CONF-9007134 - -

Proceedings of the Workshop on Software Tools for Distributed Intelligent Control Systems

The Lighthouse Hotel
Pacifica, California
July 17-19, 1990

Editor: Charles J. Herget

September 1990

Lawrence Livermore National Laboratory

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Workshop on Software Tools for Distributed Intelligent Control Systems

Lighthouse Hotel
Pacifica, California

July 17-19, 1990

CONF-9007134--

DE91 001755

Co-Chairs:	Dr. Charles J. Herget Lawrence Livermore National Laboratory
	Col. John R. James U.S. Army
Administrator:	Carol Richardson Lawrence Livermore National Laboratory
Program Committee:	Dr. Charles J. Herget Lawrence Livermore National Laboratory
	Dr. David Hislop U.S. Army Research Office
	Col. John R. James U.S. Army HQ TRADOC
	LTC Erik Mettala DARPA
	Dr. Abraham Waksman U.S. Air Force Office of Scientific Research
Meeting Facilitators:	Dr. Margaret Barbee, Principal Facilitator Lawrence Livermore National Laboratory
	Marianne Clark Lawrence Livermore National Laboratory
	Linda Donald Lawrence Livermore National Laboratory
	Helen Holmes Lawrence Livermore National Laboratory
	Ronald Weinberg Lawrence Livermore National Laboratory

CONTENTS

EXECUTIVE SUMMARY

INTRODUCTION

The Workshop on Software Tools for Distributed Intelligent Control Systems was organized by Lawrence Livermore National Laboratory for the United States Army Headquarters Training and Doctrine Command and the Defense Advanced Research Projects Agency. It was held at the Lighthouse Hotel in Pacifica, California, on July 17 to 19, 1990.

DARPA has recently expressed interest in architectural based software development methodologies. DARPA is now putting into effect a program for the development of common architectures and models of computation for particular applications to reduce the rapdily increasing cost of the life cycle of software. One of the more important areas of domain specific software architecture is that of vehicle management systems. The intention is to build a software engineering environment for intelligent control systems for military vehicles which would improve the productivity of control design engineers and lower the software costs to DOD.

The goals of the workshop were to (1) identify the current state of the art in tools which support control systems engineering design and implementation, (2) identify research issues associated with writing software tools which would provide a design environment to assist engineers in multidisciplinary control design and implementation, (3) formulate a potential investment strategy to resolve the research issues and develop public domain code which can form the core of more powerful engineering design tools, and (4) recommend test cases to focus the software development process and test associated performance metrics.

Recognizing that the development of software tools for distributed intelligent control systems will require a multidisciplinary effort, experts in systems engineering, control systems engineering, and computer science were invited to participate in the workshop. In particular, experts who could address the following topics were selected: operating systems, engineering data representation and manipulation, emerging standards for manufacturing data, mathematical foundations, coupling of symbolic and numerical computation, user interface, system identification, system representation at different levels of abstraction, system specification, system design, verification and validation, automatic code generation, and integration of modular, reusable code.

There were 48 attendees from industry, government, and academia. A complete list of attendees is contained in an appendix.

During the first two days of the Workshop, presentations were made by all of the attendees in an attempt to establish the state-of-the-art in distributed intelligent control systems.

On the morning of the third day, the attendees were divided into five working groups, each group having representatives from the three technologies: systems engineering, control systems, and computer science.

Each of the working groups was given the tasks: (1) develop a short term plan, (2) recommend a research plan, and (3) identify potential test beds for implementing software tools for distributed intelligent control systems.

On the afternoon of the third day, the entire group recovened to form a consensus of the five working groups. A meeting facilitator, Dr. Margaret Barbee, from LLNL coordinated the daily activities of the Workshop. Additional facilitators from LLNL were available to assist each of the Working Groups on the morning of the third day.

This report contains a summary of the findings of the participants of the Workshop. It also contains a collection of papers submitted by the participants.

SUMMARY OF FINDINGS

Short Term

1. Perform a state of the art review of distributed intelligent control systems.

   It was felt that the status of the technologies which are required for distributed intelligent control were adquately presented at the Workshop; however because of the diverse nature of the individual technologies, it was not possible to form a comprehensive review in a three day workshop. In general, there was agreement that distributed intelligent control, as an emerging discipline, is currently ill-defined. The usefulness of testbeds to focus issues and demonstrate the technology was emphasized by each group. The NASA/NBS Standard Reference Model for Telerobotic Control System Architecture (NASREM) was the only software architecture model offered as a central system view around which software tools can be built. At least one group concluded that there is low technical risk in applying Discrete Event Dynamic System (DEDS) mathematics to the NASREM model to achieve an initial set of software tools within two years.

2. Develop a taxonomy of currently available tools.

   It was felt that there are numerous tools which are available to perform various aspects of distributed intelligent control. It was recommended that a taxonomy of them be made. Areas where applicable tools were identified as being available included artificial intelligence (AI), data base (DB) management, computer-aided control systems design (CACSD), computer-aided software engineering (CASE), computer-aided design (CAD), and simulation.

3. Develop a tool which will perform a high level integration of existing tools.

   It was felt that the cost of developing all new tools would be prohibitively expensive. Furthermore, it was felt that the taxonomy would identify a large number of available tools. What is missing is a high level tool which some preferred to call Computer Aided Software Systems Engineering (CASSE).

4. Develop a technology transfer plan.

5. Establish a repository for the software developed.

<u>Long Term</u>

The following research areas were identified.

1. Theory of distributed intelligent control systems. It was felt a theory of distributed intelligent control systems is not available and should be established on its own.

2. Theory of other than linear control system, e.g knowledge based control systems.

3. Stability of knowledge based control systems

4. Theory on multi-agent interactions.

5. Related computer science topics:

   real time programming, e.g. languages to specify tasks which incorporate notions of time, function and events at each level;

   software re-use, e.g. the indexing problem to store and index;

   automatic programming, e.g. develpment of a design specification language for distributed intelligent control systems;

   real time control, e.g. imprecise computation, time-limited computation, concurrency, and data communications; and

   real time operating systems, e.g. scheduling.

Recommended Test Beds

Potential test beds which were identified included the Advanced Field Artillery System (AFAS), unmanned air vehicles (UAV), unmanned underwater vehicles (UUV), and the Block 3 tank.

It was recommended that there should be a laboratory test bed as well as a field test bed; that the problem should be very focused, i.e. select a specific application from the given test bed, not the entire test bed; and there should be at least two efforts, each focusing on a different test bed.

General comments

There was not unanimous agreement among all of the participants on what the target of the software should be; however, those representing the control systems community generally favored the discrete event dynamic system model (DEDS).

There was general agreement that the development of a comprehensive package from scratch would require hundreds of millions of dollars. It was strongly recommended that a more modest effort be undertaken as outlined in the short term goals. It was felt that the short term goals could be achieved within five years with a budget of tens of millions of dollars. After feasibility is demonstrated, it was felt that larger funding would become available. Several corporate attendees indicated that matching funds would be made available by corporations if general agreement could be reached on a standard architecture. This would leverage government investment in software tools.

It was recommended that the long term research be carried out simultaneously but separately from the short term program, with the short term program providing inspiration and applications.

Toward a Reference Model Architecture for Real-Time Intelligent Control Systems (ARTICS)

by

James S. Albus and Richard Quintero
Robot Systems Division
National Institute of Standards and Technology

INTRODUCTION

This position paper is a condensed version of a paper titled "Concept for a Reference Model Architecture for Real-Time Intelligent Control systems (ARTICS)", [Al 90], which advocates the development of a reference model open-system architecture as a means to accelerate the pace of technological development in automation and robotics. We believe many of the major bottlenecks in the development of intelligent machine systems could be alleviated, if not eliminated, by the development of a set of ARTICS guidelines.

The pace of commercial and military technological advancement in the fields of robotics, intelligent machine systems and automation is falling short of expectations. Problem complexity is one of the major contributors to this problem. Intelligent robot systems projects typically require bringing together teams of technologists with a broad mix of engineering disciplines and a high level of expertise. Robotics and automation manufacturers must make large investments in both developing custom test-beds and in recruiting and training competent engineering teams in order to compete in this market area. A second problem is the lack of a widely accepted theory, or system architecture model that ties together the many disciplines involved in intelligent robot systems. This limits the dissemination of intelligent machine systems technology developed in different parts of the robotics community. This prevents new projects from building upon the foundations laid by previous efforts.

A set of ARTICS guidelines would reduce the impact of problem complexity and would provide an efficient means of transferring technology between projects. Manufacturers will adopt ARTICS guidelines if they believe that their potential profits would be enhanced by an expanded market. This must be driven by traditional market forces (user demand). We need a way to create automation building blocks so that more complex systems can be developed without making the technologies more difficult to understand and to apply and without "reinventing the wheel" each time a new project begins. We believe that a common hardware/software shell structure would facilitate the incremental improvement which would produce rapid advancement in automation and robotics technology.

To summarize the goals advocated by this paper, we suggest that an Architecture for Real-Time Intelligent Control Systems (ARTICS) is needed to:

* reduce the impact of problem complexity in the development of robotic applications

* expand the market for intelligent control system components through open-system interface guidelines and protocols.

* promote portability, inter-operability and modularity of intelligent control system software and hardware

* facilitate technology transfer between intelligent control system projects

* reduce the time, cost, risk, and initial investment required in bringing new, world class, intelligent machine systems and control system products into the market place

2. ARTICS VISION

ARTICS guidelines would specify a reference model infrastructure of hardware components, software components, interfaces, communications protocols, and application development tools. Such a set of guidelines would make it possible for industry to develop and market a diverse line of control system components which could be interchangeable and realizable on many different vendors' control systems platforms.

ARTICS would be designed to facilitate technology and component transfer among the various users and developers, taking advantage of commonalities among otherwise disparate applications such as manufacturing, construction, environmental restoration, mining, space exploration telerobotics, medicine, and military applications of air, land, space, sea-surface, and undersea robotics.

A commercially manufactured ARTICS implementation product would come with libraries of algorithms for planning, task execution, sensor processing and world modeling. These libraries would be user expandable and replaceable. An ARTICS implementation would be fully documented so that users could easily modify or replace any module with a minimum of effort. It would also be commercially maintained, so that users would be able to get help in fixing bugs and making system modifications. In addition, vendors would offer training services to help the user community apply ARTICS products to their applications.

Widely available ARTICS off-the-shelf products would include a target computer system with a backplane and bus configured as a card cage, a local area network to link distributed applications and interface workstations for human/computer interface and software development, a real-time multi-processor/multi-tasking operating system, compilers, debuggers, and CASE tools. ARTICS compliant products could be integrated into an extendible open-system architecture with complete documentation of all hardware and software components.

The ultimate goal would be for ARTICS to evolve into a set of standards for real-time intelligent control systems. Figure 1 illustrates a possible common system configuration for a Version 1 ARTICS system. It would be organized into three levels.

The top level would consist of a number of workstations on an Ethernet for off-line software development and testing. A number of Computer Aided Software Engineering (CASE) tools, shell programs, simulatoıs, debugging and analysis tools, and compilers for at least C, Ada, and Common LISP would be available.

These workstations might include one or more SUNs, LISP machines, VAXes, Butterflys, Connection Machines, graphics engines and display and image processing machines.

One or more of the workstation machines might also be used for on-line real-time control of processes where response time can exceed 1 second, and in situations where weight, power, and other environmental requirements permit. A real-time, multi-computer, multi-tasking operating system such as real-time UNIX, or MACH would be provided to support this type of operation.

The top level would also support an interface to a gaming environment such as the DARPA SIMNET. This would provide a low cost means for testing and evaluating the performance of intelligent machines in a war gaming environment against manned systems, or other unmanned systems. It would also provide an environment for developing tactics and strategy for using large numbers of intelligent vehicles and weapons systems in large scale battle simulations.

The middle level of the ARTICS system would consist of target hardware, such as single board computers and memory boards of the 680X0 variety, using VME or Multi-bus communications. More than one such bus might be connected via bus gateway cards. This middle level would have a real-time, multi-processor, multi-tasking operating system such as pSOS, VRTX, or MACH capable of supporting response times of ten milliseconds or greater.

The bottom level would consist of special purpose hardware which would interface to the VME or Multi-bus. This level would support high speed parallel processing for images, as well as servo controllers with response times between ten microseconds and ten milliseconds.

Figure 2 shows a possible reference model architecture based the Real-time Control System (RCS) concepts NIST has developed since 1980. These have been implemented in a number of applications including the Automated Manufacturing Research Facility (AMRF), the NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM) [Al 89], the Air Force Next Generation Controller for machine tools and robots, and the control system architecture research conducted for the NIST/DARPA Multiple Autonomous Undersea Vehicle (MAUV) project [Al 88].

The version of ARTICS shown here consists of six hierarchical levels: servo, path dynamics, elemental tasks, individual (vehicle), group (squad), and cell (platoon). The top (platoon) level of this reference model architecture would have interfaces to a higher (company) level in a battle management system. The bottom (servo) level would interface to actuators and sensors, and operator interfaces would be defined for all levels.

NIST hopes to enlist the cooperation of experts from industry, academia, and government in developing and modifying these concepts into an agreed upon initial set of guidelines. NIST also intends to sponsor research and enlist others to sponsor research, into advanced concepts that will permit the ARTICS guidelines to evolve as technology advances.

3. REQUIREMENTS

The following "strawman" list of requirements is intended to encourage the robotics community to begin the discourse. A discussion of these requirements can be found in [Al 90].

3.1. Extensibility
3.1.1. Functional Extensibility
3.1.2. Temporal Extensibility
3.2. Human/Computer Interface Flexibility
3.3. Level of Automation Flexibility
3.3.1. Teleoperation and Remote Control
3.3.2. Computer Aided Advisory Control
3.3.3. Traded Control
3.3.4 Shared Control
3.3.5. Human Override
3.3.6. Human Supervised Control
3.3.7. Autonomous Control
3.3.8. Sensory Interactive Control
3.3.9. Mixed Mode Control
3.4. Real-Time and Temporal Reasoning
3.5. Distributed System
3.6. Graceful Degradation
3.7. Application Independence
3.7.1. Software Portability
3.7.2. Compatibility and Inter-Operability
3.8. Ease of Use
3.9. Cost Effectiveness
3.10. Development Environment
3.11. Simulation and Animation

4. APPROACH

To implement the ARTICS concept a consensus must be achieved in several areas of the common control system architecture. Such a conceptual framework would provide developers with a common design philosophy to guide the development of new robotic applications and control system products. A number of control architectures should be considered and evaluated against some set of agreed upon common control system requirements and finally a common conceptual architecture must be derived from the results of the process. More than likely such an architecture would include the ideas of a number of researchers as well as strong input from the user community. The following is a list of recent research in this area:

- Action Networks [Ni 89]
- Autonomous Land Vehicle (ALV) [Lo 86]
- Automated Manufacturing Research Facility (AMRF) [Si 83, Al 81]
- Control in Operational Space of a Manipulator-with-Obstacles System (COSMOS) [Kh 87]
- COmmunications Database with GEometric Reasoning (CODGER) [Sh 86]
- Field Materiel-Handling Robot (FMR) [Mc 86]
- Generic Vehicle Autonomy (GVA) [Gr 88]
- Hearsay II [Le 75]
- Hierarchical Control [Sk 89, Sk 87, Ko 88, Ko 88]
- Hierarchical Real-time Control System (RCS) [Ba 84, Al 81

- Intelligent Control [Sa 85]
- Intelligent Task Automation (ITA) [Bl 88]
- Manufacturing Automation System/Controller (MAS/C) [Ho 88]
- Multiple Autonomous Undersea Vehicles (MAUV) [Al 88]
- NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM) [Al 89]
- Pilot's Associate [Sm 87]
- Robot Control "C" Library (RCCL) [Ha 86]
- Robot Schemas [Ly 89]
- Soar: Architecture for General Intelligence [La 86]
- Subsumption Architecture [Br 86]
- Task Control Architecture (TCA) [Si 89]
- Tech-based Enhancement for Autonomous Machines (TEAM) [Sz 88]
- University of New Hampshire (UNH) Time Hierarchical Architecture [Ja 88]

There are a number of government efforts under way that should be factored into the process of defining an initial set of common architecture components. Some of these include:

- The NIST Federal Information Processing Standard (FIPS)
- The NIST Government Open Systems Interconnection Profile (GOSIP)
- The Navy's Next Generation Computer Resources (NGCR) program
- The Air Force's Next Generation Controller (NGC) program
- The Army's Standard Army Vetronic Architecture (SAVA) program

In addition there is a Department of Energy interest in establishing guidelines for robotic systems needed in their Environmental Restoration and Waste Management Program, the U.S. Bureau of Mines Pittsburgh Research Center is conducting research in automation systems for coal mining and there are a number of DARPA programs (past, present and on-going) which are producing relevant technologies.

5. REFERENCE MODEL DEVELOPMENT PLAN

ARTICS must be able to evolve as technological progress is made. It will be important to create an organizational structure that can coordinate the process of evaluating change and update proposals and a process for achieving consensus on the release of new versions of the ARTICS guidelines. Such an organization will need a steering committee made up of leading experts in the field of robotics, intelligent machines and automation from industry, academia and government.

An ARTICS development effort could take the form of a voluntary organization much like the Initial Graphics Exchange Specification (IGES)/Product Data Exchange Specification (PDES) organization [Ig 89] chaired by NIST. Alternatively the reference model could be developed by a major user of the technology such as the Department of Defense in the form of a military specification (MILSPEC). In either case working groups will be needed to steer the ARTICS development and to document and distribute the results. Once an initial set of ARTICS guidelines has been agreed to it can be submitted to one or more national or international standards organizations as a proposed standard (e.g., ANSI, EIA, IEC, IEEE, ISO, RIA, etc.).

6. REFERENCES

[Al 90] J.S. Albus, R. Quintero, R. Lumia, M. Herman, R, Kilmer, and K. Goodwin, "Concept for a Reference Model Architecture for Real-Time Intelligent Control Systems (ARTICS)," NIST Technical Note 1277, April 1990.

[Al 89] J.S. Albus, H.G. McCain, and R. Lumia, "NASA/NBS Standard Reference Mode for Telerobot Control System Architecture (NASREM)," NIST (formerly NBS) Technical Note 1235, April 1989 Edition.

[Al 88] J.S. Albus, "System Description and Design Architecture for Multiple Autonomous Undersea Vehicles Project," NIST Technical Note 1251, September 1988.

[Al 81] J.S. Albus, "Brains, Behavior and Robotics," BYTE/McGraw-Hill, Petersborough, NH, 1981.

[Ba 84] A.J. Barbera, J.S. Albus, M.L. Fitzgerald, and L.S. Haynes, "RCS: The NBS Real-Time Control System," Robots 8 Conference and Exposition, Detroit, MI, June 1984.

[Bl 88] R. Blair, "Intelligent Task Automation Interim Technical Report," Report No. II-6, Honeywell Corporate Systems, Golden Valley, MN, August 1988.

[Br 86] R.A. Brooks, "A Robust Layered Control System For A Mobile Robot," IEEE Journal of Robotics and Automation, Vol. RA-2, No. 3, March 1986.

[Gr 88] M.D. Grover, et. al., "An Autonomous Undersea Vehicle Software Testbed," Proceedings of the Fifteenth Annual Technical Symposium of the Association for Unmanned Vehicle Systems, San Diego, CA, June 6-8, 1988.

[Ha 86] V. Hayward, R.P. Hayward, "Robot Manipulator Control Under Unix RCCL: A Robot Control "C" Library," International Journal of Robotics Research, Vol. 5, No. 4, Winter 1986.

[Ho 88] Honeywell Inc., "MAS/C System Overview," MA70-100, Release C, Honeywell Inc., Phoenix, AZ, May 1988.

[Ig 89] "Welcome to IGES/PDES, Newcomer Material," July 1989, available through the Chairman, IGES/PDES, NIST, Bldg. 220, Room A150, Gaithersburg, MD, 20899.

[Ja 88] J. Jalbert, "EAVE III Untethered AUV Submersible," Proceedings of the Fifteenth Annual Technical Symposium of the Association for Unmanned Vehicle Systems, San Diego, CA, June 6-8, 1988.

[Kh 87] O. Khatib, "A Unified Approach for Motion and Force Control of Robot Manipulators: The Operational Space Formulation," IEEE Journal of Robotics and Automation, Vol. RA-3, No. 1, February 1987.

[Ko 88] W. Kohn, T. Skillman, "Hierarchical Control for Autonomous Space Robots," Proceedings of the AIAA Guidance, Navigation, and Control Conference, Minneapolis, Minnesota, Aug 15-17, 1988.

[Ko 88] W. Kohn, "Declarative Theory for Relational Controllers," Proceedings of the IEEE Control Decision Conference, Austin, Texas, Dec 7-9, 1988.

[La 86] J. E. Laird, A. Newell, P. S. Rosenbloom, "Soar: An Architecture for General Intelligence," Carnegie-Mellon University paper CMU-CS-86-171, December 1986.

[Le 75] V.R. Lesser, et. al. "Organization of the Hearsay II Speech Understanding System," IEEE Trans. on ASSP, Vol. 7, No.1, 1975, p. 11.

[Ly 89] D. M. Lyons, M. A. Arbib, "A Formal Model of Computation for Sensory-Based Robotics," IEEE Transactions on Robotics and Automation, Vol. 5, No. 3, June 1989.

[Lo 86] J. Lowerie, et. al. "Autonomous Land Vehicle," ETL- 0413 Martin Marietta Denver Aerospace, Denver, Co, July 1986.

[Mc 86] H. G. McCain, et. al. "A Hierarchically Controlled Autonomous Robot for Heavy Payload Military Field Applications," Proceedings of the International Conference on Intelligent Autonomous Systems, Amsterdam, The Netherlands, December 8-11,1986.

[Ni 89] N. Nilsson, "Action Networks," Proceedings from the Rochester Planning Workshop: From Formal Systems to Practical Systems, J. Tenenberg, et al. (eds.), University of Rochester, New York, 1989.

[Ro 89] Robotic Technology Incorporated, "Standard Architecture for Real-Time Control Systems (SARTICS), Final Report," NIST Contract No. 43NANB909769, dated 21 Nov. 1988, Submitted to NIST Robot Systems Division, Gaithersburg, MD, 3 July 1989.

[Sa 85] G.N. Saridis, "Foundations of the Theory of Intelligent Controls," IEEE Workshop on Intelligent Control, Troy, August 1985, p. 23.

[Sh 86] S.A. Shafer, et. al. "An Architecture for Sensor Fusion in a Mobile Robot," Proceedings of the IEEE International Conference on Robotics and Automation, San Francisco, CA, April 7-10, 1986, pp. 2002-2010.

[Sk 89] T. L. Skillman, K. Hopping, "Dynamic Composition and Execution of Behaviors in a Hierarchical Control System," SPIE Mobile Robotics IV, Philadelphia, Nov 7- 8, 1989.

[Sk 87] T. L. Skillman, W. Kohn, et al."Blackboard Based Hierarchical Intelligent Control System," Proceedings of the AIAA Conference on Computers in Aerospace VI, Wakefield, MA, Oct 7-9, 1987.

[Si 89] R. Simmons, T. Mitchell, "A Task Control Architecture for Mobile Robots," Computer Sciences Dept., Carnegie Mellon Univ., Pittsburgh, PA, AAAI '89 Spring Symposium, Robotic Navigation, 1989.

[Si 83] J.A. Simpson, R.J. Hocken, and J.S. Albus, "The Automated Manufacturing Research Facility of the National Bureau of Standards," Journal of Manufacturing Systems, Vol.1, No. 1, pg. 17, 1983.

[Sm 87] D. Smith, M. Broadwell, "Plan Coordination in Support of Expert Systems Integration", Proceedings of the Knowledge-Based Planning Workshop, Austin, TX, December 8-10, 1987.

[Sz 88] S. Szabo, H. A. Scott, R. D. Kilmer, "Control System Architecture for the TEAM Program," Proceedings of the Second International Symposium on Robotics and Manufacturing Research, Education and Applications, Albuquerque, NM, November 16-18, 1988.

ARTICS VERSION 1
COMMON SYSTEM CONFIGURATION

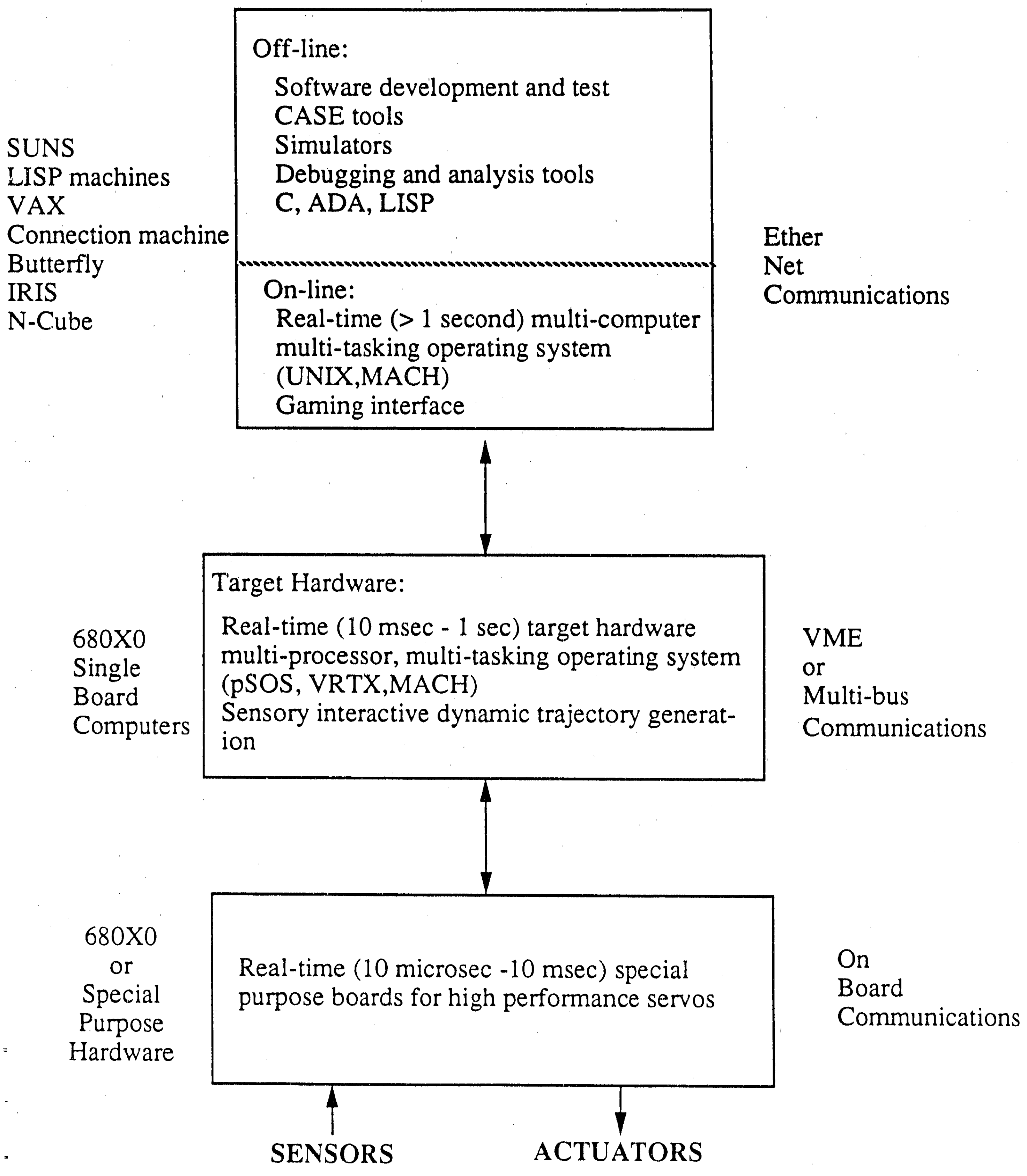

SUNS
LISP machines
VAX
Connection machine
Butterfly
IRIS
N-Cube
Off-line:
Software development and test
CASE tools
Simulators
Debugging and analysis tools
C, ADA, LISP
On-line:
Real-time (> 1 second) multi-computer multi-tasking operating system (UNIX,MACH)
Gaming interface
Ether Net Communications
680X0 Single Board Computers
Target Hardware:
Real-time (10 msec - 1 sec) target hardware multi-processor, multi-tasking operating system (pSOS, VRTX,MACH)
Sensory interactive dynamic trajectory generat-ion
VME or Multi-bus Communications
680X0 or Special Purpose Hardware
Real-time (10 microsec -10 msec) special purpose boards for high performance servos
On Board Communications
SENSORS
ACTUATORS

Figure 1

REFERENCE ARCHITECTURE FOR REAL-TIME INTELLIGENT CONTROL

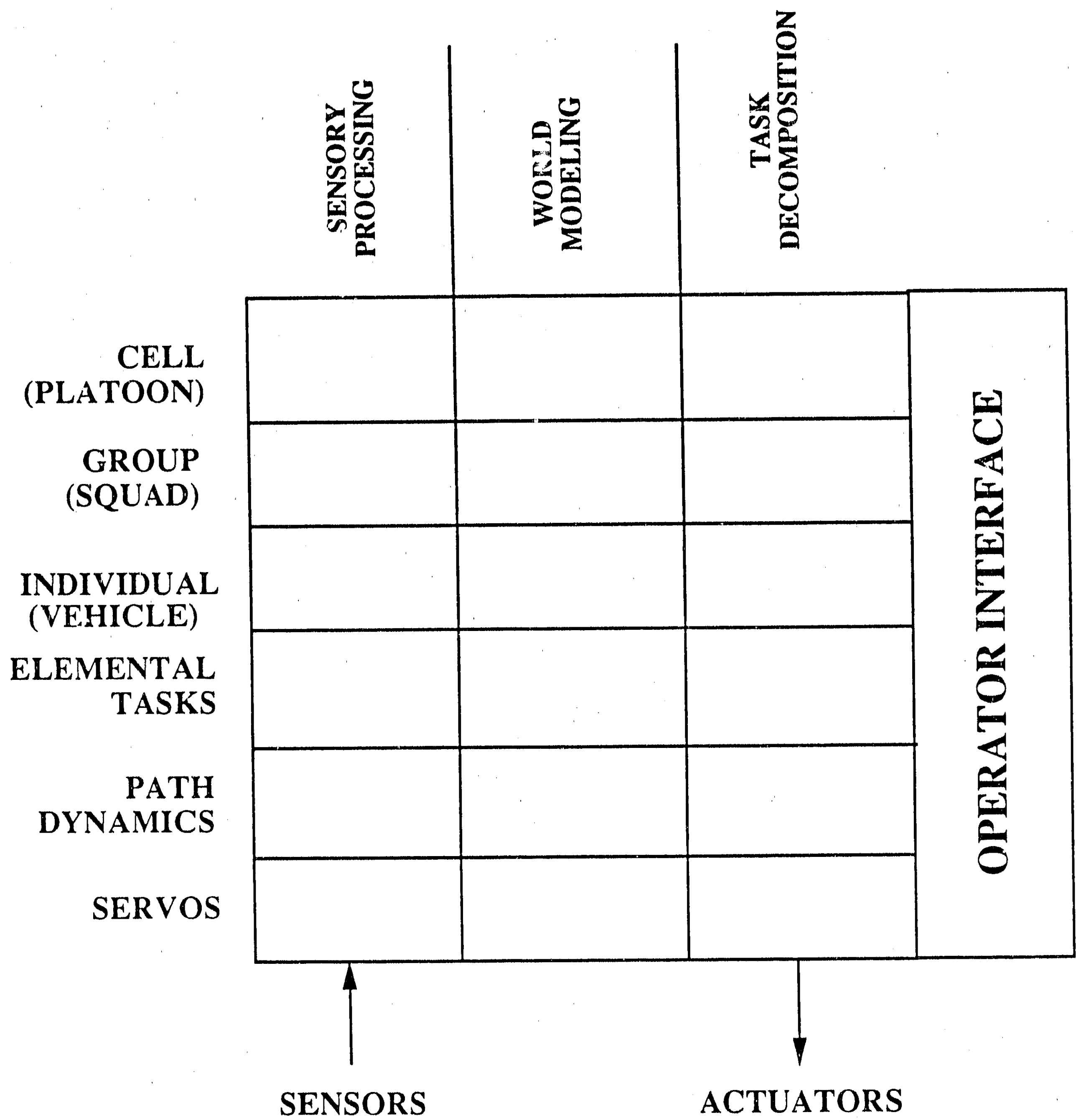


Figure 2

Applying a Computer Aided Prototyping System to the Software of an Autonomous Underwater Vehicle

Thomas E. Bihari*, Robert B. McGhee, Luqi, Yuh-jeng Lee

Department of Computer Science
Naval Postgraduate School
Monterey, California 93943
(408) 646-2449

*** Adaptive Machine Technologies, Inc.**
1218 Kinnear Road
Columbus, Ohio 43212
(614) 486-7741

I. Introduction

This workshop addresses an important direction for tool development. Within the current state of the practice, a great deal of duplicated effort is spent developing similar software systems within a particular application domain, such as distributed, intelligent vehicle control. The experience gained during these projects is wasted if it cannot be used to aid the development of subsequent, similar projects.

We believe that this experience can provide the basis for developing a common ground for all the applications in a domain. The development of domain-specific architectures and tools supporting the essential processes and properties of particular application domains will allow reuse of the domain knowledge and domain-specific solution techniques that comprise the most expensive part of the effort to develop new systems.

