View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by NASA Technical Reports Server

18-516 : 514-63
88
VE 909080

Execution Environment for Intelligent Real-Time Control Systems

J. Sztipanovits
Vanderbilt University
Nashville, TN 37235

1. ABSTRACT

Modern telerobot control technology requires the integration of symbolic and non-symbolic programming techniques, different models of parallel computations, and various programming paradigms. This paper describes the Multigraph Architecture, which has been developed for the implementation of intelligent real-time control systems. The layered architecture includes specific computational models, integrated execution environment and various high-level tools. A special feature of the architecture is the tight coupling between the symbolic and non-symbolic computations. It supports not only a data interface, but also the integration of the control structures in a parallel computing environment.

2. INTRODUCTION

There is an ever-increasing demand for improving the information processing capabilities of robot controllers, measurement systems, or process control systems. The ultimate goal is to enhance system autonomy, adaptivity and functional performance. Since some of these features were previously provided by human operators, systems exhibiting these properties are often qualified to be "intelligent."

Essential extension of capabilities is always based on the application of new techniques. Current trends in intelligent systems include: (1) the use of parallel/distributed computing architectures, (2) the use of parallel/distributed programming models, and (3) the application of various artificial intelligence (AI) programming techniques. Since the new techniques typically are not substitutes but extensions of the conventional techniques, integration has become a key factor in building intelligent systems. Symbolic and numerical computations, various programming paradigms and different parallel computing models have to be merged in the frame of a possibly unified architecture.

The critical system component, where most of the implementational problems of system integration have to be solved, is the execution environment. The execution environment provides run-time support for the architecture, and couples various programming models to each other and to the underlying hardware system.

This paper describes an experimental architecture - the Multigraph Architecture - and the corresponding execution environment developed for building integrated systems. After the summary of the background of this research, the design considerations are outlined. Then, the main components of the Multigraph Architecture are discussed, which is followed by the summary of various applications and future plans.

3. BACKGROUND

A. Intelligent Systems

There is a rapidly growing research interest in the application of AI techniques in robot controllers, measurement systems and process control systems. General, architectural issues of intelligent systems are analyzed in [1]. The generic architecture of intelligent systems is characterized by the introduction of the "knowledge-level," which includes "knowledge-intensive" system components providing high-level perception, modelling and planning functionalities.

The structure and operation of the knowledge-level system components are typically model-driven. New possibilities offered by knowledge-based, model-driven automation in telerobotics are described in [2]. Architectural issues of model-driven instrumentation are discussed in [3] and the application of new techniques in the Knowledge-based Experiment Builder of a magnetic resonance imaging system is discussed in [4]. The prospective of model-driven, knowledge-based systems in controllers is outlined in [5].

A common view regarding the structure of intelligent systems operating in real-time environment is that they must have layered architecture where "high-level," knowledge-based system components synthesize, monitor and control the operation of the "low-level" sensory and processing activities.

B. Graph models of computations

An important class of parallel computational models are the graph models. The computational graphs (or control graphs) are directed graphs, where nodes represent units of computations and arcs represent dependency relationships. The general properties of graph models are analyzed in [6], and their classification is given in [7]. In dataflow models, arcs arise from data dependences, and data are passed along the arcs in execution time. In control-flow models, not the data, but pointers to the data are carried; therefore, this model requires the availability of shared memory. The computational units can be scheduled data-driven or demand-driven. Data-driven scheduling means that a unit is executable if the necessary input data are available. In demand-driven scheduling, only those nodes which are necessary to provide the requested data are activated.

Graph models have essential advantages in the context of intelligent real-time systems.

- Graph models can uniformly describe parallel computations for different multiprocessor architectures, such as distributed and shared memory systems.

- The granularity of the model can be "tuned" by selecting the size of the computational units.

- The "imperative" parts of the computation (i.e. the code of the computational units) are naturally separated from its logic structure represented by the control graph. This separation makes it possible to dynamically modify the computational structure.

- The control graph can be easily represented in declarative form. The declarative representation is the key for using symbolic processing techniques to synthesize various computational structures, such as real-time signal processing systems [4].

