S‴ the content of the channels will not exceed the given limits. However, it is also obvious that we can obtain the situation S by starting at S‴ and redo each blocked action up to the state it had at T in the original execution PE. As this will only involve 'receive'-calls, the limits are obviously also obeyed in S. □

Now, assume that we have shown that all complete I-executions of a certain PE-program will never fill the channels above certain limits, when executed in an unrestricted system. Equipped with the above proposition, we can then reason as follows:

First assume that we enforce the given limits as capacities and have a P-execution PE that deadlocks. We then know that this deadlock will not include any unsatisfied 'send'-calls, and the reason is that PE could equally well be executed in the unrestricted system, and then (at least) one such 'send'-call could be performed, resulting in a situation where the limits were exceeded. However, this is against Proposition 4. Thus, the deadlock includes only 'receive'-calls, and it will therefore also constitute a deadlock in the unrestricted system.

The other way around is even simpler: Assume we have a P-execution in the unrestricted system that deadlocks. As the limits are never exceeded in this execution it can also be executed in the restricted system, and will here obviously deadlock in the same way.

Thus, under the above assumptions, we know that the PE-program will never deadlock in the unrestricted system *if and only if* it will never deadlock in a system where these limits are enforced as capacities.

5.2. Equivalence Statements for Capacity-Restricted Systems

The two verification techniques discussed above will either not be able to catch all facts about a system or can only be used in special cases. It would therefore be convenient if we could obtain results more like those of Propositions 1 and 2, saying that something is valid for all (P-)executions *if and only if* it is valid for all I-executions.

However, with capacity restrictions things obviously are a little more complicated. To see this, look at the following example (conforming to case F11): A system has two processes A and B, and a channel with capacity one leading from A to B. Each action of A makes two 'send'-calls and those of B make one 'receive'-call. If actions of A and B may overlap this works fine, but obviously no I-execution is possible for this system.

This means that if we want each P-execution which is performable within the capacity restrictions to have an obs-equivalent I-execution, then we must somehow relax the capacity restrictions for the I-executions.

By introducing suitable capacity relaxations and case restrictions, we shall be able to give an equivalence statement similar to Proposition 1. However, in one direction this new proposition goes through with fewer restrictions than in the other, and as this may be of interest in itself, we first state and prove this one-way relation. We first make the following definitions:

Case U1*. A system is said to conform to case U1* if it conforms to case U**, and in addition (1) each action makes at most one 'receive'-call, and (2) only one process can call 'send' for each channel.

The name U1* is chosen because this restricts case U** in a way similar to the restrictions of case F1*.

Weakly restricted I-executions. Assume that the channels of a system have capacity restrictions, and that for each channel only one process is allowed to send messages to it. We shall then say that a complete I-execution is 'weakly restricted' by these channel capacities if, immediately before an action of a certain process is started, none of the channels to which this process can send messages have their capacity exceeded. In addition, after the whole I-execution the number of messages in each channel should not exceed the capacity.

Thus, in an I-execution which is weakly restricted by given channel capacities the number of messages in a channel may often exceed the given capacity. However, at certain specified points of time the restrictions should be obeyed, but these points are not the same for all channels. To avoid misunderstandings, we will sometimes write 'fully restricted by the capacities' when we simply mean 'restricted by the capacities'.

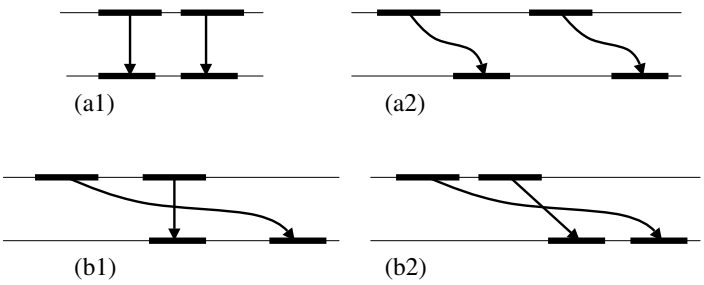

Fig. 7. The above examples ((a1) and (a2) for case F1*, and (b1) and (b2) for case U1*) demonstrate Proposition 5. See text for further comments.

In case F1* and case U1* there is only one process that can call 'send' for each of the channels. Thus, we can talk about weakly restricted I-executions, and we can prove the following proposition:

Proposition 5. For capacity-restricted PE-systems satisfying case F1* or case U1* the following is true: For each complete P-execution restricted by given channel capacities there is an obs-equivalent (complete) I-execution which is weakly restricted by the same capacities.

