A Configurable System for Automation Programming and Control

James U. Korein
Georg E. Maier[1]
Russell H. Taylor
Lawrence F. Durfee

Manufacturing Research Department
IBM T. J. Watson Research Center
Yorktown Heights, New York, 10598 USA

Abstract: This paper discusses an environment for configuration, programming, and control of robot workcells. The controller is intended to support research in automation programming and motion control, and to provide a vehicle for conveniently integrating new sensors and other devices into a workcell in a useful way. The system consists of an interactive *Programming System* connected through a shared memory to a multiple-processor *Real Time System* that performs time-critical operations. The Programming System executes programs written in an enhanced version of AML and transmits high level commands, called *verbs*, to the Real Time System for execution. Verbs may either be *simple*, consisting essentially of a process specification and termination conditions, or they may be compositions of other verbs. The processes themselves are specified in terms of lower level entities called *real time application subroutines, state vector variables,* and *data flow graphs* which describe computations to be performed in the Real Time System.

Introduction

Rapid progress in robotics and programmable automation is limited by the lack of flexibility and configurability of currently available systems, and the difficulty with which these systems are extended. These problems, among others, have also been detrimental to the integration of programmable automation on the factory floor. One critical problem is to be able to take advantage of the wide variety of new sensor and actuator technologies in the field, and to use them in meaningful ways without redesigning the system. A second is the large amount of labor involved in coordinating a variety of special purpose controllers.

It is the contention of the authors that a general purpose automation controller with an *open system* philosophy could improve the situation in several ways. The first is by allowing users make new devices known to the system and to build new real time commands incorporating these devices. Second, a general purpose controller could be used to provide a common interface to a variety of different robots and other devices, so that these devices may be programmed in a homogeneous manner.

A general purpose automation controller must be readily configured to control a wide variety of equipment, including robots, XY-tables, tools, process equipment, inspection apparatus, conveyors, etc. It must provide sufficient computational power to support the functional requirements of the equipment being interfaced to it. The programming interface must be powerful enough to describe complex behavior, while still allowing simple tasks to be programmed simply. This interface should support a spectrum of users from line attendants and maintenance personnel to robotics researchers.

Robotics research control architectures have taken a number of different approaches to meeting the requirements of configurability, computational power, and programming. Early systems [e.g., 1, 2, 3, 4, 5, 6, 7] typically relied on a mainframe or large minicomputer, or in some cases to a mini tied to a mainframe, to provide both programming and control functions. More recent systems [e.g., 8, 9, 11, 12] have tended to employ larger numbers of processors, interconnected in various ways, and there has been considerable attention recently to the use of specialized computational hardware [e.g., 13, 10] for manipulator control. A number of systems [e.g., 14, 15, 16] have recently paid more explicit attention to the problems of workstation integration. For example, Barberra et al. at NBS [14] define a multiple level control hierarchy and relies on a number of communication interfaces to tie together computers controlling various equipment within the manufacturing cell.

The system described here has been designed with a focus on dynamic configurability, layered user interfaces and incorporation of sensor-based real time operations into new commands. It is these features which distinguish it from earlier work. The system is currently being implemented at IBM for research purposes and internal use, an outgrowth of programmable automation research which has been ongoing since 1972 [e.g., 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27].

System Hardware

The overall structure of the hardware is illustrated in Figure 1. The system consists of a *Programming System,* which provides an environment for the development and execution of programs, and one or more *Real Time Systems,* which perform time-critical tasks such as motion control and sensor monitoring. The Programming System is connected to the Real Time System by a *Real Time Bridge* containing a shared memory and synchronization hardware.

The rationale for separating the Programming System from from the Real Time System is that their needs are quite different. The programming system must support a conventional operating system, a wide variety of development tools, disk and communications network. The Real Time System must provide fast computation, I/O for sensors, and actuators and guaranteed latencies for real time computations. Systems which are best suited to one of these purposes are usually not ideal for the other. Moreover, the forces driving their evolution can be expected to widen the gap. The division allows independent evolution on both sides.

[1] Visiting Scientist from the Department of Automatic Control and Industrial Electronics, Swiss Federal Institute of Technology ETH, CH-8092 Zurich, Switzerland

Programming System

The Programming System consists of a computer workstation of conventional design with the usual array of input/output peripherals, including a hard disk, local area network (LAN) interface, display, keyboard, etc. The primary requirements for this component are reasonably high computation rates, floating point capabilities, large (several megabyte) physical and virtual memory address space, acceptable interrupt response latency, and a wide menu of available hardware and support software for program development and factory communications.