4. DESIGN CONSIDERATIONS

The Multigraph Architecture (MA) provides software framework for building intelligent systems in real-time, parallel computing environment. The main layers of the architecture are: the (1) **Physical layer**, (2) **System layer**, (3) **Module layer** and (4) **Knowledge base layer**. The basic properties of the individual layers are summarized below.

A. Physical layer

Computational heterogeneity, various physical constraints (such as distance between computing nodes), and the typically high computation load require the support of different multiple-processor configurations: tightly-coupled architectures with shared memory, loosely-coupled computer networks, and their combination. Special hardware components such as array processors or i/o devices might also belong to the hardware configurations.

B. System layer

The primary function of the system layer is to provide access mechanisms to the hardware resources. In the current implementations of MA, the system layers are off-the-shelf operating systems, which facilitate services such as standard i/o, task management, intertask (interprocessor) communication and synchronization and real-time clock. An important requirement for the higher-levels is flexibility to ensure the portability to different operating systems.

C. Module layer

One of the most critical requirements for MA is to support the synthesis and dynamic modification of various low-level computational structures (signal processing systems, control systems, etc.) in parallel computing environment. The key idea in the solution is the introduction of the module layer, which serves as an interface between the knowledge base layer and the system layer. The module layer has a special graph-oriented computational model, the **Multigraph Computational Model (MCM)**, which provides the following possibilities:

- high-level (possibly very high-level) declarative languages can be defined on the knowledge-based layer to represent various computational structures such as procedural networks, constraint networks, reasoning networks etc.;

- these declarative forms can be interpreted and mapped into a computation graph on the module layer;

- the run-time support of MCM can schedule the elementary computations and "pass" them to the system layer for execution, taking advantage of the available parallelism of the computational structures;

- appropriate interpretation techniques can ensure the dynamic modification of the computation graph.

The name "Module layer" suggests the view that this layer is a "module library" consisting of typically small program modules written in C, Fortran, LISP etc. These modules are structured to form a complete program by the definition of the computation graph.

D. Knowledge base layer

High-level, symbolic computations are implemented on the knowledge base layer. Although the actual structure of the knowledge-based system components are strongly application dependent, the parallel computing environment and the features of the underlying module layer make the elaboration of a generic programming model desirable. The main purposes of the high-level programming model are:

- to support the structurization of the knowledge-based operations into concurrent activities,
- to facilitate a standardized, high-level communication system among the activities, and
- to provide interface to the module layer and MCM.

A strict requirement is that these services have to be implemented as extensions to one of the standard LISP systems in order to preserve the compatibility with different AI toolsets.

5. OVERVIEW OF THE MULTIGRAPH ARCHITECTURE

The basic computing models used on the different layers of MA and their relationships are represented in Figure 1.