Proof. We assume that a P-execution PE obeying the restrictions of case F1* or case U1* is given. As in the proof of case F1* for Proposition 1 and legal also for case U**, and thus also for case U1*, we choose an O-time for each action immediately after the only 'receive'-call (if any, otherwise before any 'send'-call), and in the normal way we construct an I-execution IE where the actions are performed in the order of their O-times. As shown in the proof of Proposition 1, this makes up a consistent obs-equivalent I-execution if we disregard the capacity restrictions.

We now claim that IE obeys the capacity restrictions in the weak sense. To show this we look at a specific process P, a specific channel C 'out of' P and a specific action A of P, and we denote by OA the O-time of A. Obviously, the capacity restriction for C is obeyed at OA in PE (as all restrictions are fully obeyed all the time in PE).

We first assume that action A either has no 'receive'-call, or that it has a 'receive'-call to another channel than C. We then look at the point of time BA immediately before A in IE. To see what is in C at this time we first observe that exactly the same number of 'send's are done to C at BA in IE and at OA in PE. This is because P is the only process making 'send'-calls to C. Further, because of the way the O-times are chosen, we see that all actions that have made a 'receive'-call to C before OA in PE are done (fully) before BA in IE, and those actions that have made such a 'receive'-call after OA in PE are done after BA in IE. Thus, the number of 'send'-calls and 'receive'-calls made to C before BA in IE are exactly the same as those made to C before OA in PE. Thereby, the capacity constraint for C is obeyed immediately before A (at BA) in IE.

Finally, we should consider the case where A itself has a 'receive'-call to channel C, and we then look at the time OA' immediately *before* this 'receive'-call in PE (after all previous calls to C). The capacity restriction for C is obviously also obeyed at this point in PE, and by the same argument as above we see that the number of messages in C at BA in IE is the same as at OA' in PE. Thus things are OK also for this case.

Obviously, the capacity restrictions are also obeyed after IE has finished. □

One may notice here that Proposition 5 also works fine if some channels have capacity zero; see, for example, Fig. 7, where the P-execution (a1) of case F1* is resulting in the weakly restricted I-execution (a2). Also notice that the P-execution in (b1) should be considered legal for case U1* in Fig. 7 with a channel capacity of one. Here, the edge pointing straight down represents a rendezvous in which a message can pass directly from the sender to the receiver even if the channel is full. The resulting weakly restricted I-execution is (b2).

We have only proved Proposition 5 for case F1* and case U**A, and one may ask whether we could have proved it for other cases. Figure 8 gives an example showing that it can at least not be proved for the full cases F11, F*1 or U**. Concerning cases F11 and F*1 we have not been able to prove the proposition for restrictions of these cases that allows situations not already covered by case F1*.

To make the above proposition go through also in the other direction, we have to restrict the cases a little further, as follows:

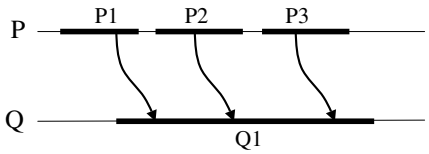

Fig. 8. The above example, conforming to cases F11, F*1 and U** (but not to case U1*), shows that we could not allow these cases in full in Proposition 5. All messages are sent on the same channel, and its capacity is one. In an I-execution, action Q1 obviously has to come after P3, thus forming an I-execution which is not weakly restricted by the capacities (failing at the start of P3).

Case F1*B (Case F1* with restriction B). A system is said to conform to case F1*B if it conforms to case F1*, and in addition all 'send'-calls made in one action are done to the same channel.

Case U1*C (Case U1* with restriction C). A system is said to conform to case U1*C if it conforms to case U1*, and in addition each action makes at most one 'send'-call.

We can now prove the following proposition:

Proposition 6. For capacity-restricted PE-systems satisfying case F1*B or case U1*C, the following is true: For each complete P-execution restricted by given channel capacities there is an obs-equivalent complete I-execution which is weakly restricted by the same capacities, and vice versa.

We give the proof separately for case F1*B and case U1*C, and we first look at case F1*B.

*Proof for case F1*B.* The 'forward' direction follows directly from Proposition 5 proved above. For the proof in the other direction, we assume that a complete weakly restricted I-execution IE is given, and our aim is to construct a (fully) restricted obs-equivalent P-execution PE. The idea of the proof will be to construct a directed graph where the vertices correspond to the 'send'-calls and 'receive'-calls of IE, and where there is a directed edge (u, v) whenever there is a reason why u has to occur before v in a fully restricted P-execution. We shall call this graph the 'ordering graph' of IE.

