A KNOWLEDGE-BASED CONTROL PARADIGM FOR REAL-TIME SYSTEMS

John H. Painter, Shih K. Lin, and Emily Glass

Department of Electrical Engineering
Texas A&M University
College Station, Texas 77843-3128

ABSTRACT

This paper examines the application of knowledge-based symbolic control to the management of execution and configuration of a complex numerical control system. Symbolic processing is used to implement inference of system 'state' and internal communication for inference and control. The Flavor System provides an object-oriented programming environment in which the inference engine and knowledge base for the Symbolic Controller are realized. System communication is accomplished by asynchronous message passing using mailbox facility.

INTRODUCTION

Knowledge-based Control may be defined as the integration of symbolic processing with procedural numerical control. This paper examines its application to the management of execution and configuration of a complex numerical control system, such as a guidance system. Here, symbolic processing is used to implement inference of system 'state' and to implement internal communication for inference and control.

The concept of Intelligent Controls was first proposed by K.S. Fu, to link Artificial Intelligence (AI) and Automatic Control [1]. An important contemporary example application is industrial automation, where machines (robots) function without complete a priori knowledge of the work environment. Current implementations are rudimentary in terms of some previous promises of AI [2]. This paper's approach is pragmatic, however, following the suggestion of [3] :

"... treating AI as an engineering discipline having a new set of software tools and techniques that can create more powerful and natural systems, regardless of any similarity to human solution techniques." [3]

This research was supported, in part, by E-SYSTEMS, Inc., Garland Division, of Dallas Texas, under Contract 4901 EP-40169

A 'hands-on' approach has been employed in the present work. A specific application has been chosen as a target around which to develop general results. The particular target is a 'software-intensive radio,' which is envisioned as being digitally implemented. Symbolic processing is used to internally control the radio down to the module level. Testing is via computer emulation (Monte Carlo). The symbolic processing architecture so developed is held to be generic and not dependent on the target application, per se.

A GENERAL SYSTEM ARCHITECTURE

An architecture is presented for integrated symbolic/numeric processing, similar to one postulated by Saridis [4], but developed independently thereof. Saridis viewed such an architecture as a hierarchy, according to his tenet, 'Principle of Increasing Precision with Decreasing Intelligence.'

Our architecture (See Figure 1., below) is four-level and is inverted from Saridis' view, in that the upper levels are high precision (numerical processing), corresponding to the target application. Sensory input to the application is at the top level. The top two levels comprise numerical processing, with the topmost level being the prime 'Application' level and the next level down being numerical 'Support' modules. These latter modules may support the top level or the third level down. The third level contains symbolic processing modules, functioning to draw inferences and to provide control (generalized) for the top levels. The final, fourth level comprises the 'Data-base,' or library, which supports the'Inference/Control' level.

This paper is concerned primarily with the third ('Inference/Control')level. This level generally has two different types of modules serving different purposes. The purpose of one type is restricted to real-time ('decision- directed') control (management) of the target application. Control tactics are implemented here, which may not be compatible with an inflexible, 'hard-wired,' direct encoding of algorithms

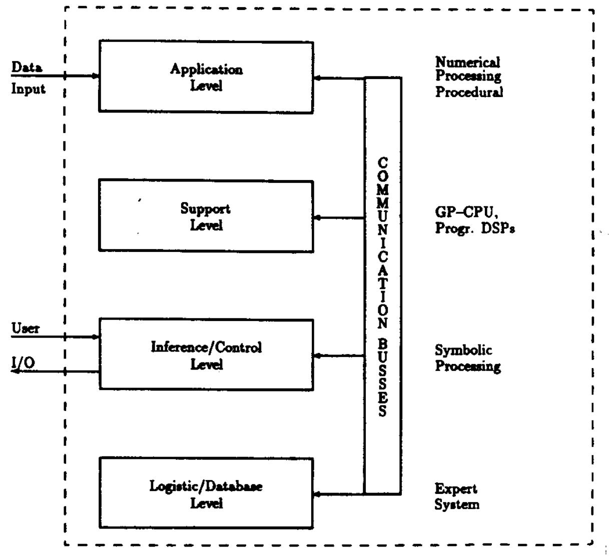

Figure 1. Total System Architecture

in the target system, itself. Such control tactics are, for example, those depending on heuristic 'rules,' such as might be used by a human operator. The second type of symbolic module at this level is that which interprets, not the state of the numerical processor, but the data flowing through the processor. This second type of symbolic inference module is not dealt within the present paper.

