```
X: = 1
X: = 0; X: = X + 1;
```

The first fragment sets X directly to 1; the second sets it to zero first and then increments X by 1. Both should lead to X being 1, but only as long as there is no interference between them. If they execute concurrently and interfere, the following may happen: Thread 2 executes and sets X to 0; then it is interrupted and thread 1 executes setting X to 1; then thread 2 continues and increments X by 1, so that it becomes 2! Due to concurrency, the memory that is holding variable X is serving two masters, which can lead to undesired effects.

To define the meaning of concurrent programs and to understand their behavior, researchers invented calculi for concurrent systems in the late 1980s. Two well-established approaches are Milner's CCS (Calculus for Communicating Systems) and **C. Antony Hoare**'s CSP (Communicating Sequential Processes).

The idea of CCS is to capture the behavior and structure of concurrent systems. The notion of an observer is the key: Two systems are considered equivalent if an observer cannot distinguish them. To formalize this idea, CCS contains the notion of bisimulation: Two systems, A and B, are equivalent if whenever A is able to carry out an action a and then becomes A, B can also carry out a and becomes B, and A and B are equivalent. In short, A and B can "simulate" each other—hence the name *bisimulation*. Another important property underlying CCS is the idea of a congruence (i.e., structure-preserving equivalence): If two systems A and B are considered equivalent and A is part of C, we can safely replace A in C by B without changing the observable behavior of C.

Bisimulation and congruence define the notion of equivalence of systems, but how does CCS capture the essentials of concurrent systems? CCS comprises two types of actions: sending a message and receiving a message. Furthermore, it comprises five basic operations: *sequential composition* allows one to define that statements execute one after another, *parallel composition* allows one to define that two processes execute concurrently, *nondeterministic choice* allows one to define that the process has two alternatives to execute, *renaming* allows one to rename actions, and *restriction* allows one to forbid the occurrence of actions.

The main motivation of CCS is to capture the structure and behavior of concurrent systems in order to reason about them. One is interested in verifying the and safety properties of programs. A safety property states that some property holds eventually in all states the system may get to. A liveness property states that the system can reach a state in which some property holds. Checking such properties enables system designers to verify that a system guarantees some properties, which is essential in safety-critical applications.

FURTHER READING

Hoare, C. A. R. *Communicating Sequential Processes.* Upper Saddle River, N.J.: Prentice Hall, 1987.

Milner, Robin. *Communication and Concurrency.* Upper Saddle River, N.J.: Prentice Hall, 1995.

—*Michael Schroder*

Conference on Data Systems and Languages

The Conference on Data Systems and Languages (CODASYL) was an organization founded in the late 1950s by the U.S. Department of Defense. Its purpose was to propose and develop programming languages for computers. It later evolved into several committees of volunteers and eventually disappeared in the mid-1990s. Largely remembered today for its definition of **COBOL** (Common Business Oriented Language), CODASYL was also involved in other activities, most notably in the database area. The term CODASYL is sometimes used as a synonym for a specific database model.

CODASYL's main contribution to the field of programming languages was COBOL, a highly successful language for business processing. Some programmers had been developing interpreters or compilers for programming languages in the early 1950s, when an industrywide team led by Joe Wegstein initiated the effort that would lead to COBOL. **Grace Hopper Murray** (1906–92) was one of the contributors to the definition of the language. She had developed Flowmatic, a language that used

natural language-like phrases to describe computer operations. COBOL itself is rather verbose and uses natural language-like constructs.

With the advent of large data repositories, database programs became necessary. COBOL was useful for this purpose, because the definition of the data is contained in a section separated from the routines that access the data. CODASYL proposed an extension for COBOL to deal with databases. The Data Description Language (DDL) and the Data Manipulation Language (DML) formed the basis of the CODASYL database model. The DDL is used for specifying the structure and integrity conditions on the database schema (e.g., the type of data contained in each field, i.e., numeric, alphanumeric, etc.) The DML is used for creating (inserting), updating, and deleting data. A query language is used for retrieving a subset of the database that satisfies user-specified search conditions.

The basic CODASYL concept was a "network" view of the database. Records of data could be linked in one-to-many relationships: For example, a department record that points to many linked records of employees. This makes the database very useful, since many different kinds of relations can be expressed. It is sometimes difficult to retrieve the information when the current links do not map well to the query, but there is always a way of telling the computer how to retrieve the information needed. Therefore, the CODASYL model is a procedural database model rather than a declarative one like SQL (**Structured Query Language**), which is based in the relational model. In a procedural database model, the programmer tells the computer explicitly how to retrieve the information needed, that is, which links it has to follow.

—Raúl Rojas

Connection Machine

The Connection Machine is a high-performance computer developed in the 1980s at the Massachusetts Institute of Technology (MIT) that incorporates a very large number of simple processors. It is one of the most radical departures from the classic **von Neumann architecture** in computing history.

Computers following the von Neumann architecture have one central processor, connected to a separate memory. Although this concept offers important advantages that have made it the predominant construction paradigm, it has several shortcomings as well. The processor, which is always busy, makes up just a small portion of the available transistors, while the rest, the memory, is accessed one memory cell at a time: This means that a large percentage of the hardware is always idle. Also, when enlarging a von Neumann computer, one enlarges more or less only the memory, making the inefficiency even worse. Computing time becomes dominated by memory accesses.

In the 1980s, **Daniel Hillis** (1958–), a Ph.D. candidate at MIT, started studying how to avoid the inefficiency created by the processor memory split. Hillis believed this *von Neumann bottleneck*, as it is often called, to be the reason for the inferiority of machines to human brains on problems like image recognition and knowledge retrieval.

The Connection Machine was his answer. Hillis designed a computer that should consist of a great number (at least in the range of tens of thousands) of simple independent processing cells, each with its own memory and connected by a very general network. The sheer number of cells should make it possible to achieve very high parallelism—assigning, for example, one processor to each pixel in an image-processing application or one processor to a transistor in a **very large scale integration** (VLSI) simulation. Hillis conceived his machine initially as a vehicle for **artificial intelligence** research.

The first prototype was called CM-1. It contained 65,536 processors, each with 4096 bits of memory. The processors, a custom-made type, 16 to a chip, were extremely simple. Their basic operation was to take 2 bits from memory and a flag, combine them as requested, and output one bit to memory and a flag. The choice of operation was as general as possible, allowing any of the 256 possible Boolean functions with three variables to be applied to the input. Rather than guessing which operations would be important, Hillis decided to offer them all. On their chips, the processors were connected to a router, which controlled the access to the network. Additionally, the processors were