The main part of the proof will be to show that this ordering graph has no cycles (except in a special case where they do no harm). Thus, a basic theorem in graph theory tells us that there is an ordering of the 'send'- and 'receive'-calls that obeys all time-ordering requirements. This ordering can be found by a 'topological sort', and will form a skeleton over which the wanted P-execution PE can easily be constructed.

So, let all 'send'- and 'receive'-calls of IE be vertices of a graph, and below we will describe the edges. We shall say that an edge 'goes forward' if it leads from an earlier to a later call in IE. The edges of the ordering graph will be of four types (see Fig. 9, where the FIFO-edges and the capacity-edges are shown):

Process-edges. If two calls u and v of IE occur in the same process, and u occurred before v, then include the edge (u, v). These edges will ensure that in the final PE the sequencing within each process will be the same in PE as in IE. All process-edges will obviously go forward in IE.

Message-edges. If in IE a certain 'send'-call u delivered a message that was read by a 'receive'-call v, then we include the edge (u, v). These edges will ensure that PE is consistent with respect to inserting a message in a channel before it is fetched. All message-edges will go forward in IE.

FIFO-edges. These edges will ensure that the resulting PE is consistent with the FIFO-structure of the channels. Therefore, between two consecutive 'receive'-calls to the same channel there should be an edge leading forward with respect to IE. Note that, for a given channel, the process-edges will automatically preserve the order of the send calls, as they all come from the same process.

Capacity-edges. This type of edge should ensure that the capacity restrictions are obeyed in the resulting PE, and they will lead from a 'receive'-call to a 'send'-call for the same channel. To define these edges we, for each channel C, number the 'send'-calls in the order they occur in time, and likewise for the 'receive'-calls:

$$C.send_1, C.send_2, ..., C.send_k$$

$$C.receive_1, C.receive_2, ..., C.receive_m$$

Assume that the capacity of C is K. As the capacities are obeyed at the end of the execution, we know:

$$m \leqslant k \leqslant m + K$$

As we have FIFO-channels, we know that the message sent by $C.send_i$ is read by $C.receive_i$, for $i = 1, 2, ..., m$.

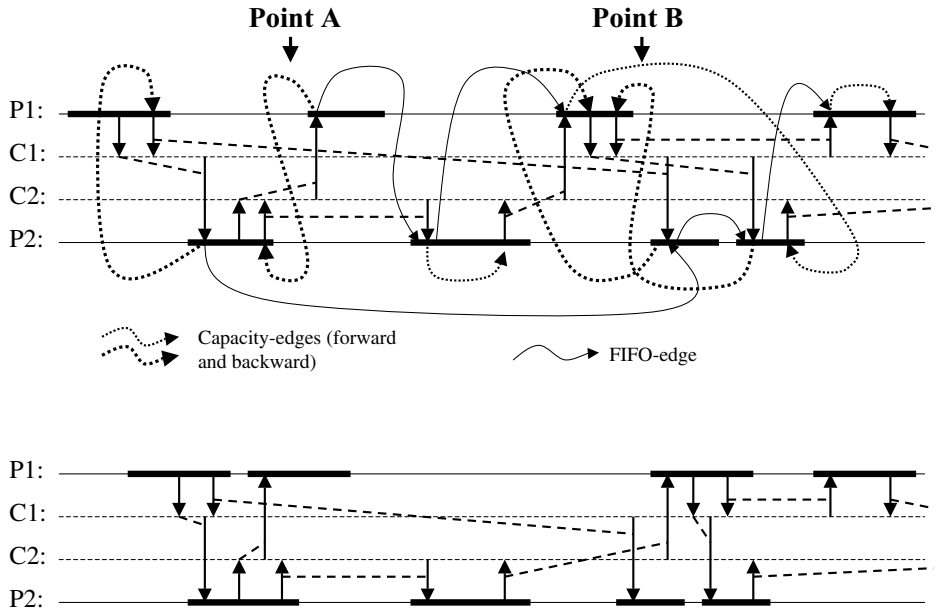

Fig. 9. Illustration of the proof of Proposition 6 for case F1*B: A PE-system with two processes P1 and P2 and with two channels C1 and C2, both with capacity one. The system conforms to case F1*B, where P1 can send to C1 and P2 can send to C2 (and both processes can receive from both channels). The execution on top is a weakly restricted I-execution, where capacity- and FIFO-edges are indicated. Note that it is not fully restricted, as e.g. C2 has two messages at point A and C1 has three messages at point B. The bottom execution is an obs-equivalent fully restricted P-execution, where the 'send'- and 'receive'-calls come in topological sorted order relative to the ordering graph.

