ANDREWS83
Andrews, G.R., Schneider, F.B.; Concepts and Notations for Concurrent Programming; ACM Computing Surveys 15, 1 (March 1983) pp. 3-43.

Technical advances have made it possible to build distributed systems that were in the past impractical, and this in turn has brought concurrent programming to all application areas. Theoretical advances in programming language design have simultaneously indicated the desirability of notations that can express concurrency simply and make requirements for synchronization explicit, while facilitating formal proofs of correctness.

This paper summarizes the main concepts in concurrent programming and classifies some of the language notations used for expressing concurrency. Examples of languages which use these notations are given throughout.

First a brief summary of concurrent programming concepts is given. The concepts of process and concurrent program are defined and various methods of executing concurrent processes are given, including multiprogramming, multiprocessing, and distributed processing. Communication and synchronization among concurrent processes are identified as being necessary for cooperation among the processes. Thus the three main decisions in designing a notation are how to indicate 1. concurrent execution, 2. interprocess communication, and 3. interprocess synchronization. Various methods for each of these are then discussed in detail.

One method of specifying concurrent execution is the use of coroutines, where each coroutine is seen as implementing a process. The semantics of coroutines allow execution of only one coroutine at a time. While this is sufficient where a processor is shared, it cannot express true parallel processing. Fork and join statements can express this, since the invoked and the invoker proceed in parallel. This mechanism is described as often difficult to understand, but very powerful. The cobegin statement is less confusing, and although not as powerful as fork and join, it is sufficient for most situations. The final strategy given for expressing concurrent execution is called process declaration. Its advantage is that it states explicitly which routines will be executed concurrently. Some implementations of this scheme have only one instance of each process, while others allow the programmer to create several.

Methods for communication and synchronization are divided into two groups - those based on shared variables, and those based on message passing.

When shared variables are used for communication, two kinds of synchronization are deemed useful - mutual exclusion which treats a group of statements as an indivisible operation, and condition synchronization which can delay a process until some condition becomes true. Several methods are described for implementing these two kinds of synchronization. Busy-waiting can be used successfully for both, but is seen as difficult to use, and wasteful of processor cycles. Semaphores were developed as a better alternative, but are unstructured and thus difficult to use without error. They also use the same notation for both mutual exclusion and condition synchronization. Semaphores were extended by introducing constructs which require that their use be structured. The resulting notations were conditional critical regions, monitors and path expressions.

Message passing is the second primitive for communication and synchronization between processes. It can be thought of as another extension of semaphores which allows them to not only synchronize, but to contain data. One important issue is how the channels of communication are to be specified. Communication can be made one to one, many to one, or many to many by using direct, port and global naming respectively. Also, the source and destination of a message can be either fixed at compile time (static channel naming) or computed at run time (dynamic channel naming). The other main issue of concern with message passing is how synchronization will be achieved. This is affected by the choice of either blocking or non-blocking sends and receives. Four languages which base concurrency on message passing are described - Concurrent Sequential Processes (CSP), Programming Language In The Sky (PLITS), Ada, and Synchronizing Resources (SR).

Finally, the large variety of programming languages that use the mechanisms described above for concurrency are grouped into three classes: procedure oriented languages, message oriented languages, and operation oriented languages. All three categories are considered by the authors to be equivalent in expressive power, but each suited to a different architecture, and emphasizing a different style of programming.

The authors conclude that while progress has been made in finding solutions to concurrency problems and developing notations for these solutions, there is still much work to be done. In the area of programming languages, efforts must be made to use and evaluate the utility of languages which adopt the various notations discussed, and formal techniques for constructing correct programs must be developed.