We have made a serious effort to minimize dependencies on the Programming System hardware, configuration, or host operating system. The only "custom" hardware element is the interface card attaching the Bridge to the Programming System bus. All Programming System software is written in *AML* or in a standard high level language (*C*), and all input/output relies on standard *C* subroutine calls. The principal operating system requirements beyond standard I/O are the ability to map the shared memory of the Real Time Bridge into specific addresses in user programs and the ability to awaken user processes in response to interrupts from the Real Time System through the Bridge.

Real Time Bridge

The Real Time Bridge contains shared memory addressable from both the Programming System and the Real Time System, together with a large number (1024) of indivisible test-and-set registers to provide mutual exclusion between processors. It provides the capability for processors on either system bus to interrupt those on the other. Finally, it contains specialized hardware supporting system diagnostics.

The Bridge effectively decouples performance on the two system buses, thus preventing bursts of I/O activity for factory communications, disk file transfer, or display updating on the Programming System bus to degrade access times on the Real Time System bus. At the same time, the shared memory permits high bandwidth, random access communication between subsystems, permitting system software to share global data structures. The Bridge architecture also provides a natural dividing line for future implementation upgrades, since changes in the bus architecture for one subsystem do not necessarily require changes in the other.

Real Time System

The Real Time System consists of multiple single board computers on a shared bus. Each card includes a processor, local memory (both private and shared) and standard interfaces for daughter cards to do I/O functions (Figure 2).

One of these computers is designated the *Supervisory Processor.* The Supervisory Processor executes commands constructed in the shared memory of the Real Time Bridge by the Programming System. It communicates with the other real time processors through their "local" shared memories and through "global" shared memory in the Bridge.

The workload of the system is distributed over all processors. Code is stored in the local memories of the processors to avoid bus traffic. Except for a small module in ROM, the code is downloaded into local RAM when the system is configured. Most processors perform special purpose functions such as axis servo control and sensor interface functions.

All processors are capable of being masters on the Real Time System bus. Processor communication is accomplished using shared memory and special prioritized interrupts for interprocessor communication.

Real time processing cards may be customized by the attachment of any of a number of daughter cards through an industry standard local bus [31]. Daughter cards are available to provide various types of I/O including A/D and D/A conversion, digital and serial I/O.

As with the Programming System, we have spent considerable effort to minimize hardware dependencies. Aside from a small kernel whose primary function is interrupt dispatching, all software is being written in high level languages. Beyond this, custom interfaces to robot hardware are confined to daughter cards. This isolates dependencies on specific robots, as well as keeping I/O traffic between a processor card and its daughters off the global real time bus.

Typical Configurations

A principal design objective of the system is that it should be configurable to support a wide variety of automation equipment. This is achieved by providing modular interfaces at the joint and sensor control level to permit many different I/O interfaces and protocols, by providing standard configuration tables and utilities, and by supporting standard communications interfaces in the programming system.

Typical configurations may be designed for applications involving pick and place, electronic test, visual inspection and assembly.

The workstation may include one or more robot robot arms, XY positioning tables, a vision system, analog test equipment, and a variety of devices controlled through serial and parallel ports. An abundance of digital inputs and outputs may be used to handle simple sensors and devices.

Layered User Interface

The community of system users is expected to span a wide range. At one end of the spectrum is the manufacturing engineer who wants a simple menu-driven or teach-pendant interface to assist him in setting up a manufacturing workstation in an expedient manner, without writing complex computer programs. Next, there are robot programmers developing applications programs to control the robot directly. Then, there are programmers writing programs which provide manufacturing engineers with the menu-driven and teach pendant application interfaces they require. Finally, there are robotics researchers experimenting with new control and trajectory planning strategies to extend the capabilities of robotic systems.

In order to accommodate these different levels of use, several layers of user interface are provided.

1. The outermost level encompasses menu-driven and pendant driven application interfaces which are useful for manufacturing engineers and factory personnel.
2. The next level is programming in AML [19, 20, 28, 29]. The Programming System provides an interpreter for an enhanced version of the *AML* programming language which supports exception handling and powerful mechanisms for data abstraction. AML callable subroutines may also be defined in lower level languages like *C* [30] and *FORTRAN*. A library of predefined commands for the Real Time System is provided to do motion control and condition monitoring. These commands are called *verbs*.