The capacity-edges will now be defined as follows: For a 'receive'-call u to a channel C, where $u = C.receive_i$, we make a capacity-edge to $C.send_{i+K}$, whenever $i + K \leqslant k$.

To see that an obs-equivalent P-execution with all the capacity-edges going forward really obeys the capacity constraints is simply a matter of counting. At each point of time and for each channel C no more than K extra 'send'-calls have been issued to C, compared to the number of 'receive'-calls.

Some of the capacity-edges may point backward in IE. However, because IE is weakly obeying the capacity constraint, they will do so only to a limited extent. We can prove the following lemma:

Next-Action Lemma. Assume that (u, v) is a capacity-edge and that A_v in process P is the action that made the 'send'-call v and that A_u is the action containing the 'receive'-call u. Then the action following A_v in P (if any) will come after A_u in IE.

The proof of this is simply a question of counting messages for each channel, and the fact that IE is weakly restricted. So, assume that there is an action A after A_v in P, and that $u = C.receive_i$. Thus $v = C.send_{i+K}$, and before A in P there are at least $i + K$ 'send'-calls to C. Before u there are exactly $i - 1$ 'receive'-calls reading messages from C, and if u happened after the start of A, there would be at least $K + 1$ messages in C when A started. This is contrary to the assumption that IE is weakly restricted, and we therefore know that A must start after u, and therefore after A_u. Thus the lemma is proved.

Now, we are in position to prove that, provided $K > 0$, this graph has no cycles. If $K = 0$, special two-node cycles may form, but the meaning and further discussion of this will be done separately below. So, for the time being we assume $K > 0$.

The key to see that no cycle will form is that almost all edges go forward with respect to IE. If this had been true for all edges, the conclusion would have been obvious. However, the capacity-edges may go backwards in IE, but the crucial point here is that if we along a path have followed a capacity-edge (u, v) backwards in IE, then the next few steps of the path will bring us even further forward in IE. We formulate this as a lemma:

Path Lemma. Assume that (u, v) is a backward capacity-edge in the ordering graph, and that

$$(v, w_1), (w_1, w_2), (w_2, w_3) \ldots$$

is a directed path from v in that graph. Then this path will not contain another capacity-edge until it has been through a node w_i which occurs later than u in IE.

For the proof of this lemma assume that the channel involved in defining the capacity-edge (u, v) is C with capacity K, that $u = C.receive_i$, and that $v = C.send_{i+K}$ is performed by process P. Since v is a 'send'-call, the only edges that can leave v are process-edges and message-edges. If we follow a message-edge (v, w) from v we know, as the channels are of FIFO-type, that $w = C.receive_{i+K}$. Thus, w happens after u.

If we instead follow a process-edge from v, we will either end in another 'send'-call of the same action, or in a 'send'- or 'receive'-call of a later action in P. In the latter case we can use the Next-Action Lemma and conclude that the action we are led into comes after u in IE. In the former case, say that the 'send'-call we are led into is x, and from here we may follow a message-edge. However, as x is in the same action as v, the requirement F1*B says that it must also be to C, and we must have $x = C.send_{i+K+r}$ for some positive r. Thus we know that a message-edge from x will lead to $C.receive_{i+K+r}$ which is also after u. Following another process-edge from x within the same action, and then again a message edge, will obviously bring us even further ahead of u. This completes the proof of the lemma.

Now, armed with this lemma we can reason as follows: If the ordering graph contains a cycle, then this cycle has to have a vertex x which is the latest one occurring in IE. The edge (x, y) leading from x in the cycle has to go backwards with respect to IE, and thus it must be a capacity-edge. However, according to the Path Lemma above, following any path further from y has to bring us to a vertex occurring after x in IE. Thus, the cycle would have a vertex after x, which is contrary to the assumption. This contradiction shows that the ordering graph has no cycle, and thus a legal ordering of the vertices can be found.

Channels with zero capacity

Above we postponed the discussion of the case $K = 0$, but here it comes. This restriction means that the channel cannot store messages at all, so that messages have to be transported directly from the sender-process to the receiver-process in a rendezvous (corresponding to vertical message-edges from process to process, e.g. as in (a1) of Fig. 7). In a weakly restricted I-execution where a channel C has capacity zero, the process outputting to C cannot start another action unless C is empty.

