The design of parallel systems requires care and accuracy if the results obtained from the system are to be useful. This requirement means that an accurate model of the system to be implemented must be derived and this model carefully mapped onto the final hardware architecture. Many systems have concurrency inherent in their operation and this concurrency should be used to the full in any implementation of the final system. However, concurrency introduces a number of complications which could cause errors in the final system to occur. A careful choice must therefore be made to describe and implement such requirements.

The design of software intended for a parallel implementation is demanding, requiring the understanding and application of proper design methods including techniques which exploit and control the parallel nature of the system. The software will comprise a set of processes in asynchronous concurrent execution where coordination is provided by synchronising interprocess communications. The designer is therefore presented with three interrelated problems:

a) partitioning the software into appropriate discrete processes.
b) the design of the interprocess communication structure.
c) the internal design of each individual process.

Established, 'stepwise refinement', design methods for conventional, sequential software can be applied effectively in (c). However, (a) and (b) are outside the scope of software engineering methods for sequential software. Partitioning software into a set of discrete processes requires care since an inappropriate partitioning strategy can lead to a complicated interprocess communication structure, in which it is difficult to avoid dynamic faults, such as deadlock, when processes interact.

One of the major problems in current parallel designs is how to convert sequential code (currently in place) into code that will execute efficiently on a network of parallel processing engines. It can be argued that the 'best' design strategy is to start from the inherent concurrency of the problem in hand and 'design-in-parallel' from the start. However, many applications require the transformation from sequential-to-parallel.

An additional problem when implementing systems on parallel architectures is what language should be used to specify the systems logical structure, and what operating system should be used.

The papers in this session of the workshop look at these problems in a number of different ways.

In the first paper, by Diekmann, Menzel and Stangenberg (Germany) the authors investigate the efficiency of implementing distributed algorithms from the point of view of the concurrent language (in this case occam) and from the operating system point of view. The paper presents a detailed overview of the most efficient programming techniques for implementing distributed algorithms. As well as looking at the concurrent language, a number of operating systems used for parallel systems are compared (Helios, ParC, Inmos Toolset, and Parix). The paper concentrates on implementations on transputer networks and shows how to utilise the transputer's features. Two methods of implementing task frames are compared; The first using semaphores from an operating system point of view, the second by using process synchronisation by internal channel operations. A comparative study of development platforms is made, using Dhrystone and LINPACK benchmarks.

In the second paper of the session, by Ahmed, Barriga, and Ayani (Sweden) the authors describe a way of implementing parallel discrete event simulations using space-time events. The paper states that one of the major problems in discrete event simulations is the sequential mode of execution that must be imposed on the events belonging to a logical process. The thesis of the paper is that these constraints can be relaxed by associating a scope with each event message. The logical process can then determine whether two events are independent by comparing their scopes. It is shown that by associating additional attributes to event messages the logical process
is provided with sufficient information to decide if two events can be processed out of timestamp order (i.e., are independent). Experimental results are obtained by using a parallel simulation environment called SIMA. The technique of space-time events is applied to the Conservative Time Windows scheme.

The final paper in this session, by Kalanteri, Winter, and Wilson (UK), presents a method of executing in parallel, sequential programs. The authors propose a novel framework for deterministic parallel execution of sequential programs. The method is based on the application of order-preserving protocols, originally developed for parallel discrete event simulation systems. They propose that automatic parallelisation of sequential code, despite the absence, or with insufficient, data dependency information, can be achieved. The scheme automatically executes sequential programs on MIMD systems such that the logical order and unity of operations on the data is maintained. Thus, the authors argue that the machine will provide a deterministic parallel execution environment for sequential programs. It is shown that the serial ordering parameters can be extracted from a sequential program, and transferred to the parallelised version of the program providing a deterministic parallel execution.