3. The next layer is the *verb composition* level at which new *verbs* may be constructed, using a set of primitive AML callable subroutines provided by the system. At this level, the building blocks for constructing new verbs are other verbs which have previously been defined. A detailed knowledge of the system is not required. New verbs take on the appearance of old ones.
4. Still deeper is the *primitive verb definition* level, which allows users to develop verbs which are fundamentally new. This is necessary to incorporate new types of devices into the Real Time System, and to experiment with new types of motion control. At this level, *verbs* are defined in terms of *specifications of real time computations*. Programming the system at this layer allows the introduction of radically new types of hardware and software into the system, but requires knowledge and caution on the part of the user.

All levels are built on top of a core of system subroutines.

The intent of this layered architecture is to provide a vehicle for experimentation with new technologies in programmable automation, and to provide a path by which the fruits of this research can flow quickly and painlessly into productive manufacturing systems.

Verbs

Verbs are commands from the Programming System to the Real Time System. They may be applied to *devices*, e.g. to monitor sensing devices or to move robots as well as other devices.

The conceptual representation of a verb is depicted in Figure 3. The verb consists of a *process*, which is the action which transpires during the execution of the verb, and a set of *termination conditions* (T1,...,T*n*), any of which may cause the verb to terminate. This concept is a natural generalization of the *guarded move* concept [17].

In a typical guarded move, for example the *process* would be the motion of a robot arm, and termination conditions might include reaching a desired position within some tolerance and encountering a large force.

In addition, the verb, like a subroutine, has a set of formal parameters. In the guarded move example, some parameters would be the device to move, the position goal, and a threshold level on the force.

A verb is a generic command which may be applied to devices of different structures. For example, the verb *MOVE* may be used to move one or several joints of a robot:

```
MOVE(j2,goal);
MOVE(<j1,j3>,<goal1,goal3>);
```

Application of a verb to a device (or a set of devices) is called a *verb instance*. A verb is *re-entrant* in the sense that several instances of the same verb may be executed concurrently (but independent of each other).

Depending on the termination condition which actually occurs, different sets of values may be returned by the verb (r1,...,r*n*). If a guarded move reaches its destination, indicating this with an appropriate return code may be sufficient. If, however, the motion is terminated due to a force, it may be desirable to return several values including the measured force vector, the position at which it was encountered, and the position at which the arm stopped.

Verb Composition

Verbs are modular building blocks which can be composed to build new verbs. These new verbs, which are called *compound verbs* have the same external structure as do *simple verbs* and can be used in exactly the same ways. Compound verbs may be constructed as *verb graphs* whose nodes are themselves verbs.

An example is given in Figure 4. One node is designated as the *starting node*. For each termination condition of each node, there is exactly one *arc* which points either to another node in the graph, or to a termination condition for the entire compound verb, in which case it is called a *terminating arc*.

The starting node is executed first. When a node terminates, subsequent action is determined by the arc leaving the termination condition which occurred. If this arc goes to another node, that node is to be executed next in sequence. However, if the arc is a terminating arc, the entire compound verb terminates with the specified termination condition.

When the compound verb is defined, a mapping between the parameters of the compound verb and those of each of the constituent verbs is established. The number of parameters for all constituent verbs may be large. A default mechanism is provided so that all these parameters need not be specified for the compound verb.

Just as for a simple verb, a set of returned values must be defined for each termination condition of the compound verb. Each terminating arc must map the returned values from its source onto those of its destination. That is, returned values of a verb node are mapped onto those for the compound verb.

An example of a useful operation which is conveniently implemented as a compound verb is shown in Figure 5. This verb, called *centering grasp* is used to grasp an object without wasted motion. The verb is invoked when an object to be grasped is known to be between the fingers of the gripper of a robot arm. The verb has four nodes, each of which is a guarded move (*GMOVE*). The first closes the gripper until either or both fingers hit the object, or until the gripper closes to a specified width. This last condition is used to check for the contingency that the object is not between the fingers, in which case the compound verb reports the erroneous condition "Too Small". If both fingers happen to hit the object at the same time, a final *GMOVE* is used to tighten the grip on the object. For the sake of simplicity in this example, the tightening operation is assumed always to work correctly. When the fingers are being closed, if the left finger hits the object before the right, another guarded move is invoked to compensate for the skew by moving the gripper left while it is closing. If the second finger then hits, control passes to the tightening verb. However, if the first finger which hit looses contact, control passes back to the original finger closing verb. As before, if the fingers get too close together for the desired object to be between them, the verb terminates with the "Too Small" condition. The fourth node in the graph handles the case where right finger hits first.[2]

Composition of verbs provides a convenient abstraction for the definition of complex motion commands from simple ones in a modular way.