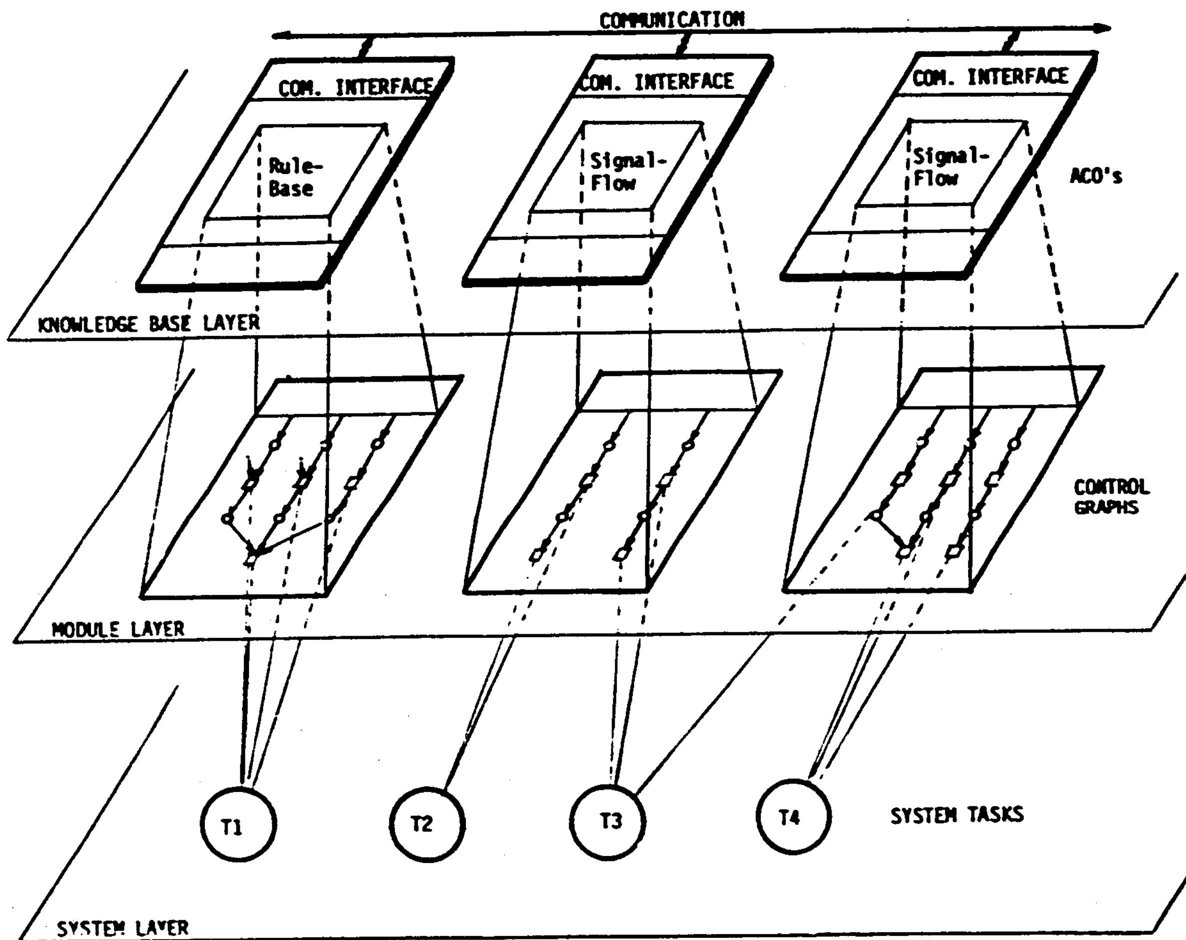

FIGURE 1. Layers of the Multigraph Architecture

A. Autonomous Communicating Objects (ACO)

The basic system structurization principle on the knowledge base layer is provided by the concept of **Autonomous Communicating Objects.** ACO is a straightforward extension of the "object" concept of object-oriented languages such as Flavor [8] in the following sense:

- ACO's are fully autonomous systems that can run virtually or physically parallel,
- ACO's can be dynamically allocated and can compete for the same resources,
- they communicate with each other by means of a fully asynchronous communication protocol.

The main purpose of ACO's is to provide a standardized "object shell" around a variety of heterogeneous knowledge-based system components. The communication "methods" are standard elements of the object shell, and hide the details of an actual implementation from the application programmers.

Various object types have been developed for supporting specific applications. These objects typically include a "knowledge base," which is represented by a special representation language. Some of these object types, such as **Procedural Network Object** (PNO) and **Rule Network Object** (RNO) are described in [9].

B. Multigraph Computational Model (MCM)

The different object-types are facilitated with an appropriate interpreter or incremental compiler, which maps the actual knowledge base into a computation graph on the module layer [9]. While ACO's serve as a symbolic representation and interface to possibly complex functional components of the system (e.g., a signal processing system, rule-based system, associative database system, etc.), the computation graph on the module layer constitutes their actual execution environment. This relationship between the ACO's and their execution environment has the following advantages:

- Execution of the operations represented by ACO's occurs in a parallel execution environment offered by MCM.
- The interpreter (or incremental compiler) "methods" of ACO's, which build the computation graph, can dynamically modify the graph, as a response to an external message, or to a feedback from the execution environment (a mechanism for implementing "self- modifying" signal processing systems is described in [4]).
- The system fully integrates two different parallel computing models. ACO's form the "macro-structure" of the system, and they communicate by using the services of loosely coupled distributed systems (typically message passing). The computation graphs provide the "micro-structure" of the system components. MCM efficiently supports medium-level (subroutine size) computational granularity, and can take advantage of tightly-coupled multiprocessor architectures with shared memory.