In the ordering graph used in the proof of Proposition 6, a channel C with $K = 0$ will result in a situation where each capacity-edge from a 'receive'-call to C will have a message-edge going between the same nodes, but in the opposite direction. In the complete I-execution IE, the capacity restrictions are obeyed at termination, and when $K = 0$ this means that all messages that are sent to C are also read by 'receive'-calls. Thus, for channels with $K = 0$, for each 'send'-call there is a 'receive'-call reading the message, and each such pair of calls forms a two-node cycles in the graph. Also, by the same arguments as earlier, we can conclude that these are the only cycles in the graph.

Now, such a two-node cycle simply says that in the P-execution we want to construct, the 'send'- and 'receive'-call of the cycle has to happen simultaneously, which dictates a rendezvous in the P-execution. Thus, this case falls nicely into our intuition about what $K = 0$ should mean, and it can thereby be handled as any other case. □

*Proof for case U1*C.* Also for this case the 'forward' direction follows directly from Proposition 5. The proof in the other direction will be similar, but slightly different from, the one for case F1*B. One should all the way remember that it is the *number* of messages that counts for the capacity restrictions, not exactly which messages are inserted into and removed from the channels.

So again, we assume that a complete weakly restricted I-execution IE conforming to case U1*C is given, and our aim is to construct a (fully) restricted obs-equivalent P-execution PE. As before, we construct an ordering graph with all the 'send'- and 'receive'-calls as nodes, and with the process-, message- and capacity-edges defined as above. However, we include no FIFO-edges.

As the channels no longer are of the FIFO-type, we cannot assume that a message received by $C.receive_i$ is sent by $C.send_i$. This has the consequence that the two-node cycles that occurred for case F1*B when the capacity was zero may in case U1*C also occur with non-zero capacities. An example is shown in Fig. 10. Thus, in the proof for case U1*C we have to take these cycles into account all along, not only for the special zero-capacity cases. This will make the proof below slightly different from the proof for case F1*B.

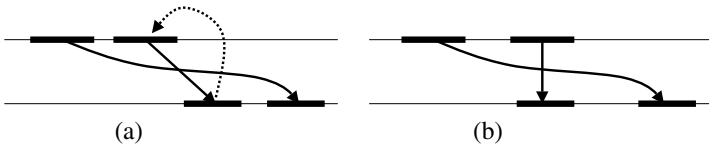

Fig. 10. Here (a) shows an I-execution conforming to case U1*C, where the capacity of the channel is one. The message-edges (fully drawn) and the capacity-edges are shown, and a two-node cycle is formed. (b) Shows a fully restricted obs-equivalent P-execution (where the edge pointing straight down represents a rendezvous in which a message can pass directly from the sender to the receiver even if the channel is full).

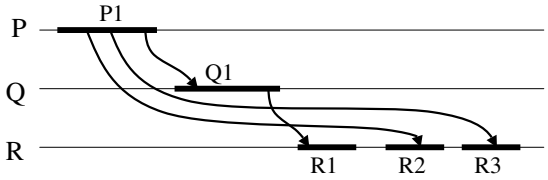

Fig. 11. This example shows that we somehow had to restrict case F1* for Proposition 6. In the situation above there is one channel between each pair of processes, and all channels have capacity one. The given weakly restricted I-execution conforms to case F1* but not to case F1*B, and we can easily see that no fully restricted obs-equivalent P-execution can be found.

However, we can prove the Next-Action Lemma for case U1*C exactly as for case F1*B. The key here is that the proof simply relies on counting of messages.

We can also prove a slightly modified version of the Path Lemma:

Path Lemma. Assume that (u, v) is a backward capacity-edge in the ordering graph. Then, from v a message-edge may lead back to u, but any other edge from v will lead to a node w which occurs later than u in IE.

For the proof of this, assume that A_u and A_v are the actions containing u and v respectively. Since v is the end of a capacity-edge, it is a 'send'-call, and the only edges leaving a 'send'-call are process-edges and message-edges.

Because of restriction U1*C we know that v is the only 'send'-call in A_v. Thus any process-edge will lead into an action following A_v (if any) in that process, and from the Next-Action Lemma we then know that it leads to a node occurring after u in IE. Thus we are done with the process-edge case.

So assume we follow a message-edge (v, w) from v. There is at most one such edge, and below we will show that w cannot occur before u in IE. Thus the message-edge part of the lemma above will be proved.