Predefining compound verbs allows large tasks to be performed by the Real Time System with a minimum of traffic between the Programming System and Real Time System.

[2] A similar method was developed by R. Paul in the early 1970's and appeared in both the *WAVE* and *AL* systems. Interestingly, this *CENTER* command was first implemented as a macro-operation in WAVE, but constant use led it to be reimplemented as a primitive [33].

Verb graphs are not implemented as conventional programs but as data structures. This allows them to be easily manipulated in order to remove the overhead introduced by nesting, and permits them to be configured dynamically, without the overhead of a language interpreter. A graphical interface is planned to allow convenient construction of compound verbs.

Specification of Real Time Computations

Function Blocks and State Vector Variables

A *function block* specifies a generic piece of computation with a well defined interface which consists of input and output ports, conditions used to report verb termination, and formal parameters with default values.

A *state vector variable* is a global data buffer in the shared memory of the Real Time System which may be used to communicate between different function block instances. State vector variables have a *type* and may hold a number of previous values.

An *instance* of a function block may be obtained by binding its input and output ports to state vector variables. Several instances of a function block, which are bound to different state vector variables, may exist at the same time. A function block instance may be scheduled to be executed once, which results in a *single invocation*, or it may be activated for repetitive execution, which results in *multiple invocations*.

Actual parameter values are passed to a function block instance before it is scheduled. Parameters are then local to the instance and cannot be accessed from outside.

The *input* and *output ports* allow an instance of a function block to access state vector variables. The value associated with a port is local to the instance during one invocation: It is guaranteed that an input does not change and an output is not accessible from outside while the function executes. The ports have a *type* and can only be bound to state vector variables of matching types.

Computations in the Real Time System may be performed at different frequencies. For example, some low level control computations need to be executed very often, whereas the generation of set points may be done less often. Therefore, a function block port has a *mode* which defines whether this port will be accessed during each invocation or not:

- *synch*(default): The port is accessed during each invocation (updated input value assumed, new output value produced).
- *fixed_ratio*: The port is accessed during each k^{th} invocation, where k is a constant.
- *var_ratio*: The port is accessed during each n^{th} invocation, where n is a variable.
- *asynch*: The port may be updated asynchronously to the invocations of the function block.

An *execution interval* may be associated with a function block instance, if it must always be executed at a certain frequency.

A port with the mode *synch* may have the type *trigger*. Such a port cannot be read or written but only affects the flow of control. A trigger port is not bound to a state vector variable but to a *trigger source*. An input trigger port may either be bound to an interrupt (e.g., from the real time clock) or to an output trigger port of another function block instance.

The following commands may be used in *command lists* to schedule function block instances:

- *install*: A function block instance must be installed in the Real Time System before it may be executed. All state vector variables bound to output ports and all physical devices used are reserved in order to detect any *resource conflict*.
- *remove*: After execution, a function block instance may be removed to allow another function to write the same state vector variables and to use the same physical devices.
- *exec_once*: Execute a function block instance once (single invocation).
- *activate*: Schedule a function block instance for repetitive execution as specified by its execution interval.
- *deactivate*: Do not invoke a currently active function block instance anymore.
- *wait*: Wait until one of the active function block instances reports a condition (illegal, if no instance is active).

Execution of such a command list terminates as soon as one of the involved function block instances reports a condition different from *success*. In case of a simple verb, which is defined by one command list, the reported condition terminates the verb instance and is returned to the Programming System as a verb termination code. In case of a compound verb, which is defined by one command list for each of the nodes of the verb graph, the reported condition determines which node is executed next or whether the verb terminates. At the time a verb instance terminates, any of its function block instances which are still active or installed are automatically deactivated and removed.

Application Subroutines

Basic function blocks, coded as *C functions*, are called *application subroutines*. A set of *coding conventions* must be used to meet the function block interface requirements. A few *support functions* allow an application subroutine to report a condition, to access the current time, to get the execution interval, and to determine whether the current invocation is the first one.