Expensive tool implementation efforts can be wasted if tool construction is started without a clear idea of the problems the tools are supposed to solve and without a systematic and formalized set of solution techniques to be incorporated in the tools. Constructing tools is both labor intensive and skill intensive, and involves knowledge both of software engineering principles and of the application domain. Because this combination is hard to find, tools developed by software engineering researchers are often "demo driven" and lack applicability, while those developed by application domain experts are often ad hoc and lack strong foundations.

The best tools are those which are based on strong theoretical principles, and also driven by a strong, vocal user community. This can be difficult to achieve when developing tools that push the state of the art, regardless of whether the effort is done by industry or by universities. The perceived reward structure for researchers in the tool provider and user communities usually makes such interaction seem undesirable - more work with little direct payoff. DOD support is essential in building an interaction between the providers and users of domain-specific software development tools. Appropriate modes of interaction should be identified and supported.

This paper presents the result of a study of the potential for interaction between two on-going research projects at the Naval Postgraduate School. The Computer-Aided Prototyping System (CAPS) project[1] is developing models and support tools for rapid prototyping of embedded real-time software. The Autonomous Underwater Vehicle (AUV) project[2] is developing a computer-controlled submersible vehicle.

The purpose of this study was to examine the goals of the AUV project and its resulting software requirements, and the goals and capabilities of the CAPS project, and to determine the benefits of pursuing joint research in the application of CAPS to the AUV software.

II. The Computer-Aided Prototyping System Project

The goal of the Computer Aided Prototyping System (CAPS) project [Luqi88] [Luqi88a] [Luqi88b] [Luqi89] is to enable rapid prototyping of parallel and distributed real-time software, as a way of increasing productivity and decreasing software costs. The CAPS project focuses on automated methods for retrieving, adapting, and combining reusable components based on normalized module specifications; establishing feasibility of real-time constraints via scheduling algorithms; simulating unavailable components via algebraic specifications; automatically generating translators and real-time schedules for supporting execution; constructing a prototyping project database using derived mathematical models; providing automated design completion and error checking facilities in a designer interface; and establishing a convenient graphical interface for design and debugging.

CAPS is a set of software tools, sharing a common basis consisting of a rapid-prototyping software development methodology, an enhanced-dataflow computational model and a prototyping language. The CAPS tool set includes a graphical editor, a syntax directed editor, a database of existing software components, a database of existing software designs, a translator which converts the prototyping language into a particular implementation language (e.g., Ada), static and dynamic task schedulers, a debugger, and others. The tool set is running on a Sun SPARCstation under UNIX and X-Windows, and is portable to any system with UNIX and X-Windows. The product produced by the system is Ada code, which is portable to any system with an adequate Ada compiler. The system can be used to design distributed and intelligent systems [Luqi89a].

CAPS' support for the rapid-prototyping methodology makes it possible for prototypes to be designed quickly and to be executed to validate the requirements. CAPS manages the entire prototyping process, from the development of the software design, through the retrieval or creation of reusable Ada software components, to the generation, execution, and analysis of the resulting Ada program. CAPS users may iterate through this process until they are satisfied with the software's behavior.

1. The CAPS project is directed by Dr. Luqi and is supported by the National Science Foundation under grant number CCR-8710737.

2. The AUV project is directed by Dr. A.J. Healey and is supported by Naval Postgraduate School funding.

The CAPS computational model and tools provide the designer of a software system with a way to draw an augmented dataflow diagram which contains the necessary timing and control constraints for specifying embedded software systems. The model maximizes parallelism by enforcing timing and control constraints only where necessary. The graphical design and the constraints drawn through the CAPS graphical editor for an embedded control system are based on the syntactical structure of the Prototype System Description Language (PSDL) [Luqi88a].

The specification part of the PSDL program describes the basic attributes of required software components. (Currently the components are Ada units, but other languages can also be supported.) This information is used by a tool which searches for appropriate reusable components stored in the software base. If no suitable component is found in the existing software base, the designer may choose to create a completely new component from scratch or to create a new component by combining or modifying an existing set of components. When the design is completed, the PSDL program is translated into Ada code which has the structures for realizing the timing and control constraints built in. The Ada program is then compiled.

The designer may then execute the program and evaluate the prototype's behavior against the behavior that he expected it to have. If the comparison results are not satisfactory, the designer may modify the prototype and evaluate the prototype again. This process continues until the prototype meets the requirements.

CAPS was designed to be used for developing prototypes for real-time systems. In CAPS, a hard real-time constraint is a bound on the response of a process which must be satisfied under all operating conditions. CAPS specifications can represent a variety of real-time constraints, including (1) maximum execution times for modules, (2) minimum calling periods, (3) synchronization of processes with sporadically-arriving external data or interrupts, (4) delays required by limitations on input/output devices, (5) maximum response times, and (6) periodic system actions.

The CAPS model and tool set have been applied to real-time software designs in several areas, including C3I [Luqi89] and process control [Luqi88a].

III. The Autonomous Underwater Vehicle Project

III.1. Overview of the AUV-II

The AUV-II is the second in a series of autonomous underwater vehicles developed at the Naval Postgraduate School. It is described in detail in [Cloutier90] [Healey89] [Kwak90]. Briefly, the AUV-II is a self-contained vehicle, approximately 16 inches wide, 10 inches deep, and 93 inches long. It displaces approximately 387 pounds and is powered by on-board batteries.

The AUV-II is a research vehicle designed as a testbed for research in mission planning, path planning, sonar data analysis and world modeling, navigation through obstacle fields, and other intelligent behaviors.

The AUV-II is propelled by two main screws (port and starboard aft) and four tunnel thrusters (fore and aft vertical, and fore and aft athwartships). These may be used to control five degrees of freedom; the AUV-II's roll axis is not controlled. However, when the AUV-II is moving with sufficient speed, the control surfaces (bow planes, stern planes, and fore and aft rudders) may be used in conjunction with the screws and thrusters to control all six degrees of freedom, including roll.

The AUV-II's sensor system consists of four pencil-beam sonar transducers mounted in the AUV-II's nose, a full suite of inertial sensors (three rate gyros, three accelerometers, a vertical gyro and a directional gyro), a flux gate compass, a paddle-wheel speed sensor, and individual motor RPM sensors.

Because the AUV-II is a research vehicle, its computational requirements are subject to change. It is important that the on-board computer hardware be modifiable and extensible, by adding more raw computing power, memory, and I/O devices, and by adding different types of these components.

The on-board computer is centered around a 12-slot GESPAC G-96 bus. The bus currently hosts one GESPAC MPU-20HF single-board computer (25 MHz Motorola 68020 and 68882 processors, 2.5 Mb of RAM, and up to 4 Mb of EPROM), and 5 other boards containing interfaces to a 200 Mb hard disk, parallel and serial communication ports, analog-to-digital input channels interfaced to the AUV-II's sensors, and digital-to-analog output channels interfaced to the AUV-II's effectors. There are currently 6 free bus slots. These are expected to be used for additional GESPAC MPU-30HF (68030-based) boards, a Transputer board, and other devices as necessary.

The GESPAC computer uses Microware's OS-9 real-time operating system. OS-9 is a full operating system, with a file system, native compilers and other development tools. OS-9 uses a time-slicing, prioritized, task scheduler. Intertask communication is via global memory, pipes, signals, and BSD4.2 sockets for inter-processor communication.

In addition to the AUV-II itself, a laboratory computer identical to that in the AUV-II, and a graphical simulation of the AUV-II and its environment (running on a Silicon Graphics workstation), are available for software development.

III.2. Characteristics of the AUV-II Software

The AUV-II's software is described in detail in [Cloutier90]. Our main interests for this study were not in the particular guidance and control algorithms, but in the duties performed by the software, the software's real-time requirements, the overall software design, and the expected life cycle of the software.

The AUV-II's software is designed as a layered architecture in which higher levels pass requested vehicle states to lower levels. The lower levels attempt to meet the requests, possibly modifying them to make them feasible, and may pass information back to the higher levels, allowing the higher levels to modify future requests. There are currently three levels. The top level consists of the Mission Planner (which is off-board), and the Mission Replanner (an on-board planning subsystem which may override the off-board planner).

These subsystems generate sets of paths describing particular missions.

The middle level consists of the Guidance sub-system, which receives paths from the Mission (Re)Planner and calculates individual "postures" to be achieved by the AUV-II. The bottom level consists of the Autopilot subsystem, which servos the AUV-II's effectors to achieve the requested postures. This is performed on a 100 ms period.

To provide input to the Autopilot's servo control loop, the state of the AUV-II must be determined from the inertial, depth, and speed sensors. This must be done by the Navigation sub-system at rates sufficient to provide an accurate current state. The AUV-II state is currently updated every 100 ms. Project goals call for data from the sonar sensors to be integrated with other sensor data, and with pre-loaded obstacle maps, in several phases. Initially, sonar data will be used to correct inertial sensor drift. Eventually, sonar data will be used for collision avoidance and for revising the existing world model. Sonar data will be collected at 100 ms periods, and world modeling will occur at somewhat longer periods.

The relative steering effectiveness of the thrusters vs the control surfaces depends on the AUV-II's forward velocity. It is anticipated that the control surfaces' effectiveness will vary from zero at zero velocity to approximately four times that of the thrusters at maximum velocity. Therefore, the AUV-II motion control strategy, and the control software, has been divided into two modes: Hovering Mode and Transit Mode. The software must be able to cleanly switch between these modes while in operation.

In addition to these "normal" operations, reflex actions like collision avoidance may be triggered by special circumstances and must produce quick responses, sometimes overriding existing activities.

The software characteristics of the AUV-II are both similar to and different from those of the Adaptive Suspension Vehicle (ASV), a three-ton, self-contained, six-legged walking vehicle[1] with which we have been associated in the past [Bihari89]. Both vehicles perform sensing, world modeling, motion planning, and servo control in real time.

For the most part, the AUV-II's guidance and control software is not required to meet extremely tight real-time requirements. Sensing and servo control periods are on the order of 100 ms or greater, with some allowable slippage. This is easily within the capabilities of existing computer hardware, real-time operating system, and software technologies. The ASV has tighter real-time requirements than the AUV-II. For example, the ASV's leg servo control software executes with periods of 10 ms or less, with serious consequences if servo cycles are missed. The ASV's real-time requirements are well-defined and generally situation-independent, however.

The AUV-II is required to be completely autonomous, while the ASV has an on-board operator performing many of the high-level world modeling and motion planning duties. The AUV-II must maintain a larger view of time (e.g., for an entire mission). For example,

1. Sponsored by the Defense Advanced Research Projects Agency under contracts MDA903-82-K-0058, DAAE07-84-K-R001, and MDA972-88-C-0031.

planning may take a significant amount of time, and the amount of time may be situation-dependent. The AUV-II must be capable of reasoning about time, and of "planning to plan", and the enforcement of the resulting timing constraints must be handled by the underlying operating system and support tools.

Furthermore, much of the information contained in the AUV-II system is time-dependent. That is, the AUV-II's perception of the state of itself, obstacles, and mission plans is dependent on the relationship between the current time and the time at which the information was created (e.g., the age of the data). Portions of the AUV-II's software may resemble a temporal database.

The AUV-II is experimental, and the software's duties range from low-level sensor data processing and servo control to high-level planning and world modeling. Ideally, the AUV-II software development environment would support the integration of a variety of programming paradigms, including procedural, functional, object-oriented, logic-based, and rule- or frame-based. Practically, a system supporting Ada and Common Lisp could provide a basis for most of these paradigms. (This would be a step forward in the state of the practice. Almost all existing AUVs, including the AUV-II, are programmed in C.)

In summary, the AUV-II software system has the following characteristics:

From a software architecture standpoint:

1. It is hierarchically structured, and it can best be understood by viewing it at different levels of abstraction for different purposes.

2. It consists of subsystems, some of which are tightly coupled, others of which are loosely coupled (and execute at different rates).

3. It operates in at least two separate modes.

4. It must occasionally perform reflex actions which override normal operations.

5. Most of the computations have real-time constraints.

6. It includes time-dependent representations of the states of the AUV-II and environment.

From a software management standpoint:

1. The specification, design, and implementation of the entire system (mechanical hardware, electrical and electronic hardware, and software) will evolve as existing research questions are answered and new questions are asked.

2. Small changes to the software can be expected to occur frequently. That is, software development will follow an experimental, iterative, implement-execute-evaluate cycle. The software may also need to be specially configured for specific missions.

3. Multiple versions of the software may be "active" at the same time, as different

researchers conduct independent experiments using specialized components integrated with a common software base.

4. The software base can be expected to outlive (in a project sense) most of the software developers. Software development methodology support and enforcement is important.

5. It must be possible for different people to understand and manipulate the system at different levels of abstraction (e.g., as "black boxes"), so they not have to learn the entire system in order to perform useful research. It must not take too long to "come up to speed".

6. Different languages and programming paradigms may be most effective for different components (or different versions of the same component). A uniform framework for managing these disparate components is needed.

IV. The Potential for Further CAPS-AUV Project Cooperation

In theory, the interaction of a real-time software tool provider (the CAPS project) with a real-time software tool user (the AUV project) has many advantages. The CAPS project would benefit from the availability of a realistic application. The AUV project would benefit from an improved software development methodology and support tools. In practice, the interaction of two such research projects must be realistic and well-defined if it is to be beneficial to both parties.

In our view, CAPS provides an appropriate and extremely useful methodology for developing real-time control software like that of the AUV project. The concepts supported by CAPS generally match those we expect for the AUV-II's life cycle. The integrated tool set should lead to easier software development and strict enforcement of the software development methodology. PSDL seems to contain the features necessary for the AUV software.

There are practical considerations, however. For example, the current AUV software is written in C, while CAPS supports only Ada at this time. The CAPS tools currently run under X Windows on a Sun SPARCstation, while the AUV tools are running on an IBM PC/AT compatible. Resolution of these practical matters could consume valuable "research" time. Some care is also needed because a complete treatment of the problem requires solutions to two unsolved research problems: real-time databases and real-time scheduling. Domain-specific assumptions and approaches must be developed to provide adequate solutions to these problems. Some progress in these directions has already been made [Galik88] [Guentenburg89] [Huskins90] [Mostov90] [Sun90] [White89], but these solutions have not yet been incorporated into the current implementation of CAPS.

We see the potential for a step by step increase in interaction between the CAPS and AUV projects. This should begin by establishing a realistic set of goals. Those goals might include, for example:

1. Formulate the AUV-II software design in PSDL and critique the design.

2. Translate the AUV-II's existing C code to Ada, and move the AUV-II development

environment to a platform with appropriate Ada tools and the X-Windows support needed by CAPS (e.g., Sun or DEC MicroVAX).

3. Form the AUV-II's Ada modules into CAPS reusable components and develop a complete AUV-II software version under CAPS.

And so on.

It is important to avoid over-integrating the two projects. In order to avoid delaying the progress of either project, the projects should maintain independent critical paths. For example, the AUV programmers should continue to develop C code until the Ada development environment is fully operational. A significant benefit might be gained by interaction at the design level (e.g., Goal 1) regardless of the eventual implementation of AUV-II software under CAPS.

V. Conclusion

The number and complexity of intelligent, autonomous, real-time systems are expected to grow, driven by the need to perform missions for which human supervision is unavailable or not cost-effective. The development and maintenance of software for such systems is an important area of research. We believe that progress in this area is achieved best by the cooperation of the providers of real-time software engineering technology (e.g., CAPS) and the users of that technology (e.g., AUV). Appropriate modes of interaction must be found.

References

[Bihari89] Bihari, T., Walliser, T. and Patterson, M., "Controlling the Adaptive Suspension Vehicle", IEEE COMPUTER, pp. 59-65, June 1989.

[Cloutier90] Cloutier, M., "Guidance and Control System for an Autonomous Vehicle", Masters Thesis, Naval Postgraduate School, June 1990.

[Galik88] Galik, D., "A Conceptual Design of a Software Base Management System for the Computer Aided Prototyping System", Masters Thesis, Naval Postgraduate School, December 1988.

[Guentenburg89] Guentenburg, H., "Automatic Generation of an Aircraft Inertial Navigation System", Masters Thesis, Naval Postgraduate School, May 1989.

[Healey89] Healey, A., Papoulias, F., and MacDonald, G., "Design and Experimental Verification of a Model Based Compensator for Rapid AUV Depth Control", Proceedings of the 6th Unmanned, Untethered, Submersible Technology Conference, Washington, D.C., June 12-14, 1989.

[Huskins90] Huskins, J., "Issues in Expending the Software Base Management System Supporting the CAPS", Masters Thesis, Naval Postgraduate School, June 1990.

[Kwak90] Kwak, S., Ong, S. and McGhee R., "A Mission Planning Expert System for an Autonomous Underwater Vehicle", IEEE Symposium on Autonomous Underwater Vehicle

Technology, pp. 123-128, June 1990.

[Luqi88] Luqi and Berzins, V., "Rapidly Prototyping Real-Time Systems", IEEE Software, pp. 25-36, September 1988.

[Luqi88a] Luqi, Berzins, V. and Yeh, R., "A Prototyping Language for Real-Time Software", IEEE Transactions on Software Engineering, pp. 1409-1423, October 1988.

[Luqi88b] Luqi, "Knowledge-Based Support for Rapid Software Prototyping", IEEE Expert, pp. 9-18, Winter 1988.

[Luqi89] Luqi and Davis, T., "A Software Prototype of the Message Processor in Navy C3I Station", Naval Postgraduate School Technical Report NPS52-90-010, August 1989.

[Luqi89a] Luqi, Berzins, V., Kraemer, B., and White, L., "A Proposed Design for a Rapid Prototyping Language", Naval Postgraduate School Technical Report NPS52-89-045, March 1989.

[Mostov90] Mostov, I., "A Model of Software Maintenance for Large Scale Military Systems", Masters Thesis, Naval Postgraduate School, June 1990.

[Sun90] Sun, J., "Developing Portable User Interfaces for Ada Command & Control Software", Masters Thesis, Naval Postgraduate School, June 1990.

[White89] White, L., "The Development of a Rapid Prototyping Environment", Masters Thesis, Naval Postgraduate School, December 1989.

OS-9 is a registered trademark of Microware Corp.

Ada is a registered trademark of the U.S. Government, ADA Joint Program Office.

G-96 is a registered trademark of GESPAC SA.

UNIX is a registered trademark of AT&T.

All other brand or product names are trademarks or registered trademarks of their respective holders.

Concurrent Processing Environments for Distributed Intelligent Control Systems

J. Douglas Birdwell and Sheng Liang
Department of Electrical and Computer Engineering
University of Tennessee
Knoxville, TN 37996–2100

Abstract

Factors affecting the development of concurrent processing environments for knowledge–based systems in real–time applications are discussed. A collection of cooperating small and simple knowledge–based systems is an attractive alternative to a large centralized one. There are, however, significant questions concerning the relative performance and survivability of these systems. This paper will address some of these issues, including suitable communication frameworks, and the relative merits of various features of the knowledge base description language and approach. There are several advantages to distributed implementations including more efficient evaluation of smaller knowledge bases and specialization of knowledge bases within specific domains of expertise. Potential disadvantages include communication delays, performance under degraded operating conditions, and non-deterministic operation. Some discussion will be provided concerning an expert system shell design for applications in power electronics.

Introduction

Our research explores the development of an intelligent controller for a power electronics–based inverter drive for induction motors. The controller is to be implemented using a dual processor architecture using an Intel 80386 microcomputer and a Texas Instruments TMS320C30 digital signal processor (DSP). The software residing on the microcomputer is to be controlled by a rule–based expert system while the TMS320C30 DSP carries out time–critical control and decision action.

One project objective is to explore alternate expert control system architectures for implementation on microcomputers. A modern control algorithm contains significant algorithmic complexity; along with the necessary protective mechanisms, interfaces, and discrete state logic, it often stresses an implementor's ability to meet performance specifications using the available hardware technology. Now we envision the addition of "intelligent" control functions which many hope will accomplish things which previous, more algorithmic, approaches have failed to do. We view segmentation of process knowledge, either within a single processor or over a distributed network of processors, as a necessary strategy to provide sufficient processing capability and flexibility to meet our long–range goals. This is accomplished in two ways. By reducing the size of each "intelligent" module, the efficiency with which it can be evaluated is increased, and this increase should more than offset the added overhead of coordination between cooperating modules. Second, migration to a distributed computing architecture is nearly transparent. An added benefit is derived from the use of an object–oriented module structure; modules may be implemented and tested in a somewhat independent manner. An attractive approach is to introduce a collection of relatively small distributed and cooperating knowledge–based systems, or ***knowledge sources (KSs)***. This method's concurrency tends to favor real-time processing since the execution of one knowledge source will not block the remaining knowledge sources from real-time events. Our interest is on a development tool rather than a single expert system. We intend tu explore the essential requirements of real-time distributed knowledge-based systems and design an appropriate development tool.

A significant body of literature exists documenting previous research in this area; however, we believe our work is somewhat unique in the following areas: First, our goal is to develop an expert system tool capable of supporting very high speed real-time processing; as such, our interest in segmentation of domain expertise into independent, cooperating knowledge sources is for efficiency by limiting the size and scope of each knowledge source. Second, rather than focus upon the coordination behaver between knowledge sources, as is done in much of the existing literature, we focus upon the eventual use of the tool in a complex application; thus, knowledge is decomposed along easily recognizable boundaries, such as interprocessor communication, human interface, economic optimization, and measurement database functions, and messages are designed based upon "engineering expertise," rather than upon any rigorous foundation. Third, we sacrifice some flexibility for efficiency; specifically, classes of objects (frames) and data types are supported and used to further segment each knowledge source's fact base, inference is (at present) restricted to forward chaining, and only limited pattern matching is supported rather than unification. Fourth, we aim to implement this tool on a very limited architecture and operating system environment, using the 80386 processor and the MS–DOS operating system with an extender which provides protected–mode (flat address space) operation. This makes it easier to embed the system in real-time hardware.

Our interest spans more than 80386 applications; in that regard, we are developing an equivalent Unix–based tool using multiple processes and lightweight processes, and intend to explore the implementation of a similar tool in a multi-processor environment using an Intel iPSC/860 hypercube. Such tools can be used for future research on performance and reliability of object–oriented expert systems on distributed architectures, and for development of embedded and distributed control applications.

Background

At the outset, one must ask why we should begin development of yet another expert system tool. Existing tools abound; why are they unlikely to satisfy our requirements? First, we are interested in embedded applications using relatively small computing resources. Second, we require real–time processing in a fairly demanding (with respect to the rate at which information is received and must be interpreted, at least) application. Many existing tools, such as KEE [1] and CLIPS [2], which represent two extremes of the cost scale, are not designed to deal with real–time applications. Systems such as G2, Muse and Cronos [3], which are intended for real–time applications, are too costly for our applications, and are probably too complex to successfully embed in the intended computer architecture. We wish to take advantage of "intelligence" in the control system design; however, we need a "lightweight" environment in which to implement the required functionality. At present, this does not exist commercially, or, to our knowledge, within the research community.

Numerous real–time expert systems for specific applications have been developed in the past ten years. While the term *real time* doesn't say in general how "fast" the system should be, people have generally agreed that a real–time system should be fast enough for use by the process being served and should guarantee a response in a strict time limit. Some guidelines are important for the development of real–time systems and tools. Real–time systems must deal with nonmonotonicity of incoming data, asynchronous events, an interface to external environments (sensors, actuators, and possible a human supervisor), and uncertain or missing data. Other requirements include the ability to maintain continuous operation (not interrupted by garbage collections), guaranteed response time, controlled focus of attention, and temporal reasoning [4].

Blackboard systems have proved to be successful in dealing with complex problems with large solution spaces [5]. Under the blackboard framework, knowledge is divided into multiple knowledge sources that perform the subtask of finding partial solutions. Partial solutions are posted on the blackboard so they are globally accessible. Controlled by a scheduler, knowledge sources take turns generating new partial solutions. It's easy to segment a task into modules and use multiple reasoning methods; however, knowledge sources in blackboard systems are tightly coupled. Each of them is only part of a intelligent system just like a piece of the human brain. They must be well-organized and tightly connected. In this way, modularity of each knowledge source is limited. System response time to events is not guaranteed since the activation of a knowledge source will normally not end until it terminates. Thus, a traditional blackboard system is essentially a centralized system with the additional features controlling focus of attention and partitioning knowledge. Researchers have extended this to build distributed blackboard systems for real–time applications, such as distributed sensor networks (DSN) [6]. These systems use several blackboard systems which communicate via message passing. Although they were distributed intelligence systems which ran in real–time, they were structurally tailored for a specific application. This approach is suitable for some problems, but it is not a general framework.

A type of loosely coupled, distributed cooperative problem-solving system [7] has gained some interest; in addition, it fits the technology of distributed systems well. Each element in the problem–solving network works rather autonomously as a member in a cooperative team with little centralized control. Better modularity is introduced. Unlike blackboard systems, which have been developed for real–time applications, this class of systems has as yet no large application, although it seems promising for real-time control.

Blackboard system and CDPS are so far the two most influential approaches to distributed knowledge based system design. Blackboard systems have undergone more then 15 years of evolution. The technology is mature due to the numerous large applications which have been implemented, but they are generally expensive to run and develop. CDPS systems research, on the other hand, started with an interest in the cooperative behaver of seperate intelligent systems. Researchers have worked on how to maintain coherent cooperation under limited communication, including the means to communicate more intelligently. These results are so far primarily of theoretical value. Most of the work on CDPS is based on prototype implementations and simulation, for the probable reason of the overwhelming acceptence of blackboard systems in complex real–world applications. To date, CDPS suffer from the lack of application and development tools. The only way to heal this situation is to test CDPS in applications of a realistic size.

Researchers in artificial intelligence (AI) seem to be overlooking the possible wide use of their complex frameworks in low–end control systems, such as embedded controllers. In this area traditional control argorithms and simple AI approaches still dominate. Currently complex systems at most work at a very high level of control systems where real–time features and restrictions are lost. A significant dependence upon the support services of the operating system has been assumed during development of most of these systems; such support is unlikely to be present in low–end applications. These factors make it hard to migrate the framework to a low–end processor for real–time processing; on the other hand, this migration may alleviate the current complexity of the control algorithm once more capable AI frameworks reside on low–end processors. This gap can be filled by the development of suitable expert system tools that meet the following: First, they should support the development of expert system under a new and effective real–time AI framework. Second, they should provide guaranteed response to external events. Third, these systems should be simple to learn and easy to use by control system engineers.

Framework

We are interested in exploring the use of cooperative distributed knowledge-based systems for real-time control and in developing an environment for testing. Our framework differs from most existing expert system shell's in that we introduce fully concurrent inference on distributed knowledge bases. Many of the ideas are from CDPS systems; however, since it's a development tool, it leaves the upper layer coordination protocol to the expert system's developer, while supporting only the concurrent inference and communication framework.

Each knowledge source is an independent system. Knowledge sources residing on a single processor are scheduled by a preemptive scheduler. In our system the scheduler knows nothing about what's going on within each knowledge source; it only guarantees the execution time allocated to a knowledge source will not exceed a pre-determined time slice, and that knowledge sources will be scheduled according to the assigned priorities. The scheduler knows nothing about the reasoning, and is unable to organize the execution of each knowledge source on the basis of the state of the reasoning process. Alternate methods are the design of non-preemptive or self–scheduling processes. In a self–scheduling system, the active process decides when to suspend itself and which other process to activate. Blackboard systems use non-preemtive scheduling, as do some CDFSs such as the AF system [8]. This approach introduces less overhead on task switching and synchronization; also it appears more favored in producing a trackable process. We favor our approach because of the following reasons:

- **It enhances structural simplicity and modularity.** Unlike non-preemptive scheduling, which needs a centralized scheduler, and self–scheduling processes, which assume each KS performs proper scheduing, scheduling is transparent to the KS implementer. Each KS is responsible only for itself. Structural simplicity is obvious compare with blackboard systems. Also, self–scheduling increases the connectivity of KSs and decreases modularity.
- **Reliability is improved.** Dangerous situations like system deadlock due to errors in one KS are prevented. Fatal error in a single KS can be detected and, by reseting that particular KB, the rest of the system will still work. This leads to better error recovery and system diagnosis. In a non-preemptive system, this situation normally leads the entire system into an unwanted state forcing a system–wide reset.
- **Quicker response to external stimuli is obtained.** The maximum delay for response to an asynchronous event is the length of the time slice. The scheduler can also be designed to immediately start re-scheduling upon receipt of an urgent event. For a non-preemptive system, the KS writer must either take meticulous care to guarantee each KS's maximum time consumption, or must let activation of a KS be interruptable by an event. The later solution violates non-preemptive scheduling, so the action that deals with events can not be considered a standard KS. This leads to increased structual complexity.
- **Implementations and behavior on single and networked computers are similar.** KSs can run concurrently on networked computers. The behavior of the system is essentially the same as the case of a single computer (up to non-deterministic solutions which depend upon the relative timing of KS execution). The overhead of multitasking vanishes, although it is replaced by communication delays and overhead.
- **Embedded system implementations are easier to support.** For an morden embedded system, most state of the art microprocessors have hardware support of multitasking with little overhead. They do not, however, have support for specialized language features which may have been assumed in current implementations.

Communication between knowledge sources is limited to message passing, which allows maximum flexibility in the implementation; instead of considering each knowledge source as a large rule as in blackboard systems, here it can be considered as a large object capable of reasoning. We believe introducing this kind of powerful object can greatly increace the power and integrity of our framework. From one KS's point of view, all other's internal implementations are hidden. Not only can different knowledge sources use different inference schemes; they can also be implemented with different languages. A knowledge engineer or control system engineer can choose his own way of adding a piece to the whole system.

We consider each knowledge base identically in spite of the fact some of them are not intelligent systems at all. These members only do routine jobs and are considered part of the problem solving team. They may have lower priority and be subordinate to more intelligent knowledge sources. This approach puts a heavier burden on the central scheduler, but appears to be a good trade–off between efficiency and structural simplicity for our control problems. For larger applications distributed across multiple processors, this approach becomes natural and introduces no overhead in addition to that already present due to communication delays. In this way symbolic and numeric processing are easily coupled and become a powerful problem solver.

Approach and Discussion

Language

The distributed cooperating framework doesn't require the definition of a specific knowledge base description language. The KS designer can choose any conventional programming language (C,Fortran), AI language (Lisp, Prolog), or expert system development tool (CLIPS). In this case, the operating system or distributed operating system serves as scheduler and deals with messages; however, we design a new language because of the following reasons: First, our goal is to develop a tool which should provide a convenient knowledge representation, an efficient inference mechanism, and good support for debugging and maintenance. The language is an essential part of the tool to support these. It also provides a standard programming environment. Second, we want a language that can best fit our framework and real-time applications. Third, our current application is to run on a Intel 80386 under MS-DOS. We have no operating system support for multitasking. Thus we have to write a small multitasking kernel ourself. Last, in implementations on a single processor, we wish to share code segments between KS processes; it would be difficult to adapt an existing expert system shell to fit these conditions. This last consideration becomes extremely important for embedded systems, where memory space may be restricted.

The basic element of this language is the frame or object. Currently, a frame consists of only slot-value pairs. Inheritance between frames is supported. Frames have proved to be an effective way of knowledge representation. We find them also quite suitable for describing real-time control systems. Each frame belongs to a class. Classes define the structure of frames and serve as type information. This reduces the flexibility of knowledge representation but gains efficiency for the inference engine.

The language is currently a forward chaining system. Frames are treated as facts to express the current status of inference. *If ... then* rules with condition and action parts describe the inference. Rules can be assigned a priority. Pattern matching and function evaluation are closely coupled in the condition part of the rule. We deem this necessary for control system applications. Variables serve as wild cards in pattern matching. Normally the inference engine searchs through all instances of variables to test a pattern match. The user can also provide his own control of the pattern match process. The rule with the highest priority which has its condition satisfied fires by executing its action. Typical actions of a rule include assertion, retraction, or modification of frames. Other actions include sending messages to other knowledge sources. If no rule can be fired, the knowledge source will sleep, waiting for an external events (the arrival of a message) to wake it up.

The type of a frames's slot must be defined as either symbolic or numeric in its class definition. Symbolic values serve as atoms for pattern matching in inference. Floating point numeric values represent real-time data. This approach sacrifices some flexibility, but allows the language to be efficiently compiled for both symbolic and numerical processing.

The language is implemented entirely in C. Some ideas are from the CLIPS system, which allows the user to add functions written in C. The knowledge base and inference engine can be embedded in a user-written C main program. A small C-style embedded language is designed to allow the definition of functions as part of the knowledge source. These functions are compiled to an internal form, and interpreted by the inference engine; however, unlike functional programming languages, functions are statically defined and can't be mixed with data.

The language introduce a special type of *when ... then* rule to deal with asynchronous events and messages. When rules are fired not by conditions, but by events. The priority of a when-rule defines the priority of the corresponding event. The difference between a condition and an event is that a condition can change while an event is saved. Even if the condition of an *if ... then* rule has once been satisfied, it may never fire if another rule with a higher priority fires first and changes the condition. On the other hand, every event that occures will be stored in a queue and kept in the queue until it is either acted upon by a *when ... then* rule or is removed from the queue by a clear queue action. Events can arrive at any time. The arrival of message is an event; the result of a function evaluation can also be an event. This support has direct application in dangerous situations or error detection and alarm processing. Each when-rule has a corresponding event queue. The system will update the corrsponding queue each time a message arrives or a sensitive variable (one whose value contributes to a function evaluation used by the condition of a when–rule) is changed. Also note repeated satisfaction of same condition can send multiple events to the corresponding when–rule even if they are not immediately processed. This simple but effective approach gives the system its ability to deal with real–time events such as hardware interrupts.