Due to the 'real-time' requirement for control, there is a tradeoff between which control functions can be implemented in symbolic processing and which functions must be implemented in directly encoded numerical algorithms. In order to differentiate the two, the first are characterized as 'Expert Control,' while the latter are called 'Hard-wired.' *Those control functions requiring execution speeds beyond the capabilities of symbolic processing must be hard-wired.* Here, we consider only symbolic processing for Expert Control.

The symbolic module, which is to infer 'state' and to control the target system, is simply called the 'CONTROLLER.' This controller gathers from the target system such indications of performance as are required to infer the 'operational state' of the controlled system. Note that an implicit assumption is made that the targeted applications are not of such large scale that distributed ('democratic') control is required. Rather, control responsibility and authority is concentrated in a single module.

THE INDIRECT CONTROL STRATEGY

STATE : It is conceived that some of the application system's modules are sufficiently complex that their 'states' may change from time to time, and that these changes are mode dependent. That is, an operational mode change may imply a state change, and vice versa. For example, consider a module which implements a tracking loop, or some other synchronization module, in the target application. As this module is activated, de-activated, and re-activated during operation, various internal 'signals' (waveforms), when observed, indicate the module's instantaneous operational status. Rudimentary sensing algorithms may be implemented in the module which then indicate module status to the external world. When automated, such indications take the form of (binary) 'flags.' *The 'state' of a module may then be defined to be the set of all its flags, indicated by a digital word.* By extension, *the 'state' of the controlled system is the superset of module states, indicated by an ordered collection of words.*

OPERATIONAL MODES : The operation of the application system is conceived as consisting of a sequence or set of a finite number of operational 'MODES.' Defining each operational MODE then spawns sets of describing 'characteristics' which are true for the controlled system, mode-by-mode. These characteristics describe the dynamic architecture of the controlled system, defined as the set of modules, each of which actively processes data during the indicated mode of operation.

The first describing characteristic of the controlled system, for any particular mode, is 'PATH.' PATH implies a set of instructions indicating where each numerical processing system module is to pass its output data. The second describing characteristic is 'SEQUENCE.' This spawns a set of instructions describing the sequence in which the various numerical processing modules are to execute in a given mode, as data passes through the architecture. SEQUENCE is essentially the instantaneous 'connection diagram' of the numerical system's modules, mode by mode. Now, SEQUENCE and PATH may seem to be redundant. However, this definition makes provision for dealing with a synchronous application system, having explicit parallel internal structure, by using separate, independent (unsynchronized) hardware modules, wherein the synchronization burden is carried by inter-module communication. A final describing characteristic for the application system is 'PARAMETER.' This spawns sets of numerical coefficients which each module uses as it performs numerical processing, mode by mode.

Control is imposed on the application system in an indirect way. CONTROLLER reads STATE and, consulting data-bases of rules, determines MODE, SEQUENCE and PARAMETER. SEQUENCE is used internal to CONTROLLER to further determine PATH. CONTROLLER then communicates PATH and PARAMETER to each active module. Each processing module communicates to the modules in its path the fact that it has produced data (buffers) which are available for further processing. Thus, CONTROLLER does not intervene (directly) in the flow of data through the application system.

COMMUNICATION: Inter-module communication is implemented asynchronously, to give a control strategy which is generally applicable to synchronous or asynchronous applications, which may also be distributed. The communication method which corresponds to the control described above is called 'message-passing.' [5,6,7]. Space does not here allow citation of the extensive recent literature concerning this subject, which is associated with distributed processing, concurrent systems, and threads of control.