C. System tasks

The computational model on the system layer is provided by the actual operating system. The key concept is the "system task," which represents a "slice" from the processing capacity, and can access to various resources. The elementary computation units that are scheduled by the run-time support of MCM are executed by the system tasks.

6. MULTIGRAPH COMPUTATIONAL MODEL

MCM can be characterized as a **control-flow model**. The control structures of computations are represented by bipartite graphs that are built of **actornodes, datanodes** and **connection** specifications (see Figure 2).

The actornodes are associated with the elementary computational units, called **scripts**, which can be written either in LISP or in any other language, such as C, Fortran, Pascal, etc. The scripts do not know about their position in the control graph: they communicate with other graph components through the input/output ports of the actornodes. The actornodes are associated with a local datastructure, called **context**, which can be accessed by the script. If the code of the script is reentrant, it can be attached to several actornodes. In different computation problems the scripts may be quite different: a script may be an interrupt-driven i/o handler, a transformation of the input data arriving to the actornode, or an interpreter module, which interprets the symbolic form stored in the context of the actornode.

Datanodes store and pass the data generated by actornodes. They can be either **streams** with multiple output ports, or **scalars**. The streams maintain the partial sequence order of the data generated during the computations, which preserves the overall consistency.

The control graph can be operated in **data-driven** or in **demand-driven mode**, or in a combination of the two modes. In data-driven mode, the data sent to a datanode propagate a "control token" to the connected actornodes. The actornodes will fire according to the specified control discipline: in ifall mode, at least one

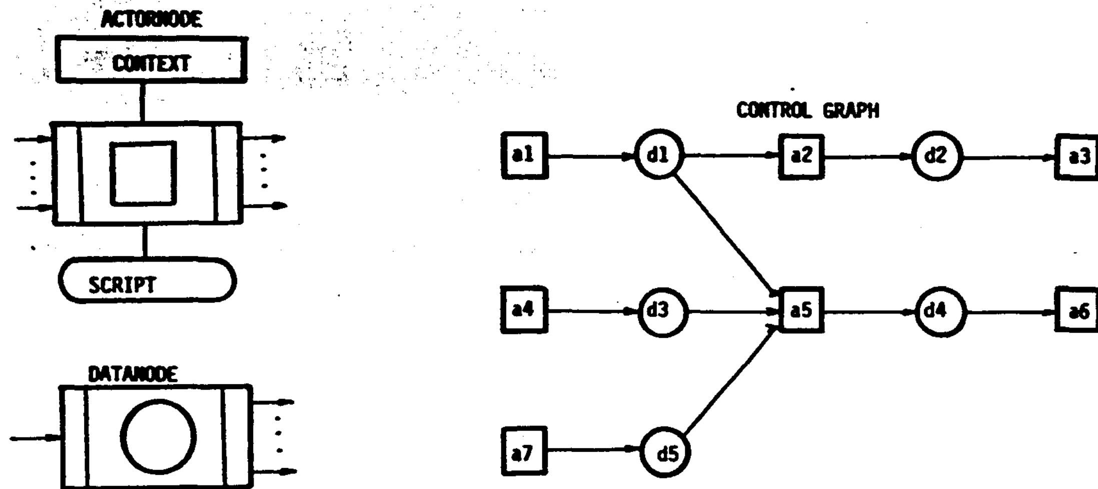

FIGURE 2. Components of the Multigraph Computation Model

control token must be sent to all of the inputs of the actornode: in ifany mode, every received control token will cause firing. In demand-driven mode, the request for data sent to a datanode will generate a control token if the datanode is "empty." This control token will fire all of the actornodes that are potentially able to provide the requested data. The demand propagates backward along the control graph until data is generated. From this point a forward propagation starts, which finally provides the requested data. A more detailed description of the computational model can be found in [10, 11].

The run-time support for the MCM is provided by the **Multigraph Kernel** (MK). The structure of the MK is shown in Figure 3. The control graph is represented in the **descriptors**, which are manipulated by various kernel functions. The **Control Interface** functions are used for dynamically building and modifying the control graph. These functions are imbedded in a LISP system, where the various graph-builder interpreters and incremental compilers are implemented. The **Module Interface** includes the data/demand propagation kernel calls for the scripts. An important feature of the system is that actornodes with scripts written in different languages can be mixed in the same control graph. (The necessary transfer routines are invisible to the user.) This feature is used for creating tight coupling between symbolic and non-symbolic computations. Tight coupling means that not only data structures can be passed between the two kinds of computations, but there may be a fully integrated control structure.