Several special features are designed as part of the language's support for real–time applications. Issues like dealing with asynchronous events and guaranteed response time have already been discussed. This language keeps track of the time information. The following time information is kept internally:

- the time a frame was last created, used, or modified,
- the time a slot was last used or modified,
- the time a rule was last fired, and
- the time of arrival of each message.

This information can be retrieved by calling built–in functions. This gives direct support to temporal reasoning. This luxury can be disabled for efficiency. In addition, the frame design of the language is implemented in a manner which is garbage–free, eliminating the need for garbage collections and their impact upon system performance. All freed memory is incrementally collected, consuming very little overhead.

Messages

Message passing is the only means by which knowledge sources are able to communicate. In our language, messages have the same syntax as frames, although their internal data structures are different. This provides a consistent message format. However, unlike transmitting simple ASCII string as messages, as many other systems do, knowledge sources don't have to do translations. This approach requires consistency of class definitions in different knowledge sources.

Message passing between knowledge sources is done by the scheduler. Messages can be assigned with priority. When a knowledge source is going to send a message, it calls a built–in send function. The system translates the call into a request to the scheduler. The scheduler picks messages in order of their priority and sends them to their targets, perhaps via a network. The scheduler is also responsible for putting messages into receivers' event queues and waking them up. Based on the urgency of the message, the scheduler also decides whether to immediately start a knowledge source or wait until the current time slice is over. This way message management and the sleep/wake mechanism is completely transparent to knowledge sources. Knowledge sources implemented by different languages, on different machines can be treated as identical in terms of message passing.

Although knowledge sources are logically unlayered, they can be organized as a complex physical structure. Just as in the case of distributed operating system, efforts are made to hide this physical complexity so every knowledge source is accessed by name. However, the designer of the expert system should be aware of this physical arrangement. Each knowledge source is responsible for it's own work, but local ones may form a group with frequent message exchange between members while exchanging information with remote KSs ata a more leisurely rate. Close coupling with a distributed operating system and direct utilization of much of it's facilities is a characteristic of our system.

Detailed coordination protocols are left to KS designer. They can be implemented using message passing primitives. At some special occasion, knowledge sources that share memory space on the same machine (such as lightweight processes) can pass addresses as message instead of large data segments, so shared memory can also be supported.

Summary

A design for an expert sytem environment for real–time control applications has been discussed. This environment is intended to support multiple knowledge sources cooperating via message passing and residing either on a single processor or a network of processors. A language for expert system applications has been defined which integrates message passing and a class/object structure designed for control applications. The end result can be viewed as providing support for a cooperating network of intelligent objects on small platforms suitable for embedded control system implementations.

Acknowledgments

This research has been supported by the Electric Power Research Institute's Power Electronics Applications Center under contract TCRD05–CR89–D070 with the Tennessee Center for Research and Development.

References

[1] *KEE Software Development System for EXPLORER Systems.* Sperry Corporation, 1985.

[2] J. C. Giarratano, *CLIPS User's Guide, Version 4.3.* Artificial Intelligence Section, Lyndon B. Johnson Space Center, NASA, Aug. 1989.

[3] S. E. Sallé and K. Årzén, "A comparison between three development tools for real–time expert systems: Chronos, G2, and Muse," in *1989 IEEE Control Systems Society Workshop on Computer–Aided Control System Design*, pp. 50–57, IEEE, Dec. 1989.

[4] T. J. Laffey, P. A. Cox, J. L. Schmidt, S. M. Kao, and J. Y. Read, "Real–time knowledge–based systems," *AI Magazine*, vol. 9, pp. 27–45, Spring 1988.

[5] H. P. Nii, "Blackboard systems: the blackboard model of problem solving and the evolution of blackboard architectures," *AI Magazine*, vol. 7, pp. 38–53, Summer 1986.

[6] J. R. Delaney, R. T. Lacoss, and P. E. Green, "Distributed estimation in the MIT/LL DSN testbed," in *Proceedings of the American Control Conference*, pp. 305–311, AACC, June 1983.

[7] E. H. Durfee, V. R. Lesser, and D. D. Corkill, *Cooperative Distributed Problem Solving*, pp. 83–147. Vol. 4, Reading, MA: Addison–Wesley, 1989.

[8] P. E. Green, *AF: A Framework for Real–Time Distributed Cooperative Problem Solving*, pp. 153–175. London: Pitman, 1987.

Hierarchical Approach to Specification and Verification of Fault-tolerant Operating Systems

JAMES L. CALDWELL II & RICKY W. BUTLER
NASA Langley Research Center
Hampton, VA. 23665-5225

BENEDETTO L. DIVITO
Vigyan, Inc.
Hampton VA. 23666

9 June 1990

Abstract

The goal of formal methods research in the Systems Validation Methods Branch (SVMB) at NASA Langley Research Center (LaRC) is the development of design and verification methodologies to support the development of provably correct system designs for life-critical control applications. Specifically, our efforts are directed at formal specification and verification of the most critical hardware and software components of fault-tolerant fly-by-wire control systems. These systems typically have reliability requirements mandating probability of failure $< 10^{-9}$ for 10-hour mission times. To achieve these ultra-reliability requirements implies provably correct fault-tolerant designs based on replicated hardware and software resources.

1 Introduction

The application of theorem provers to verification of critical properties of real-time fault-tolerant digital systems is being explored at NASA Langley. Specifically, we are interested in fly-by-wire digital avionics systems. Typically these systems continuously read sensor values, perform computations implementing the desired control laws, and output the resulting values to actuators. Sensor values might include airspeed or input on the attitude of the aircraft. The actuators control engines, flaps, and/or rudders.

The reliability requirements for commercial aircraft are very high — probability of failure less than 10^{-9} over 10-hour mission times. This level of reliability is often referred to as *ultra-reliability.* If quantification of system reliability to this level seems a questionable endeavor, consider the problem of latent design errors. Design errors affect system reliability in unpredictable ways and measuring their effects in the lab is infeasible. In systems containing latent design errors, failures of individual replicated processors are not independent and render the reasoning behind replicated str: tegies for fault-tolerance impotent.

A current approach to solving the problem of latent design errors is based on notions of design diversity. This approach is typically implemented by independent design groups working from common specifications. However, in an often cited paper [2], Knight and Leveson have shown, at least in the software domain, that design diversity does not necessarily ensure independence of design errors. Moreover, quantification of software reliability in the ultra-reliability range is not feasible in the presence of design errors [5]. Historically, quantification of hardware unreliability due to physical failure has not been viewed as a problem and, reliability analysts assume hardware components are immune from design errors. However, as we move into the nineties, hardware description languages, silicon compilation, ASIC's, and microcoded architectures are blurring the boundaries between hardware and software development methodologies. Based on this observation, we believe caveats regarding quantification of unreliability attributable to design errors now apply to hardware as well.

Hence, we argue for verification through mathematical proof, rather than design diversity, as a partial solution to the serious problem of design errors in digital systems. Our approach is to formally specify and verify the correctness of mechanisms that implement the required fault tolerance. Management of the distributed resources that implement the required fault tolerance is a complex systems problem. Considering the obvious requirement that the voted results produced by the replicated processors must be voted in a fault-tolerant fashion seems to lead to a vicious circle. A second difficulty arises from the fact that voted results mask errors only if each replicate receives the same inputs; thus sensor values must also be distributed to each processor in a fault-tolerant manner. Ingenious algorithms have been developed to perform these tasks [4]. Verifying that these algorithms have been correctly incorporated into the fabric of a distributed operating system is at the heart of reliable fault-tolerant system design.

2 A Science of Reliable Design

Mathematical reliability models provide the foundation for a scientific approach to fault-tolerant system design. Using these models, the impact of architectural design decisions on system reliability can be analytically evaluated. Reliability analysis is based on stochastic models of fault arrival rates and system fault recovery behavior. Fault arrival rates for physical hardware devices are available from field data or empirical models [7]. The fault recovery behavior of a system is a characteristic of the fault-tolerant system architecture.

The justification for building ultra-reliable systems from replicated resources rests on an assumption of the failure independence between redundant units. The alternative approach of modeling and experimentally measuring the degree of dependence is infeasible, see [5]. The unreliability of a system of replicated components with independent probabilities of failure can easily be calculated by multiplying the individual probabilities. Thus, the assumption of independence allows fault-tolerant system designers to obtain ultra-reliable designs using moderately reliable parts. Often complex systems are constructed from several ultra-reliable subsystems. The subsystem interdependences (e.g. due to shared memories, shared power supplies, etc.), can still be modeled (assuming perfect knowledge about the failure dependencies) and the system reliability can be computed. Of course, the reliability model can become very complex.

The validity of the reliability model depends critically upon the *correctness* of the software and hardware that implements the fault tolerance of the system. If there are errors in the logical design or implementation of the fault-recovery strategy or in the design of individual system components, failures between redundant units may no longer be independent. The quantification of system unreliability due to physical failure would be meaningless.

Based on this analysis, the validation of the reliability of life-critical systems can be decomposed into two major tasks:

- Establishing that design errors are not present.
- Quantifying the probability of system failure due to physical failure.

The first task is addressed by formal specification and mathematical proof of correctness. The second task is addressed by the use of reliability analysis models and tools to analytically evaluate the effects of individual component failure rates on the overall system reliability.

3 Formal Methods

The major difference between the approach advocated here and approaches used for design of more traditional fault-tolerant operating systems is in the application of formal methods. This approach is borne from the belief that the successful engineering of complex computing systems requires the application of ***mathematically based analysis*** analogous to the structural analysis performed before a bridge or airplane wing is built. The mathematics for the design of a software system is ***logic***, just as calculus and differential equations are the mathematical tools used in other engineering fields.

The application of formal methods to a development effort are characterized by the following steps.

1. Formalization of the set of *assumptions* characterizing the intended environment in which the system is to operate. This is typically a conjunction of clauses $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ where each A_i captures some constraint on the intended environment. Typically $\mathcal{A}$ has many models although the author of a specification generally has a particular model in mind.

2. The second step is the formal characterization of the system *specification* in the formal theory. This is a statement $\mathcal{S}$ characterizing the properties which any implementation must satisfy.

3. The third step is formalization in the theory of an *implementation* $\mathcal{I}$. Typically, an implementation is a decomposition of the specification to a more detailed level of specification. In a hierarchical design process there may be a number of implementations, each more detailed than its specification.

4. The final stage is a proof that the implementation $\mathcal{I}$ satisfies the specification $\mathcal{S}$ under the assumptions $\mathcal{A}$. Formally, this is a *proof* of the statement $\mathcal{A} \supset (\mathcal{I} \supset \mathcal{S})$, where $\supset$ denotes logical implication. That is, under any model of $\mathcal{A}$, $\mathcal{I}$ is an implementation of the specification $\mathcal{S}$.

Some comments are in order. If the set of assumptions proves to be inconsistent, i.e. there is no model of $\mathcal{A}$, then any implementation satisfies all specifications and the entire effort is in vain. This suggests a strategy of minimizing both the number and complexity of the assumptions. The assumptions can be seen as constraints on the operating environment in which the specified component is to be placed.

The author of the formalizations typically has some specific model in mind which he is trying to characterize in the formal statements $\mathcal{A}$, $\mathcal{S}$, and $\mathcal{I}$. From the perspective of methodology, it is a good idea to prove some *putative theorems* about these statements to ensure that the intended model has been faithfully captured. For example, in a formal characterization of a memory, say $\mathcal{M}$, it is important to ensure that the specification correctly captures notions of reading and writing. One property of interest might be that reading the contents of address a at times t_1 and t_2 will yield the same value, v, as long as there is no write to a of a value $u, u \neq v$, during the interval (t_1,t_2). This property should surely hold in any model of $\mathcal{M}$. Proving such a theorem builds confidence that $\mathcal{M}$ correctly characterizes the intended models.

It should be noted that, strictly speaking, this property could *only* be shown by reasoning about the specification; no amount of testing can establish that this property holds. In fact, many of the properties of interest in fault-tolerant design are within the domain of formal methods and their verification depends on reasoning as opposed to testing-based approaches. The existence of formal characterizations of a system provides a basis for such reasoning.

3.1 Hierarchical Proof

The methodology outlined here is inherently hierarchical. Under the assumptions $\mathcal{A}$, if implementation $\mathcal{I}_1$ is shown to be an implementation of a specification $\mathcal{S}$ and $\mathcal{I}_2$ is shown to be an implementation of $\mathcal{I}_1$ we conclude that $\mathcal{I}_2$ is also an implementation of $\mathcal{S}$. The sentence above can be formally restated as an inference rule:

$$\frac{\mathcal{A} \supset (\mathcal{I}_2 \supset \mathcal{I}_1),\ \mathcal{A} \supset (\mathcal{I}_1 \supset \mathcal{S})}{\mathcal{A} \supset (\mathcal{I}_2 \supset \mathcal{S})}$$

Logically, this is a simple consequence of the transitivity of implication. Its significance for a hierarchical verification strategy is obvious; it provides formal justification for linking together a chain of formal proofs of correctness to show the lowest level decomposition of a series of decompositions is an implementation of the original specification.

3.2 Levels of Application

Formal methods are the applied mathematics of computer systems engineering. In other engineering fields, applied mathematics are utilized to the extent that they are required to achieve acceptable levels of assurance for safety, performance, or reliability. It is often assumed that the application of formal methods is an "all or nothing" affair. This is not the case. There is a useful taxonomy of *levels of application* identified here.

0. No application of formal methods.
1. Formal Specification of all or part of the system.
2. Paper and pencil proof of correctness.
3. Formal proof checked by mechanical theorem prover.

Significant gains in assurance are possible in existing design methodologies by formalizing the assumptions and constraints, the specification, and the implementation. Experience shows that application of *level 1* alone often reveals inconsistencies and subtle errors that might not be caught until much later in the development process, if at all. It is generally accepted that the later a design error is identified the more costly is its repair, therefore this level of application can provide significant benefits.

The use of paper and pencil proof in the design process adds another level of assurance in design correctness. *Level 2* application forces explicit consideration of the relationships between the implementation and the specification and often reveals forgotten assumptions or incorrect formalizations.

A proof of correctness is only as good as the prover. Even stronger evidence for correctness can be established by forcing proofs through a mechanical theorem prover. This is *level 3* application of formal methods. It must be noted that there is no guarantee that the implementation of the mechanical prover is correct or that the hardware on which the mechanical verification was performed was not faulty. Thus, there is no absolute guarantee of the correctness of an implementation even after a mechanical proof has been performed. What is gained by the additional effort is a detailed argument for the correctness of the implementation. The process of "convincing" a mechanical prover is really a process of developing an argument for an ultimate skeptic who must be shown every detail.

Partial application of any of the levels is possible for different parts of the system. We advocate the application of level 3 formal methods only for the most critical (and hopefully reusable) system components. What is classified here as level 1 and level 2 formal methods are being widely applied in the U.K.

4 Architectural Approach

In our research at Langley on provably correct fault-tolerant control systems we consider architectures consisting of four or more electrically isolated processors that can communicate with one another. Typically these systems run with a static multi-rate schedule with tasks scheduled periodically. Each processor synchronously executes the same schedule, and the system votes all actuator outputs to mask individual processor faults. The fault models used are worst case models in which faulty processors can maliciously co-operate in attempts to defeat the fault tolerance of the system. Under this worst case model, $3m + 1$ processors must be working in order to tolerate m faults [1]. If we assume the existence of a fault-tolerant basis providing clock synchronization and interactive consistency, then a simple majority of working processors suffices to out vote any minority of faulty processors.

Empirical evidence indicates that transient faults are significantly more common than permanent faults. If designed correctly, these systems are able to recover gracefully from transient faults. Each computation generally only depends on a short part of the input history and typically has a minimal amount of global state information. If the global state is voted periodically and internal state is recoverable from sensors, it is clear that after some finite time errors can be flushed from the system.

The approach adopted here for the design of the distributed aspect of the system is motivated by Lamport's paper [3]. At the base of the system is a distributed clock synchronization algorithm, allowing the system to be viewed as a synchronous system. Under contract to NASA, Rushby and von Henke [6] formally verified Lamport and Melliar-Smith's [4] clock synchronization algorithm[1] providing a key system building block. In a system relying on exact match voting it must also be ensured that each processor receives the same inputs from the sensors. This is accomplished by a Byzantine resilient interactive consistency algorithm running on the distributed system. With these algorithms as a base, the voter ensures that as long as a majority of the processors are working then the replicated system produces the same results as an *ideal* non-faulty processor would.

[1]Interestingly, they found at least one error in the published proof that had remained undiscovered through the social review process.

5 Conclusion

It has been argued that quantification of system reliability in the ultra-reliable range depends on the provably correct implementation of fault tolerance. Absolute correctness is unattainable. However, formal methods provide added assurance of correctness by forcing detailed consideration of the assumptions, the specification, and the implementation in a formal setting. Hierarchical design proofs provide a formal framework to allow consideration of these details at the appropriate level of abstraction. These methods are being applied in research efforts underway at NASA LaRC. A NASA technical report outlining the first phase of design specification and proof of a fault-tolerant operating system for control applications will be available in the near future.

References

[1] D. Dolev, J. Y. Halpern, and H. R. Strong. On the possibility and impossibility of achieving clock synchronization. In *Proceedings of 16th Annual ACM Symposium on Theory of Computing*, pages 504–511, Washington, D.C., April 1984.

[2] J. C. Knight and N. G. Levenson. An experimental evaluation of the assumptions of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, Jan 1986.

[3] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems*, 6(2):254–280, April 1984.

[4] Leslie Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1987.

[5] Doug Miller. Making statistical inferences about software reliability. Technical Report CR-4197, NASA, December 1988.

[6] John Rushby and Friedrich von Henke. Formal verification of a fault tolerant clock synchronization algorithm. Technical Report 4239, NASA, June 1989. Contractor Report.

[7] *Reliability Prediction of Electronic Equipment.* U.S. Department of Defense, January 1982. MIL-HDBK-217D.

An Emulation/Simulation Environment For Intelligent Controls

Dr. N. Coleman
U.S. Army Armament Research, Development and Engineering Center
Picatinny Arsenal, NJ 07806-5000

ABSTRACT: This paper describes a rapid prototyping tool for intelligent control system software development which supports both knowledge based simulation and real time emulation capability. The tool was developed to help bridge the gap between the disciplines of artificial intelligence and control system theory by providing a system architecture and software development environment compatible with both low level, high bandwidth control applications and higher level low bandwidth cognitive processes involving perceptual reasoning and planning. A simple example illustrating an application of the tool in support of on-going weapon platform automation research within the Armament Research and Development Center is presented.

Introduction: There has been a growing interest within the Army and the DOD community over the past several years in the areas of artificial intelligence and robotics and the application of these technologies to enhance the performance and effectiveness of future armament systems while reducing development and operational cost. One means of achieving this goal is through "intelligent" task automation of on-board crew functions thereby permitting reduction and eventual elimination of crew requirements for certain high risk, limited duration missions. The levels of automation required dictate the need for highly sophisticated on-board real time expert systems which are tightly coupled and fully integrated (in a closed loop sense) with on-board sensors (i.e., trackers, radar, flir, etc.) and actuation devices (i.e., weapon/sensor controls, vehicle controls, loading devices, etc.). The purpose of this paper is to discuss progress on two aspects of this "intelligent" automation issue, namely (1) the definition of a platform control architecture and (2) software development tools required to support prototyping experimentation and evaluation of advanced automation software.

Hierarchical Control Architecture: The system architecture selected to provide the basic framework for intelligent control/automation, is based on concepts originally developed by Albus. The Control structure, as originally described, consists of a hierarchy of finite-state machine modules, each of which implements a local hybrid-state feedback controller corresponding to the performance of a particular subtask. The inputs to each local controller consist of a command input from the module above and acknowledgements from the modules below it in the hierarchy; the outputs consist of an acknowledgement/error message to the module above and commands to the modules below. The lowest level modules are interfaced to the physical environment through sensors and actuators as shown in Figure 1. The nominal actions of the controller are to (a) decode the command input, (b) properly command and synchronize the concurrent actions of the modules below it, and (c) generate an acknowledgement when the subtask is completed. If off-nominal conditions occur, the module must determine whether it can control the resources to take corrective actions; if so, the actions are taken; if not, an error signal to the next highest module is generated. An analysis of the finite-state case of this architecture has been given in 2, 6, and 8.

Generalizations of the original modular hierarchical control concept are necessary in order to support the requirements dictated by intelligent/autonomous military systems.

o Ability of the module hierarchy to reconfigure itself

o Incorporation of task planning/replanning and perceptual reasoning in some (higher level) modules; enhansed error analysis capabilities

o Direct interfaces between some higher level modules and the physical environment (e.g. intelligent tracker or sensor based robotic loader)

o Provision of knowledge bases

o Multiple communication channels between modules (e.g., message passing as well as common memory)

All of these features have been incorporated into the extended architecture and fully supported by the software development tool described below.

Laying out the control hierarchy for a particular task is to a large extent, ad hoc due to the lack of a comprehensive theory of intelligent control. Some of the parameters which must be defined by the designer are:

o Number of levels

o Number of independent concurrent processes at each level

o Connectivity, maximum branching factor

o Information rates between modules (messages/channel/time unit)

o Processing capacity (computation/node/time unit)

o Closed-loop response time (time-scale for each level)

These parameters should be selected so as to balance the response time scales and computation loads at each level of the hierarchy.

Simulation/Emulation Environment: The inherent high cost and complexity of embedded software for future battlefield robotic systems and automated weapon platforms makes the need for powerful development, evaluation, validation and rapid prototyping tools absolutely essential. The modular hierarchical control approach is well suited to rapid software development in a multi-programmer environment, to interface standardization and to multi processor implementation. Some of the important features associated with this first generation prototyping tool for intelligent controls are:

o Task/subtask layout capability

o Interactive graphics tool for design and implementation of Finite State Machine Modules (FSM). This tool is hosted on Symbolics 3675 and generates FSM files which are communicated over ethernet to a VAX 750 which executes the State Machine Hierarchy

o Assignment of modules to processors or emulated processors. Ability to examine alternate network configurations

o Emulation of a hierarchical control system

o Simulation of the environment and controlled subsystems (terrain, sensors, tactics, weapon platforms, visibility, etc.)

o Emulation of hierarchical control systems and multiple cooperating systems

o Knowledge based simulation of hostile forces

o Graphics display capability (map, system status displays)

o Interface with digital terrain data base

o On-line monitoring and modification of control system status for debugging purposes

o Off-line performance analysis and evaluation

o Provision for emulation to drive physical subsystems

o Incorporation of prerecorded field test data

o Knowledge base interface for control system emulation and simulation (e.g., interface with tactical knowledge bases and knowledge based threat scenario simulation, etc.)

o Incorporation of procedure libraries with different source languages (LISP, Flavors, C, PASCAL, etc.)

The structure of the emulation is shown in figure 2. Its use involves three stages: (a) laying out modules and specifying control systems and simulation logic; (b) running the emulation, monitoring control system status and debugging; (c) port processing for performance analysis and evaluation. The next section describes a prototype application developed with the emulation tool.

<u>Example</u>: Figure 3 illustrates the major elements of a platoon level control emulation consisting of a world model terrain data base, blue tank platoon, red tank platoon and two expert system modules for mission planning and threat/situation assessment. The world model coordinates all elements of the tactical simulation including tank dynamic models, sensor models, terrain models, turret stabilization models, knowledge bases and tactic for red platoon, etc. The blue tank platoon commander module emulates the command and control functions of the blue tank platoon leader including mission planning, engagement planning, route planning, command and control functions and threat assessment. This module interfaces directly with the route/mission planning expert system and the situation assessment expert system running on a Symbolics 3675 with low resolution color display. A simplified state transition diagram for the platoon leader module is shown in fig 4. Each tank commander module communicates, coordinates and controls the functions of a gunner, loader and driver module as shown in fig. 5. The driver module controls a simulated tank (speed and direction), the gunner module controls a simulated tank turret and interfaces also with a laboratory automatic target recognition (ATR) and tracker subsystem. The loader FSM module interfaces with and controls a Puma 560 robot which is configured as shown in fig. 6. The Puma 560 recieves a load command from the loader FSM and uses its own vision sensor to determine locating and orientation of projectiles and breech. It uses a Lord force/torque sensor for active compliance control during loading and a polaroid range sensor for obstacle avoidance. The loader hardware configuration is shown in fig. 7.

Although detailed discussions of the expert system modules within the emulation is beyond the scope of this paper, a few comments will be made for purpose of clarification. The route planning module is somewhat unique in its hierarchical structure which uses object oriented programming (Flavors) to generate a family of feasible routes from the current tank location to the objective location. A rule based expert system is used to reason about and evaluate the feasible paths based on tactical considerations, mission requirements and resource constraints. Fig. 8 gives a view of the commanders display showing the planned route for the commanded blue tank and a projected route for a hostile red tank based on expert knowledge of terrain and tactics.

Conclusion: Future weapon platforms will be required to operate in increasingly hostile environments with fewer crew members and ultimately must be capable of operation with minimal operator interaction or intervention. Powerful and general purpose software prototyping and development tools will be required to design such systems. The Intelligent Hierarchical Control System Emulation described in this paper is a first generation tool of this type. Hierarchical Intelligent Weapon Platform Control Concepts have been demonstrated in a prototype system which is capable of autonomous operation.

REFERENCES

1. Albus, J. S., Barbera, A. J. and Nagel, R. N. "Theory and Practice of Hierarchical Control", Proc. 23rd IEEE Computer Society International Conference, Washington, D.C., 1981.

2. Johnson, T. L., "Hierarchical Decision-In-The-Loop Processes", Proc. Oakland University, Conference On Artificial Intelligence, Rochester, MI, April 1983.

3. N. Coleman, T. Johnson, "A Hierarchical Control Architecture for Autonomous Systems", Proc. Oakland University Conference on Artificial Intelligence, Rochester, MI, April 1984.

4. N. Coleman, "An Approach to Intelligent Weapon Platform Automation", ADPA Fire Control Symposium, April 1984.

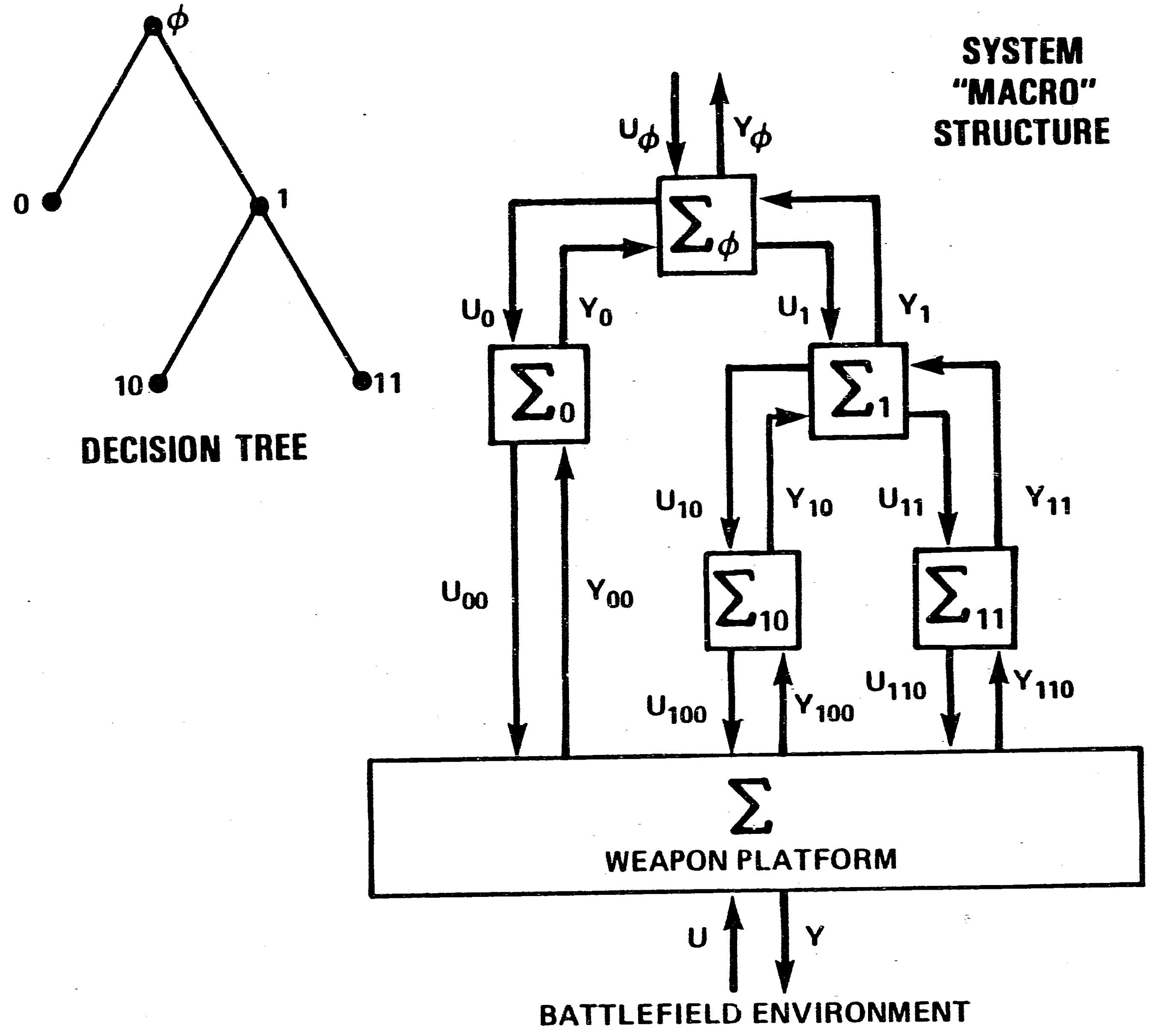
5. N. Coleman, S. Redington, "Hierarchical Intelligent Control", ARDC Report SCS-C-IR(SC) 83-005.

6. N. Coleman, "A Hierarchical Control Architecture for Intelligent Systems", ARDC Report 84-007.

7. N. Coleman, "A Hierarchical Approach to Intelligent Weapon Control", Proc. IEEE/NSF Workshop on Intelligent Control, RPI, April 1985.

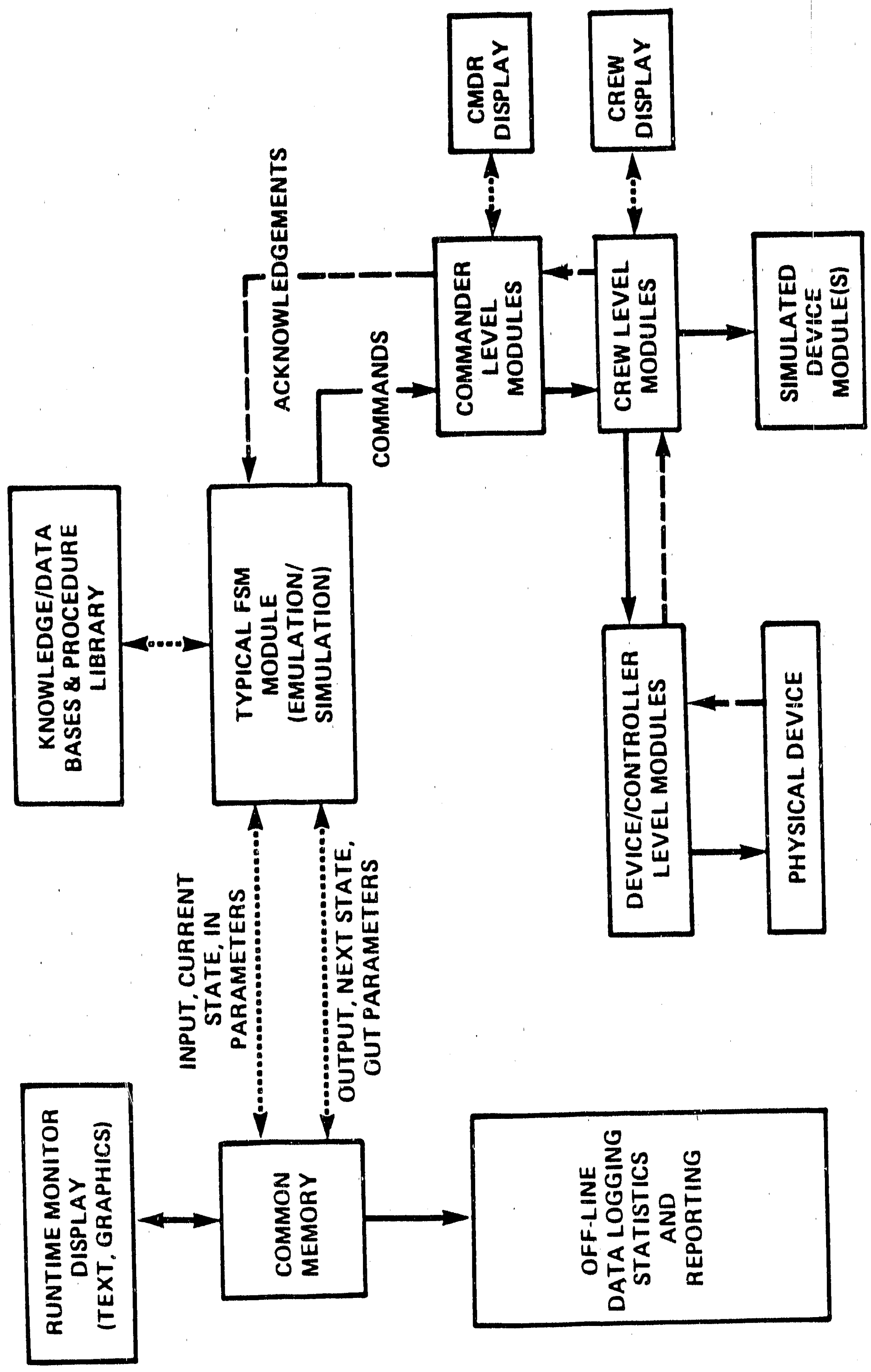
8. N. Coleman, "An Approach to Intelligent Weapon Platform Automation", WESCON, November 1985.

