ICAS-90-2.4.3

A Systems Approach to Avionic Multiprocessing Architectures

J. Dennis Seals
AT&T Bell Laboratories
Whippany, N.J. 07981, USA

Abstract: Commercial, military, and aerospace airframes are now being developed that require avionics systems with processing capabilities that rival today's largest supercomputers. To meet these requirements, avionics designers will have to address issues such as software support, multiprocessor control, and open machine architectures as an integral part of the processing system design. This paper describes a heterogeneous multiprocessing architecture (support software, operating system, and hardware) that will meet these real-time processing requirements and significantly reduce software development and support costs. The application support software augments the Ada toolset with a powerful graph language that functions as a program design language and, in many cases, is machine-translatable into Ada code. A hybrid control mechanism handles the complexities of multiprocessing control while providing a transparent interface between the application user and physical machine. The machine architecture is based on a modular building block concept and asynchronous communication network that permits processors with different functions, clock speeds, and data bandwidths to be integrated into a common system without major protocol problems or data bottlenecks.

The Problem

Avionics systems are emerging that require cooperative, real-time, multiprocessing throughputs that will rival today's supercomputers. In addition, the systems will have to meet demanding environmental and packaging requirements, and provide fault-tolerant computing. As challenging as the requirements are, they pale in comparison with the software problem. Estimates for embedded software range from several hundred thousand lines to over a million lines of high-level code per system, and the life cycle support of this code promises to run into tens of billions of U.S. dollars. To meet these processing requirements and stem the rising costs of software, new approaches to language and application capture, software support, operating systems and machine architectures have to be developed. Unlike conventional systems that approach these components more-or-less independently, new systems must design and integrate them as an integral part of the overall processing system design.

Software costs are rapidly reaching crisis proportions, with estimates now ranging from 50% to 80% of a processing system's total life-cycle cost. Estimates of the embedded software needed to support programs such as the U.S.'s advanced tactical fighter, lightweight helicopter, and national aerospace plane range from 0.5 to over 2 million lines of high order code. Even new commercial airplane avionics will require several hundred thousand lines of code to support new structure and diagnostic systems and intelligent pilot aids. Current estimates of the development cost of a line of code range from $30 to $100 US, depending on the mission criticality of the code. But design and development is only part of the overall cost of software. Life-cycle support cost can often exceed development costs many times over. To grasp the size of the problem, one has only to consider how long its would take to scan 1 million lines of code. Assuming six seconds per line and eight-hour days, it would take over six months just to scan this software. Now imagine how long it would take to find an error or make an update! Simple arithmetic shows that life cycle cost of each of these future software programs will range from several hundred million to several billion US dollars each unless new approaches can be found.

As avionics systems increase in complexity, so must the operating systems that control them. Most large commercial processing systems use complex operating systems that utilize from 40% to 60% of the available computational resources. These systems typically control only a few processing units and are optimized for a particular type of processing, such as real-time control or distributed processing. While the next generation of avionics operating systems will not have to deal with the intricacies of virtual memory and disk-based control, they will in many ways be far more complex. They will have to dynamically assign tasks, share resources (and non-system assets), meet millisecond latency requirements, support highly complex fault management, and provide data security. In addition, these systems will need to concurrently control a wide variety of cooperating processors without incurring excessive processing resource overhead.

Finally, future avionic applications will call for machine architectures to support a diverse mix of signal, data, control, and decision processing. This necessitates an open architecture and high-speed communication network in which processors with different functions, clock speeds, and data bandwidths can communicate and exchange data without creating major bottlenecks. As systems become highly integrated, the machine architecture must provide the necessary hardware and communication links to support the fault detection and recovery needed for critical flight systems.

AT&T Bell Laboratories has developed and is now testing an avionics multiprocessing architecture that promises to greatly mitigate these problems. This architecture, called the Advanced Avionics Core Processing (AACP) architecture, embodies several innovations in software language, application development, operating system and hardware design. It represents an *integrated design* approach since support software, operating system, and hardware system were designed specifically to work together.