When an application subroutine is compiled, in addition to the object file a so called *symbol file* is created which includes function names, type definitions, variable names, field names etc. These symbol files are automatically read and processed at the time an object module is downloaded to the Real Time System. As a consequence, the Programming System is able to check whether the coding conventions have been observed and to automatically configure the application subroutine interface for the most part. Only port types different from *synch*, condition values and formal parameter default values different from zero must be configured explicitly.

Application subroutines are the only functional objects in the system which cannot be introduced at any time but must be compiled and linked before the application is downloaded to the Real Time System.

Data Flow Graphs

A function block composed of one or more other function blocks is called a *data flow graph*. Such a graph is defined by

- a set of named function block nodes,
- a set of named data nodes which are either a port of the data flow graph or a typed data cell internal to the graph,
- a function which maps each input port of each function block node to a data node,
- a function which maps each output port of each function node to a data node,
- a function which maps each condition of the graph to one or more conditions of one or more function block nodes, and

- a function which maps each formal parameter of the graph to one or more formal parameters of one or more function block nodes.

Internals of Verbs and Devices

A verb is internally represented by

- a formal parameter list (including default values, types, and keywords),
- one or more function blocks and/or function block instances,
- one or more command lists to install, execute (once or repetitive), deactivate, and remove function block instances,
- an input parameter table to define the distribution of actual parameters to function block instances, and
- one or more output mapping tables (each associated to a set of termination conditions) to define the return values.

When a verb is called, the device(s) and/or state vector variable(s) passed as actual parameters are used to build a new verb instance which includes instantiation of the function blocks of the verb. According to the input parameter mapping table, the remaining actual parameters are then distributed to the function block instances, and the verb instance is passed to the Real Time System to be executed. The termination code returned to the Programming System determines which of the output mapping tables is used to compute the actual verb return values.

A *device type* is internally represented by a *verb*. For example, the device type *joint* would be represented as a verb to continuously servo a joint to a desired position. Application of a device type results in a new *device* which is internally represented by a *verb instance*. In the case of a joint, this would bind a joint servo to specific sensors and actuators as well as to a specific state vector variable where a new desired joint position is picked up repeatedly as soon as the device is enabled.

When a verb is applied to a device, the necessary communication is actually performed through one or more state vector variables, each either written by the verb instance and read by the device or vice versa. Before a verb may be applied to a device, the device must be *enabled* which results in the execution of the verb instance representing the device. When a device is no longer needed, it may be *disabled*, resulting in terminating the corresponding verb instance.

Example

The concept of verb implementation is illustrated by the example of a *guarded move* applied to a single joint *j1*. Figure 6 shows all the function block instances which are involved.

The device *j1* is represented by a verb instance which consists of the function block instances *pdInit* and *j1__servo* and the command list

```
install(pdInit);
install(j1__servo);
exec__once(pdInit);
activate(j1__servo);
remove(pdInit);
wait;
```

which is executed in the Real Time System when the joint is enabled by the statement *ENABLE(j1)*. The function block instance *j1__servo* stays activated, i.e. is invoked each 5 ms, until the joint is disabled again.

The verb instance corresponding to *GMOVE(j1,...);* consists of the function block instances *plan*, *genSetPoint*, and *monitor* and the command list

```
install(plan);
install(genSetPt);
install(monitor);
exec__once(plan);
activate(monitor);
activate(genSetPt);
remove(plan);
wait;
```

which is executed when the verb instance is passed to the Real Time System. The function block instances *monitor* and *genSetPoint* stay activated, i.e. are invoked each 20 ms, until the goal position is reached or a force is encountered, and the verb instance is terminated.

Real Time Execution Model

Supervisor

The system code called the *supervisor* resides on the supervisory processor of the Real Time System. The supervisor receives the verb instances to be executed from the Programming System and controls the behavior of the Real Time System by interpretation of the corresponding command lists. Whenever a verb instance terminates, the supervisor returns a termination message to the Programming System.

In order to minimize overhead, a function block instance is passed to the Real Time System as a set of *action sequences*. An action sequence is a list of subroutines to be called strictly sequentially on the same processor. An action sequence is typically bound to an interrupt, so that it may be triggered by simply issuing that interrupt.

In the simplest case (e.g., a sequence of application subroutines passing only information from one to the other through ports with the mode *synch*), an action sequence contains calls to application subroutines only. Cases where a function block instance is distributed over several processors or uses port modes other than *synch* are more complex. In these cases, an action sequence contains additional function calls to trigger other action sequences and to provide mutually exclusive access to state vector variables.