9. N. Coleman, S. Milligan, "Planning For Autonomous Weapon Control", Proc. Oakland University Conference on Int. Machines and Systems, April 1985.



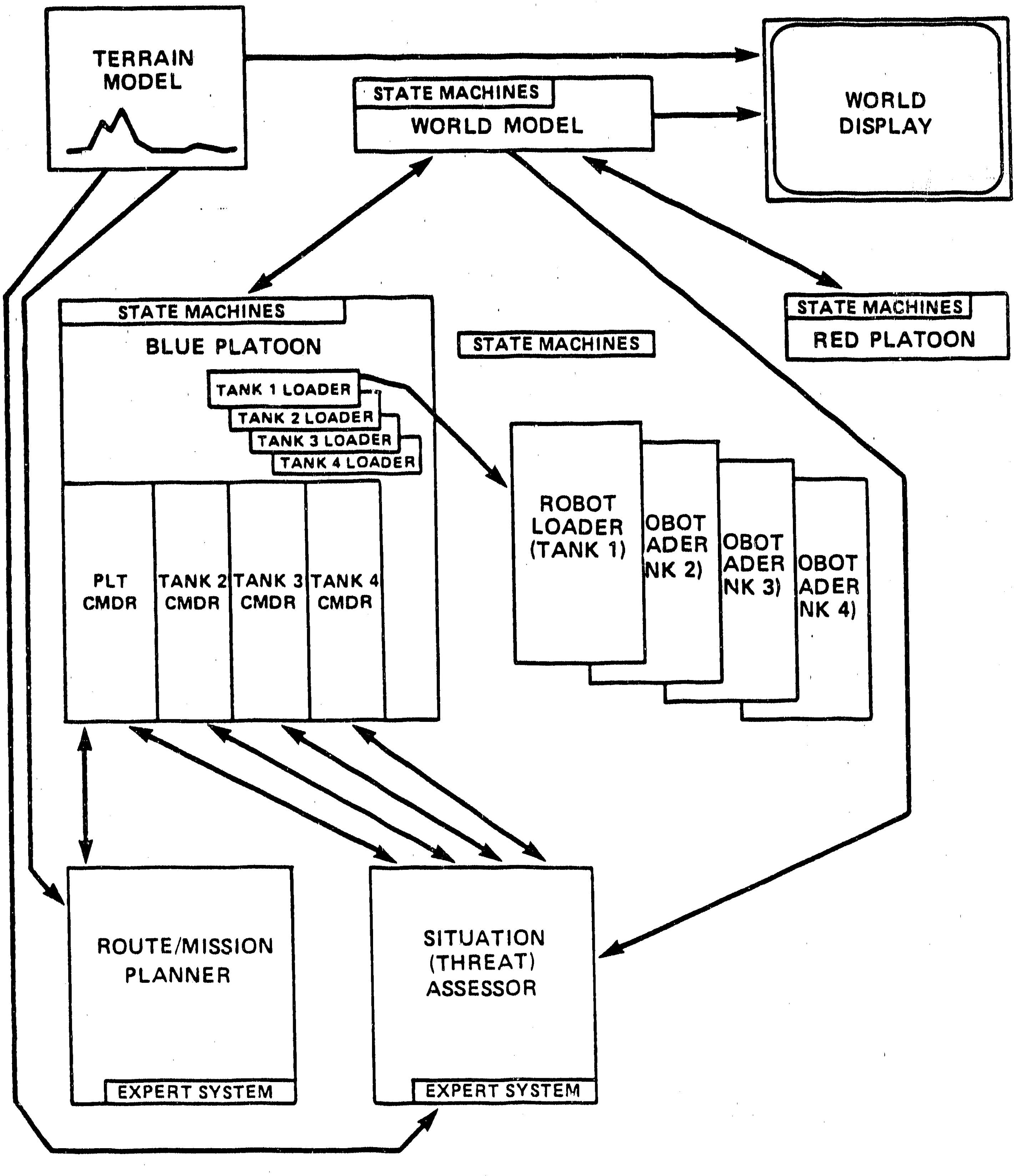
Modular Hierarchial FSM Control Architecture

Figure 1


RUNTIME MONITOR DISPLAY (TEXT, GRAPHICS)
COMMON MEMORY
OFF-LINE DATA LOGGING STATISTICS AND REPORTING
INPUT, CURRENT STATE, IN PARAMETERS
OUTPUT, NEXT STATE, OUT PARAMETERS
KNOWLEDGE/DATA BASES & PROCEDURE LIBRARY
TYPICAL FSM MODULE (EMULATION/ SIMULATION)
DEVICE/CONTROLLER LEVEL MODULES
PHYSICAL DEVICE
ACKNOWLEDGEMENTS
COMMANDS
COMMANDER LEVEL MODULES
CREW LEVEL MODULES
SIMULATED DEVICE MODULE(S)
CMDR DISPLAY
CREW DISPLAY

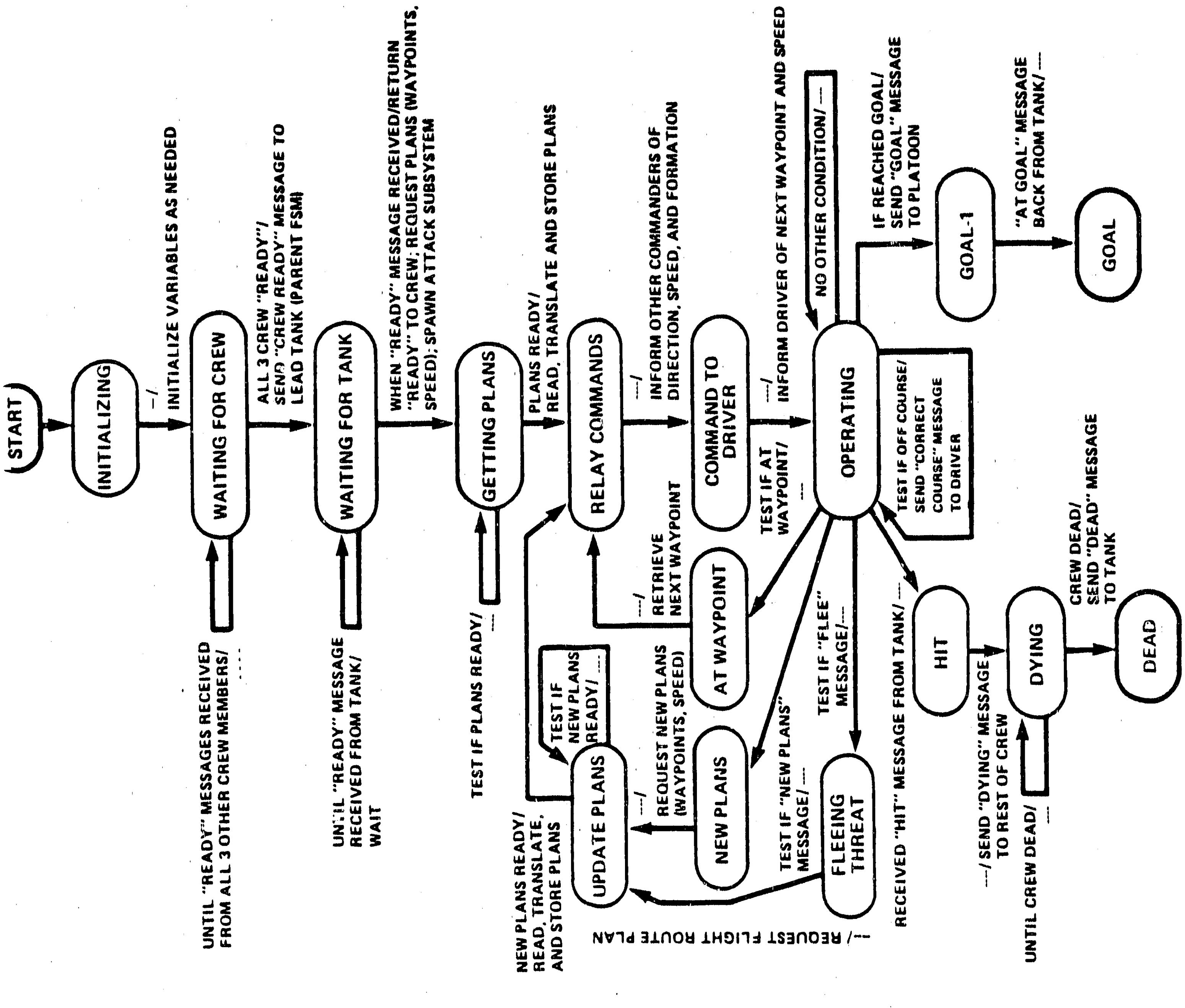
Hierarchial Control Emulation Structure

Figure 2

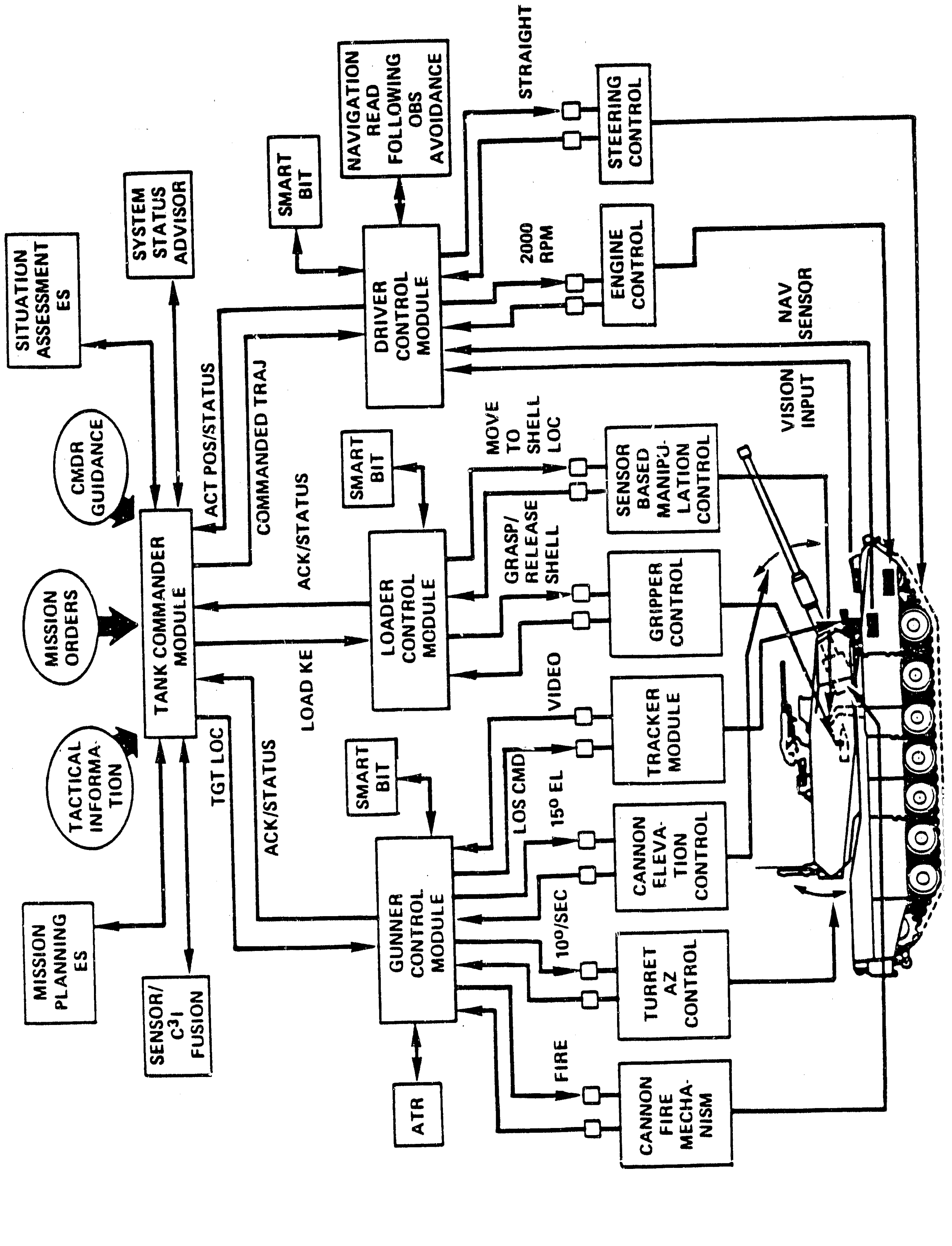


Simulation/Emulation Architecture

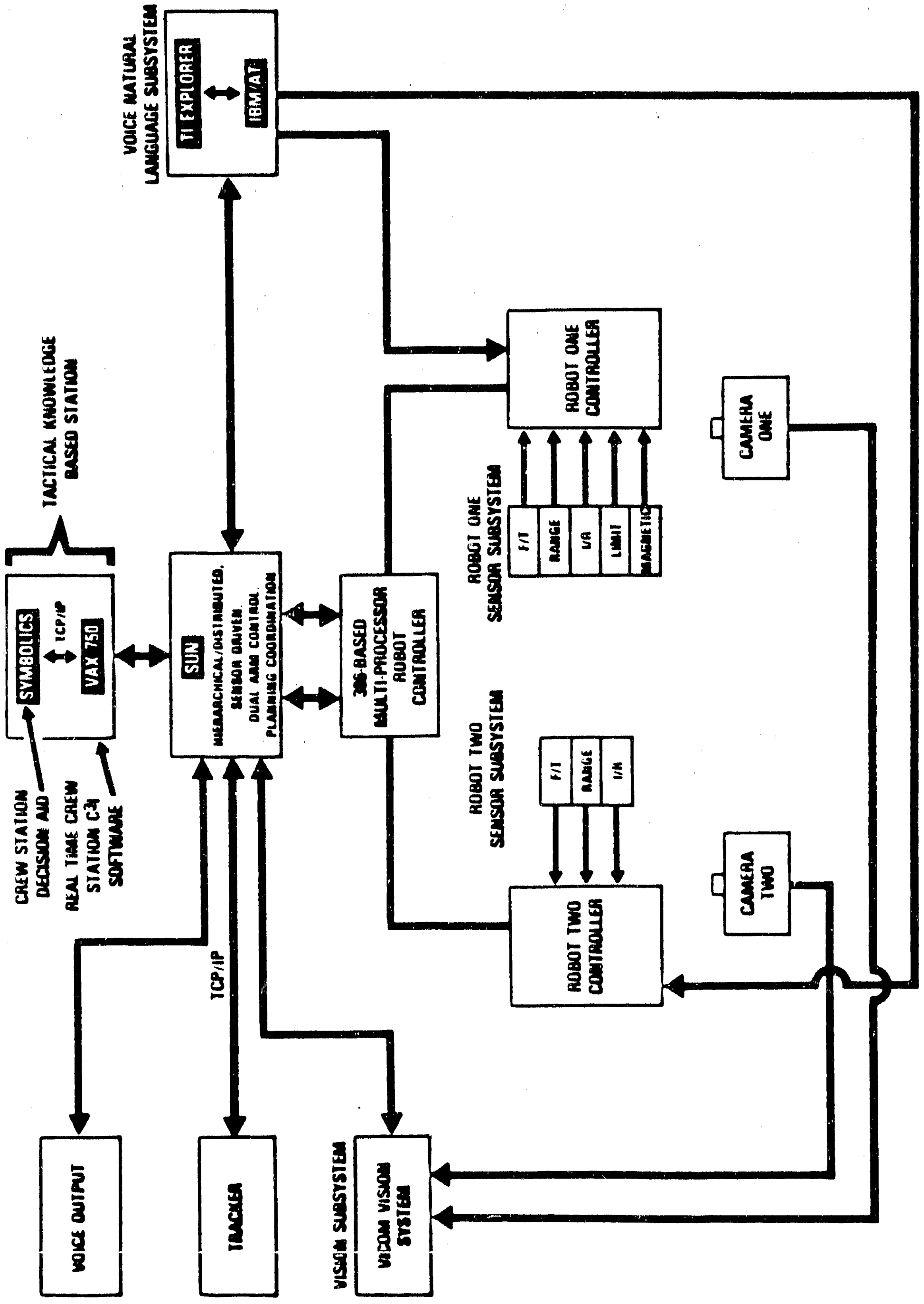
Figure 3


START
INITIALIZING
—/ INITIALIZE VARIABLES AS NEEDED
WAITING FOR CREW
UNTIL "READY" MESSAGES RECEIVED FROM ALL 3 OTHER CREW MEMBERS/ ...
ALL 3 CREW "READY"/ SEND "CREW READY" MESSAGE TO LEAD TANK (PARENT FSM)
WAITING FOR TANK
UNTIL "READY" MESSAGE RECEIVED FROM TANK/ WAIT
WHEN "READY" MESSAGE RECEIVED/RETURN "READY" TO CREW; REQUEST PLANS (WAYPOINTS, SPEED); SPAWN ATTACK SUBSYSTEM
GETTING PLANS
TEST IF PLANS READY/ —
PLANS READY/ READ, TRANSLATE AND STORE PLANS
RELAY COMMANDS
—/ INFORM OTHER COMMANDERS OF DIRECTION, SPEED, AND FORMATION
COMMAND TO DRIVER
—/ INFORM DRIVER OF NEXT WAYPOINT AND SPEED
OPERATING
NO OTHER CONDITION/ —
IF REACHED GOAL/ SEND "GOAL" MESSAGE TO PLATOON
GOAL-1
"AT GOAL" MESSAGE BACK FROM TANK/ —
GOAL
TEST IF OFF COURSE/ SEND "CORRECT COURSE" MESSAGE TO DRIVER
TEST IF AT WAYPOINT/ —
AT WAYPOINT
—/ RETRIEVE NEXT WAYPOINT
TEST IF "NEW PLANS" MESSAGE/ —
NEW PLANS
—/ REQUEST NEW PLANS (WAYPOINTS, SPEED)
UPDATE PLANS
TEST IF NEW PLANS READY/ —
NEW PLANS READY/ READ, TRANSLATE, AND STORE PLANS
TEST IF "FLEE" MESSAGE/ —
FLEEING THREAT
—/ REQUEST FLIGHT ROUTE PLAN
RECEIVED "HIT" MESSAGE FROM TANK/ —
HIT
—/ SEND "DYING" MESSAGE TO REST OF CREW
DYING
UNTIL CREW DEAD/ —
CREW DEAD/ SEND "DEAD" MESSAGE TO TANK
DEAD

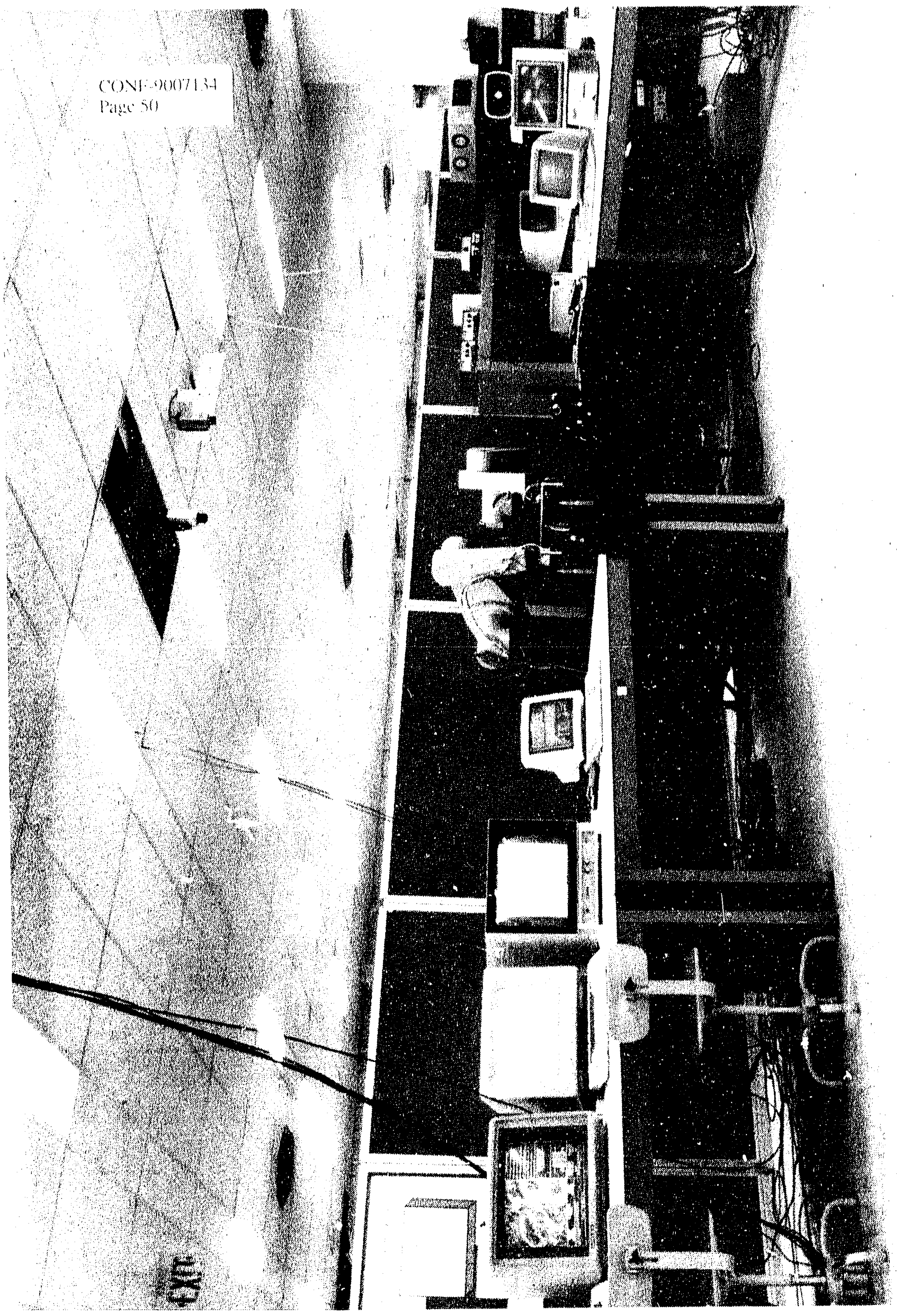
Intelligent Weapon Platform Automation



CREW STATION AUTOMATION ARCHITECTURE


VOICE NATURAL LANGUAGE SUBSYSTEM
TI EXPLORER
IBM/AT
TACTICAL KNOWLEDGE BASED STATION
SYMBOLICS
TCP/IP
VAX 750
CREW STATION DECISION AID
REAL TIME CREW STATION C3 SOFTWARE
SUN
HIERARCHICAL/DISTRIBUTED, SENSOR DRIVEN, DUAL ARM CONTROL, PLANNING COORDINATION
386-BASED MULTI-PROCESSOR ROBOT CONTROLLER
ROBOT ONE CONTROLLER
ROBOT ONE SENSOR SUBSYSTEM
F/T
RANGE
I/A
LIMIT
MAGNETIC
ROBOT TWO SENSOR SUBSYSTEM
F/T
RANGE
I/A
ROBOT TWO CONTROLLER
CAMERA ONE
CAMERA TWO
TCP/IP
VOICE OUTPUT
TRACKER
VISION SUBSYSTEM
VICOM VISION SYSTEM

CONF-9007134
Page 50



Software Engineering Environments for Military Vehicle Management Systems

Prof. Michael Fehling
Laboratory for Intelligent Systems
321 Terman Center
Stanford University
Stanford, CA 94305-4025

Dr. Charles F. Hall
Lockheed Artificial Intelligence Center
O/96-20 Bldg. 259
3251 Hanover Street
Palo Alto, CA 94034

Abstract

The costs associated with software development and maintenance seem to increase without bound on some programs. This has led to comparing software to entropy. It has been proposed that a way to control software costs is to provide an appropriate software engineering environment (SEE) and make sure that it is used throughout the program life cycle. This approach has been successful in some cases and in others disastrous. Three major issues are 1) all software development programs do not fit neatly into a single space, 2) what is worse, we do not seem to have as yet a methodology for matching program requirements to software environment attributes, and 3) programmers like to use what they like to use. We feel that any attempt to develop a standardized software engineering environment must address all three of these issues if it is to be accepted and successful. Based on these issues, the problem can be partitioned into three tasks 1) developing a taxonomy of problem domain characteristics that can be used to partition the problem space, 2) developing a taxonomy of SEE attributes and appropriate benchmarks and tests that can be used to define an environment given a particular problem, and 3) defining a generic framework based on accepted software standards (e.g., X-windows, Ada, Etc.) within which system developers can, and will, want to work. This position paper will discuss how some of these issues are being addressed for a specific military vehicle domain, the autonomous control of underwater vehicles.

Introduction

The main emphasis at this workshop is the development of a software engineering environment (SEE) for building military vehicle intelligent control systems. The goal is to improve the productivity of control design engineers and lower the total life cycle costs to DoD. The issues go beyond a suitable set of tools for building distributed intelligent control systems. The total software life cycle, appropriate methodologies, standards, and computer-aided software engineering (CASE) tools need to be considered.

Given that an appropriate life cycle model and SEE have been defined, what specializations -- if any -- are required for specific application domains of interest? In particular, are there characteristics of problem domains that can be used to partition the space of all problem domains? We believe the answer is yes. Moreover, there will be sets of overlapping characteristics common to sets of domains, as well as characteristics unique to specific domains. Given such a partitioning, the question becomes, can a mapping from domain characteristics to SEE attributes be found? If so, a set of specialization requirements for a particular SEE can then be defined.

How can the problem domain partitioning and the requirements to attributes mapping be found? An attempt to take a top down approach by defining characteristics of all problems and breaking that into subsets, etc., seems to be a very difficult task. We propose therefore, a bottoms up approach in which a specific problem is chosen to constrain the solution. Once this domain is thoroughly understood and an appropriate SEE and set of tools is defined, that solution could then be generalized.

Since the emphasis at this workshop is intelligent control systems for military vehicles we have selected a specific instance of this general domain -- the autonomous control of underwater vehicles.

In the next section we will briefly review some of the software life cycle and general SEE issues. Then we shall discuss some of the special issues that are associated with the engineering of intelligent control systems. This will be followed by a discussion of the Schemer architecture, a distributed control executive for real-time control. We will then discuss how we are using Schemer to prototype a control system for an autonomous underwater vehicle and then make some concluding remarks.

Software Engineering and Life-Cycle Issues

The software life cycle can be broken down into three phases: definition, development, and maintenance. A complete software engineering environment (SEE) should support the development cycle through all three phases. Moreover, there are several different procedural approaches that may be taken to software engineering. The classic life cycle approach -- or "waterfall model" -- is shown in Figure 1. This approach is characterized by little or no formal feedback from step to step. It also requires, in its purest implementations, the freezing of requirements and design specifications at an early stage of the life cycle. This approach is appropriate when the requirements are completely specified and the problem is well defined. It has been successfully used for some very complex and ill defined system developments; but, only under carefully managed conditions. A poorly managed software development

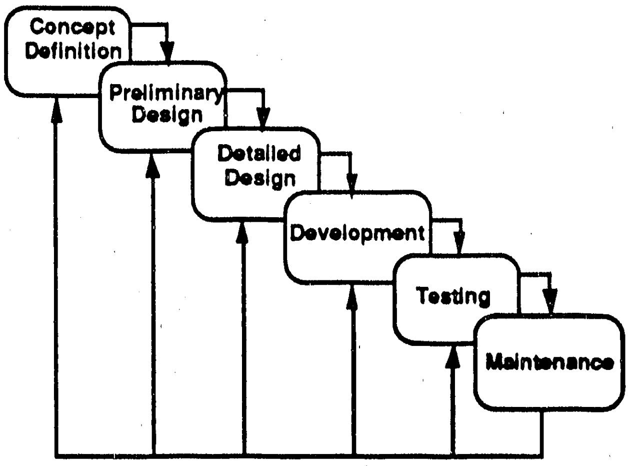


Figure 1. Standard S/W Development Life Cycle

team can produce disastrous results using this model.

A life cycle paradigm that circumvents some of the problems of the classic apprcach is the evolutionary or rapid prototyping model. There are several variations of this model, one of which is shown in Figure 2. The two key aspects are the tightly coupled iteration between the concept definition and rapid prototyping phases; and, the multiple feedback paths that occur in each step of the process. The former ensures a "quick" convergence to a demonstrated agreed upon set of requirements that can then be passed on to the detailed design phase. The latter

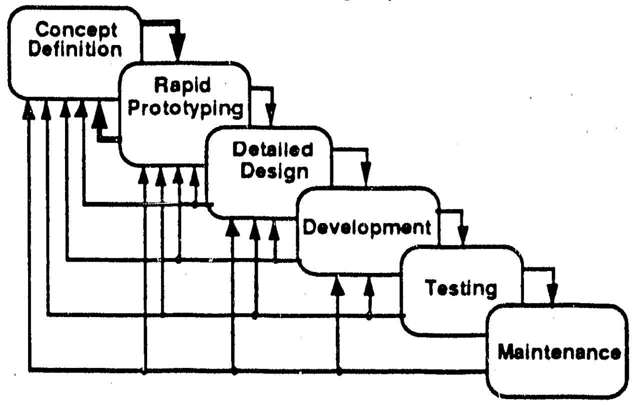


Figure 2. Evolutionary Model

ensures that potential "gotchas"are identified and fed back into the process at the appropriate level as soon as possible. Ideally, the long feedback paths will have little traffic. In general, this paradigm works best when heavy human-computer interactions are required, when complex output is to be produced, or when new or untested algorithms are to be supplied. It is less beneficial for large, batch-oriented processing or embedded process control applications. As the case with the classic model, this approach provides substantial benefit when properly applied and managed. It should be viewed as complementary to the classic life cycle model.

Regardless of the life cycle model used, one should select a software engineering environment (SEE) that supports not only the creation and manipulation of source code, but methodology as well. That is, the environment should provide assistance for software requirements analysis, design, and test, as well as aids for project planning, tracking, and control. In addition, a desktop publishing capability should be provided for efficient production of high quality documentation. Such a computer-aided software engineering (CASE) system is a software engineer's assistant, taking the drudgery out of software engineering that leads to low productivity and quality. The selected SEE needs to be built on top of, and be compatible with, existing standards to insure portability, maintainability, and interoperability.

Engineering Intelligent Control Systems

To summarize, our view of the software engineering issues entails three principal areas of concern — (1) defining a computational architecture that supports an appropriate generic approach to the performance and other requirements of the application domain, (2) providing a SEE that encourages, and even enforces, use of an appropriate engineering methodology both in general, and tailored to the special needs of the application domain, and (3) supporting general and domain-specific aspects of life-cycle maintenance. Although limitations of space preclude a detailed discussion, let us briefly review some of the more salient issues in each of these three areas for the focus of this workshop, software for the intelligent control of complex, military vehicles.