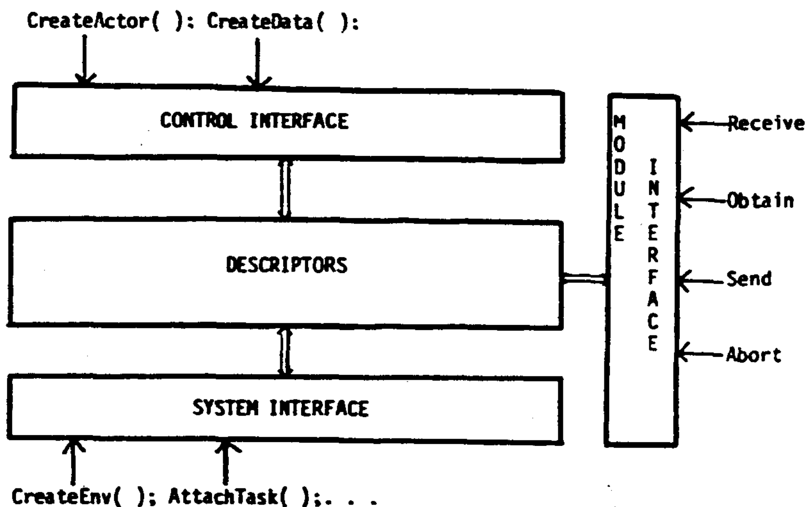

FIGURE 3. Structure of the Multigraph Kernel

The most complex part of MK is the System Interface. The System Interface schedules the elementary computations that are defined during the data/demand propagation. The computational units (the scripts of the fired actornodes) are passed to the available system tasks for execution. The environment mechanism of the System Interface ensures that subsets of the control graph can be dynamically associated with one or more system tasks that include the necessary resources for executing the scripts. This mechanism provides a very straightforward way for dynamic resource management in multiprocessor configurations.

Two important implementation issues are the granularity and the memory model. The lower limit for the reasonable computational granularity is basically determined by the overhead of MK. Since MK currently is implemented in software, the overhead is introduced by the control token propagation functions. On the 68000 processor-based IBM 9000 system (clock frequency is 8Mhz), the overhead is about 800 microseconds: on the VAX 785 implementation is less then 200 microseconds. Due to the construction of the MK, the overhead is basically independent from the size of the control graph.

Since MCM is a control-graph model where the pointers to data structures rather than the data are passed along the graph, the model requires the presence of shared memory for those tasks (and processors) that are assigned to the same subgraph as execution resource (see Figure 4). In order to provide flexibility toward architectures which do not support shared memory access for the processors (hypercube architectures or distributed computer configurations), a simple mechanism is implemented to link control graphs that are allocated in the local memory of the separate nodes. The scripts of receiver and transmitter actornodes provide a logical link between the separated subgraphs and implement the data transfer by using the actual services of the underlying system layer (e.g., the message passing services of DECNET in the VMS/DECNET implementation). At these links, the control-graph model is "transformed" to dataflow model, since the actual datastructures - and not just pointers - are passed to the "remote" nodes.

This method makes it possible to generate large processing networks from their symbolic representation in distributed computing environment [12].