The action sequences representing a data flow graph are prepared as soon as it is defined. First, the data flow graph is reduced to the corresponding *control flow graph*. This may raise an exception, because there does not exist a control flow graph for any arbitrary data flow graph. The control flow graph includes two solutions. The first produces a sequence to execute the data flow graph on a single processor, the second shows the highest degree of parallelism permitted by data flow constraints. Fortunately, most of the relevant applications map to one of these two solutions, and in other cases satisfactory results may be obtained by manually assigning each application subroutine instance to a processor. The problem of automatic load distribution onto multiple processors with minimal overhead for dispatching and synchronization has not yet been investigated.

Dispatcher

The dispatcher is distributed over all processors of the Real Time System. It actually executes the action sequences and provides the operations to trigger other action sequences and to perform the synchronization necessary to mutually exclude concurrent accesses to state vector variables.

Conclusions

This paper has described the design philosophy, structure and some key concepts of a general purpose controller for programmable automation.

At this time the system is partially implemented. When the system is complete, it will be used for the investigation of automation applications, the development of new motion control techniques, and the incorporation of new sensors into new verbs.

As well as supporting new research in robotics, the system provides a convenient way to describe distributed real time computations. Some interesting questions are raised about the translation of verb graphs and data flow graphs into efficient computations.

We contend that verb concept is a useful model for describing actions, and that the methods described for building and composing them make them a modular tool as well. By the creation of verb libraries, we hope to provide some of the wide variety of behaviors that are captured in the verbs of natural language.

Acknowledgements

We would like to thank our colleagues for their technical contributions to the work discussed in this paper. Lee Nackman, Mark Lavin, Jim Colson, Dave Klein, Ken Morgan and Barry Russell have contributed to the programming system. Martin Sturzenbecker, Kevin Short, Bob French and Gary Franklin have contributed to the design and development of the Real Time System. Robert Eng and Jim Brewer have done substantial work in hardware design and implementation. Many others at IBM Research in Yorktown Heights, at the Manufacturing Systems Products group in Boca Raton and at The Entry Systems Division Process Development Lab in Austin have made contributions.

References

[1] H. A. Ernst, *MH-1, A Computer-Operated Mechanical Hand*, Sc.D. Thesis, MIT, December 1961.

[2] Richard Paul, *Modelling, Trajectory Calculation, and Servoing of a Computer Controlled Arm*, Ph.D. Dissertation, Report STAN-CS-72-311, Stanford University, November 1972.

[3] Carl Ruoff, "PACS: An advanced multitasking robot system", *The Industrial Robot*, June 1980, pp 87-98.

[4] D. Silver,*The Little Robot System*, MIT Artificial Intelligence Laboratory Memo 273, January 1973.

[5] R. Finkel, R. Taylor, R. Bolles and J. Feldman, "AL, A Programming Language for Automation", Stanford Artificial Intelligence Laboratory Memo AIM-243, Stanford University, 1974.

[6] S. Mujtaba and R. Goldman, "AL user's manual." Memo AIM-323, Artificial Intelligence Laboratory, Stanford Univ., Stanford, CA, 1979.

[7] Clifford Geschke, *A System for Programming and Controlling Sensor-Based Robot Manipulators*, Ph. D. Dissertation, University of Illinois, December 1978.

[8] F. Ozguner and M. L. Kao, "A Reconfigurable Multiprocessor Architecture for Reliable Control of Robotic Systems", *Proceedings of the 1985 IEEE International Conference on Robotics and Automation*, pp. 802-806, March 1985.

[9] R. Nigam and C. G. S. Lee, "A Multiprocessor-Based Controller for the Control of Mechanical Manipulators", *Proceedings of the 1985 IEEE International Conference on Robotics and Automation*, St. Louis, pp. 815-821, March 1985.

[10] S. Ahmed and C. B. Besant, "Motion Control of Industrial Robots with Closed-Loop Trajectories," *Proceedings of the 1984 IEEE International Conference on Robotics*, pp. 305-309, March 1985.

[11] I. Lee and S. Goldwasser, "A Distributed Testbed for Active Sensory Processing", *Proceedings of the 1985 IEEE International Conference on Robotics and Automation*, St. Louis, pp. 925-930, March 1985.

[12] D. Siegel, D. Kreigman, S. Narasimhan, G. Gerpheide, J. Hollerbach, "Computational Architecture for the Utah/MIT Hand", *Proceedings of the 1985 IEEE International Conference on Robotics and Automation*, St. Louis, pp. 919-924, March 1985.