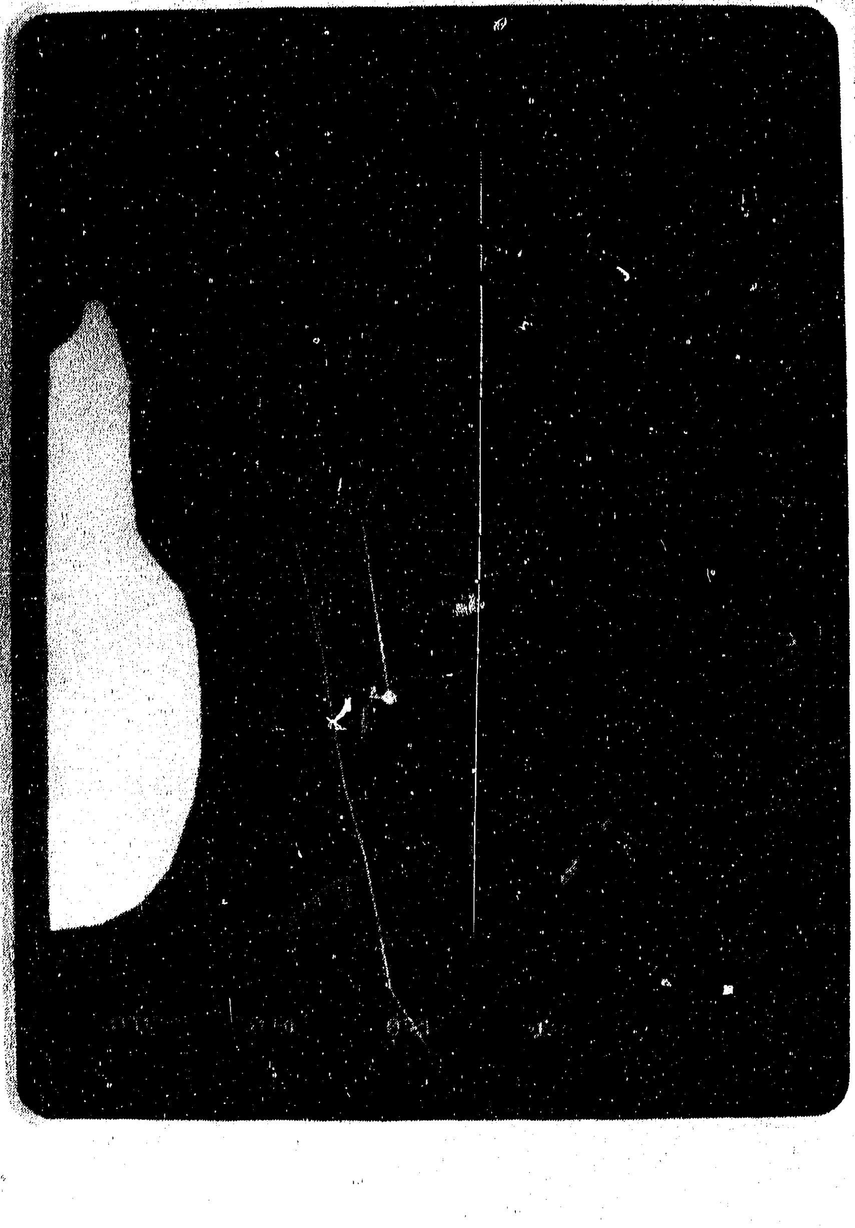
Functional Requirements:

We believe that the research and development community can avail itself of a reasonably mature view of the functional requirements of intelligent vehicular-control systems. These functional requirements derive from the need of these control systems to interact effectively and efficiently with conditions in the complex, dynamic environments in which the vehicle must perform. These dynamic conditions constrain (and sometimes serendipidously enhance) system performance, including operations of the control software itself. In other papers, one of us (Fehling) has examined and analyzed a number of important functional requirements that are imposed on "intelligent control" software intended to manage complex systems that are embedded in, and interact with, real-world domains. The following are among the most important of these requirements:

• *Real-time performance* — Intelligent vehicular-control systems must enable the vehicle's prompt reaction to, and interaction with, its environment. Fehling and his colleagues (e.g., Fehling et al., 1986; Fehling & D'Ambrosio, 1990) have discussed the demands imposed upon software embodying intelligent problem-solving capabilities that must be realized under real-time performance constraints. To deliver real-time performance, the control software must support the vehicle's ability to (a) react to critical events by promptly changing the focus problem-solving or other actions to bear upon tasks that embody appropriate responses to those critical events, and (b) gracefully "trade-off" extent, precision, or quality of the vehicle's responses against the amount of time available to complete those responses. To provide these abilities, the intelligent control system must be able to guarantee, or at least reliably estimate, the time required by candidate responses.

• *Uncertainty management* — Military vehicles must successfully accomplish missions under uncertain conditions. The intelligent vehicular-control system must be able to reason about how and when to perform tasks in the face of this uncertainty. As Fehling and his colleagues have pointed out, this requires that the vehicle's problem solving system and intelligent-control methods (a) are robust in the face of this uncertainty, (b) can be adapted to provide the highest quality response possible in the face of the limitations imposed by the uncertain information on which they are based, and (c) allow the intelligent controller to opportunistically mandate actions that may reduce the level of uncertainty upon which future actions are based.

• *Autonomy/Flexibility* — Intelligent control requires that the vehicle be capable of maintaining its operational integrity in the face of conditions and events in its environment that have the potential to impair the vehicle and its ability to survive. Autonomy requirements must be traded against the criticality of the vehicle's mission and the extent to which undertaking



actions to achieve the mission might impact the vehicle's autonomy or survivability. As we are coming to see these issues, we believe that an intelligent control system must include a principled basis for (a) evaluating the relative priorities or "utilities" of predicted, alternative outcomes of its actions under anticipated conditions, and (b) managing the vehicle's commitment to a course of action so as to maximize the expected utility of its activities (in the full decision-theoretic sense). Thus, for example, the costs of damage to the vehicle must be weighed against the benefits of full completion of the vehicle's operational mission.

The preceding views about these categories of functional requirement result from our own experience in building intelligent control software for applications to advanced avionics systems, command-and-control (C2) systems, and industrial process control systems, as well as vehicular control systems. Our efforts in developing and applying intelligent-control architectures such as Schemer (discussed below) have helped to evolve these views.

SEE and Engineering Methodology:

While we agree that the issues and approaches of software engineering for conventional systems can shed valuable light on intelligent-control engineering, we feel, nevertheless, that the application domain of vehicular control imposes important, idiosyncratic requirements on software engineering. For example, the preceding functional requirements discussed impose special software engineering requirements that challenge conventional software engineering concepts and the assumptions that underlie conventional CASE tools. In particular, system developers' lack of information about the domain includes uncertainty about the proper way to model that domain as well as uncertainty about the details of events (e.g., how and when these events will be realized) that are expressible within some particular model. (Cf., the presence of "modeling uncertainty" is a situation that is well known to control theorists.) Due to our belief in the importance of dealing with such modeling uncertainty, we feel that SEEs for intelligent vehicular control must especially support engineering activities such as the following:

• *Performance estimation* — Engineers building intelligent control systems require tools that aid them in estimating the performance of various candidate control methods under a conditions that are most likely to occur in the application domain. In our own work, for example, we have found it useful to provide engineers with a tool that calculates upper bounds on performance times that will result from applying a problem-solving method such as a rule-set in a rule-based system.
• *Empirical testing* — Engineers must be able to construct and carry out experiments with their partially completed control systems and sub-systems. To obtain realistic information, the tools that support empirical testing must allow engineers to simulate the operation of vehicles and vehicular subsystems being managed as well as simulating the dynamic flow of critical events and conditions in the operational environment.
• *"Impact assessment"* — This is really a special case of the two previous requirements. In discussing the functional requirements of intelligent control we noted the importance of managing tradeoffs among alternative courses of action in terms of their impact on such things as

successful mission completion and vehicular autonomy. Engineers must examine these tradeoffs in the most concrete way possible when faced with the requirement to design tradeoff policies into an intelligent controller. Thus, engineers need support in analyzing the potential impact of a control strategy under expected operational conditions so that they can evaluate and implement policies with the highest expected positive impact on system performance.

Life-cycle Maintenance:

If we are successful in building and deploying intelligently controlled vehicles that survive to carry out their assigned missions, then we will face issues of life-cycle maintenance similar to those faced by developers of conventional systems. Unfortunately, the scientific and engineering community has so little experience with fully deployed intelligent-control systems, that little can be said at this time. However, this is not meant to deemphasize the importance of this topic. Rather, we caution that the engineering community not prematurely apply to intelligent-control systems approaches to life-cycle maintenance merely because they appear to work well for other types of software. As earlier preceding remarks indicate, intelligent control entails unique functional and engineering-methodology requirements. Thus, the issues of life-cycle support are also likely include some unique approaches.

We have a great deal to learn before we can confidently determine the required features of SEEs and software engineering methodology across the full spectrum of intelligent control applications. This is particularly true for the issues of engineering methodology and life-cycle support of these systems.

For this reason, we believe that a "bottom-up" approach is called for. In carrying out such an approach we urge the development of computational architectures and development-support tools that are especially tailored for use in building intelligent-control software. Initially, the methods and tools should probably be further restricted to apply to specialized subdomains rather than attempting from the outset to construct an SEE for the full range of intelligent control applications. As SEEs become mature for various domains, their common features can be abstracted and SEEs can be developed that support a broader range of applications. This approach promises quick payoff for certain domains of intelligent control without compromising our overall interest in producing a general-purpose SEE for the full range of intelligent control applications.

Schemer

As our previous remarks imply, the approach to a practical software engineering methodology and SEEs for intelligent control begin with the definition and implementation of a suitable computational architecture tailored for such applications. Fehling (Fehling et al., 1989) describes such a computational architecture. This architecture, called Schemer, has been developed especially for intelligent control applications. Schemer has been successfully used to build over two dozen intelligent-control applications in domains such as advanced aerospace avionics, resource-deployment, industrial process management and process control, and autonomous vehicle control. At this time at least four Schemer applications have been fully deployed and put into regular use as part of commercial and fielded military systems.

Here again space precludes a detailed discussion of Schemer. However, we summarize the following properties of this architecture:

• *Real-time performance* — At this time we believe that Schemer is the only problem-solving architecture that has been designed and implemented specifically to address the issues of distributed, real-time applications. Schemer provides important basic features to support real-time performance. These include (a) an approach to problem-solving control that supports the simultaneous management of multiple problem-solving tasks, (b) preemptive, prioritizing control of these concurrent tasks, (c) full encapsulation facilities for modularization and "transaction protection" under preemptive, prioritizing control, (d) formal models of control and data-flow, (e) a "high-level" language for specifying computational control among combinations of Schemer problem-solving elements, and (f) full support for event-driven and data-driven control to achieve reactive and interactive styles of computation.

• *Development-support Tools* — To support application development, we have augmented the basic Schemer architecture with tools and other facilities that support the builder of distributed, real-time applications. These facilities (a) include stepping, tracing, checkpointing, and checkpoint editing at the level of interaction among Schemer constructs as well as in the underlying implementation language (usually LISP), (b) provision of a "library" of module templates that provide basic problem-solving sub-systems (e.g., a forward- or backward-chaining rule-based system) that might be included as part of the implementation of the intelligent control system, and (c) performance metering packages that can measure (again at the level of Schemer constructs) the speed and complexity of computation of specified implementation elements.

• *Simulation Testbed Support* — In recent projects using Schemer, we have begun to explore tools and techniques that allow implementers to rapidly construct simulations of the application domain and other aspects of application environment. Unlike our work on the basic Schemer architecture and the types of tools just sketched, our understanding of simulation testbed facilities remains far more primitive. We are hopeful, however, that our early successes in developing such testbed capabilities for Schemer applications will soon lead us to a more mature view of these issues.

• *Iterative, "Bottom-up" Development* — In keeping with our earlier remarks, we are evolving our Schemer-based approach to an SEE for intelligent control by working in a "bottom-up" manner on circumscribed domains and then integrating across these domains as commonalities and higher-level abstractions reveal themselves and allow us to generalize Schemer's design and the construction of the software engineering tools within this architecture.

AUV Control System Development

As an example of our bottom-up approach, we are using Schemer to prototype an intelligent control system for an autonomous underwater vehicle (AUV). This AUV controller monitors the conditions during a mission, including the condition of the AUV itself, determines the presence of conditions and events that potentially compromise the mission, and specifies actions to cope with such unanticipated events.

Schemer is being used as the basis for an integration framework for building AUV control applications such as this one. This includes the use of Schemer as a run-time framework within which to manage the activities of the AUV and its subsystems. In particular, Schemer provides the preemptive, prioritizing, multi-tasking management of the controller's actions as well as the actions of the other AUV subsystems.

Schemer also provides the architectural basis and set of tools for this application. In fact, in developing this AUV application we are finding it useful to augment the basic tools and templates (as noted earlier) with special purpose tools and methods that are tailored to, and embody detailed knowledge of, the application domain.

Finally, we note that we are finding Schemer to be especially useful as an open architecture, suitable for encapsulating and managing the interactions among diverse problem-solving elements. This Schemer capability is an aspect of SEE architectures that we have ignored so far in this paper.. We are using Schemer to encapsulate and manage problem-solving elements that are based on a very diverse range of programming and problem-solving approaches. This includes the integration of AI programming methods with conventional mathematical control and optimization methods, for example. In addition, many of these elements were originally developed independently of Schemer. Schemer strong support for encapsulation and its data communication constructs are serving well to provide an "open architecture" within which to easily integrate diverse styles and approaches to system development. At least for AUV applications, we are finding that Schemer's open architecture is crucial.

As noted earlier, we have used Schemer in a similar manner to support development and deployment of intelligent control applications for other domains. As our use of this approach matures we are exploiting opportunities to merge concepts and methods across these areas. In fact, the generality we have already achieved with Schemer is due to this evolutionary, bottom-up approach.

Conclusions

We have outlined some of our concerns about the tools and methods needed to support the engineering of software to intelligently control complex military vehicles. We believe that the research and development community should attend closely to the full range of software engineering issues in developing tools and approaches for such applications. This includes attention to the issues of maintainability and life-cycle support. However, we also caution that, at this time, the issues of intelligent control are rather poorly understood. For this reason, we urge an evolutionary approach to determine the proper solution to this important problem in software engineering. We have adopted such an evolutionary approach in our our work. This has helped us to deploy a number of important intelligent-control applications and to keep an evolving focus on crucial research issues that must be addressed. This evolutionary strategy is serving us well in making progress toward our eventual goal of producing truly general-purpose computational architectures and development-support tools for building intelligent control systems.

References

Fehling, M., Altman, A., & Wilber, B.M. (1989) "The HCVM: An Instance of the Schemer Architecture for Real-time Problem-Solving." In R. Jagganathan, R. Dodhiawala, and L. Baum (Eds.) *Blackboard Systems and Applications*, New York, Academic Press.

Fehling, M., Sagalowicz, D. & Joerger, K. (1986) "Knowledge Systems for Process Management," In Proceedings of ISA-86.

A Second-generation Expert System for Computer-aided Control System Design

Dean K. Frederick
DKF Consulting Service, Inc.
Ballston Lake, NY 12019
(Rensselaer Polytechnic Institute, Troy, NY)

ABSTRACT

The CACE-III expert system [1] for designing series lead-lag compensators for single-input/single-output plants to satisfy three frequency-response specifications has been implemented using the GoldWorks inference engine and expert-system development tools. The implementation makes use of the frames and user-interface features of GoldWorks to provide an environment from which further extensions of the design approach or other design methods can be created in an orderly fashion. The control-analysis program Matrix$_X$ has been used to perform the required control-system modeling and analysis calculations at the request of the expert system.

A number of ways in which the present design system can be improved and expanded are discussed. Among these are a hybrid organization of the rules that combines the known procedural knowledge of the design process with an ability to constantly look for off-nominal or unexpected trends in the design process and to react to them. Another way is to make greater use of object-oriented programming techniques in conjunction with the frames and their slot values. The tools that have been developed for handling the varied interactions between the expert system and the numerical applications program are described, with specific attention paid to their robustness. Details of the work done to date may be found in [2].

INTRODUCTION

The project described in this paper relates to the development of an expert-system environment for the computer-aided design of feedback control systems and to the creation of a rule base specifically tailored to the design of lead-lag compensators for single-input/single-output (SISO) systems. It is an extension of earlier work by James, Frederick, and Taylor [1] that had led to a package called CACE-III. That version ran on a VAX computer and used a proprietary inference engine called Delphi from General Electric, and the Cambridge Linear Analysis and Design Program (CLADP) from Cambridge University in England for the control-system model-building and analysis calculations. The block diagram in Fig. 1 shows the components of the feedback system under consideration. The plant is linear and time-invariant and the controller is a series lead-lag compensator, entirely in the forward path.

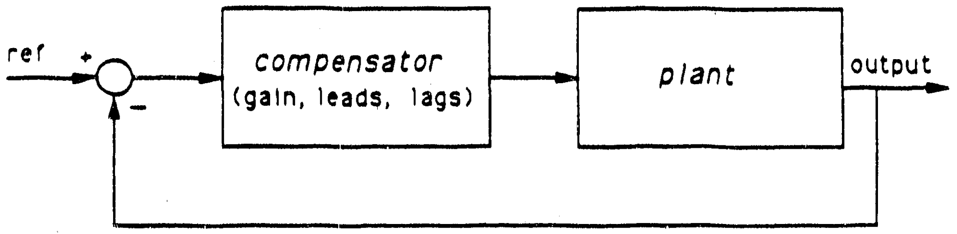


Figure 1. Feedback systems being considered.

The objective of the present study has been to build a knowledge-based system for control design that uses a state-of-the-art inference engine and can be run on an IBM-PC/AT compatible computer. The current phase of the project has been to establish the initial hardware and software of the system.

Work has been directed at establishing sufficient expertise with the GoldWorks expert-system development environment to enable the effective use of its many and varied capabilities. The knowledge-representation techniques available in the GoldWorks environment have been used to implement a knowledge-based system for aiding engineers in the design of lead-lag compensators for SISO systems. The work has been performed on two personal computers that are operated in parallel. The expert system runs on a Zenith-248 that has an eight-megabyte memory board in order to accomodate the extensive GoldWorks code. The numerical control-related calculations are done on an IBM PC/XT using the commercial package Matrix$_X$/PC.

Additional tasks that have been pursued during this phase of the project include:
(1) writing rules without embedded Lisp code so as to maintain a clear separation between the knowledge and the procedural portions of the system,
(2) making use of the frames allowed by the GoldWorks inference engine to organize the knowledge about the control system design,
(3) several improvements in the design algorithm,
(4) expansion of the lead-lag design capabilities to handle plants with lightly-damped modes,
(5) enhancing the robustness of the design process so the user can recover from a variety of problems, should they arise during a run,
(6) improving the user interface so as to provide up-to-date status information and pop-up forms, data-entry menus, and acknowledgement messages,
(7) providing a mechanism whereby the user can interrupt the run and modify the specifications or the parameters of the controller, and
(8) allowing the user to continue a completed design with altered specifications.

Figure 2 shows the elements of the expert system and the means by which they interact.

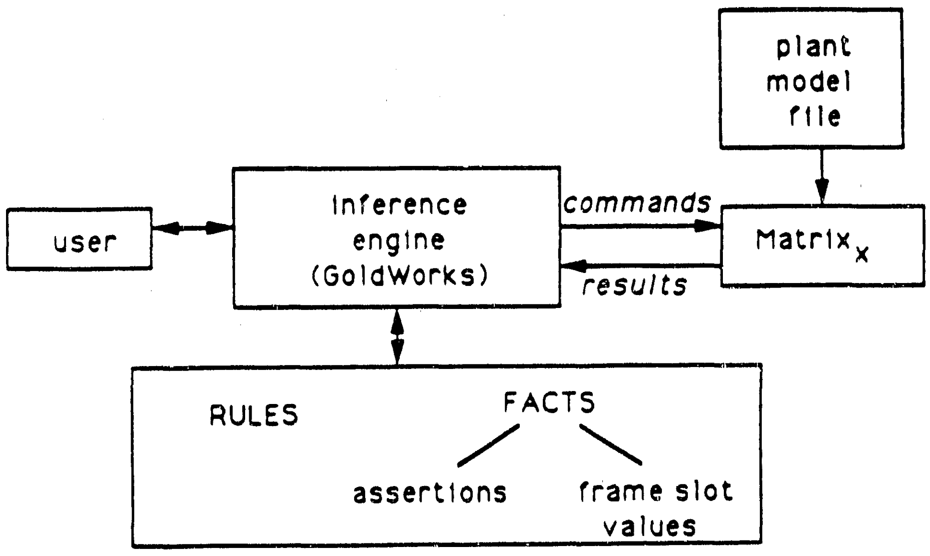


Figure 2. Principal elements of the design system.

The knowledge base is made up of the rules and the facts (assertions and slot values). Matrix$_X$ is used for all of the control-related calculations such as determining the closed-loop bandwidth and implementing the models of the plant and the compensator (controller). A set of plant models has been written in Matrix$_X$ and this can easily be augmented by the user to include additional plants. When a session begins the user selects the plant to be considered and is prompted to enter the specifications.

The development thus far has not resulted in the exploitation of the object-oriented nature of the GoldWorks environment to provide better justifications to the user. However, the restructuring of the lead-lag design heuristic into the frames and rules of GoldWorks has provided the underlying knowledge representation which can now be used to provide that kind of information. Perhaps more importantly, that same representation method has resulted in a knowledge base which is easier for the developer to create and maintain.

FRAME ORGANIZATION

The first major design decision was the definition and organization of the frames so as to properly fit the tasks to be accomplished, namely the design of control systems. Figure 3 shows the organization that has been used.

TOP-FRAME is part of GoldWorks and is always the starting point for the user-defined frames. Below that we show the frames that have been created for the control-design problem, namely:
DESIGN-FLOW CONTROL contains information relating to the design process itself, rather than the control system,
PLANT contains information pertaining to the plant or process that is to be controlled,
OL-SYS contains information that relates to the open-loop system comprising the plant and the controller,
CONTROLLER contains controller information and has child frames for each of the components (gain, leads, lags, and possibly a notch filter) that make up the controller (compensator),
CL-SYS stores information relating to the closed-loop behavior of the plant and controller,
INIT stores information relating to the particular design run being made, and
SPECS contains the specifications, their tolerances, and whether or not the specifications are met by the current controller.

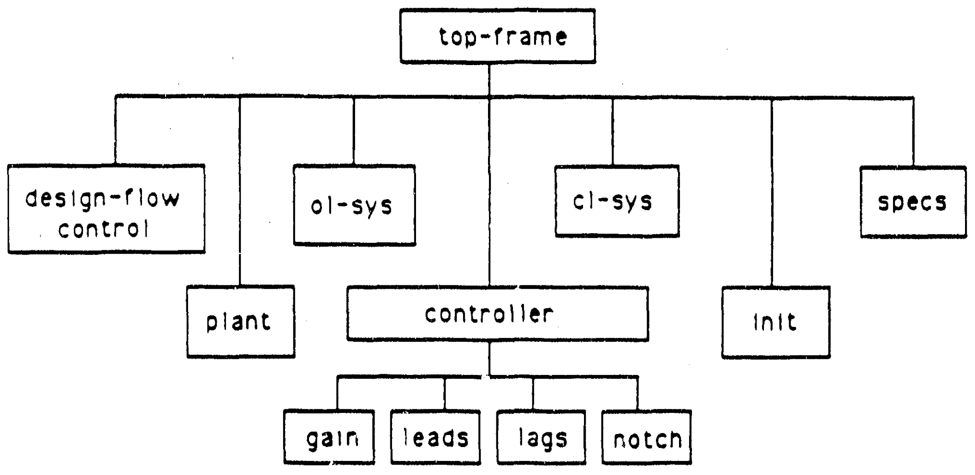


Figure 3. The control-system-related frames.

DESIGN ALGORITHM

The design method used for the original version of CACE-III [1] has been retained, with several modifications. As suggested by Fig. 4, there are three main parts to the algorithm. First, the initial version of the compensator is created to contain a gain, lead, and lag. Then an iteration is begun during which the gain and the alpha, i.e., the pole-to-zero ratio, of the lead are adjusted until the gain margin and the closed-loop bandwidth specifications are met (hopefully). Then the lag is given a final adjustment and all three of the specifications are tested. In the following, we will discuss the algorithm in more detail and conclude with some suggestions for further improvements.

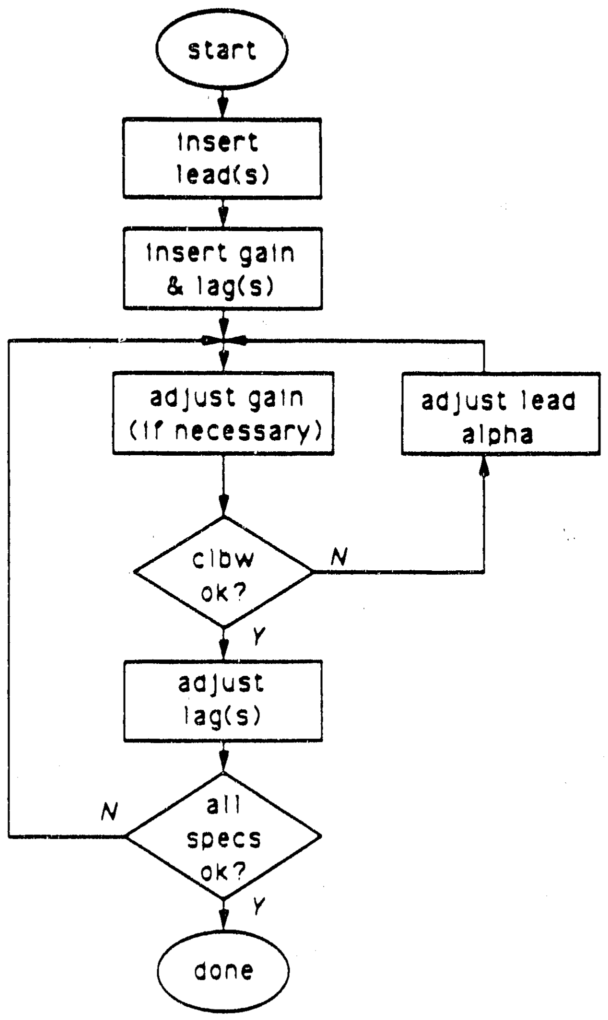


Figure 4. Flow chart of the design algorithm

The basic idea for starting the compensator is to insert one or more leads based on the phase angle of the frequency response of the plant alone, measured at the specified closed-loop bandwidth frequency. The center frequncy of the the lead is set equal to the closed-loop bandwidth specification and the value of alpha is determined so as to make the phase angle at that frequency of the combined plant and lead to be -175 degrees if the plant is type 0 and -140 degrees if the plant is type 1. The objective of this heuristic is to start the design process from a point that is in the "ballpark" as far as being able to attain the desire closed-loop bandwidth.

Once the initial lead(s) have been set, a calculation is done in Matrix$_X$ to determine the gain margin and the frequency at which the gain margin is calculated. Then the gain is adjusted to meet the gain-margin specification and one or more lags are inserted to meet the specified low-frequency gain (type-0 plant) or velocity constant (type-1 plant). At this point we know that two of the three specifications have been met, so we next have Matrix$_X$ evaluate the closed-loop bandwidth. In the unlikely event that the value falls within the tolerance of the specification, the design has been completed because a controller has been found that satisfies all three of the specifications, within the tolerances.

The more likely situation is that we have not satisfied the closed-loop bandwidth specification. In this event, an adjustment will be made in the alpha parameter of the lead(s) so as to increase the amount of phase lead if the actual closed-loop bandwidth is low and to decrease the phase lead if the closed-loop bandwidth is above the specified value. A relatively simple heuristic rule is used to select the change in the value of alpha (the pole-zero ratio) of the lead(s) and no change is made in the center frequency at present.

Assuming that the adjustments of the lead and the gain finally result in the closed-loop bandwidth specification having been met, there is a final adjustment made to the lags to ensure that the low-frequency-gain or velocity constant specification has been met. Then the other two specifications are retested, because we must expect to retest the specifications after any change has been made to the compensator.

Another significant change from the original version of CACE-III is the step taken to recognize and compensate for lightly-damped modes of the plant. It has been known that the presence of such modes (say complex poles with a damping ratio less than 0.2) will cause the original CACE-III algorithm to yield unsatisfactory results. If such a mode is discovered during the diagnosis phase the expert system inserts a notch filter in series with the plant.

INTERFACE CODE AND FILES

Because of the necessity of developing and modifying the mathematical model of the control system and of carrying out a multitude of control-related calculations, it is not practical to implement the entire expert system for the design of lead-lag controllers in GoldWorks. Because Matrix$_X$ has been used for the control calculations, it has been necessary to develop a mechanism for having GoldWorks write the necessary Matrix$_X$ commands and to interpret the results.

For example, when the compensator must be modified, the alpha of the leads must be increased. The rule that decides this task must be performed will call a specialized Lisp procedure. This procedure will extract any required numerical values, such as the new values of the compensator parameters (gain, lead center frequency and alpha, and lag center frequency and alpha) from the slots of the appropriate frames. Then it will use them to write the required commands to a file that is read by Matrix$_X$. After any required modification of the compensator and calculations are completed, Matrix$_X$ will write the results and an error code to a file. GoldWorks will read the file, test the error code, and digest the results. In Fig. 5, we show the result of a design where the final compensator is composed of a gain, two leads, and two lags.

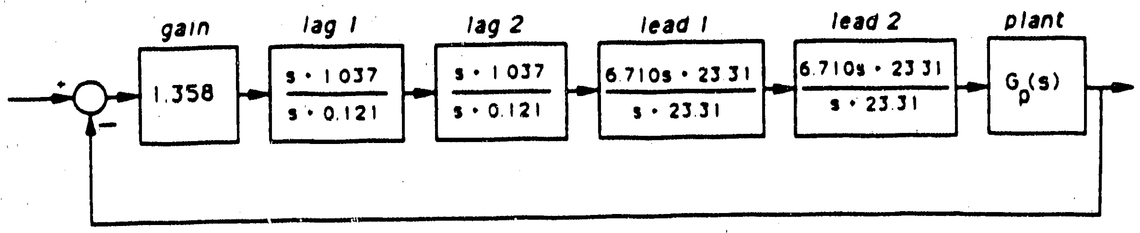


Figure 5. Block diagram of the final feedback system.

RESULTS

During the course of this project a number of significant changes have been made that have resulted in a vastly-improved implementation of the original design methodology and made modest improvements in the design algorithm. With these changes, the opportunity now exists for further growth. In the remainder of this section we will list and give brief descriptions of these changes.

A. *Changes from the original CACE-III*

Expert-system shell: The current work has been done with the original version of GoldWorks that was produced by Gold Hill Computers, Inc. of Cambridge, MA in 1987.

Rules: The rule base has been greatly simplified from that of the original version. Part of this reduction came from a decision not to implement the model-development feature of the original version, but for the most part it came from having had the benefit of the earlier work and being able to use a second-generation expert-system shell.

GoldWorks-Matrix$_X$ interface code: A considerable amount of effort went into the development of the code for the interface between GoldWorks and Matrix$_X$. Because Matrix$_X$ handles its models in a very different fashion from CLADP (its counterpart in the original version of CACE-III), it was necessary to completely redefine the manner in which the expert system and the application program keep track of and modify the compensator and open- and closed-loop models. For example, GoldWorks thinks of the compensator in terms of poles, zeros, and center frequencies, whereas Matrix$_X$ thinks of it as a packed matrix consisting of the A, B, C, and D matrices of the state-space representation. When a pole or zero is modified, the corresponding slot value is changed in GoldWorks, but in Matrix$_X$ the entire compensator is discarded and a new one is built. Also, error flags, that had not been used in the original implementation, were introduced in the Matrix$_X$ code. The net result is that the present system is far more robust and extendable than its predecessor. For the most part, the Matrix$_X$ side of the interface does the state-space model manipulations caused by the adjustments in the compensator and the control calculations required for the design algorithm and the evaluation of the specifications. The GoldWorks side of the interface, written in Lisp, does the compensator manipulations in terms of the alphas and the center frequencies, the writing of the Matrix$_X$ command files, and the interpretation of the results received from Matrix$_X$.

User interface with GoldWorks: The user-interface development tools of GoldWorks have made it possible for the user to enter information such as the plant name and the specifications with a minimum of typing and with reduced chance for error. The specifications are entered by editing the default values that appear on a form, with their tolerances, and the names of any predefined plant models can be entered by clicking on a menu. Additional displays have been created to keep the user informed as to the status of the design process, in terms of the open- and closed-loop system characteristics and the controller parameters.

Post-run review and restart: All of the files of Matrix_X commands and results that are passed back to GoldWorks during a run are saved. Also the firing of every rule is recorded in a GoldWorks log file running on the Zenith, and a Matrix_X diary file records all of the information that is displayed on the screen of the XT. In addition, the values of all of the Matrix_X variables are saved at each step of the design in binary files that can be loaded into the Matrix_X work space. With this information, it is possible to follow and reconstruct the details of the design process after the run has been completed.

Lightly-damped plant modes: The rule base has been expanded to be able to handle plants with lightly-damped modes which the original version of CACE-III was unable to do. This is accomplished by inserting a notch filter in series with the plant and then proceeding with the design algorithm.

B. *Possible extensions*

Although numerous improvements have been made so far over the original implementation of CACE-III, there are still a number of features that can be added by taking advantage of the GoldWorks environment, and there are further enhancements to the existing implementation that can be incorporated.

Rule-base consolidation and greater use of object-oriented programming: The rules have developed to a point where they are no longer as clean and orderly as we would like, and the division of tasks within the Lisp code is not as clear as it could be. Also, we have not taken full advantage of the object-oriented-programming techniques that GoldWorks has put at our disposal. For example, handlers can be created to send messages to the frames (objects) and a slot value in that frame might be the name of the procedure for calculating the closed-loop bandwidth. It is anticipated that simplifications can be made and GoldWorks programming techniques incorporated that will prove beneficial when adding some of the new features that follow.

Faster algorithm convergence: It is believed that the convergence of the current design algorithm can be speeded up by using a more sophisticated rule for selecting the amount of phase lead change based on the error in the closed-loop bandwidth specification. Such a rule would use the mathematical model of the plant, which is available to the expert system, to analytically predict the expected changes in the closed-loop bandwidth and adjust the lead alpha accordingly. One could also incorporate a learning mechanism, perhaps tied to the plant model.

Vary compensator center frequencies: It might be possible to get some benefit when more than one lead is involved by allowing the center frequencies of the leads to differ from the desired closed-loop bandwidth frequency or by letting them differ from one another. Similar changes could be made in the center frequencies of the lag(s), based on recognizing distinctive geometric ch[illegible]cteristics of the Nichols plot.

Reduce overshoot at plant input: Once the basic algoritm has met the specifications, the expert system can alleviate the high overshoot that will occur at the input to the plant following a step change in the reference input. This can be done by moving the lead part of the compensator from the forward path to the feedback path. However, this will affect the zeros of the closed-loop transfer function and the step response, so the specifications will probably no longer be met. It will be necessary to have some additional rules for adjusting the compensator.

Sampled-data designs: It should also be possible to do sampled-data designs by using the bilinear transformation of the discrete-time model to a pseudo continuous-time model, carry out the design with the present rules, and transform the controller back to the discrete-time domain.