Solving the Software Problem

The AT&T architecture embodies two advances that can significantly reduce software life-cycle costs: functional programming and decoupled control. Functional programming permits applications to be captured as data-flow graphs - a type of structured flow chart. While the architecture supports conventional programming approaches (e.g., Ada programs), functional programming can provide a more than two-fold reduction in life-cycle costs. Decoupling the application and control software simplifies system integration by removing complex control issues, such as cooperative control, fault/overload management, and mode changes from the application programs and placing them into independent high-level command programs.

The AT&T architecture is based on a three-level approach to embedded software consisting of application programs, a run-time executive, and local operating systems (See Figure 1). The application program defines the sequence of operations needed to solve a user's application problem. It may be captured either as Ada programs or a graph language. The run-time executive software performs the high level scheduling and resource allocation functions for the application programs. Its major function is to hide the complexities of the machine hardware and local operating system from the application user. This approach not only simplifies application programming but allows growth and technology insertion to occur with little or no impact on existing software. The local operating system software provides lower level services that support the executive. This software is tailored to a specific processor class, permitting different processor classes to be included in the architecture.

Programming Language & Style - AACP application software has three basic components: application procedures, I/O procedures, and command programs. Each application procedure defines a sequence of functions or transformations to be applied to a particular data stream. Application procedures may be captured either as Ada programs or data flow graphs. The data flow graph is a functional programming approach in which application programs are defined as directed graphs and are quite similar to conventional flow charts. The data flow graph provides a very compact and natural program design language (PDL). For many applications such as signal processing, these graphs can be machine compiled into an intermediate code (Ada, SPGN) without programmer intervention. Government-sponsored evaluations of this programming methodology has shown a reduction of more than 80% in development time and testing, and project a halving of the life-cycle costs.

A data flow graph (See Figure 2) comprises three basic components: nodes, queues, and graph entities. A node defines the basic function or application process that is being applied to the data inputs. Queues provide the primary data storage and transfer mechanism between nodes, and can be conceptualized as an elastic FIFO buffer. Graph entities are data structures that specify node and graph parameters, such as the number of filter taps, or current entries in a threat file. The data flow graph also provides a natural partitioning of the application for parallel processing (when coupled with the appropriate control mechanism). This approach enforces a high level of code reuse and provides hierarchical viewing of an application process.

Command programs control the actions and interactions of application programs and their external interfaces. For example,