For the proof, assume that w does occur before u. We know that $u = C.receive_i$ for some channel C, and then $v = C.send_{i+K}$, where K is the capacity of C. Thus, before v (and thereby before A_v) there are $i + K - 1$ 'send'-calls to C, while before u there are $i - 1$ 'receive'-calls to C. As A_v happens before u, and as one of the 'receive'-calls to C before u (that is, w) is used for reading the message from v, we can conclude that at most $i - 2$ of the $i + K - 1$ messages sent to C before A_v is read when A_v starts, and thus at least $K + 1$ are remaining. However, this is contrary to the assumption that IE is weakly restricted at the start of A_v. Thus, the assumption that w occurs before u must be wrong, which completes the proof of the Path Lemma.

From the above we know that the ordering graph may contain cycles of the special two-node type with a capacity-edge and a message-edge, but we should show that there are no other cycles. So assume that there is one. We can then obviously also (by 'short-cutting') find one such cycle which is 'simple', that is, with no node occurring twice (utilising the fact that different two-node cycles of the above type never have nodes in common). This cycle must have a latest node in IE, and the next node in the cycle must obviously be a backward capacity-edge (u, v). As the cycle is simple and not of the special two-node type, the next edge is not (v, u), and from the Path Lemma we can then conclude that it will lead to a node later than u in IE. However, this is contrary to the assumption, and therefore no such cycle exists. □

Thus, Proposition 6 is proved both for case F1*B and for case U1*C.

One may wonder whether we really had to restrict case F1* (to case F1*B) and case U1* (to case U1*C) to be able to prove Proposition 6. The examples in Figs 11 and 12 show that allowing case F1* or case U1* in full will not work.

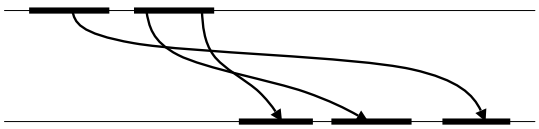

Fig. 12. This example shows that we somehow had to restrict case U1* for Proposition 6. In the situation above all messages are sent on the same channel, which has capacity one. The given weakly restricted I-execution conforms to case U1* but not to case U1*C, and we can easily see that no fully restricted obs-equivalent P-execution can be found.

5.3. Consequences for Verification

The consequences of Proposition 6 for verification of PE-programs are rather straightforward. For PE-programs of type F1*B or type U1*C we simply know that an SI-predicate is valid for all fully restricted complete P-executions if and only if it is valid for all weakly restricted complete I-executions. Thus one can concentrate on the latter types of executions when studying properties that are expressible as SI-predicates.

Proposition 5 tells us that for PE-programs of type F1* or type U1* all SI-predicates that are valid for all weakly restricted complete I-executions are also valid for all fully restricted complete P-executions, but not necessarily the opposite. Thus one can try the latter method first, as it may allow larger actions, and thereby perhaps give simpler proofs. If this does not work, one can use Proposition 6 (if necessary with smaller actions) which in principle allows us to settle the case for all P-executions by considering only I-executions.

6. Deadlock and Finite Channel Capacities

As discussed earlier, deadlock for the case with finite channel capacities is slightly more involved than for the case with infinite capacities. This is because also 'send'-calls may be blocked, if the channel is full. However, it turns out that we can prove a natural combined extension of Propositions 2 and 6, thus obtaining an equivalence statement for case F1*B and case U1*C.

For this proposition we should define what it means for a PE-program to be executed in receive-safe I-mode with weak capacity restrictions. This is straightforward: We execute actions from the different processes strictly one after the other. Whenever an action is finished we are only allowed to proceed with an action A of process P if (a) the content of the channels will allow the receive-calls of A to be satisfied and (b) the number of messages in all channels to which P may send messages are within their capacity limits.

Thus, we will say that such an execution has deadlocked if it stops 'prematurely', or if the capacity-restrictions are not fulfilled when all processes have terminated. More exactly, this means that the system has stopped in one of the following two situations: (1) There are processes that have not terminated, but none of these are allowed to start another action, either because there are not enough messages to make it receive-safe or because some output channel of that process has too many messages (or both). (2) All processes have terminated, but there are channels with more messages than the capacity allows.

We can now formulate the announced proposition:

Proposition 7. A PE-program of type F1*B or type U1*C will not deadlock when executed in (fully restricted) P-mode if and only if it will not deadlock when executed in receive-safe I-mode with weak capacity restrictions.

Proof. We do the proof by showing that there is a deadlocking (fully restricted) P-execution if and only if there is a deadlocking receive-safe weakly restricted I-execution.