Allow additional specifications: It would be desirable to be able to accomodate a wider range of specifications, such as percent overshoot, settling time, rise time, and steady-state error to a disturbance input. At present, the diagnosis of the plant detects the presence of a lightly-damped mode and the expert system will insert a notch filter to compensate for it. It should be possible to have other characteristics of the plant detected and used to determine it the user's specifications are reasonable and, if not, how they might be modified.

Multiple gain-margin values: When Matrix$_x$ detects multiple values for the gain margin, i.e., multiple crossings of the -180 degree phase line, it reports these to GoldWorks but at the present time the rules required to use this information intelligently are not available.

Explanations: The expert system could be strengthened by making better use of the capabilities of GoldWorks to provide explanations of the design process to the user.

Design tradeoffs: Another area for possible improvement is to provide the user with assistance in making design tradeoffs, once the specifications have been met or it has been determined that they can not be met.

Constraints: Along with this, it would be useful to incorporate some constraints on the design that the user could adjust, if desired. Examples of these would be maximum number of leads and lags, maximum value of the parameter alpha for the leads and lags.

Phase margin: At present, the phase margin and the phase-margin frequency are not used in the design procedure. It is well known that one can have a satisfactory gain margin but still have a very oscillatory response if the phase margin is too low. Also, there are plants for which the phase margin is defined but the gain margin is not. The original version of CACE-III could deal with such plants, but this feature has not been carried along.

Hybrid rule base: It might make sense to have two classes of rules. One class would implement the design algorithm, say as it is presented in Fig. 4, and the other class of rules would respond to problems or exceptions that might arise. Examples would include multiple gain-margin values, unusual bandwidth values, lack of convergence, failure of compensator changes to yield expected results, etc. One approach would be to create subframes of the OL-SYS and CL-SYS frames that would be used to store the patterns recognized as being anomalous. When detected, a pattern could be used to fire the rules for a variation in the design algorithm. In such a case, the expert system would be using the heirarchical inheritance mechanism of the frames to define the problem and the situation-action mechanism of the rules to perform the design.

Application to real-time control: The question of using an expert system as part of a real-time control system has not been specifically addressed. However, we believe that much of what we have accomplished so far is applicable to that goal provided that the design is done in a supervisory role by using the rules to adapt to parameter variations and uncertainty.

CONCLUSION

It has been demonstrated that the combination of GoldWorks and Matrix$_x$ provides an effective environment for carrying out the design of SISO control systems. In doing so, much of the infrastructure that is necessary for doing this type of work has been put into place in a way that it is adaptable to a variety of other applications.

REFERENCES

[1] James, John R., Dean K. Frederick, and James H. Taylor, "Use of expert-systems programming techniques for the design of lead-lag compensators", IEE Proceedings, vol. 134, pt. D, no. 3, May 1987.

[2] Frederick, Dean K., John R. James, Alfred Antoniotti, and Hiroyuki Nitta, "A Second-generation Expert System for computer-aided control system design", DKF Consulting Service, Inc., Ballston Lake, NY, February 1990.

ACKNOWLEDGEMENTS

The author was assisted in this work by Col. John R. James, HQ TRADOC, Fort Monroe, VA and by Alfred Antoniotti and Hiroyuki Nitta, both of the Electrical, Computer, and Systems Engineering Department, Rensselaer Polytechnic Institute, Troy, NY. The work was performed for Dr. Norman Coleman of the U.S. Army Armament Research and Development Center at Picatinny Arsenal, Dover, NJ. The research was performed under Contract No. DAAL03-86-D-0001, Delivery Order 1100, of the Scientific Services Program, Battelle Research, Research Triangle Park, NC. The views, opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

Control Systems Engineering in an Open–Architectured Object–Oriented Software Environment

Donald T. Gavel
Lawrence Livermore National Laboratory
P.O. Box 808, Livermore, CA 94550

ABSTRACT

Over the past decade, the practice of control systems engineering has been profoundly affected by the explosion in computer hardware and software technology. This paper describes the emerging concepts of open–architectured object–oriented software design which allow for both compatibility and expandability of computer based engineering tools in the face of ever expanding user needs.

1. Introduction

Fueled by the explosion in computer workstation and networking technology, there has been a rapid increase in both the availability and capability of the software tools most useful to control systems engineers. Today, software technology has advanced to a point where significant standardization issues have been worked out and very low cost or even free software is available for many of the important computational tasks performed by the control systems engineer, such as symbolic equation manipulation, numerical calculations, graphical display, and documentation.

While incorporating all of the control system design tools into a single "package" seemed the obvious answer years ago, by far the most popular solution now is the "open environment" approach, that is, using a good general purpose operating system, such as Unix, and a small set of well documented data formats, e.g. IEEE floating point standard, TIFF graphics format, etc., which are now widely accept across *multiple* engineering disciplines. The principal advantage of the open environment is that tools are allowed maximum flexiblity for future growth and enhancement.

In order to encourage wide dissemination of software across computer hardware architectures and operating systems, much of today's software is made available in an "open–architectured" format. The open–architectured software package includes all of the source files and all of the script files necessary to regenerate the working program. This approach allows users to tailor programs to their own needs by editing the source or input files. Unfortunately, however, software is often written so obscurely that its operation is hard to determine from source listings, and users are tempted to start over from scratch, resulting in a wasteful duplication of effort.

The object–oriented approach, which has developed over the past several years, has helped to aleviate this problem by making source files more manageable and easier to debug [1]. The technique and has been widely adopted by modern software designers. The Ada and C++ are programming languages are clear indications of the tendency in this direction.

This paper will focus on a new open-architectured, object-oriented engineering design tool called MatC, which can do most of the matrix-based numerical calculations for control systems design, as well as for a number of other disciplines. MatC integrates well with other tools, both open- and closed-architectured, that are useful to engineers, for example: Mathematica (symbolic manipulator), VIEW (image processing/data display), Emacs (editor/window system), and TEX(document typesetter). As shown in figure 1, an engineer can go from conceptual design to parametric design to final report, all electronically, which lends itself well to traceability and repeatability of the design process.

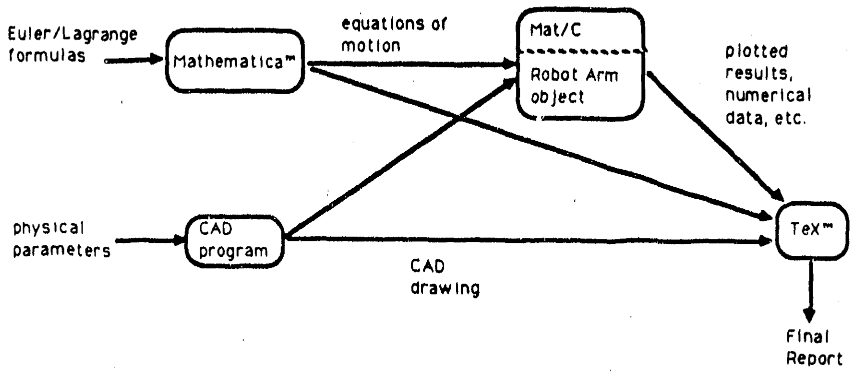


Fig. 1. *The Flow of Engineering Data During Control System Design*

In section 2 we describe the operation of MatC in a window based environment. Section 3 describes the software objects from which MatC is constructed, and explains how they can be reused in other programs. In section 4, we discuss the other important pieces of software that form an integrated engineering software environment. Finally, in section 5 we will use the design of a robot control system as an example of modern computer-based engineering development.

2.The MatC Program

MatC is basically an interactive calculator. The user types commands and the computer responds with the calculated answer. The simplest "commands" are arithmetic calculations, for example

```
<> 8*3
ANS =
    24
```

The program has calculated the product $8 \times 3 = 24$ and stored the answer in the variable `ANS`, which always contains the result of the most recent calculation. MatC allows you to specify and do calculations on matrices:

```
<> a = [ 1 2
```

```
         3 4 ];
<> b = [ 1
         2 ];
<> a*b
ANS   =
    5
    11
<>
```

We have set the variable a equal to the matrix $\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix}$ and set b equal to $\begin{pmatrix}1\\ 2\end{pmatrix}$ then multiplied them according to the rules of matrix multiplication to get the result $\begin{pmatrix}5\\ 11\end{pmatrix}$.

MatC has a large number of built-in matrix functions that can be called interactively. For example, since a is a square matrix, we can compute its eigenvalues

```
<> eig(a)
 ANS   =
 -0.3723
  5.3723
<>
```

or, we can generate a complex number

```
<> i = sqrt(-1)
 i   =
  0.0000  + 1.0000  i
```

MatC calculations work on both real and complex numbers. The MatC functions all allow complex arguments, and may produce complex results. Figure 2 shows a list of MatC functions generated by the "WHAT FUN" command.

```
------- Builtin Functions (* = not implimented yet) -- 77 items -------
  ABS      ASTIG    ATAN     BASE     CHOL     CLIP
  COND     CONJ     CONV     COS      CURV     DET
  DIAG     DIARY*   DISP*    EDIT     EIG      ERF
  ERFC     EXEC     EXP      EYE      FFT      FFT2D
  FLOP     GAUSS    GRID     HESS     HILB     HIST
  IMAG     INV      ISTHERE  KRON     LABEL    LEV
  LOAD     LOG      LU       LYAP     MAGI     MINM
  NORM     ONES     ORTH     P3D      PCOLOR   PINV
  PLOT     POLY     PRINT*   PROD     QR       RAND
  RANK     RAT      RCON     REAL     RICC     ROOT
  ROUN     RREF     SAVE     SCHU     SIN      SIZE
  SORT     SQRT     SUM      SVD      TRACE    TRIL
  TRIU     _EPS     _I       _INF     _PI
------------------------------------------------------------------------
```

Fig. 2. *A List of MatC Functions*

The features of MatC are:

- At the most basic level, the program acts as an interactive calculator for matrices. Users familiar with

BASIC, FORTRAN, or even a hand calculator find MatC easy to learn and use.

- The MatC language is a complete interactive programming environment, with structured programming constructs (FOR, WHILE, IF), function calls, database manipulation, etc.

- MatC has an extensive built-in function library. Matrix calculations are based on the proven LINPACK and EISPACK software packages.

- There is a variety of ways to enter data into MatC: it can be entered from the terminal, read frcm script files, read from MatC formated data files, or read from ascii data files written by other programs.

- There is a similar variety of ways to save data on disk files. All or part of the MatC database can be saved for later use, either in future MatC interactive sessions, or for transfer to other programs.

- MatC has extensive interactive plotting capabilities. Both 2–D and 3–D plots can be generated. The resulting graphical display can be sent to a printer or transfered as picture data to other programs such as word processers, viewgraph makers, etc.

- MatC has a standard interface to "external" software. Users can easily add their own subroutines or entire libraries of functions into the MatC environment, thus specializing the program to the a particular application's needs.

- Th. C language sources are available to users. MatC was designed and programmed using an object oriented approach. Each object is a data structure and the set of operations relevant to that data structure. For example, the MatC objects are familiar items such as **Matrix**, **Plot**, **TextFile**, etc. Typical operations are **Print**, **Copy**, **Dispose**, etc. MatC's objects can be easily modified at the C source level to suit a user's specialized needs. The objects can also be incorporated into other programs.

The interactive environment is both intelligent and user friendly. Windows are used to display one or more interactive MatC sessions (each with its own database). Script files can be edited in seperate windows, then "sent" to the MatC processor in one keystroke. Each plot has its own window and a set of smart operations that go with it. "Help" files are on-line and can be viewed while running MatC in a separate smart window with a topical search facility. All of the documentation is available on-line in TEXfiles.

The MatC software package can be distributed completely electronically, on tape or disk, or over a network, which makes it easy to access all of the information in one place when it is needed, cuts down on paperwork, and gives users easy access to the most up-to-date version. The open–archtecture allows for customization to a particular application. An example of a customized robot control design application is given later in this report.

2. Objects

The development of MatC was made easier through the use of objects. An object is a data structure combined with all of the operations (methods) that are valid on that data structure. Ideally, objects are conceptually easy to grasp entities, such as a **Matrix** or a **Graph**, for which high level operations, such as **Plot** or **Save**, convey immeadiate meaning. The actual implimentation of an object's methods, as well as the structure of the object itself, can be hidden from all the other routines that use the object.

The data hiding feature, called encapsulation, turns out to be an extremely valuable technique for reusability of code, ease of enhancement, and isolation of bugs. For example, in the process of developing an enhanced version of the code, an object's capabilities can be expanded or modified, even its data structure can be changed, without requiring any changes in the existing software that uses the object. The upgraded object is always user-compatible with old version. In fact, the tendency in object-oriented design is to never throw away old code (since it is reusable) but only to add on completely new capabilities.

Most objects understand a generic set of methods. *Object*_**New** and *Object*_**Dispose** create and dispose instances of objects. *Object*_**Print** displays object data at the terminal. *Object*_**Copy**, *Object*_**Rename**, *Object*_**Size**, etc. also do obvious operations. Particular objects may actually impliment these methods differently, depending on what is appropriate. Particular objects also have their own unique methods, expanding upon the basic set.

Some of the objects used in MatC are explained below.

- **Matrix** — contains the size, data, and name of a matrix. Methods are **Matrix_Multiply**, **Matrix_Divide**, **Matrix_Eigenvalues**, etc.
- **SymbolTable** — maintains the MatC database. Typical methods are **SymbolTable_Add**, **SymbolTable_Remove**, **SymbolTable_Lookup**, **SymbolTable_Sort**, etc.
- **Symbol** — is the entity stored in a **SymbolTable**. It contains the **Matrix** or **Function**, creation time, and protection bits.
- **Lexer** — is the MatC command interpreter. The method **Lexer_ParseAndExecute** absorbs the user's command string and sends it on to the parser. Since MatC understands infix mathematical syntax and uses a specialized syntax for matrices, a rather complex parser is used. The parser was constructed from a syntax description file using the Yacc parser generator program. A subordinate **Machine** object executes the parsed code. **Lexer** also handles commands read from text files using the **Lexer_ExecFile** method.
- **Graphic** — contains the graphical information about a plot or other drawing.
- **Window**, **TE** (TextEdit extension), **MenuBar**, and **EventHandler** — are specific to the Macintosh environment and maintain the user interface. On Unix, these operations are taken care of in a Vterm, Xterm, or EMACS window.

The reusablity and encapsulation properties of MatC objects make them ideal as seperate entities that can be incorporated in other programs. For example, if one wants to read the data stored in a MatC "save" formated file into another program, that program can be modified to use the **Matrix** object. The **Matrix_Load** method will read the data from disk, and other **Matrix** methods are available to extract the important information contained in the matrix.

Alternatively, the **Matrix** object can be modified so that it can read and write data stored in the other program's format. As MatC develops, we will see an increasing tendency toward compatibility with other programs and data formats. This make sense in light of MatC's open–architectured philosophy and the fact that MatC has such an extensive interactive data manipulation and display capability. It is the logical choice as a data "switching station."

4. Integrated Engineering Software Environment

In order to assure a systematic and repeatable design process, engineers demand a sophisticated operating system as well as a significant amount of support software. In the ideal open environment, there is always be allowance for modification, enhancement, partial execution, and general twiddling by the design engineer. The objective is to make the tedious parts of the engineering job easy to accomplish in a systematic manner and not to favor any particular engineering design approach.

A comfortable operating system and terminal screen environment is flexible enough to allow the simultaneous display of large amounts of information. For example, in Figure 3 the on-line help, a script file editor, directory listings, and the program doing calculations are all simultaneously visible.

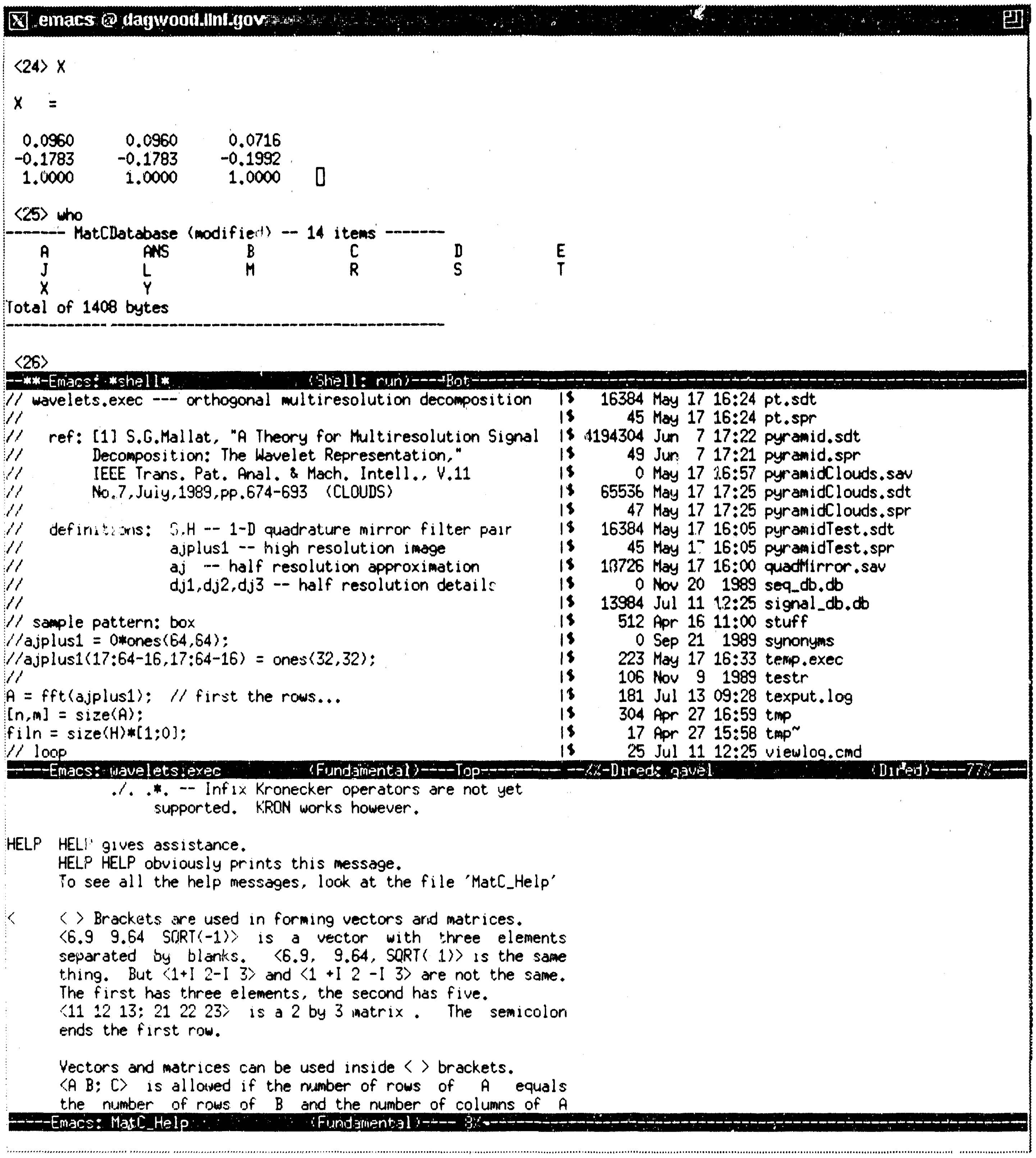
Several tools make it possible today to have this arrangement using almost all open–architectured software. To mention a few: the MIT X-Windows system provides an excellent, window system for graphics workstations, and is available free of charge; GNU EMACS is an excellent multi-port environment that will work on non graphics terminals (GNU offeres a variety of open–architectured Unix utilties, including a C compiler, debugger, make facility, etc.); Mat/C has the basic tools for control systems design, general calculation, data analysis, simulation, and plotting; TEXis a document typesetting program that can be tailored to allow text and graphics pasting.

5. Robot Control System Design Example

As an example problem we consider the design of a robot motion controller. The dynamics of a robot system are highly nonlinear, uncertain, and depend on the load, operating speed, and position of the arm. Our design process will roughly follow the path outlined in Figure 1. We start from a basic mechanical description of the system (say, a parameterized CAD drawing), the basic inertial parameters (mass, CG, moments of inertial of each part), and the geometry of the arm (the Denavit-Hartenberg parameters), and proceed to develop a controller and anaylze the behavior.

The Lagrangian for the system is

$$L = T - V = \frac{1}{2}\dot{\mathrm{q}}^T J(\mathrm{q})^T M J(\mathrm{q})\dot{\mathrm{q}} - \mathrm{g}(\mathrm{q}) \qquad (1)$$



```
emacs @ dagwood.llnl.gov

 <24> X

 X   =

  0.0960      0.0960      0.0716
 -0.1783     -0.1783     -0.1992
  1.0000      1.0000      1.0000     []

 <25> who
------- MatCDatabase (modified) -- 14 items -------
    A          ANS         B          C          D          E
    J          L           M          R          S          T
    X          Y
Total of 1408 bytes
----------------------------------------------------

 <26>
--**-Emacs: *shell*              (Shell: run)----Bot------------------------------------------
// wavelets.exec --- orthogonal multiresolution decomposition  |$  16384 May 17 16:24 pt.sdt
//                                                             |$     45 May 17 16:24 pt.spr
//    ref: [1] S.G.Mallat, "A Theory for Multiresolution Signal |$ 4194304 Jun  7 17:22 pyramid.sdt
//         Decomposition: The Wavelet Representation,"         |$     49 Jun  7 17:21 pyramid.spr
//         IEEE Trans. Pat. Anal. & Mach. Intell., V.11        |$      0 May 17 16:57 pyramidClouds.sav
//         No.7,July,1989,pp.674-693  (CLOUDS)                 |$  65536 May 17 17:25 pyramidClouds.sdt
//                                                             |$     47 May 17 17:25 pyramidClouds.spr
//    definitions:  G,H -- 1-D quadrature mirror filter pair   |$  16384 May 17 16:05 pyramidTest.sdt
//                  ajplus1 -- high resolution image           |$     45 May 17 16:05 pyramidTest.spr
//                  aj  -- half resolution approximation       |$  10726 May 17 16:00 quadMirror.sav
//                  dj1,dj2,dj3 -- half resolution details     |$      0 Nov 20  1989 seq_db.db
//                                                             |$  13984 Jul 11 12:25 signal_db.db
// sample pattern: box                                         |$    512 Apr 16 11:00 stuff
//ajplus1 = 0*ones(64,64);                                     |$      0 Sep 21  1989 synonyms
//ajplus1(17:64-16,17:64-16) = ones(32,32);                    |$    223 May 17 16:33 temp.exec
//                                                             |$    106 Nov  9  1989 testr
A = fft(ajplus1);  // first the rows...                        |$    181 Jul 13 09:28 texput.log
[n,m] = size(A);                                               |$    304 Apr 27 16:59 tmp
filn = size(H)*[1;0];                                          |$     17 Apr 27 15:58 tmp~
// loop                                                        |$     25 Jul 11 12:25 viewlog.cmd
-----Emacs: wavelets.exec        (Fundamental)----Top----------  --%%-Dired: gavel          (Dired)----77%--
          ./. .*. -- Infix Kronecker operators are not yet
               supported.  KRON works however.

HELP  HELP gives assistance.
      HELP HELP obviously prints this message.
      To see all the help messages, look at the file 'MatC_Help'

<     < > Brackets are used in forming vectors and matrices.
      <6.9  9.64  SQRT(-1)>  is  a  vector  with  three  elements
      separated  by  blanks.   <6.9,  9.64, SQRT(-1)> is the same
      thing.  But <1+I 2-I 3> and <1 +I 2 -I 3> are not the same.
      The first has three elements, the second has five.
      <11 12 13; 21 22 23>  is a 2 by 3 matrix .   The  semicolon
      ends the first row.

      Vectors and matrices can be used inside < > brackets.
      <A B; C>  is allowed if the number of rows  of   A   equals
      the  number  of rows of  B  and the number of columns of  A
-----Emacs: MatC_Help            (Fundamental)---- 3%-----------------------------------------
```

Fig. 3 *Running MatC under EMACS*

where T is the kinetic energy, V is the potential energy, $\mathbf{q}$ represents the vector of robot joint displacements or rotation angles along the n degrees of freedom of the robot. M is an $2n \times 2n$ diagonal matrix containing the link masses and moments of inertia of each moving link. The Jacobian, $J = (J_v^T, J_\omega^T)^T$ is defined by

$$J_v(\mathbf{q}) = \partial \mathbf{v}/\partial \dot{\mathbf{q}}; \qquad J_\omega(\mathbf{q}) = \partial \omega/\partial \dot{\mathbf{q}}. \tag{2}$$

From the Lagrangian, the Euler–Lagrange equations of motion can be derived

$$\tau_{\text{external}} = (\dot{\frac{\partial L}{\partial \dot{\mathbf{q}}}}) - \frac{\partial L}{\partial \mathbf{q}} = H(\mathbf{q})\ddot{\mathbf{q}} + C(\mathbf{q}, \dot{\mathbf{q}}) + G(\mathbf{q}) \tag{3}$$

The functions $H(\cdot)$, $C(\cdot,\cdot)$, and $G(\cdot)$ are very complicated expressions in terms of the original robot inertial parameters and robot geometry [3]. These expressions are best derived using a symbolic math manipulation program such as Mathematica. The problem of performing tedious but straightforward symbolic calculations arises in many control and state estimation tasks, particularly those involving non-linear systems. It is important to have computer assistance in performing these calculations so that they can be done without error.

Given the equations of motion, we can use MatC to perform analysis and design. For example, say we wish to compare a decentalized design (that is, independent parallel controllers for each joint) to the computed torque approach, which requires a centralized control computer. Computed torque is given by

$$\tau_{\text{applied}} = H(\mathbf{q})[-k_d\dot{\mathbf{e}} - k_p\mathbf{e} + C(\mathbf{q}, \dot{\mathbf{q}}) + G(\mathbf{q})] \tag{4}$$

where $\mathbf{e} = \mathbf{q} - \mathbf{q}_d$ is the difference between desired and actual joint position, and k_p and k_d are proportional and derivative gains, respectively. The closed loop system has predictable $2n$–order linear behavior, but note that the method relies on calculating $H(\cdot)$, $C(\cdot,\cdot)$, and $G(\cdot)$ in real time with a centralized control computer.

A decentralized control law, which could be implimented using commercially available PID controller chips at each joint, will be of the form

$$\tau_{\text{applied}} = \rho D(-k_d\dot{e} - k_p e). \tag{5}$$

In this control law, D can be conside :d a diagonal approximation to the inertia matrix $H(\cdot)$. The parameter ρ is to be adjusted in the design process. The performance of the independent joint controller will not in general be as good as perfect computed torque control. Our aim is to select control parameters, k_d, k_p, D, and ρ that optimize system performance given the contrant of a decentralized law.

The degree of stability for the decentralized controller is given by

$$\mu = \lambda_{\min}[I - \rho^{-1}K^T W(\mathbf{q})^T H(\mathbf{q}) W(\mathbf{q}) K]\sigma 0 \tag{6}$$

where σ_0 is the magnitude of the real part of the *computed torque* closed–loop eigenvalue closest to the $j\omega$ axis, $K = (k_p, k_d)^T$, and

$$W(\mathbf{q}) = H(\mathbf{q})^{-1}D - I \tag{7}$$

which is a measure of how closely D approximates $H(\cdot)$. Note that the best possible performance is $W = 0$ giving $\mu = 1$. The closed loop settling time is

$$T_s \sim 3/\mu \tag{8}$$

and the closed loop tracking error is given by

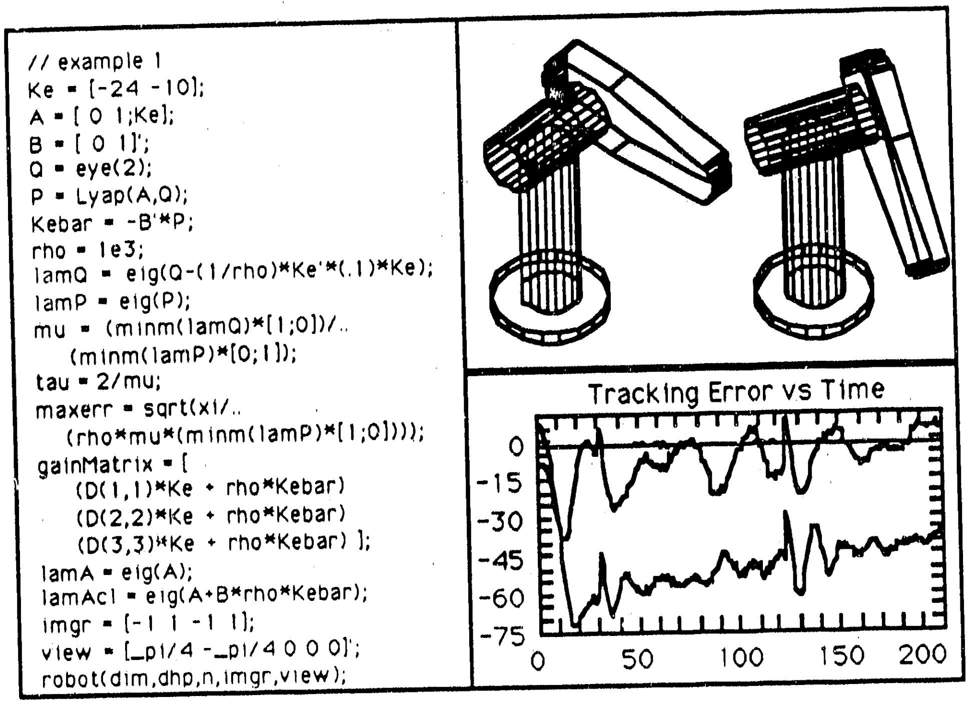
$$||e|| \leq \left| \frac{\xi}{\rho\mu} \right|^{1/2} \tag{9}$$

where ξ is a function of the desired trajectory accelerations and the magnitude of Coriolis and gravity forces. The designer must iterate on choices of K, D, and ρ in order to maximize μ, and minimize T_s and $||e||$. A portion of the MatC script file that does this is shown in Figure 4 which also shows the robot arm in simulated configurations and plots of simulation results.

The drawing of the robot arm is an example of how MatC can be customized for particular applications. A new object **Robot** was coded and attached to MatC through the standard function interface module (called **UserFunctions**). This new object constructs a 3-D polygonal representation of the arm given some defining geometrical dimensions and the Denavit–Hartenberg parameters (which include **q**, the arm's configuration). The object has methods that create projections onto a 2-D graphic port, as shown, or writes the polygons to a file for input to a program that produces a shaded color rendition. Using a MatC script file, pictures can be redrawn at different configurations to make a movie, if desired.

6. Conclusion

As control systems engineers are relying more and more on powerful personal computer workstations to aid in their design work, open–architectured software is beginning to play a key role in the engineering software environment. MatC is an example of this kind of easily customizable software that can perform general calculations, data handling, and graphics tasks. The object oriented design, combined with user access to sources, makes MatC adaptable to expanding needs.



```
// example 1
Ke = [-24 -10];
A = [ 0 1;Ke];
B = [ 0 1]';
Q = eye(2);
P = Lyap(A,Q);
Kebar = -B'*P;
rho = 1e3;
lamQ = eig(Q-(1/rho)*Ke'*(.1)*Ke);
lamP = eig(P);
mu = (minm(lamQ)*[1;0])/..
   (minm(lamP)*[0;1]);
tau = 2/mu;
maxerr = sqrt(xi/..
   (rho*mu*(minm(lamP)*[1;0])));
gainMatrix = [
    (D(1,1)*Ke + rho*Kebar)
    (D(2,2)*Ke + rho*Kebar)
    (D(3,3)*Ke + rho*Kebar) ];
lamA = eig(A);
lamAcl = eig(A+B*rho*Kebar);
imgr = [-1 1 -1 1];
view = [_pi/4 -_pi/4 0 0 0]';
robot(dim,dhp,n,imgr,view);
```

Fig. 4. *MatC Script File for Robot Controller Design and Graphic Output*

References

1. B. Cox, **Object–Oriented Programming: An Evolutionary Approach** Addison-Wesley, Reading, MA, 1987.
2. B. Armstrong, O. Khatib, and J. Burdick, *The Explicit Dynamic Model and Inertial Parameters of the PUMA 560 Arm*, **Proceedings IEEE 1986 International Conference on Robotics and Automation**, April, 1986, San Francisco, CA, 510—518.

Hierarchical Heterogeneous Symbolic Control: Lessons Learned from TEXSYS

B. J. Glass
NASA-Ames Research Center
Moffett Field, CA 94035
glass@pluto.arc.nasa.gov

Abstract

A multilayered approach to the symbolic control of complex electromechanical assemblies is discussed. An example of this approach is given in some recent tests of the Thermal Expert System (TEXSYS) in control of the Boeing Aerospace Thermal Bus System (BATBS), a prototype two-phase Space Station Freedom thermal bus. The BATBS hardware requires read-update-act cycles of under a minute, and it is subject to dynamic reconfiguration while operating. These performance requirements are addressed by hierarchical layering of model-based expert system software on a conventional numerical data acquisition and control system. Temporal and structural reasoning capabilities are found to be needed to identify all component faults. TEXSYS test results demonstrate both nominal control and fault recovery actions with the BATBS. Dynamic modification of the symbolic model used in this approach is compared to that of a classical numerical adaptive controller.