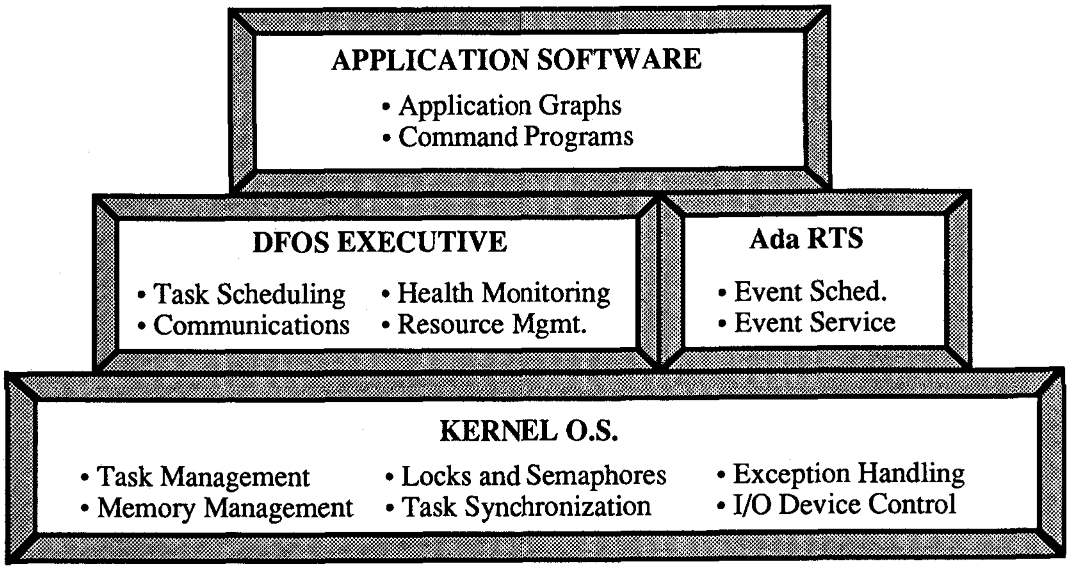

FIGURE 1 - Embedded Software Heirarchy

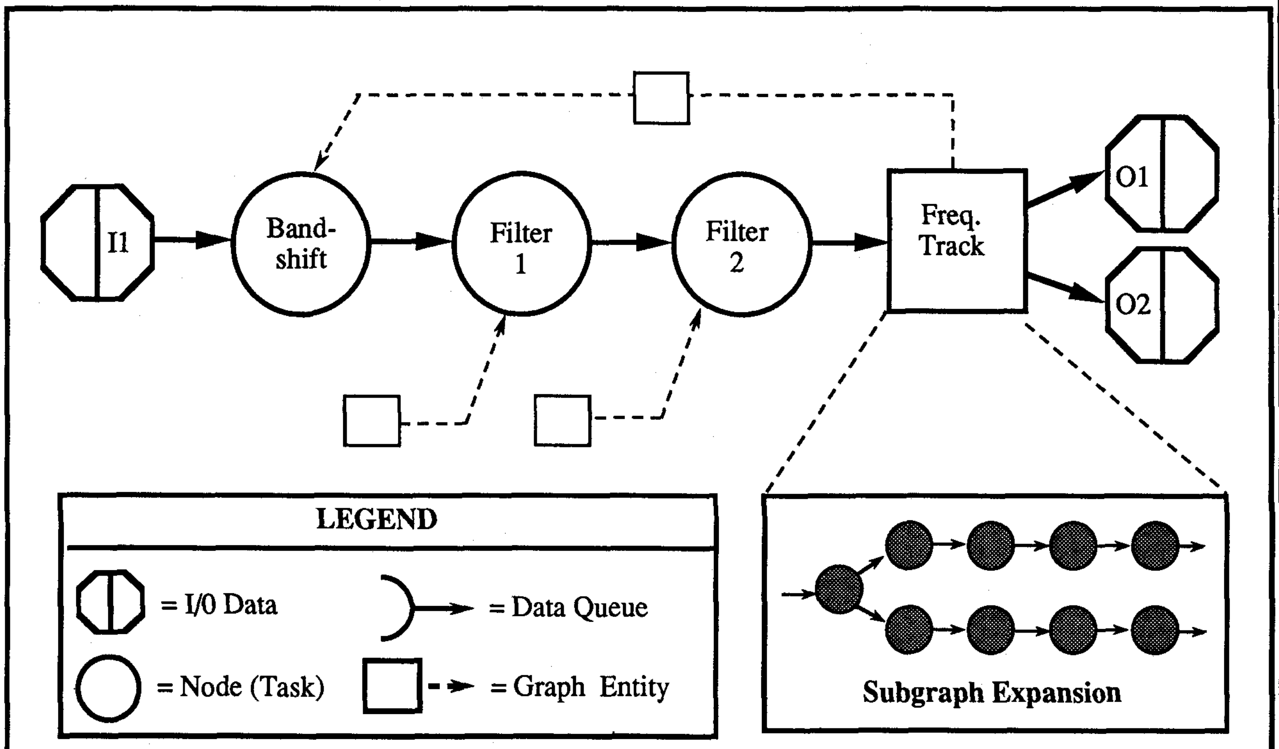

FIGURE 2 - A Simple Data Flow Graph

a command program can start and stop application programs, and control mode changes, sensor steering, revisits, overload management, and diagnostic testing. Command programs are optional and may not be needed for some single sensor or display applications. However, command programs can greatly simplify the development of highly integrated systems by freeing the subsystem application developers from the intricacies of global control. Similarly, the developers of the command program are concerned only with the global aspects of control and not the intricacies of subsystem application processing.