CONTROLLER IMPLEMENTATION

The Symbolic Controller, shown in Figure 2., is being developed in an object-oriented Common Lisp environment [8], called the Flavor System. The reason for employing this particular programming technique will become evident in subsequent discussion. Another important feature which is indispensable to the symbolic control system is the Foreign Function

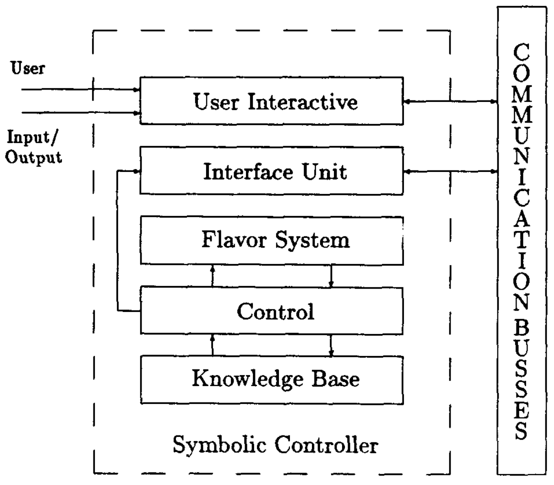

Figure 2. The Symbolic Controller

Interface services supported by the same programming environment. This inter-language interface allows a LISP program to link with compiled C, FORTRAN, and PASCAL images to support cooperation with 'foreign processes' in those languages. Communication channels between the emulated Controller (LISP) and numerical processor (FORTRAN) are implemented using a VAX/VMS interprocess communication utility, "MAILBOX" [9,10,11]. Communications internal to Controller are, however, accomplished by sharing instance variables in the Flavor System.

FUNCTIONAL DESCRIPTION: Figure 2. shows the architecture of the Symbolic Controller, consisting of five functional modules. These are the primary Control module, plus four additional modules. These are now explained. Note, however, that the ordering of functional modules does not correspond to their operating sequence in the controller.

In the total system simulation, the Symbolic Controller first conducts a series of initialization tasks before interacting with the numerical processor. These initialization activities are sponsored by the Interface Unit shown in the Controller architecture. Therefore, this functional module is first described.

Three foreign functions are defined for the Controller in the initialization stage. They are each responsible for designated tasks: SYSACCS spawns the numerical processor for the Controller as its attached concurrent subprocess. Then it creates two mailbox facilities, to which all concurrent processes in the system can have access. WRTATTN_READ is a function that queues an unsolicited read request to a mailbox, where messages sent from the numerical processor are received. Finally, WRT_TO is responsible for transmitting messages from the Controller to the processor. System-defined routines are available to support these operations [10]. The detailed communication mechanism for the concurrent processing system is explained below. For more information on system concurrency and inter-process communication, see [5, 12-15].

The User Interactive module is the interface to the external world. It deals with possible human operators and provides user I/O. In the initial system emulation mode, this module queries the user for data to initialize the numerical processor. This is both an initial operational mode and a pre-simulation step, since it also sets up the simulated 'signal generator' for the target application. Parameter initialization is accomplished interactively, and the procedure is manual driven. The user can change internal parameters such as numer of data bits, carrier amplitude, signal to noise ratio, and so on. The user can also ignore the query, in which case, a set of previously defined parameters are used instead.

The Control module uses two supporting modules, being the Flavor System and the Knowledge Base. (Note that in Figure 1., the Knowledge Base was shown separately, for clarity.) The Flavor System is actually the object-oriented programming environment, which supports AI techniques for symbolic manipulation of knowledge [16]. As related earlier, the Controller makes decisions on the numerical system's current and succeeding operation modes in order to achieve symbolic control. These decision making routines are implemented individually as objects, or in our case, by flavors. Functions associated with various decision specifications are then represented as 'methods,' which the flavors may perform. Another feature of the programming environment, called 'inheritance,' allows the combining of methods and provides internal (to the controller) communications as a side effect. An example of this concept is provided below in discussion of the internal communication design for the Symbolic Controller.

The Knowledge Base is a collection of symbolic representations of the numerical processor's internal states. They are library-like data structures, which are implemented in 'Hash Tables' [17]. Each table entry consists of a pair of associated objects, a *key* and a *value*. Desired *values* of some objects are obtained by searching for matches of their corresponding *keys*. For example, in the target application, the Hash Table MODLIB is the data base corresponds to the decision routine MODEID that determines the current operation mode for the numerical processor. Each *key* entry is a fixed number representation of the system's current state, and the corresponding *value* is the symbolic representation of that state. The system's current operating mode is, therefore, determined by matching the received binary string value with the *keys* in the Table. The operating mode is found if there exists a perfect match. For no match, 'Mode Unknown' is returned.