We first assume that a deadlocked P-execution PE for the PE-program is given, with PE fully obeying the capacity restrictions. Thus, at least one of the sequential processes of PE ends in either a 'receive'-call to an empty channel or in a 'send'-call to a full channel. The actions containing such blocked calls will be referred to as *receive-blocked* and *send-blocked* actions respectively. Those processes not ending in such blocked actions will have terminated normally.

We first observe that the channels involved in the blocked 'receive'-calls are obviously disjoint from those involved in the blocked 'send'-calls. Otherwise some processes would have been able to proceed.

We start the construction of our I-execution by adjusting PE to a complete execution PE' as follows: We remove from PE all receive-blocked actions and we run to completion all send-blocked actions (disregarding

the capacity restrictions). As we are in case F1*B or case U1*C we know that each completion of such an action will send messages to separate channels, and no messages will be sent to channels where a 'receive'-call was blocked. As each action has at most one 'receive'-call, the receive-blocked actions can be removed without disturbing the contents of any channel.

From PE' we then construct an I-execution IE', by ordering the actions according to their O-times as in the proof of Proposition 5. By the same arguments as in that proof, we find that IE' will weakly obey the capacity restrictions all along, except that the completing 'send'-calls of the send-blocked actions will make the corresponding channels exceed their capacity at the end of IE'. Note that adding or removing these final 'send'-calls in one process will not interfere with any other process, neither in PE' nor in IE'. Thus, we can conclude as follows:

All channels involved in a blocking 'receive'-call in PE are empty also after IE'. Thus the corresponding actions cannot be started after IE' in a receive-safe I-execution. All channels involved in a blocking 'send'-call in PE contain at least one message more than their capacity, as the 'send'-call that was blocked was allowed to output its message in IE'. Thus, none of the corresponding processes are allowed to proceed with another action (if any), as the execution should be weakly restricted.

Thus, if PE had receive-blocked actions, or it had send-blocked actions where the corresponding process has *more* actions to do after the completion of the send-blocked one, then there are actions that, with respect to the local process, may be started after IE'. However, none of these are allowed to start in a receive-safe weakly restricted I-execution, and thereby the system has deadlocked by point (1) in the definition above. In the opposite case (that is, there are no receive-blocked actions and each completion of a send-blocked action also terminates the corresponding process), then a deadlock has occurred by point (2) in the definition, as some of the capacities are exceeded. Thus the proposition is proved in one direction.

For the proof in the other direction, assume that IE is a receive-safe weakly restricted I-execution that has deadlocked. This means that all processes of IE ended with completed actions, and that we have one of the following two situations: (1) There are processes that have more actions to do, but these either cannot have their 'receive'-call satisfied or they have too many messages in some of their output channels (or both). (2) All processes have terminated normally, but there are channels with too many messages.

In situation (1) the channels having too many messages are obviously disjoint from those involved in processes that cannot proceed because of the receive-safety. Remember here that each action makes at most one 'receive'-call.

To obtain a P-execution PE that deadlocks, we first produce a modified I-execution IE'. For this we 'truncate' the last action of those processes that ended up with an overflowed output channel, in the following way: If the (only) output-channel used by this action ended up with an overflow of n messages, then we remove the last n 'send'-calls done by that action, and the local operations in between. As this channel was within its capacity when the action started, at least that many 'send'-calls were made in this action. At least one 'send'-call will be removed in each truncated action.

The resulting truncated execution is called IE', and to show that IE' is consistent we need to know that removing the 'send'-calls during the truncation will not disturb the rest of IE. The key here is to show that the messages that was output by the removed 'send'-calls was not read by any 'receive'-call in IE (which are the same as the 'receive'-calls in IE').

For case F1*B this is rather obvious. We know that the messages sent by the removed 'send'-calls were the last ones sent to the (only) concerned channel, and therefore (because the channels are of FIFO-type) also among those remaining in that channel after IE.

For case U1*C we can reason as follows: As each action has at most one 'send'-call, and as no output-channels of a process are overflowed when an action of that process starts, a channel can at most be overflowed by one message at any time during IE. Thus, channels that are overflowed at the end of IE are so by exactly one message, and this must be caused by the only send-call in the last action of the corresponding process. Thus, between the end of this action and the end of IE no 'receive'-call can have been issued to that channel (or the overflow would have disappeared). Therefore, removing this 'send'-call will not disturb the rest of IE.

Now, truncated actions of type F1*B or U1*C still conform to their respective cases. We can therefore consider IE' as a complete execution in itself (although not produceable as a *complete* execution by the same PE-program). This means that Proposition 6 can be applied, and we can turn IE' into a fully restricted obs-equivalent P-execution PE', which we shall use as first part of PE.