[13] *Proceedings of Special IEEE Workshop on Computational Architectures for Robotics*, St. Louis, March 24, 1985.

[14] A. Barberra, M. Fitzgerald, J. Albus and L. Haynes, "RCS, The NBS Real-Time Control System", *Proceedings of Robots 8 Conference*, Detroit, pp. 19-1 to 19-34, June 1984.

[15] K. Shin and M. Epstein, "Communication Primitives for a Distributed Multi-Robot System", *Proceedings of the 1985 IEEE International Conference on Robotics and Automation*, St. Louis, pp. 919-924, March 1985.

[16] R. D. Gaglianello and H. P. Katseff, "MEGLOS: An Operating System for a Multiprocessor Environment" *Proc. 5'th IEEE International Conference on Distributed Computing Systems*, Denver, pp. 35-42, May 1985.

[17] P. Will and D. Grossman, "An Experimental System for Computer Controlled Mechanical Assembly." *IEEE Trans. Comput.*, C-24, pp. 879-888, 1975.

[18] R. H. Taylor and D. D. Grossman, "An Integrated Robot System Architecture", *IEEE Proceedings*, Vol. 71, pp. 842-855, July 1983.

[19] R. Taylor, P. Summers, and J. Meyer. "AML: A Manufacturing Language." *International Journal of Robotics Research*, Vol. 1, No. 3, pp. 19-41, 1982.

[20] (anon.), *IBM Manufacturing System: A Manufacturing Language Reference Manual* No. 8509015, IBM Corporation, 1983.

[21] M. Lavin and L. Lieberman, "AML/V: An Industrial Machine Vision Programming System." *International Journal of Robotics Research*, Vol. 1, No. 3, 1982.

[22] R. Taylor, R. Hollis, M. Lavin, "Precise Manipulation with Endpoint Sensing", *IBM J. of Research and Development*, Vol 29, No 4, pp. 363-376, July 1985.

[23] R. L. Hollis, R. H. Taylor, M. Johnson, A. Levas and A. Brennemann, "Robotic Circuit Board Testing Using Fine Positioners with Fiber-Optic Sensing", IBM Research Report RC #11164, May 21, 1985; also in *Proceedings of the International Symposium on Industrial Robots*, Tokyo, September 1985.

[24] R. L. Hollis "A Fine Positioning Device for Enhancing Robot Precision", *Robots 9 Conference Proceedings*, Detroit, Michigan, pp. 6-28 to 6-36, June 1985.

[25] J. Ish-Shalom and D.G. Manzer, "Commutation and Control of Step Motors", *Proceedings of 14th Annual Symposium on Incremental Motion Control Systems and Devices*, Champaign, June 1985.

[26] R.H. Taylor, J.U. Korein, G.E. Maier and L.F. Durfee, "A General Purpose Control Architecture for Programmable Automation Research," *Third International Symposium on Robotics Research*, MIT Press, 1986.

[27] G.E. Maier, R.H. Taylor and J.U. Korein, "A Dynamically Configurable General Purpose Automation Controller," *Fourth IFAC/IFIP Symp. on Software for Computer Control*, Graz, Austria, May 1986.

[28] L. R. Nackman and R. H. Taylor, "A Hierarchical Exception Handler Binding Mechanism", *Software - Practice and Experience*, Vol.14 (10), pp. 999-1007, Oct. 1984.

[29] L. R. Nackman, et. al., *The Yorktown Experimental AML Reference Manual*, Document in preparation, IBM T. J. Watson Research Center, Yorktown Heights, NY.

[30] B. Kernighan and D. Ritchie, *The C Programmming Lanuguage*, Prentice Hall, 1978.

[31] *INTEL iSBX™ Bus Specification*, Intel Corporation; Manual 142686-002 Mar81

[32] Vincent Hayward and Richard Paul, "Robot Manipulator Control under UNIX," from TR-EE 84-10, Purdue University School of Electrical Engineering, pp. 22-34, Jan. 1984.

[33] Robert Bolles and Richard Paul, *The Use of Sensory Feedback in a Programmable Assembly System* Stanford University Computer Science Report STAN-CS-396, October 1973.

[34] T. Lozano-Perez, M. T. Mason, R. H. Taylor, "Automatic Synthesis of Fine-Motion Strategies for Robots", *Int. J. of Robotics Research*, Vol. 3, No. 1, pp. 3-24, Spring 1984.