The Control module thus realizes the control paradigm, and together with the Flavor System, they form the Inference Engine of the symbolic control system. Rule Matching is implemented inside the Control module. The actual rules are defined within the various methods assigned to the flavors. Metaknowledge [18] concepts can be applied to shorten the search paths to determine the succeeding mode (sequence of module operations) for the numerical system. With meta-rule implementation, the Inference Engine will not exhaustively search through irrelevant rule sets, because they will be effectively excluded from the search list. It is evident at this point that the Symbolic Controller possesses the basic structural elements of an Expert System, namely, the Inference Engine, Knowledge Base, and User Interactive. Moreoever, the architecture can be characterized as a Flavor-based Expert Controller because the bottom three functional modules, including the Hash Tables, of Figure 2. are developed mostly inside the Flavor System programming environment.

COMMUNICATIONS: Two communication mechanisms are employed by the symbolic control subsystem. They are inter-process and intra-process communications. Inter-process communication is accomplished through message passing channels established for the Controller and the numerical processor. Two virtual mailbox devices are created to accommodate the memory volume anticipated during message exchanges. VAX/VMS System Queue Input/Output (QIO) functions implements the I/O routines for the inter-process communication. The specific input function, called "set write-attention asynchronous system trap (SWAAST)," allows concurrent processes to individually queue 'read-message' requests without halting process executions, if a new message is currently unavailable. The process execution will be suspended, however, when the expecting message becomes available, and immediately after SWAAST has retrieved the message, the process resumes running. Using these VMS system services allows simulation of multi-processor architectures in a multi-user VAX environment.

Inter-process communication is simulated as follows. First, a Command/Status (CS) word data structure is designed, which transports system information such as module identifier, message type, and status block between the two processors. CS-word is implemented as a binary string data type. The current CS-word construct only incorporates the status block. Binary strings, representing the running numerical processor's current states are sent to Controller via the mailbox utility. The binary strings are then converted to symbolic equivalents inside a Hash Table where the *key* is the received binary value, which maps to its symbolic equivalent, an operational mode. Then, the inference engine, mainly the control module, does a sequence of rule-matchings on the received messages, to determine the present and succeeding states of the entire numerical processor. Then, Controller sends instructions back to the numerical processor, via another mailbox utility, for the next period (mode) of operation.

As a specific example, in the Radio Processor a five-digit binary word represents each state, such as 'STANDBY,' 'STARTUP,' 'FILTER,' 'RUN,' 'POWERUP,' 'POWERDOWN,' and 'EXIT,' respectively. The most significant bits of the word identify the module sending the message. Thus, upon receiving '00001,' Controller finds, from the Hash Table, that the Automatic Gain Control Module is in STARTUP State. When all modules have signalled 'STARTUP,' the Controller declares 'STARTUP,' and passes instructions back to the modules to move the receiver into the next operational mode, which is 'RESET.'

Internal to Controller, communication between objects (individual flavors) is realized by sharing instance variables. This results as flavors are combined so that shareable instance variables are properly included in the methods of individual flavors within the communication network. The network is currently a hierarchical 'Flavor Tree.' The flavor at the top of the hierarchy inherits all the instance variables below it. The flavors at the bottom of the Flavor Tree are considered basic, and are never instantiated alone. This Flavor Tree structure can be recognized as an inverted family tree.

For example, the current Flavor Tree for the Controller consists of four individual flavors, each of which represents a functional module, internal to the Control module of Figure 2. . From the top down, these are 'MODLMGR,' 'MODESEQ,' 'MODEMGR,' and 'MODEID.' For each flavor (module), methods (functions) and instance variables (attributes) are defined, with a predefined message, *:msg*, common to all methods in the tree. This allows combining methods in the Flavor Tree in a way that the methods executed from the bottom flavor of the Tree to the top. This means that when the instances corresponding to the numerical processor's states, defined as 'cs_word,' are input to Controller and passed via *:msg*, its modules execute sequentially, to identify the mode, manage the mode, sequence the (numerical processor) modules, and manage the (numerical processor) modules.

CONCLUSION

The indirect method of control, detailed above, formalizes and exploits a discipline which occurs naturally in the design of complex numerical systems. That discipline is 'moding,' wherein design of control for a complex system focuses on its natural modes of operation. The formal relating of the system's local states to its global modes then yields the symbolic method of control.

Several interesting issues have been sharply defined during pursuit of this work. First, continuing design of the Symbolic Controller should support a broad range of requirements from the numerical application system. That is, the design approach should be to develop a general purpose Controller Shell, to which, a multi-tasking application system can be attached. A necessary feature of such a Controller Shell is its ability to reconfigure its internal 'wiring' of control paths to meet specific needs of the attached multitasking processor.