Introduction

Some complex space-based systems require constant monitoring and control -- parameters, configuration, and component health change with time. Current operational practice generally requires human operators to scan telemetry, watching for deviations from expected performance. In real-time, large-scale applications, such as Space Station Freedom (SSF) subsystems, this will prove expensive -- since many operators are required given the data processing limitations of humans. Transmission and processing delays, coupled with human inattentiveness, also tends to reduce safety and stability margins. By automating some or most of the monitoring, troubleshooting, and control of these dynamic space systems, the need for direct human involvement is reduced and robustness is improved. This paper discusses the approach used to implement "expert" control on a representative SSF prototype subsystem.

A symbolic model-based reasoning approach to identification and control, such as is taken here, is desirable because of its ability to follow changes in parameters, varying hardware configuration, and sensor failures; and, because of its ability to construct a qualitative model even in the absence of a detailed numerical model[1]. Frequent design changes, damage, and highly nonlinear components are to be expected in the creation and deployment of complex systems. Also, operational robustness is generally enhanced by the ability to operate and to reach at least partial conclusions with incomplete data[2].

A prototype thermal control system, or external thermal bus, for the Space Station Freedom was initially selected as a representative space system for a symbolic control application. The thermal bus used for TEXSYS tests was the Boeing Aerospace Thermal Bus System (BATBS) resident at the NASA Johnson Space Center. It is a complex, self-balancing system with many independent parameters, which has thus far made conventional dynamic numerical simulation infeasible[3].

To automate some of the operational functionality of a thermal test engineer, existing artificial intelligence techniques such as frame systems[4], data-driven programming and model-based reasoning[5] were employed to create a symbolic thermal bus model. Together with rules for conflict interpretation and tasks for representing procedural knowledge, this knowledge base comprised the core Thermal Expert System (TEXSYS). The core TEXSYS was layered on conventional software for data acquisition and control, creating a hierarchical "expert" (as defined by Åström and others)[6] or "symbolic" controller.[7] In a series of tests, this approach was demonstrated to be capable of real-time monitoring and control of the BATBS during both nominal operations and induced faults[8].

The first section of this paper discusses the control requirements for the BATBS. An overview of the layered symbolic controller approach used in TEXSYS is given in the second section, followed by an example and conclusions.

Problem Setting

SSF External Thermal Bus -- a nonlinear, dynamic electromechanical plant

Ground test prototypes of several two-phase thermal bus designs for space station use have been constructed for evaluation. Initial TEXSYS development assumed a series of different thermal bus designs, but was shifted to the BATBS prototype when the original target testbed delivery was delayed. A descendant of the BATBS architecture is now the baseline design for the SSF external thermal bus.

The physical design of this thermal architecture is discussed in more detail elsewhere.[3]

Given several highly nonlinear components, no accurate dynamic numerical simulation of this architecture has yet been developed, making the design of conventional numerical control laws difficult. Empirical performance descriptions of these black-box components has thus far been substituted for accurate mathematical models. Given these poorly-known components, the model is not easily used as a state estimator.

Thermal Bus identification and control requirements

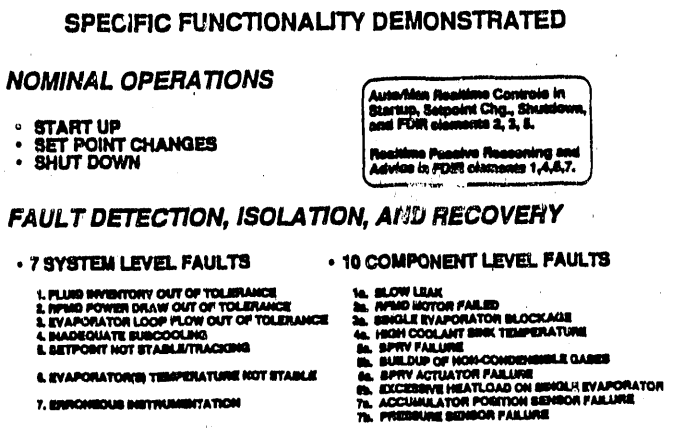
The conventional data acquisition and control system (DACS) software preceded the symbolic software at the BATBS site and was separately verified and validated. DACS sampled all sensors once every five seconds. To avoid consideration by the expert system of steady or slowly-changing data, data were filtered for significant changes. This filtering, as well as integration of the expert system with the DACS, was done by the TEXSYS Data Acquisition System (TDAS). Filter deadbands and alarm limits were defined for sensor values, trends (five-point fit: 25 sec. trend) and long-term trends (sixty-point fit: 5 min. trend), and were refined by trial-and-error during expert system validation testing. This testing procedure and the filtering services provided by the TDAS software are described in another paper.[9]

When hardware faults occur, the BATBS may reach off-nominal operating states, These states have been characterized in terms of broad, systemic faults caused by specific component faults.[10] TEXSYS was required to identify and react to all of the seven known system-level faults and to ten (of 34) component-level faults chosen by thermal engineers as most interesting or representative. Figure 1 shows a chart listing these faults. The trending capability provided by TDAS, together with discrete event histories, provided a temporal reasoning capability that was necessary in order to unambiguously identify these faults.

Most of the other 25 component faults were in fact tested with TEXSYS -- for instance, pressure sensor failure was the only formally required sensor fault, but the same capability could be (and was) also used to detect temperature, flow, delta-pressure, and position sensor failures. Certain destructive faults (e.g., explosive pipe rupture) were not tested because of safety, cost and downtime constraints.

Problems with using simple rule-based approaches

Given scores to thousands of sensors, and given that for safety's sake redundant sensors usually exist for critical parameters, simple rule-based approaches become increasing infeasible in proportion to the number of sensors. The reason for this problem is clear when one considers the combinations of sensors that would be have to be referenced in separate rules for each fault or required control action. Otherwise, sensor failures could render a simple rule-based controller useless, unable to rely on backup sensors. Also, simple rule-based expert controllers generally have no mechanism for identifying unanticipated conditions -- thereby limiting their stability.


SPECIFIC FUNCTIONALITY DEMONSTRATED

NOMINAL OPERATIONS

- START UP
- SET POINT CHANGES
- SHUT DOWN

Auto/Man Realtime Controls in Startup, Setpoint Chg., Shutdown, and FDIR elements 2, 3, 5.

Realtime Passive Reasoning and Advice in FDIR elements 1,4,6,7.

FAULT DETECTION, ISOLATION, AND RECOVERY

• 7 SYSTEM LEVEL FAULTS

1. FLUID INVENTORY OUT OF TOLERANCE
2. RPMD POWER DRAW OUT OF TOLERANCE
3. EVAPORATOR LOOP FLOW OUT OF TOLERANCE
4. INADEQUATE SUBCOOLING
5. SETPOINT NOT STABLE/TRACKING
6. EVAPORATOR(S) TEMPERATURE NOT STABLE
7. ERRONEOUS INSTRUMENTATION

• 10 COMPONENT LEVEL FAULTS

1a. SLOW LEAK
2a. RPMD MOTOR FAILED
3a. SINGLE EVAPORATOR BLOCKAGE
4a. HIGH COOLANT SINK TEMPERATURE
5a. SPRV FAILURE
5b. BUILDUP OF NON-CONDENSIBLE GASES
6a. SPRV ACTUATOR FAILURE
6b. EXCESSIVE HEATLOAD ON SINGLE EVAPORATOR
7a. ACCUMULATOR POSITION SENSOR FAILURE
7b. PRESSURE SENSOR FAILURE

Fig. 1. Required BATBS induced faults included in 1989 tests.

Approach

Hierarchical, heterogeneous levels of control

Broadly speaking, this approach implements control at four hierarchical layers. As shown in the example in Figure 2, the plant with its actuators and sensors is at the bottom. A conventional control layer provides millisecond-level responses, with hard-wired limit-checking of critical parameters. These controllers will shut down the plant if an unsafe state is approached, and cannot be altered by higher levels -- providing a fail-safe for higher levels of control. A procedural level of control implements checklists for activities such as setpoint changes. It also encompasses data filtering and general limit checks (corresponding to the DACS and TDAS software, as well as TEXSYS task operators described below). The core expert system is on top of the procedural layer, with current technology providing response times typically measured in seconds. On top is the human operator, responding to changing plant states in seconds to minutes, depending on attentiveness, data presentation, distractions, etc. The procedural and expert layers then supplant a bank of operators in reporting plant state to a given chief engineer or system manager. Other layered architectures have been proposed for real-time symbolic controllers, e.g. ARTIFACT[11]. Rather than a hierarchy of functionally-similar controllers assembled at many layers of abstraction, the architecture shown in Figure 2 has only a few layers of functionally-different controls. This relatively flat, heterogeneous hierarchy was found to speed real-time performance, perhaps at the expense of elegance.

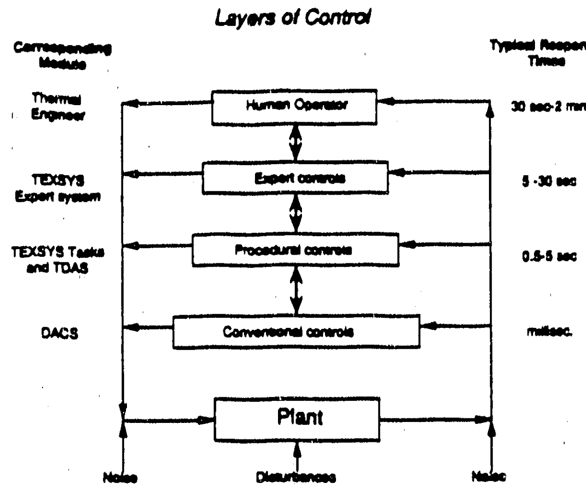


Fig. 2. A layered symbolic control architecture.

Useful AI Techniques

Mechanism modelling in TEXSYS was addressed with the Model Toolkit (MTK)[12], which used a hierarchy of frames to represent the natural taxonomy of subsystems, assemblies, branches, and components. Slots in these frames may refer to declarative knowledge (i.e., physical parameters), other frames (e.g., connections between component frames) or attached procedures. The Model Toolkit was built on the KEE[13] object-oriented programming environment. More detail on the AI techniques used by TEXSYS may be found in a separate paper[14].

Model-Based Monitoring and Control Design

Schematic-like Symbolic Model for Functional Simulation

In order to shift between hardware designs and respond quickly to field modifications of hardware, it was desired to decouple generic component behavior (how a valve behaves, how pipes or pumps function, etc.) from the behavior of a given thermal bus design. The approach taken in TEXSYS was to create a library of plant components, with default behaviors associated with them. The domain knowledge captured in this library was then preserved during changes from one target plant to another. This modular, component-oriented modelling approach also made ongoing plant design changes comparatively easier to follow, compared with procedural or simple rule-based approaches.

Given a library of relatively generic components, a specific plant design can be modelled by creating instances of components corresponding to the hardware, then creating connections between components to create a schematic-like topology. Sensor measurements can then be placed in this topology at their analogous locations and propagated along connections to create a functional simulation of the plant.

Model Conflicts and Collisions Trigger High-Level Responses

In its most general definition, model-based reasoning is diagnosis based on the comparison of a device's measured behavior with the expected behavior from the model. In TEXSYS, this comparison is handled in two ways: structural data conflicts in the model, or data collisions, are treated as abnormalities. Likewise, parameter conflicts with the model's expected values are signified by translation of a parameter value into a status which is not "nominal" (or not "steady," typically, for trends).

In the case of data collisions, all components touched by propagation of the conflicting values are marked as possible failure candidates. Candidate sets of component failures which would explain the collision are generated with a GDE-like approach [15], with joint probabilities calculated from the *a priori* expected failure probabilities of the affected components.

This architecture can then be imagined as having two fault identification processes working in parallel, responding to conflicts and collision in the model. The former uses rules to match off-nominal parameter states with known fault modes, while structural reasoning is very useful for detecting sensor failures and unforeseen faults. In either case, assertion that a given fault exists triggers an active value, which initiates a corresponding prestored fault recovery procedure and/or notifies the human operator.

Tasks: Procedural Control Layer

Executive Toolkit: Framework for Procedural Knowledge

All expert system-layer executive behavior, both for the TEXSYS system read-update-act cycle and for fault diagnosis and recovery, is implemented via intercommunicating goal-driven subprocesses, or tasks, created with the Executive Toolkit (XTK)[16]. XTK provides a high-level structured language for writing procedural task specifications. Tasks are invoked by creating a goal to be accomplished -- if multiple tasks exist which match the desired goal, processing takes the form of backward-chaining. Tasks can specify subgoals, which may trigger other tasks in turn. Tasks may access parameter values and status in the model, and may contain structured programming constructs (loops, recursion, conditionals).

The fault processing active value sets a goal of recovery from the given fault; any defined tasks which satisfy that goal are then initiated as independent processes, distinct from the TEXSYS executive tasks.

Tasks Responding to Model Changes: Spawning Independent Adaptive Agents

Some fault diagnosis and recovery tasks include branching and looping constructs, provided by XTK, which allow these tasks to control the BATBS dependent on the state of the model and its parameters. For example, the recovery task from one fault, that of excessive non-condensible gas (NCG), checks the end-to-end delta pressure (a measure of pump performance in the BATBS) periodically in a loop internal to the task. If the parameter status is "very low" at a given update, the task sets a subgoal of opening and closing a valve which vents vapor from the RFMD, which is then done by a command sent to the DACS layer; if the status is marginal or "low," the task does nothing for that cycle; if the status is nominal or better, the task marks the goal of recovery as satisfied and terminates. Given that a task may perform different control actions depending on the model's identified parameter values and statuses, TEXSYS implements a form of adaptive control somewhat analogous to a qualitative self-tuning regulator. Fault recovery and nominal control tasks are spawned and run as independent processes, separate from the main expert system loop.

Run-time Compromises Required for Real-Time Performance in TEXSYS

TEXSYS used a truth maintenance system (TMS) to index and follow which facts were true or untrue at a given time. Small floating point variations would in practice cause new strings of facts to be created along data propagation paths: the TMS would simply mark the previous facts as untrue, rather than purging them. This led to a slow accumulation of old beliefs in memory, resulting in slow performance degradation. It proved to be necessary to take TEXSYS offline every 4-5 hours for about twenty minutes to clear the TMS.

An initial performance problem was due to the slowness of using interpreted rules for propagation of data in the model, as well as for device behavior description and fault recognition. Initial TEXSYS cycle times with interpreted rules were over three hours. A rule compiler was developed which converted rules into compiled Lisp functions. These run-time rules ran roughly 3000 times faster than interpreted rules, allowing TEXSYS cycles to be measured in terms of seconds.

An inefficient indexing scheme for candidate fault sets caused expert system response times to rise linearly with the number of hypothetical worlds extant. This performance problem, plus the existence of unmodellable black-box components in the BATBS, caused more fault detection and control knowledge to be explicitly encoded in rules rather than left for the GDE mechanism to discover. Hypothetical reasoning was still found to be useful for detecting sensor failures and unforeseen faults.

TEXSYS Results

TEXSYS Met Project Requirements

In tests run with the BATBS at the NASA Johnson Space Center in August, 1989, TEXSYS successfully controlled the BATBS during all real-time nominal operations such as startup, shutdown and setpoint changes. TEXSYS successfully identified and acted on all 17 required induced system and component-level faults shown in Figure 1, as well as identifying some unplanned faults.

Multi-level Control Example

As examples, Figure 3 shows the diagnosis and recovery by TEXSYS of excessive non-condensible gases (NCGs), which causes a loss of pitot pump efficiency -- end-to-end delta pressure (BDP703)-- somewhat analogous to cavitation in an ordinary vane pump. This fault is often seen after setpoint temperature is lowered, causing the NCGs to come out of solution in the anhydrous ammonia working fluid. Since this was a steady-state mode of operation with respect to setpoint and heat loads, and the pump speed remained constant within nominal bounds (approx. 2920 rpm), and the net subcooling (another measure of BATBS performance) was nominal, then TEXSYS interpreted the drop in BDP703 beginning at 20:15 as due to excessive NCGs. It responded by notifying the human operator of the problem and requesting permission to purge the NCGs.

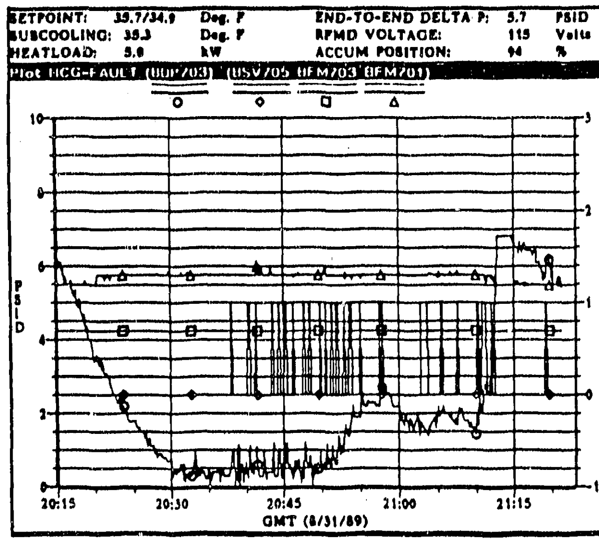


Fig.3. Excessive NCGs controlled by TEXSYS.

After operator approval was granted around 20:35, TEXSYS summoned a NCG recovery controller into existence. This independent process toggled the NCG purge valve, BSV705, as shown by the series of pulses in Figure 3. It did so by sending valve-open and valve-closed commands to the underlying DACS layer, which in turn varied voltage levels to trigger BSV705 (a solenoid valve).

The NCG controller was both value and rate-dependent on BDP703, as shown by comparing the pulses and their spacing to the BDP703 curve. Once BDP703 was stabilized within nominal bounds, the NCG recovery controller terminated itself.

Conclusions

The success of the hierarchical symbolic model-based control approach taken in TEXSYS demonstrates that a layered symbolic controller can be used to successfully control complex hardware in real time. The use of qualitative model-based and temporal reasoning in TEXSYS is one of the first applications of these techniques to online, real-time process control. Further research will focus on improving the real-time performance of these knowledge-based methods, including the processing of multiple simultaneous faults in real time.

A generic, component-oriented modelling approach is recommended in order to follow dynamic hardware configurations. A component-oriented rather than a system-oriented approach can avoid much rewriting of rules when components are rearranged in a device, as often happens during repairs, upgrades, and design changes of complex mechanisms. The operational success of TEXSYS demonstrates that the integration of a range of techniques (model-based and qualitative reasoning, classification systems, frame-based representations, temporal reasoning, and procedural reasoning) can be used to control and diagnose faults in certain relevant electromechanical systems in real time. Given their success in the TEXSYS demonstration, the model-based symbolic control methods discussed in this paper are a possible general approach to the automation of the monitoring and control of large, dynamic electromechanical systems.

Acknowledgements

The TEXSYS project reflects the work of over thirty people at two NASA centers, but space does not permit mentioning them all. However, the Executive Toolkit was created by Richard Levinson, and the TEXSYS top-level cycle was designed originally by William Erickson and Richard Owens. Mary Rudokas and Edmund Hack deserve special recognition for their knowledge engineering contributions. Project leadership was provided by Carla Wong and John Bull.

References

[1] Scarl, E., "Sensor Failure and Missing Data: further inducements for reasoning with models," *Proc. 1989 Workshop on Model-Based Reasoning*, American Assoc. for Artif. Intelligence, Detroit, MI, August 1989, pp. 1-6.

[2] Glass, B.J., and Wong, C.M., "A Knowledge-Based Approach to Identification and Adaptation in Dynamical Systems Control," *Proc. of the 27th IEEE Conference on Decision and Control*, Austin, TX, December, 1988, pp. 881-886.

[3] Bland, T.J., Downing, R.S., and Rogers, D.P., " A Two-Phase Thermal Management System for Large Spacecraft," *15th Intersociety Conference on Environmental Systems*, SAE Paper 851351, San Francisco, CA, July 1985.

[4] Minsky, M.,"A framework for representing knowledge," in P.Winston (ed.), *The Psychology of Computer Vision*, McGraw-Hill, New York, 1975, pp.211-277.

[5] Bobrow, D.G., ed., *Qualitative Reasoning about Physical Systems*, MIT Press, Cambridge, MA, 1985.

[6] Åström, K.J., Anton, J.J., and and Årzén, K.E., "Expert control," *Automatica*, Vol. 22, 1986, pp. 277-286.

[7] James, J.R., and Suski, G.J., "A Survey of Some Implementations of Knowledge-Based Systems for Real-Time Control," *Proc. of the 27th IEEE Conference on Decision and Control*, Austin, TX, December, 1988, pp. 580-585.

[8] Glass, B.J., "A Model-Based Approach to the Symbolic Control of Space Subsystems," *1990 AIAA Guidance, Navigation, and Control Conference*, AIAA Paper 90-3430, Portland, OR, August 1990.

[9] Hack, E.H., and DeFilippo, D. "Demonstrating Artificial Intelligence for Space Systems: Integration and Project Management Issues," *Sixth IEEE Conference on Artificial Intelligence Applications (CAIA-90)*, Santa Barbara, CA, March, 1990.

[10] "Space Station Prototype Two-Phase Thermal Bus System (TBS) Fault Detection, Isolation, and Recovery (FDIR)," Crew and Thermal Systems Division, NASA Johnson Space Center, September 8, 1988.

[11] Francis, J. and Leitch, R., "Towards Intelligent Control Systems," in *Expert Systems and Optimisation in Process Control*, ed. A. Mamdani and J. Efstathiou, Gower Technical Press, Aldershot, England, 1986, pp. 63-72.

[12] Erickson, W.E., and Rudokas, M.R., "MTK - An AI Tool for Model-Based Reasoning," *Third Conference on Artificial Intelligence for Space Applications*, V. II, NASA CP-2492, Huntsville, AL, November, 1987, pp. 1-5.

[13] Fikes, R. and Kehler, T., "The role of frame-based representation in reasoning," *Communications of the ACM*, Vol.28, Nov. 1985, pp. 904-920.

[14] Glass, B.J., Erickson, W. and K. Swanson, "Qualitative and Temporal Reasoning in TEXSYS: a Device-oriented Approach to Reasoning about Physical Systems," *International Workshop on the Principles of Diagnosis*, Stanford, CA, July 1990.

[15] deKleer, J. and Williams, B.C., "Reasoning about Multiple Faults," *Proc. Fifth National Conference on Artificial Intelligence*, Philadelphia, PA, August, 1986, pp. 132-139.

[16] Levinson, R., "Autonomous prediction and reaction with dynamic deadlines," *Proc. AAAI Spring Symposium*, Stanford, CA, March, 1990.

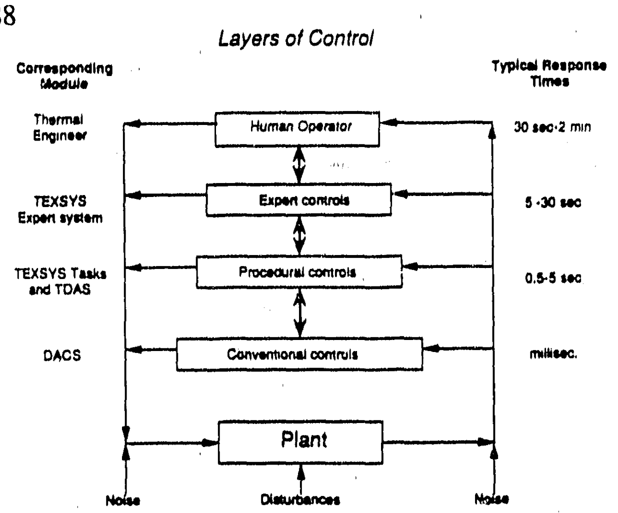


Fig. 2. A layered symbolic control architecture.

Useful AI Techniques

Mechanism modelling in TEXSYS was addressed with the Model Toolkit (MTK)[12], which used a hierarchy of frames to represent the natural taxonomy of subsystems, assemblies, branches, and components. Slots in these frames may refer to declarative knowledge (i.e., physical parameters), other frames (e.g., connections between component frames) or attached procedures. The Model Toolkit was built on the KEE[13] object-oriented programming environment. More detail on the AI techniques used by TEXSYS may be found in a separate paper[14].

Model-Based Monitoring and Control Design

Schematic-like Symbolic Model for Functional Simulation

In order to shift between hardware designs and respond quickly to field modifications of hardware, it was desired to decouple generic component behavior (how a valve behaves, how pipes or pumps function, etc.) from the behavior of a given thermal bus design. The approach taken in TEXSYS was to create a library of plant components, with default behaviors associated with them. The domain knowledge captured in this library was then preserved during changes from one target plant to another. This modular, component-oriented modelling approach also made ongoing plant design changes comparatively easier to follow, compared with procedural or simple rule-based approaches.

Given a library of relatively generic components, a specific plant design can be modelled by creating instances of components corresponding to the hardware, then creating connections between components to create a schematic-like topology. Sensor measurements can then be placed in this topology at their analogous locations and propagated along connections to create a functional simulation of the plant.

Model Conflicts and Collisions Trigger High-Level Responses

In its most general definition, model-based reasoning is diagnosis based on the comparison of a device's measured behavior with the expected behavior from the model. In TEXSYS, this comparison is handled in two ways: structural data conflicts in the model, or data collisions, are treated as abnormalities. Likewise, parameter conflicts with the model's expected values are signified by translation of a parameter value into a status which is not "nominal" (or not "steady," typically, for trends).

In the case of data collisions, all components touched by propagation of the conflicting values are marked as possible failure candidates. Candidate sets of component failures which would explain the collision are generated with a GDE-like approach [15], with joint probabilities calculated from the *a priori* expected failure probabilities of the affected components.

This architecture can then be imagined as having two fault identification processes working in parallel, responding to conflicts and collision in the model. The former uses rules to match off-nominal parameter states with known fault modes, while structural reasoning is very useful for detecting sensor failures and unforeseen faults. In either case, assertion that a given fault exists triggers an active value, which initiates a corresponding prestored fault recovery procedure and/or notifies the human operator.

Tasks: Procedural Control Layer

Executive Toolkit: Framework for Procedural Knowledge

All expert system-layer executive behavior, both for the TEXSYS system read-update-act cycle and for fault diagnosis and recovery, is implemented via intercommunicating goal-driven subprocesses, or tasks, created with the Executive Toolkit (XTK)[16]. XTK provides a high-level structured language for writing procedural task specifications. Tasks are invoked by creating a goal to be accomplished -- if multiple tasks exist which match the desired goal, processing takes the form of backward-chaining. Tasks can specify subgoals, which may trigger other tasks in turn. Tasks may access parameter values and status in the model, and may contain structured programming constructs (loops, recursion, conditionals).

The fault processing active value sets a goal of recovery from the given fault; any defined tasks which satisfy that goal are then initiated as independent processes, distinct from the TEXSYS executive tasks.

CASE Products for Knowledge Based Systems Design and Development in Ada

Dr. James R. Greenwood
Gregory Stachnick
Dr. H. Stephen Kaye
Robin Wada
Jorge Gautier

July 1990

Advanced Decision Systems
1500 Plymouth Street
Mountain View, CA 94043

Abstract

During the past decade significant progress has been made in demonstrating the utility of Knowledge Based System (i.e., Artificial Intelligence) technologies in diagnostics, situation assessment, and planning applications. These technologies have enabled more complex problems to be addressed than could otherwise be handled with classical algorithmic software approaches. Some of the underlying knowledge based technologies have matured and have been incorporated in "AI tools or shells" available on the market today. While these tools allow the application of these advanced technologies by 'experts", they have not bought knowledge based system development into the general software engineering community.

During the past several years, we have been developing engineering oriented products that allow generation of knowledge based applications on embedded systems written in Ada. These products combine pictorial CASE front-ends for design and specification, sophisticated auto-generation (i.e., compilation) systems that convert the specification of knowledge based systems directly to Ada code, and real-time run-time environments for execution of knowledge based applications within the multi-tasking Ada environment.

The first of these CASE products is an Ada Object Base[1] (AOB) that provides general purpose object-oriented facilities (i.e., classes/objects/methods), knowledge-base capabilities for incremental retraction of object changes, hypothesis management, and persistence (i.e., save/restore) delivered in the Ada run-time environment. The second CASE product is the Procedural Reasoning System[2,3] (PRS-Ada) that provides goal-directed and data-directed inferencing typically employed in advanced planning and resource allocation applications.

[1] Work supported under DARPA/MICOM initiative in Real-Time Mission Planning
Contract DAAHOI-90-C-0164

[2] Work partially supported under DARPA/BRL Airland Battle Management (ALBM) system
Contract 7840B0820M

[3] Work partially supported under RADC/DARPA Survivable Adaptive Planning Experiment (SAPE)
Contract 89925041

These first products incorporate advances in pictorial CASE, advanced development environments, compilation of knowledge base constructs into Ada, and inference control directed towards engineering embedded systems in Ada. This paper provides an overview of these products, as well as, a model for software engineering products that augment the development and delivery of knowledge based systems in Ada.

1.0 Introduction

During the last ten years significant advances have been made in demonstrating the utility of advanced technologies to automate problems that heretofore could not be automated via traditional approaches. Knowledge Based systems and/or expert systems (i.e., Artificial Intelligence) is one such technology that has been demonstrated on numerous projects and applications in both the Government and Commercial arena. However, with the maturing and wider acceptance of these technologies, the tools for developing knowledge based applications have fallen behind the needs of the market. That is, applications now require robustly engineered solutions delivered on standard hardware platforms in standard programming languages. In addition, there is wide need for software engineers to develop knowledge base applications, as opposed to a need for specialized knowledge base system developers. Thus, a need for products that can allow a typical software engineer to utilize knowledge based system technology exists.

In addition to knowledge based systems, significant advances have been made in two other areas: compilers and Computer Aided Software Engineering (CASE). Compiler technology has matured tremendously in the last decade in the ability to compile (i.e., generate/convert) advanced constructs into executable code. The advanced compilation techniques designed to compile Ada, such as generics, tasking, strong type checking, and separate compilation, are now well understood. Also, generating highly optimized code for a wide range of machine architectures is now possible.

CASE has emerged over the past five years as a set of tools and technology for defining and augmenting the software engineering process. Typically what is referred to as CASE is various forms of diagramming of the requirements, design, and specification of software. Several products allow simulation of the specification and designs to ensure correctness and completeness. Several products allow generation of code or code templates from the diagrams or specifications. CASE has been traditionally focussed on MIS or real-time front-end applications; not on knowledge based systems.

These three technologies for knowledge based systems, CASE, and compilation as combined into a single product could provide tools for software engineers tailored to knowledge based systems development.

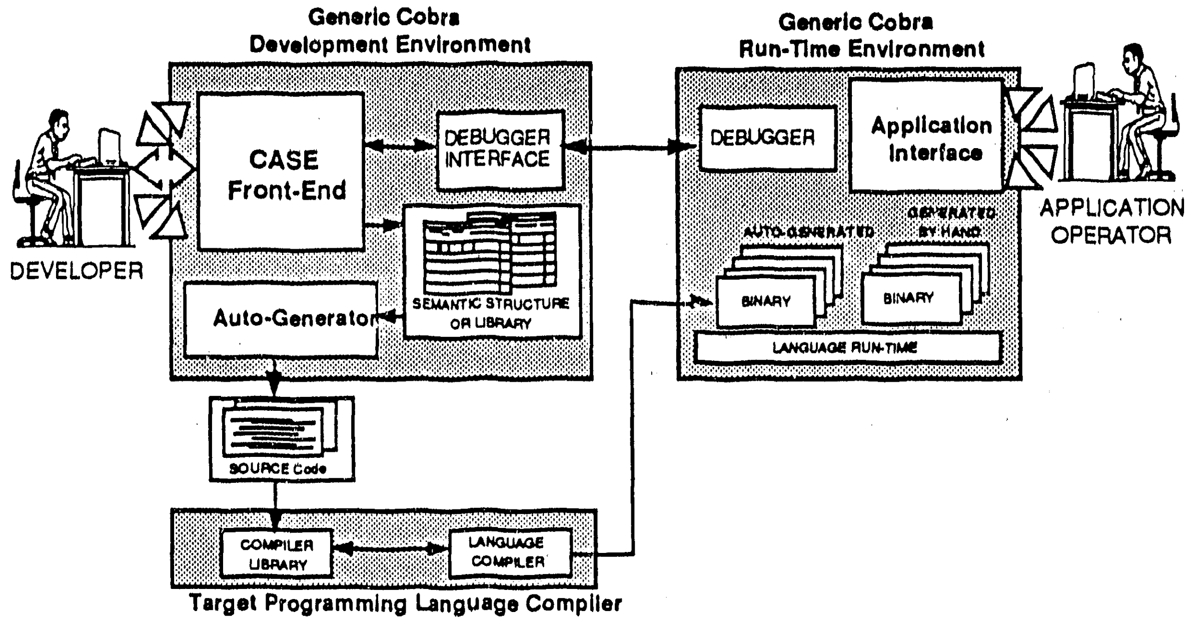
During the last three years, Advanced Decision Systems has developed a set of software engineering tools that combine CASE and compilation technology in development and run-time environments that facilitate the engineering of knowledge based systems in Ada. These tools are part of a family of products called Cobra. The first product being the Ada Object Base and the second being PRS-Ada.

2.0 Common Architecture

The Cobra family of products is is based on the approach of combining advanced development environments, CASE design tools, advanced compiler technology, and separate real-time run-time environments. The product s are designed to deliver fully engineered applications for real-time embedded system applications. The products have a common set of components:

Development Environment:
CASE Front-end - a pictorial CASE development environment for designing and specifying the software or system.
Auto-Generator - a compiler or translator that converts the design and specification to executable code.
Run-Time Environment - the run-time support for the executable code that allows testing and debugging interactively tied back to the CASE front-end.