I/O programs are valves between the external world and the application programs. They are dependent on actual hardware implementation, highly specialized in nature, and are typically bound to specific processors.

Application Programming Support - Future software support environments must reduce application code development/test efforts and simplify the software needed for parallel and cooperative processing. The application support environment provides a rich tool set for program development, performance simulation, and system test and debug. It supports a highly structured methodology for software development that naturally partitions an application program into smaller units with well-defined interfaces. This methodology permits the application programming to be allocated to groups with specialized talents and well-defined responsibilities. It promotes a high degree of code reuse and eliminates many errors that occur in the interface code.

The application development tools support application capture either as Ada programs or data flow graphs. Therefore, a graph provides a natural paradigm for the program design language. The application tool set provides a language-directed, interactive, graphical editor that enables the application engineer to capture, modify, and annotate data flow application graphs. Application capture is a top-down, hierarchical development in which the first step is to create the system or big picture graph. At this level, the graph shows the various application subgraphs as undefined black boxes. This graph level defines the flow of data and interfaces between the various processing subsystems. Once this graph is captured, the subgraphs are assigned to and defined by sensor and subsystem processing specialists. Definition of each sensor/subsystem subgraph can be carried out concurrently and independently.

Graphs are created on a high-resolution, bit-mapped terminal. The principal mode of input is via a three-button mouse that permits the user to choose graph objects, attributes, viewing levels, and edit functions from pop-up menus. The keyboard is needed only for naming objects and graph files, or for filling in templates. The graph capture process is fast, allowing graphs of several thousand nodes to be captured in a few weeks or less. The editor also provides some run-time syntax checking, catching many errors during the edit process. The output of the graph editor is either converted manually into Ada code, or depending on the application, machine-translated into Ada code. The translation process consists of creating the public (interface) part of an Ada package (for each node) around the reusable private part stored in the graph library. Aside from the graphical support tools, the application support toolset provides a complete set of library management tools, compilers, linkers, global optimizers, source debuggers, and system build tools.

A powerful set of simulation tools has been supplied to aid both the application programming and the processor design teams. This tool set has three basic simulators: the system simulator, the graph simulator, and the node simulator. The system simulator is a tool for evaluating the performance of the architecture as a whole. It provides information on processor loading and usage, memory usage, latencies, bus and network loading, etc. It enables the application software team to determine if their programs are correctly partitioned and, meet processing latency and control requirements. This simulator also allows the processor (machine) design team to determine the proper set of processing elements/groups and interconnection topology. A special compiler converts valid data flow graphs into input for this simulator.

A graph simulator is provided for interactive debugging and validation of the application graphs. This simulator is target independent and allows the user to monitor and control execution by using breakpoints and data manipulation. It is supported by a large library of reusable Ada primitives and provides convenient tools for the addition of new primitives.

The node simulator simulates the execution of nodes and graph segments in exactly the way they would execute on a given processing element. Since it can be prohibitive in time and cost to execute a complete graph, this tools enables the application user to debug and validate individual nodes and node chains.

The test support tools aid in the test and integration of application software on the target multiprocessor. They allow the user to monitor program execution, set test and breakpoint conditions, monitor bus and network messages, and otherwise test applications on the target machine. This software, coupled with the test monitor hardware, allows the user to monitor each processing element and their buses in a non-interfering, real-time mode. It is also possible for the user to introduce data or control messages on the system buses via the test monitor.