Figures

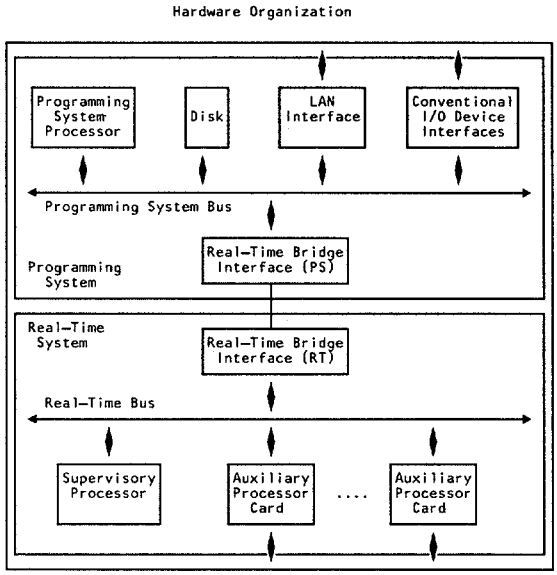

Figure 1.

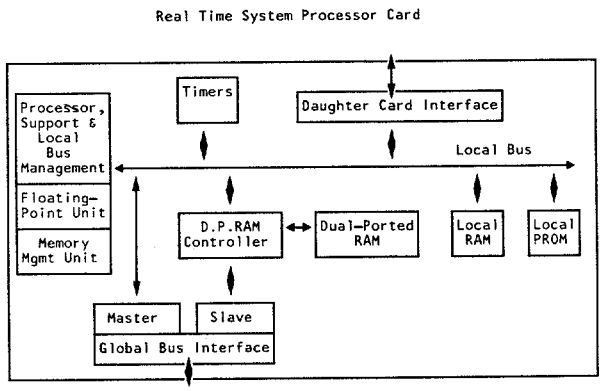

Figure 2.

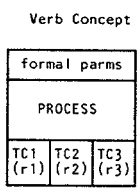

Figure 3.

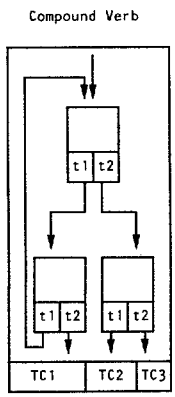

Figure 4.

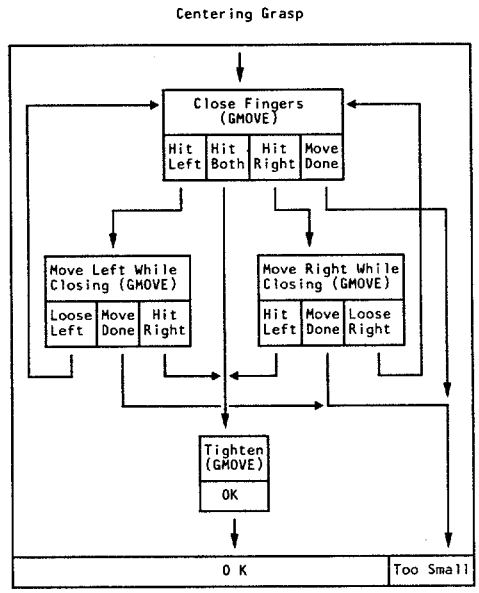

Figure 5.

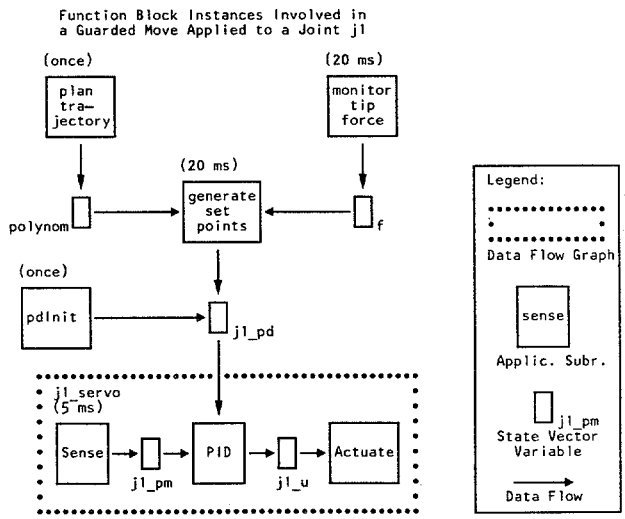

Figure 6.