To obtain a deadlocking P-execution we then, after the end of PE', start in parallel (a) all actions that could not be started after IE because their 'receive'-call could not be satisfied, and (b) all the 'rest-parts'

of the truncated actions. From the way this is constructed, we know that the following is true after PE': All channels involved in 'receive'-calls from (a)-actions are empty and all channels involved in 'send'-calls in (b)-rest-parts are full (and these two sets of channels are disjoint). Thus the (a)-actions will be blocked in their first 'receive'-call and the (b)-rest-parts will be blocked in their first 'send'-call.

Thus, the PE-program has deadlocked in P-mode. Note that this also covers the case where *all* processes of IE had terminated, but where some capacities are exceeded in the final situation. In this case there had to be truncated actions. □

6.1. Consequences for Verification

The consequences of Proposition 7 for verification purposes is again rather obvious. To study whether a PE-program of type F1*B or U1*C may deadlock we can study whether it may deadlock when executed in receive-safe I-mode with weak capacity restrictions. One should note that in receive-safe I-mode it is enough to keep track of the conditions for deadlock between actions. This, for example, allows a suitable invariant to be used for showing absence of deadlock.

7. Future Work

The work described above answers some questions, but it also opens up for further research in different directions. One direction is to study how the results above may affect other efforts towards verification of parallel and distributed programs. As indicated in the introduction, it might influence both testing techniques and techniques using more formal methods.

Another direction is to try to extend the theory further. Relevant topics here are: What can be said if we have PE-systems where a mixture of the cases F11, F*1, F1*, and U** occurs? Is it possible to say something of interest if traditional global variables are involved?

And finally, as pointed out by a referee, it would be of great interest if the theory for finite capacity channels also could take care of messages with different sizes. For this generalisation the definitions would have to be adjusted in obvious ways, but if this is done we conjecture that the propositions of Sections 5 and 6 will remain true also in this more general setting.

We think all these directions are worth pursuing.

Acknowledgements

We would like to thank Ole-Johan Dahl for reading an early version of this paper and for valuable comments. In addition we want to thank the referees for helpful feedback and suggestions.

References

[ChL85] Chandy, K. and Lamport, L.: Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[GaT90] Van Gasteren, A. J. M. and Tel, G.: Comments on 'On the proof of a distributed algorithm': always-true is not invariant. *Information Processing Letters*, 35:277–279, 1990.

[GaW96] Garg, V. K. and Waldecker, B.: Detection of strong unstable predicates in distributed systems. *IEEE Transactions on Parallel and Distributed Programs*, 7(12):1323–1333, 1996.

[KrL97] Krogdahl, S. and Lysne, O.: Verifying a distributed list system: A case history. *Formal Aspects of Computing*, 9:98–118, 1997.

[Kro78] Krogdahl, S.: Verification of a class of link-level protocols. *BIT*, 18:436–448, 1978.

[Lam80] Lamport, L.: The 'Hoare logic' of concurrent programs. *Acta Informatica*, 14:21–37, 1980.

[Lee99] Lee, E.: Local detection of exclusive global predicates. In *Proceedings of the 1999 International Conference of Parallel Processing*, pages 336–343. IEEE Computer Society, 1999.

[LyK97] Lysne, O. and Krogdahl, S.: On communication protocols, invariants and rewriting. In *Selected Papers from the 8th Nordic Workshop on Programming Theory, Research Report 248*, pages 157–165. Department of Informatics, University of Oslo, 1997.

[MaI92] Manabe, Y. and Imase, M.: Global conditions in debugging distributed programs. *IEEE Transactions on Parallel and Distributed Programs*, 15:62–69, 1992.

[MaN91] Marzullo, K. and Neiger, G.: Detection of global state predicates. In *Proceedings of the 5th International Workshop on Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1991.
[Owe92] Owe, O.: Axiomatic treatment of processes with shared variables revisited. *Formal Aspects of Computing*, 4:323–340, 1992.
[StS95] Stoller, S. D. and Schneider, F. B.: Verifying programs that use causally-ordered message-passing. *Science of Computer Programming*, 24:105–128, 1995.
[VeD95] Venkatesan, S. and Dathan, B.: Testing and debugging distributed programs using global predicates. *IEEE Transactions on Software Engineering*, 21(2):163–177, 1995.
[Ver87] Verjus, J. P.: On the proof of a distributed algorithm. *Information Processing Letters*, 25:145–147, 1987.

Received December 2000
Accepted in revised form September 2001 by U H M Martin