Building a Better Operating System

The AACP operating system consists of the Data Flow Operating System (DFOS) executive and the Kernel Operating System (KOS). The DFOS executive performs high-level scheduling, communications, and resource allocation functions. Its world consists of the dynamic flow of large grain tasks such as an image transform, diagnostic update, or data base query. It assigns each task to an available (or specified) processor and monitors its status. Once a task has been assigned to a processor, its execution is under the low-level control of the kernel operating system. After task execution is completed, control is returned to the DFOS executive. The data from this task triggers new tasks, continuing the data flow process cycle. Although the DFOS executive must control scheduling and resource allocation for a large and diverse number of processors, its task is greatly simplified because of the high-level nature of this control. This minimizes the potential for the executive to become the processing bottleneck. Since there is little or no inter-task dependency between low-level control functions, they can easily be distributed across the various processors. This allows each processor to have its own unique low-level control structure and ensures that each processor can operate concurrently and independent of others within the same system.

The DFOS Executive - This software provides three basic functions: task scheduling, resource allocation, and task assignment. It is a decentralized control mechanism that supports real-time control, parallel processing and dynamic load balancing while making the machine architecture appear as a single processor to the application user. It can provide demand and data-flow control as well as conventional thread control. This hybrid control approach evolved to meet the variety of real-time scheduling and load dynamics projected to occur in future avionics systems. Some applications such as radar processing are highly regular and driven by stringent real-time constraints and predictability. Other applications such as communications and electronic support measures are primarily driven by the dynamics of the environment and can experience near instantaneous changes in processing workloads of over three orders of magnitude. Still other processes such as diagnostic tests and airframe stress monitoring can run in the background. The run-time executive can support these different control requirements via three basic modes: threading, pooling, and chaining.

Threading permits the application user to bind a flow sequence of nodes (called a thread) to one or more processing elements. The run-time executive supports the distribution of the thread on the processors, its initialization, and firing. After firing, thread nodes (tasks) request data from their predecessors , making this a demand-driven mode. All movement of data between nodes is accomplished through logical queues maintained by the run-time executive. Data integrity and firing order is guaranteed and deterministic latencies can be calculated. This mode is typically used for applications requiring critical time constraints and fast response times (<10 ms.).

Pooling dynamically assigns nodes to available processing resources, permitting uniform load balancing in a dynamic environment. Nodes are executed when all their inputs are available, making this a data driven mode. The run time executive matches an executable node to an available processor resource, and consumes its inputs upon execution. This mode of operation offers the best utilization of processing resources but can create the longest and most non-deterministic latencies.

Although code reuse can provide large gains in software productivity, users often have to trade generality for application efficiency and control. *Chaining* solves this problem by allowing the application user to create new nodes from primitive node sequences, and to customize existing nodes with little or no cost in execution efficiency or operating system overhead. This is done by allowing a sequence of nodes to be designated and scheduled as a single super node. For example, new nodes can be created from primitive nodes such as vector subtract and dot product. Existing nodes can be customized by appending decision logic nodes. Chaining reduces processing latency, memory, and data bandwidth problems by reducing the scheduling and data handling associated with several small nodes to that for a single node. Data integrity and execution sequencing between chained nodes is guaranteed by the run-time executive. The operating system overhead associated with the execution of chains is deterministic, making it applicable to time critical applications. Chaining can be used in conjunction with threading and pooling modes.

Machine Architecture

The machine hardware designers were faced with three major challenges. First, they had to provide hardware support for the application language and operating system. Secondly, they had to design a new communication network that provided an improvement of two orders of magnitude in data bandwidth over existing avionics systems. This was necessary to support both the requirements of the distributed operating systems and the increased clocking frequencies of the next generation of CPUs and memories. Finally, the hardware had to support a diverse mix of signal, image, data, and decision processing.

The resulting machine architecture (See Figure 3) was realized as a small set of standard form/fit modules with common electrical interfaces and a high-bandwidth, asynchronous interconnection network. The basic building block (and LRM) is the module, which conforms to the SEM-E form factor. Modules can be combined (via a set of common buses) to form processing elements such as a data or signal processor. Each processing element can execute high-level tasks and is characterized by having its own executive and local operating system. Functional elements can be combined into tightly-coupled groups called functional groups. These groups provide the close cooperation needed for some applications such as radar processing. Functional elements are combined into a multiprocessing architecture via a fully-connected network switch capable of transferring billions of bits of information per second. The asynchronous nature of this switch network permits processing elements with different functions, I/O bandwidths, and clock speeds to communicate efficiently using a common set of commands and protocols. External communications with sensors, displays, and other subsystems is provided by high bandwidth optical links.