An associated concern about the general purpose Controller Shell architecture relates to the concurrency of the controller. Since the multiple tasks of the numerical application system are requesting control services simultaneously with the Symbolic Controller, it must be able to allocate proper execution quantum to individual tasks. This leads to the idea of developing a concurrent processing environment for the controller should such ability be lacking in the programming language [19].

At the present stage of development, true 'intelligence,' corresponding to Saridis' Organization Level [4], has yet to be implemented. Such implementation requires augmentation of the Controller architecture, which has been done, plus design and coding of the corresponding symbolic algorithms, which remains to be done. This will require incorporating uncertainty management capability in the Controller Shell, by which Symbolic Control can operate under conditions where insufficient or fuzzy information are encountered [20]. This last issue defines the next step in design of the symbolic controller, which is one of the core issues in the arena of so-called Intelligent Control.

In summary, the next stage of system implementation will focus on developing Controller's organization level where intelligent reasoning is emphasized. Features supported should include autonomous reconfiguration of control path, total system concurrency, and intelligent inference ability. Work continues to accomplish all these goals.

REFERENCES

[1] K.S. Fu, "Learning Control Systems and Intelligent Control System: An Intersection of Artificial Intelligence and Automatic Control," IEEE Trans. on Automatic Control, vol. AC-16, no. 1, pp. 70-72, 1971.

[2] Howard Anderson, "Why Artificial Intelligence Isn't (Yet)," AI EXPERT, vol. 2, no. 7, pp. 36-44, July, 1987.

[3] R.H. Anderson and R.B. Greenberg, "UNIX and AI, A Beautiful Marriage," UNIX WORLD, pp. 26-33, August, 1986.

[4] G.N. Saridis, "Knowledge Implementation: Structures of Intelligent Control Systems," IEEE International Symposium on Intelligent Control, Philadelphia, Penn., January, 1987.

[5] C.A.R. Hoare, "Communicating Sequential Processes," Communications of the ACM, vol. 21, no. 8, pp. 666-677, August, 1978.

[6] Carl Hewitt, "Concurrency in Intelligent Systems," AI EXPERT, Premier Issue, pp. 44-50, 1986.

[7] W.A. Mason, "Distributed Processing: The State of the Art," BYTE, pp. 291-297, November, 1987.

[8] Lucid Common Lisp : User's Guide for the VAX, Lucid, Inc., Menlo Park, CA., 1986.

[9] Guide to Programming on VAX/VMS, Digital Equipment Corp., Maynard, MA., 1986.

[10] VAX/VMS System Services Reference Manual, DEC, op.cit., 1986.

[11] VAX/VMS I/O User's Reference Manual, DEC, op. cit., 1986.

[12] P.H. Enslow Jr., "Multiprocessor Organization: A Survey," Computing Surveys, vol. 9, no. 1, pp. 103-129, March 1977.

[13] P.B. Hansen, "Distributed Process: A Concurrent Processing Concept," Communication of the ACM, vol. 21, no. 11, pp. 934-941, 1978.

[14] J.A. Stankovic, "Software Communication Mechanisms: Procedure Calls Versus Messages," Computer, vol. 15, no. 4, pp. 19-25, 1982.

[15] S.M. Shatz, "Communication Mechanisms of Programming Distributed Systems," Computer, vol. 17, no. 6, pp. 21-27, 1984.

[16] M. Stefik and D.G. Bobrow, "Object-Oriented Programming: Themes and Variations," The AI Magazine, vol. 6, no. 4, pp. 40-62, 1986.

[17] G.L. Steele, Jr., Common Lisp, The Language, Digital Press, Digital Equipment Corp., Billerica, MA., pp. 282-285, 1984.

[18] R. Davis and B.G. Buchanan, "Meta-Level Knowldege: Overview and Applications," The 5th IJCAI, Cambridge, MA., August, 1977.

[19] A.P. Bernat, "Multitasking for Common LISP," AI EXPERT, Premier Issue, pp. 68-79, 1986.

[20] L.A. Zadeh, "The Role of Fuzzy Logic in The Management of Uncertainty in Expert Systems," Fuzzy Sets and Systems, vol. 11, no. 7, pp. 199-227, 1983.