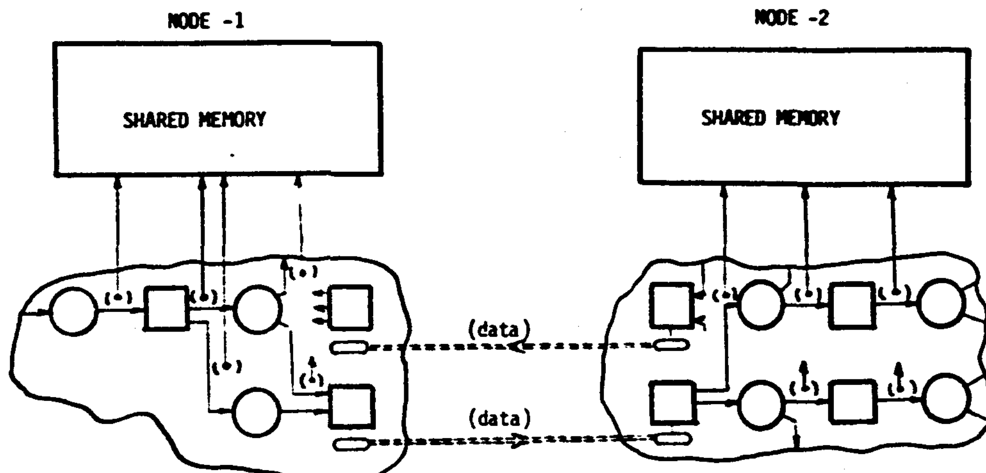

FIGURE 4. Memory Model of MCM

7. STRUCTURE OF THE EXECUTION ENVIRONMENT

The simplified structure of the execution environment supporting MA can be seen in Figure 5. MK is implemented as an additional layer to a standard operating system. Depending on the computer architecture and on the features of the particular operating system, MK may exist in one or more copies. The System Interface of MK is a well structured, modular program which makes porting the kernel relatively easy, even to devastatingly different operating systems and real-time supervisors.

The Module Interface functions of MK can be invoked from LISP as well as from other languages. This ensures that scripts can be written in different languages, and existing module libraries can be easily interfaced to MK. The Control Interface of MK is imbedded in LISP since currently we use LISP as implementation language of the knowledge base layer.

Various high-level software components such as the generic object shell for ACO's and the standard methods of different ACO types are implemented in LISP.

For distributed computer configurations, the LISP system (in the first implementation FRANZ LISP, recently changed to Common Lisp) has been expanded with the Communicating LISP System facility, which provides task management and asynchronous message passing primitives [13].

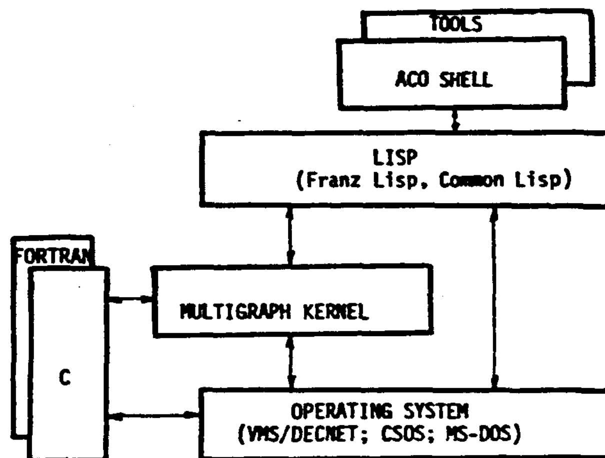

FIGURE 5. Structure of the Execution Environment

8. EXPERIENCES

MA has been implemented on very different computer configurations and has been used for various applications.

An Intelligent Test Integration System (ITIS) has been implemented in the Space Station Laboratory of the Boeing Aerospace Company in Huntsville, AL [12]. The purpose of ITIS is to support the automatic generation of test systems in real-time, distributed computing environment. ITIS is implemented as a knowledge base layer above the conventional test system components, and can build complex test configurations from the symbolic specification of test scenarios. The computing environment is a VAX network with the VMS/DECNET operating system expanded with special hardware units.

A different application of the layered MA architecture is the Knowledge-based Experiment Builder (KEB), which was developed for the M.I.T/IBM experimental MRI (Magnetic Resonance Imaging) system [4]. The core of the KEB is a high-level representation language for signal processing schemes and a smart interpreter, which can generate the appropriate version of the real-time signal processing system, and which is able to reconfigurate it for specific events. The computing environment of this system is the IBM 9000 computer with the CSOS real-time operating system.

Various computational structures have been developed and are being investigated for the MCM, such as a hierarchical planner [14], knowledge-based simulation builder [15], pattern-driven inference system [16], etc.

9. CONCLUSIONS AND FUTURE PLANS

The integration of symbolic and conventional programming techniques, parallel computing models of different granularity, and various programming paradigms are essential conditions for the successful implementation of intelligent real-time systems. The Multigraph Architecture has proven to be a good approach to solve the problems of integration. It provides a generic framework, programming models for structurizing software components and various tools for the actual implementation.