The Processing Modules - The AACP architecture had to support a diverse mix of data, signal, and decision/control processing. Since each class of processing needs unique processing capabilities, multiple processing engine modules are required. The current module inventory include four different processing modules: a general purpose data/control processor, a video processor, a fixed-point signal processor, and a floating-point signal processor. Requirements for decision processing (object or fuzzy rule-based) modules are now under investigation. The data/control processing module is based on a 32-bit RISC architecture augmented with special I/O and fault management coprocessors. These powerful coprocessors free the main CPU from the many support tasks that can rob it of over 60% of its computational utility. The signal processors are highly pipelined, parallel processing ensembles capable of providing several hundred million operations per second. A memory expansion bus and memory modules permit additional memory to be added to these processing elements.

External I/O Modules - Communications between the multiprocessing architecture and other subsystems such as sensors, displays, and weapons/countermeasures are currently handled by two modules: the sensor/video I/O module and the high speed data bus. The sensor/video I/O modules provide the basic building block for the Pave Pillar Sensor Data Distribution Network (SDDN) and Video Data Distribution Network (VDDN), while the high speed data bus provides many high level control functions. Both modules use fiber optic communication links, providing vastly improved data bandwidth and EMI/EMP protection.

The sensor/video I/O module provides four transmit and four receive fiber optic links (point-to-point), each capable of 200 Mbits/sec (data) burst transmission. The data links may be assigned as independent data paths or grouped for greater link bandwidth. The protocol uses a 8B/10B block encoding, based on an expanded version of the Fiber Data Distribution Interface (FDDI) 4B/5B block code used in fiber optic token ring networks. The design greatly reduces the number of buses and discretes normally associated with avionic systems by allowing interrupts, control, and synchronization signals to be embedded in the data streams. Special encoding rules and hardware allow these signals to be stripped from the incoming data and routed directly to a control processor with little or no impact on data movement.

The high speed data bus provides a 100 Mbit/sec data bus between the multiprocessor and other subsystems.* Its primary use is for system setup (booting and initialization), high-level control between subsystems, and fault recovery.

The Communication Network - Internal communications between processing elements and groups are accomplished via network buses and a fully connected switch network. The network bus is a message-passing bus that moves information between the various modules comprising a processing element or group. It also permits these modules to share a common port in the switch network. The switch network provides multiple, simultaneous, non-blocking communications between functional elements and groups. Simultaneous communications are limited only by the number of switch ports and by the number of processing element/group pairs requiring communication paths.

A connection path is guaranteed if the requesting (source) and receiving (destination) ports are not busy. Control of the network is completely decoupled from the actual data transfer operation, thereby permitting control operations such as arbitration, port monitoring, time-out monitoring, and connection/disconnection assignment to be performed concurrently. Each communication path is asynchronous, with the source interface suppling the clock. The switch network can easily be expanded (by adding additional switch modules) to build 8-port, 16-port, or 32-port switches. The burst transfer bandwidth is 40 Mbytes/sec per port. Each port is bi-directional, half-duplex, with full duplex operation accomplished by using two ports per functional element.

Summary

Although the AACP architecture represents major advances in software capture, hybrid control mechanisms, and machine architectures, its most important achievement lies in combining these advances to improve overall system efficiency and supportability. The performance of these systems would have been impossible to achieve with conventional design approaches. Some AACP systems have achieved sustained computational rates in excess of several billion 32-bit floating point operations per second, yet occupy a volume of less than two cubic feet. Large applications are now being captured in one-fifth the normal development time.

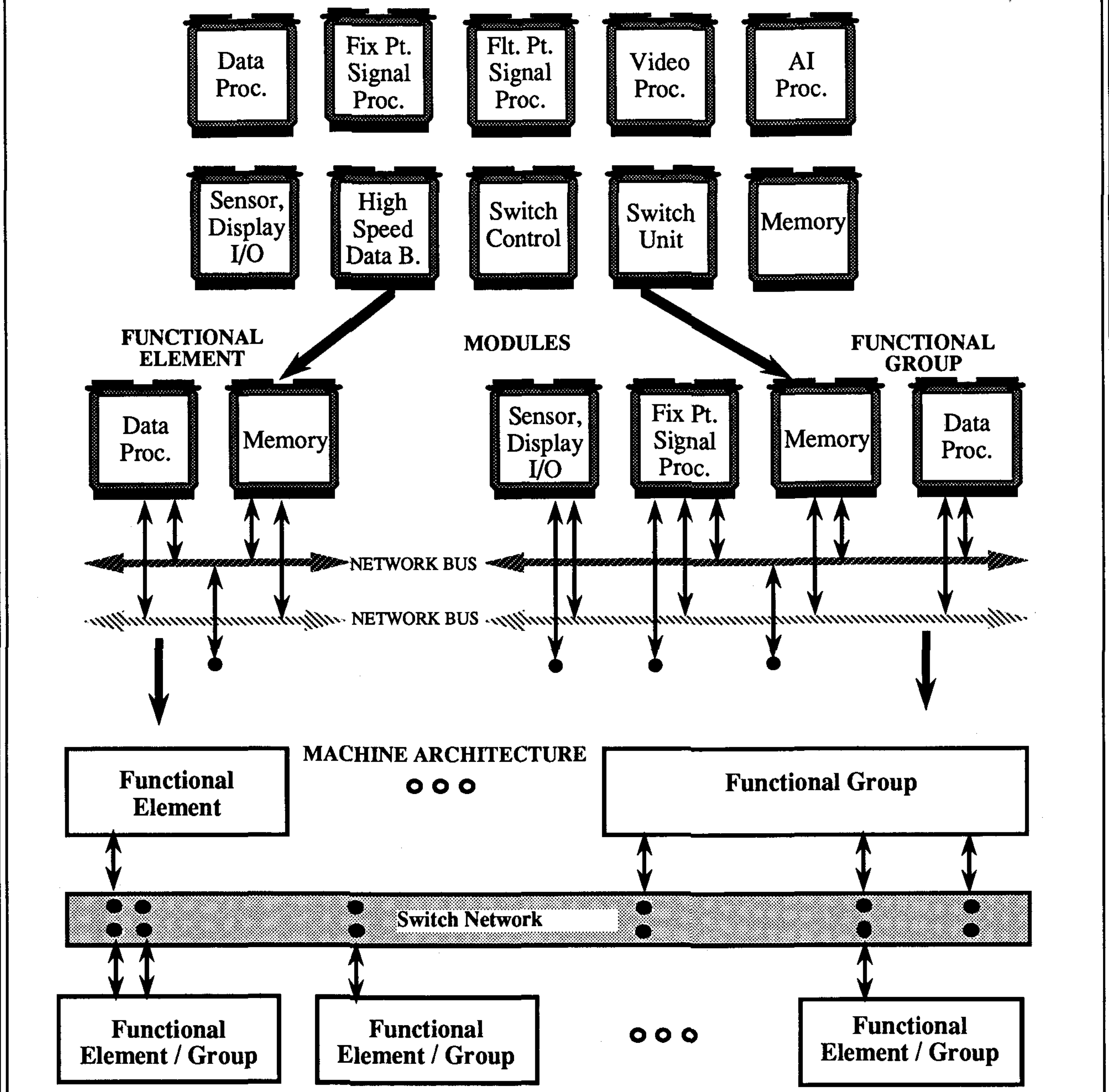

FIGURE 3 - Machine Synthesis