We have practical experiences with implementing systems in single-processor (IBM-AT/MS-DOS), single-processor multitasking (VAX/VMS and IBM 9000/CSOS) and distributed (VAX network VMS/DECNET) computing environments. As a next step, we intend to implement the execution environment for tightly-coupled multiprocessor configuration.

The system offers a convenient method to describe and implement "self-modifying" signal processing systems. Further research is needed to utilize this capability in the design of structurally adaptive measurement control systems.

10. ACKNOWLEDGEMENTS

The research described in this paper was supported in part by Boeing Aerospace Company, IBM Corporation and Vanderbilt University. The author especially wishes to acknowledge the contributions of Csaba Biegl and Gabor Karsai to the design and implementation of the execution environment, and Byron R. Purves of Boeing Aerospace Company and Colin G. Harrison of IBM to the design of the application systems.

11. REFERENCES

[1] A. Newell, "The Knowledge Level," Artificial Intelligence, Vol. 18, 1982, pp. 87-127.

[2] S. Lee, G. Bekey, A.K. Bejczy, "Computer Control of Space-Borne Teleoperators with Sensory Feedback," Proc. of the IEEE International Conference on Robotics and Automation, St. Louis, MO, 1985, pp. 205-214.

[3] J. Sztipanovits, "Knowledge-Based Approach in Measurement and Instrumentation," Proc. 3rd International Conference on Measurement in Clinical Medicine, Edinburgh, Scotland 1986, pp. 29-33.

[4] J. Sztipanovits, C. Biegl, G. Karsai, J. Bourne, C. Harrison, R. Mushlin, "Knowledge-Based Experiment Builder for Magnetic Resonance Imaging Systems," Proc. of the 3rd IEEE Conference on Artificial Intelligence Applications, Orlando FL., 1987 (in press).

[5] K.J. Astrom, "Auto-Tuning Adaptation and Expert Control," Proc. American Control Conference, Boston MA, 1985, pp. 1514-1519.

[6] J.C. Browne, "Formulation and Programming of Parallel Computations: A Unified Approach," COMPCON, Spring 1985, pp. 624-631.

[7] D.D. Galski and Jih-Kwon Peir, "Essential Issues in Multiprocessor Systems," IEEE Computer, June 1985, pp. 9-27.

[8] D. Moon, R. Stallman, D. Weinreb, "LISP Machine Manual," The MIT AI Lab., Cambridge MA, 1984.

[9] J. Sztipanovits, R. Purves, G. Karsai, C. Biegl, S. Padalkar, R. Williams, T. Christiansen, "Programming Model for Coupled Intelligent Systems in Distributed Execution Environment," Proc. of the SPIE's Cambridge Symposium on Advances in Intelligent Robotics Systems, Cambridge, MA, 1986 (in press).

[10] J. Sztipanovits, "MULTIGRAPH: Parallel Architecture for Intelligent Systems," Dept. of Electrical Engineering, Vanderbilt University, Techn. Report #86-01, 1986.

[11] C. Biegl, "Multigraph Kernel User's Manual," Dept. of Electrical Engineering, Vanderbilt University, 1985.

[12] J. Sztipanovits, B. Purves, S. Padalkar, J. Rodriguez, K. Kawamura, R. Williams, H. Biglari, "Intelligent Test Integration System," Proc. of the Conference on Artificial Intelligence for Space Applications, Huntsville, AL, 1986, pp. 177-185.

[13] S. Padalkar, "Communicating LISP System Facility," M. Sc. Theses, Dept. of Electrical Engineering, Vanderbilt University, 1987.

[14] G. Karsai, "Hierarchical Planning with Objects," Proc. of the 19th Southeastern Symposium on Systems Theory, Clemson, SC, 1987 (in press).

[15] C. Biegl, "Knowledge-based Generation of Simulation Models," Proc. of the 19th Southeastern Symposium on Systems Theory, Clemson, SC, 1987 (in press).

[16] C. Biegl, "Florence (Dataflow Oriented Inference Engine) User's Manual," Dept. of Electrical Engineering, Vanderbilt University, 1986.