This common architecture for each Cobra product is shown in the figure below:



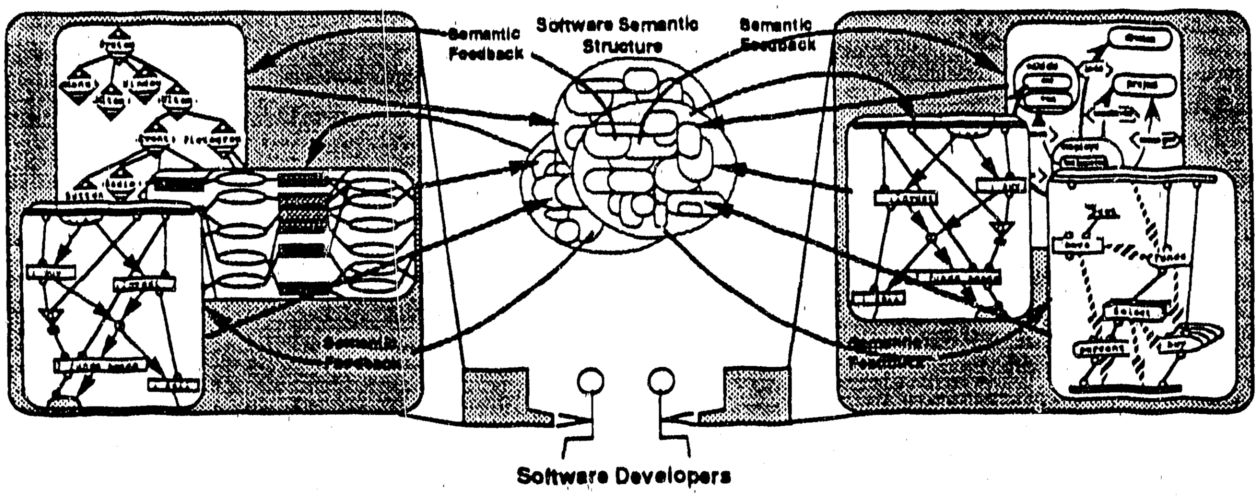
These three elements underlay the initial Cobra products PRS-Ada and Ada Object Base that are described more fully later in this paper.

2.1 Pictorial CASE

Each product contains an integrated CASE environment referred to as a Pictorial Programming System that allows the construction and generation of software. **Pictorial Programming** is constructing and testing software through a series of pictures called Pictograms. All aspects of the software design, development, and testing process are performed interactively through multiple pictograms on workstation graphic displays.

The software development process proceeds through a set of pictograms specific to the problem solving approach being employed in the software application. Pictograms for data and relationships are available, as well as, pictograms for code development. Pictograms are syntactic and semantic pictures specific for each type or operation.

Multiple developers construct the software concurrently through pictograms on their individual workstations.

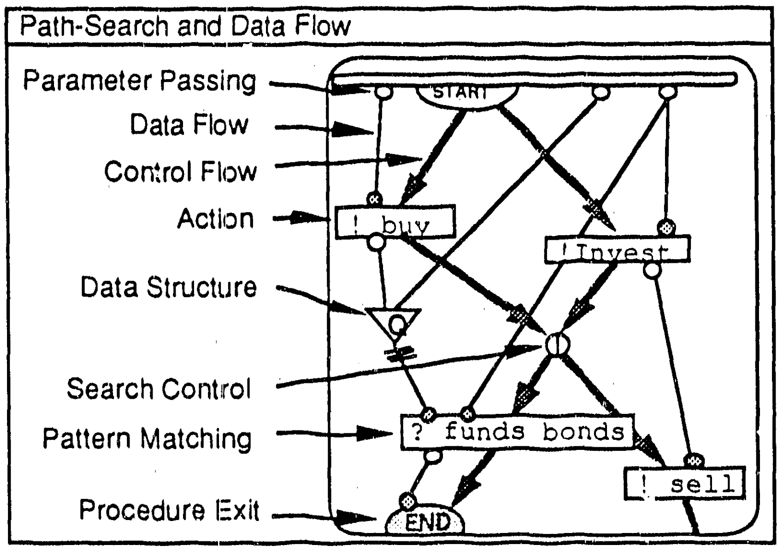


Each developer has multiple views (i.e., pictograms) into the software that is being constructed. The Software Semantic Structure is a complete representation of the concept, specification, design, code, and configuration of the software.

2.1.1 The Pictorial Language

Each Cobra product's language has been developed originally as a non-lexical pictorial language specific to the product domain (e.g., data and objects for the Ada Object Base). Typically, CASE products are merely diagramming tools that have limited underlying semantics with no compilation ability, severely limiting their usefulness. The Cobra product line supplies pictorial programming for each domain.

Each pictogram embodies specific syntax and semantics, allowable iconic representations, and rules on how the pictogram can be manipulated. These pictograms replace both design diagrams from traditional CASE tools, as well as lexical programming languages. For instance, the figure below shows a typical pictogram that embodies a complete set of programming functions in a pictorial language.

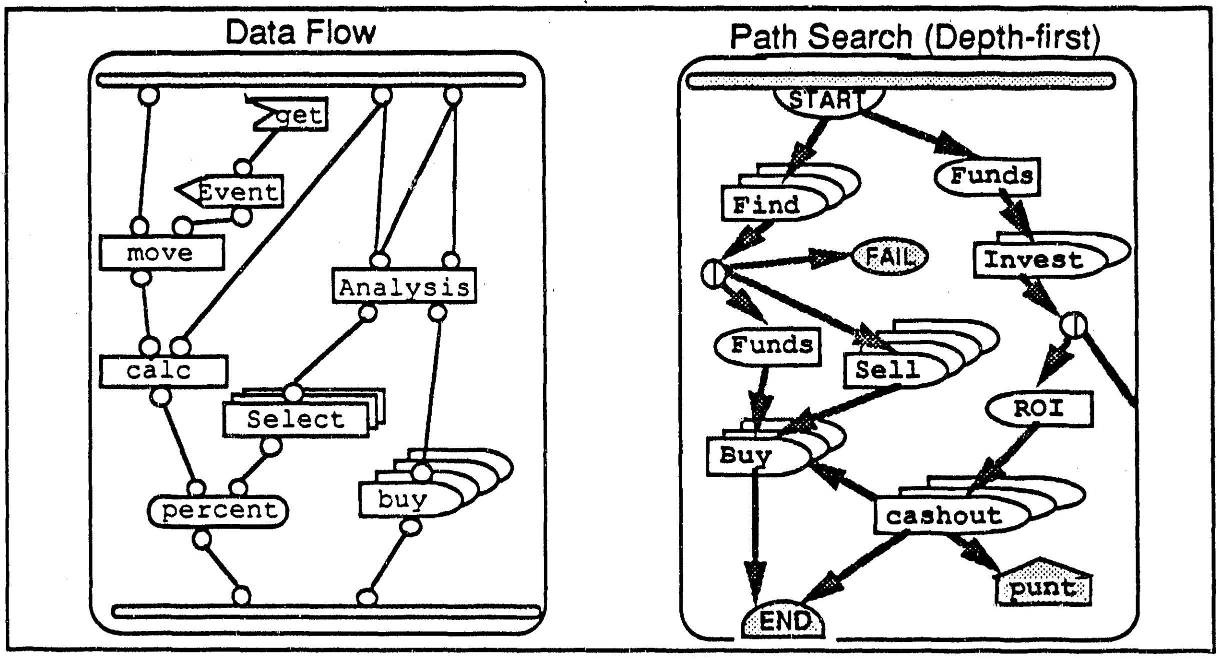


This pictogram can be converted by the Auto-generator into directly executable code.

2.1.2 Pictograms

Pictograms are the fundamental representation of design, implementation and execution of programs in the Cobra system. These pictograms are grouped into several categories: concept formation, execution paradigm, group, relation, physical, and interface categories. Examples of the execution paradigm pictograms are described below.

Execution Paradigm - These pictograms are the programming (or coding) level of the development system such as PRS-Ada. Execution pictograms compile directly into executable code and replace the traditional lexical programming language of the system.



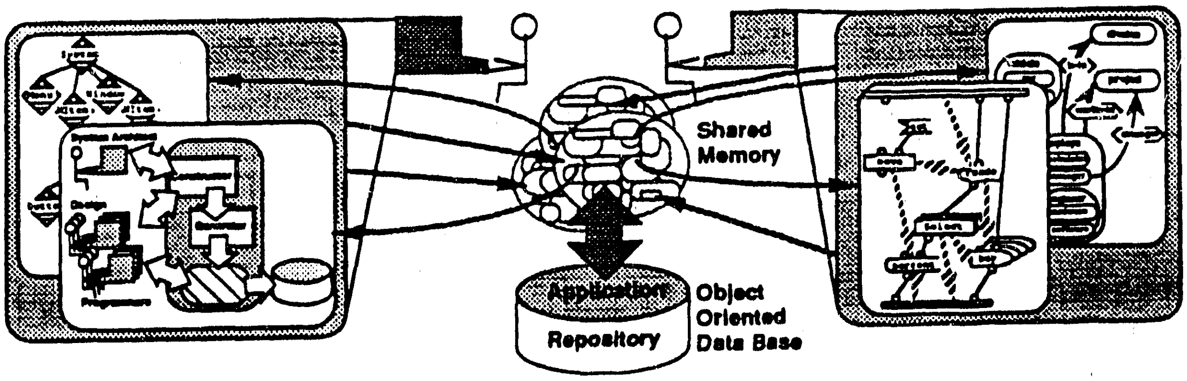
2.1.3 Pictogram Connections

Even though the pictograms can be viewed as top-down design, development need not proceed in that manner. A developer will typically move "up and down" through sets of pictograms, modifying the design so that coding is easier, changing data structures to reflect prototype results, and making other such changes. This feature is supported by the Software Semantic Structure within the pictorial programming environment.

Pictograms are interconnected through the Software Semantic Structure that contains the representation of the software being developed. This complex structure provides multiple levels of abstraction to represent the design and description of the software. This Structure keeps these views consistent and semantically correct. Changes in one pictogram are reflected directly as changes in related pictograms, provided the change is semantically correct between the pictograms. This extensive "checking" substantially reduces the chance for error and speeds the development process. Incomplete or inconsistent modifications of the design or code for the system are nearly impossible to make. Thus, higher quality software can be developed and maintained with less people in less time.

This Cobra architecture provides high development performance and compatibility with evolving CASE standards. Development performance is gained by having the Software

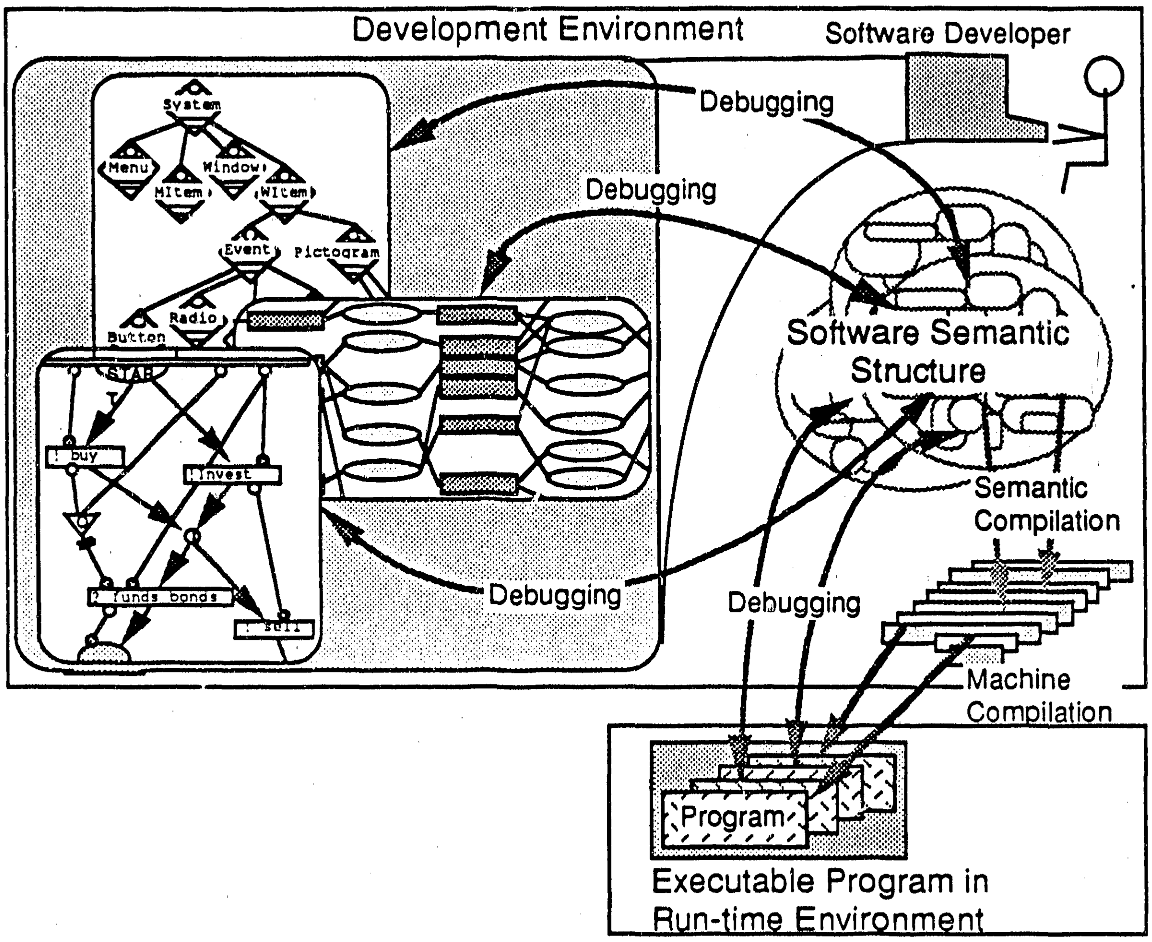
Semantic Structure in a distributed shared memory between the workstations, not on a disk or in a data base.



Archival to disk of the software design and development information is only done when necessary to preserve information between development sessions. This allows a factor of 10 to 100 improvement in the interactive development and checking aspects of the system over comparable repository-based CASE products. Cobra is specifically designed to satisfy the professional developer who is sensitive to this development performance.

2.2 Pictogram Compilation

A unique aspect of the Pictorial Programming System is that these pictograms compile directly into executable code that can be directly debugged at the pictorial (i.e., graphical) level. The Cobra products incorporate a unique compilation process that compiles the pictograms into executable code as depicted in the figure below.



The development environment compiles, links, and loads the software into a separate run-time environment on a target processor. The executable program is linked back to the pictograms in the development environment for full interactive debugging of the software through the pictogram representations. This approach allows detailed debugging and test of real-time software through the high-level pictogram representations. When debugging and test is complete, the connection between the development and run-time environment can be removed to allow the resulting software to run independently, perhaps on an embedded real-time processor.

3.0 The Ada Object Base

The Ada Object Base (AOB) product is the first Cobra product. It provides a CASE tool to build Ada object bases for applications, and the run-time support in Ada to test, debug, and deliver the applications. AOB is one of the few tools that really allows transition demonstration of Artificial Intelligence knowledge bases into operational systems written in Ada.

The AOB is an object oriented knowledge base for the Ada domain, and a development environment for specifying, creating, and browsing the objects/knowledge. The Ada Object base provides all the facilities typically associated with an object oriented language (e.g., C++) and a knowledge base development system, but delivered in the Ada environment.

The AOB supports for an Ada framework the following features:

Object Oriented Language Features:
- Class Definitions
- Dynamic Object Creation (Instances)
- Inheritance
- Methods on objects and classes in Ada or in PRS-Ada
- Specialization of Methods
- Class Variables
- Object initialization

Knowledge Base Features:
- Persistence at Class, Instance (i.e., object), or module (i.e., package) level
- Transaction History supporting checkpointing and incremental retraction.
- Multiple Hypotheses Support (i.e., worlds)

CASE Development Features:
- Pictorial Entity-Relationship Specification system
- Pictorial Browser/Editor/Debugger for Class, Objects, and Methods

The Ada Object Base is patterned after the object-oriented systems developed in CLOS, SmallTalk, C++, and Objective-C. The AOB execution structures, and object representations mimic implementation details of Objective-C, but mapped to the Ada record, pointer, and packaging constraints.

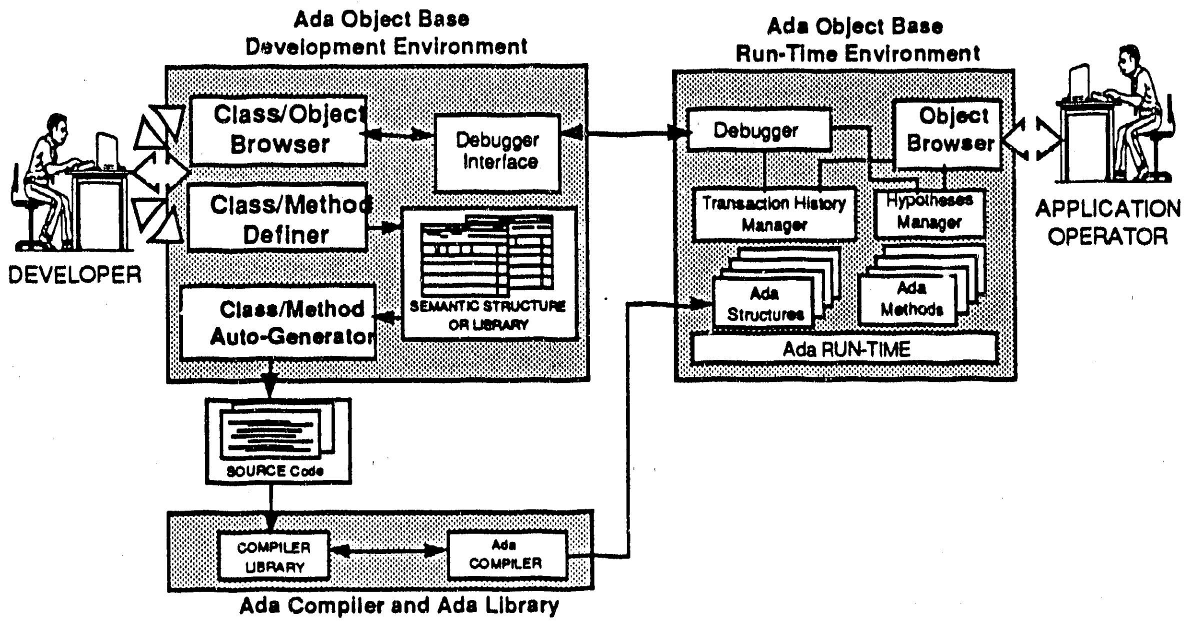
The AOB knowledge base features allow RAM-resident knowledge bases that support data dependent backtracking, multiple hypotheses support, and methods to save and restore the data. The complex internal mechanisms and bookkeeping for these capabilities are handled by the AOB, freeing the application programmers to concentrate on reasoning strategies. The initial features are useful in planning, resource allocation, scheduling, and data fusion applications.

The development environment for the AOB is a mini-CASE environment that allows pictorial definition of the classes and objects via extended entity-relationship diagrams that are auto-generated (i.e., compiled) into the Ada data declarations and execution structures. The development environment also provides pictorial editing and browsing of the classes, objects, and methods.

The AOB mimics product features from Objectworks, from the KEE frame system, and from data base CASE tools (e.g., Oracle CASE tool) in a single tool for developing knowledge bases targeted to applications written in Ada. The AOB is either a stand-alone product or can be utilized as an integral part of PRS-Ada product (See next section).

3.1 Architecture

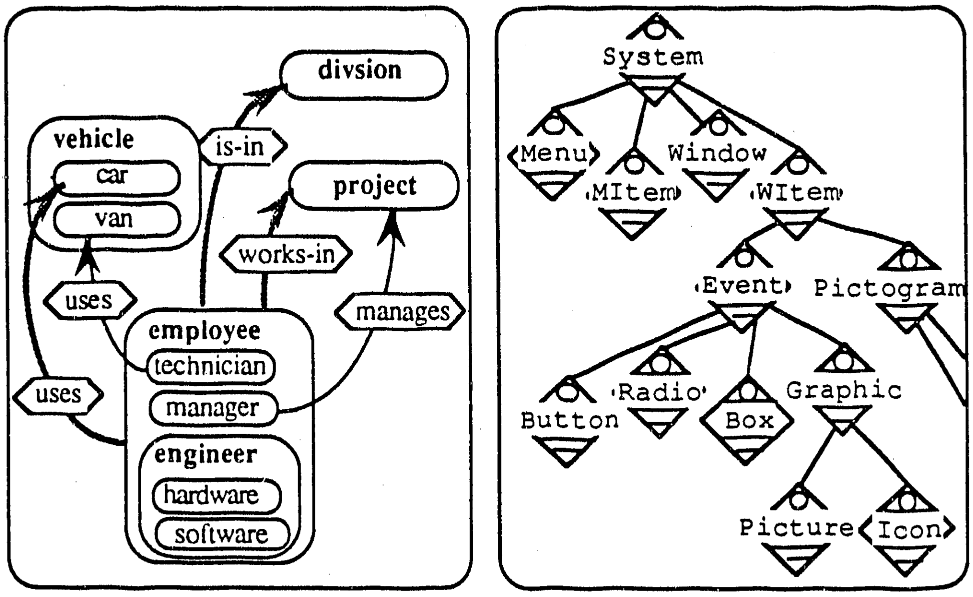
The figure below the shows the architecture of the AOB development and delivery environments.



As with all the tools in the Cobra product family, the AOB tool has a pictorial CASE front end, an auto-generate feature for producing code, an application run-time, and an interactive debugging environment coupled back to the pictorial CASE tool.

3.1 E&R and Class Pictograms

Pictorial Specification: The class definitions, class hierarchy, and composite class definitions are defined via the Class pictogram shown on the right in the figure below. Composite objects (i.e., part-of relationships) are defined with the extended entity-relationship (E-R) diagrams (i.e., pictograms) shown on the left below.



The extensions provide the extra object oriented features typically not found in traditional entity-relationship diagrams. These Class and E-R pictograms are converted by the auto-generator to the Ada source code declarations and run-time structures that provide the object base features.

Class/Object Browser: The browser utilizes the pictorial class, object, and method pictorial representation, showing structures, values, relationships, inheritance, and composite pictograms. Interacting through these pictograms allows rapid understanding of the specified object base. The browser provides the interface to the persistence (i.e., save/restore) mechanism and overall development environment.

AOB Classes, Objects and Methods: The AOB classes are defined both as Ada types and as associated Ada data declarations that hold method connections and class variables. The object instances are dynamically created and discarded from the Ada heap. Both class and instance methods can be provided in Ada, or as PRS-Ada procedures. Ada methods are written as Ada procedures with a specific calling sequence. Methods are bound to the classes or object instances at compile time. Method specialization is supported. Composite objects that provided "part-of" relations are also provided.

3.2 Run-time Support

The AOB supports the data dependent backtracking feature allowing both incremental backtracking and checkpointing to changes to the object base. In addition, the object base supports a multiple hypothesis (i.e., world) mechanism necessary for reasoning or handling multiple alternatives or sets of data. The persistence mechanism is auto-generated ada code that saves and restores object instances and their definitions to/from a disk. Single objects, all objects in a class, and all objects in a module can be save/restored. Future versions of the AOB will allow interface to data bases for loading (i.e., populating) the AOB. The AOB run-time also has a remote debugging facility that allows the object base to be browsed and/or debugged through the graphical CASE facilities.

3.3 Auto-Generator

The auto-generator is the component of the AOB development environment that converts the pictorial specification, method definitions, and class structures into a complex set of ada source code that provides the application specific AOB. The resulting Ada source code can be linked with "hand-written" Ada code to comprise an application written in Ada. The development environment runs any standard Ada compiler and Ada library mechanism as a slave to this generation process. The Ada compiler produces the executable program from the auto-generated Ada code and the hand-written Ada code.

3.4 Development History

The AOB has been under development during the past year in support of the Survivable Adaptive Planning Experiment (SAPE) for RADC/DARPA, the Advanced Planning System (APS) for RADC/TAC, and the Real-time/Mission Planning (RT/MP) for DARPA/ASTO. A subset of the AOB system is now in beta test on SAPE.

The AOB is a unique capability being developed at ADS in support of multiple projects. It ultimately provides a path for knowledge based applications into an Ada delivery environment of operational systems.

4.0 The PRS-Ada Product

PRS-Ada is a graphical (i.e., pictorial) software development tool for designing and delivering knowledge based systems in an Ada domain. PRS-Ada provides facilities for pictorial knowledge capture, pictorial development of reasoning strategies, and pictorial testing and debugging. PRS-Ada provides an automatic code generation facility that converts the pictorial forms to standard Ada source code.

The PRS-Ada pictorial language and run-time support provides the following capabilities:

Goal-Directed Reasoning - The language supports the execution of procedures to accomplish specified goals. The execution strategy allows the search for a line of reasoning that satisfies a set of goals. Satisfied goals are stored in a global achievement base.

Dependency-Directed Backtracking - As part of the execution strategy for procedures, dependency-directed backtracking with retraction of knowledge changes is supported. This provides capabilities similar to Prolog that are useful in planning and other goal-directed search strategy systems..

Data-Directed Invocation - Trigger conditions can be specified on procedures that match against data in either the goal achievement base or knowledge in the Ada Object Base (AOB). When a trigger condition is met then the associated procedure will execute.

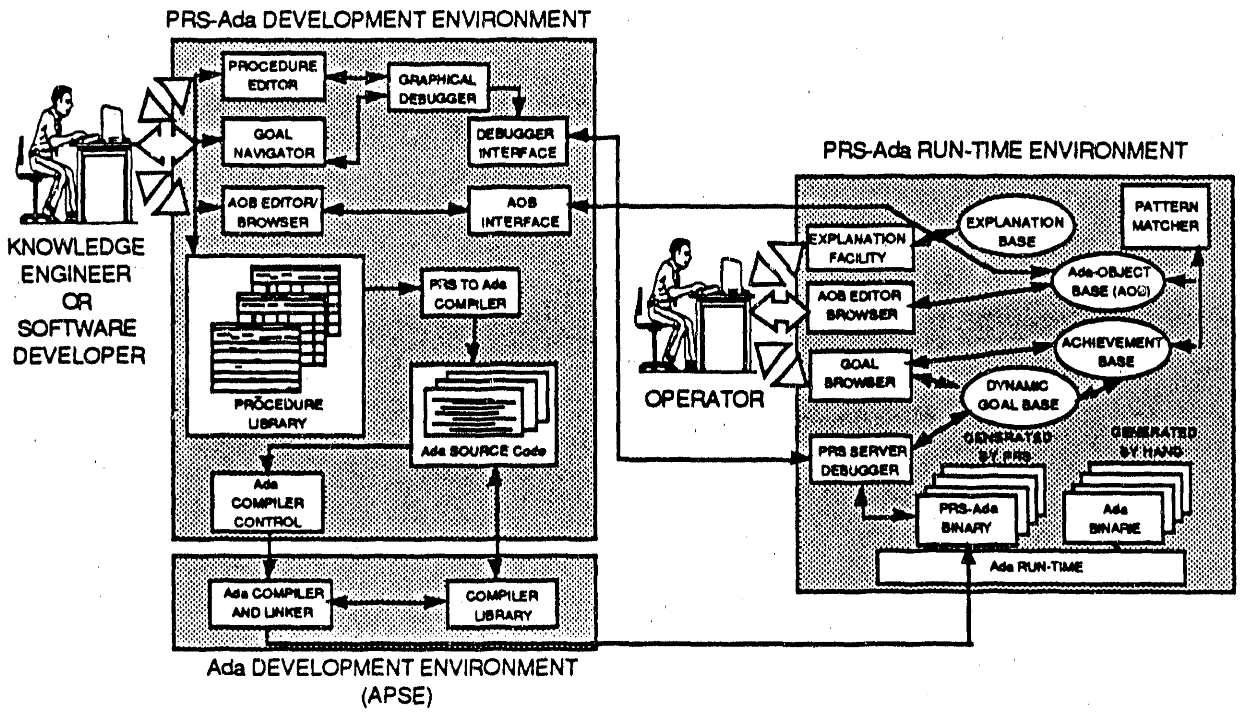
Multiple Hypotheses Support - The procedure execution, achievement base, and object base support multiple hypothesis reasoning. That is, alternative lines of reasoning can be pursued providing a means to generate multiple alternatives directly. This provides a feature similar to the world mechanism in KEE by Intellicorp but designed to provide much more efficient execution.

Pattern Matching on Achievements and Data - PRS-Ada pattern matches against achievements in the achievement base and against data/facts in the AOB. Matched facts or data are utilized to control inference. For instance, this allows reasoning about the plan generation (i.e., reasoning) process, and the generated plan in a planning system.

Explanation Facility - The explanation facility provides interactive evaluation and interpretation of the reasoning results through a flexible hierarchical explanation capability.

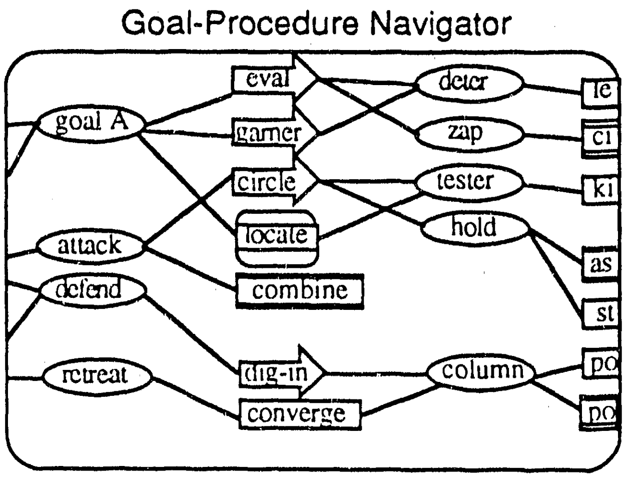
4.1 Architecture

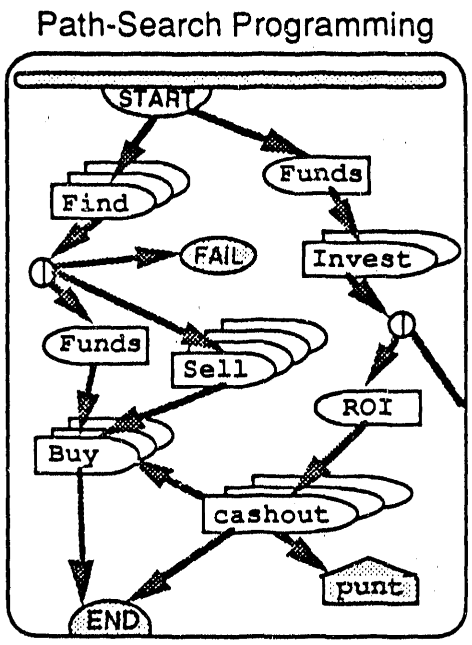
The PRS-Ada system architecture is shown in the figure below:



4.2 Pictograms

PRS-Ada utilizes two primary pictograms: Path-Search and Goal-Procedure Navigator. These pictograms shown below allow goal-directed reasoning processes to be encoded, and compiled directly into source code.





The path-search pictogram allows encoding of lines of reasoning. Specific operators for achieving goals, retrieving goals from the achievement base, and for testing if goals have been achieved are provided by icons in the language. The flow between the icons specifies a depth first search strategy during execution.

The Goal-Procedure Navigator pictogram specifies the relationship between goals and sub-goals, and between goals and the procedures that can achieve those goals. These two pictograms are coupled through the semantic structure that represents the software system. Other pictograms are being added to the PRS-Ada to allow other search strategies (i.e., breadth-first) to be employed.

4.3 Development Environment

Like all the Cobra products, PRS-Ada has distinct development and run-time environments. The development of procedures and knowledge proceeds in a two ways: first, reasoning strategies are captured through knowledge elicitation and captured in the graphical procedures; and second, knowledge (i.e., objects, facts, data) is solicited and captured in the Ada Object Base (AOB) (see: previous section - AOB). The development environment supports the compilation of these declarations and definitions directly into Ada source code and dynamic knowledge bases based on Ada constructs and functions. The resulting reasoning system is thus directly compatible with any Ada application or any Ada compiler and run-time support system. PRS-Ada has been interfaced to three different Ada compilers/environments: Alsys, Verdix, and Telesoft on two different hardware platforms (e.g., Sun-3 and Sparcstation). The current system runs under X-windows/UNIX on the Sparcstation. Porting PRS-Ada to additional hardware platforms is underway.

4.4 Auto-Generator

The auto-generator is the component of PRS-Ada development environment that converts the pictorial specification into Ada source code that provides the application specific reasoning code. The resulting Ada source code can be linked with "hand-written" Ada code to comprise a full application written in Ada. The development environment runs any standard Ada compiler and Ada library mechanism as a slave to this generation process. The Ada compiler produces the executable program from the auto-generated Ada code and the hand-written Ada code. Several auto-generation options are available allowing various levels of debugging information to be embedded in the generated code. This allows fully interactive graphical debugging of down-loaded (into the run-time system) code, and then later recompilation to remove debugging hooks in the final production embedded systems code.

4.5 Run-time Environment

After the reasoning strategies and knowledge have been compiled into Ada the Ada source is compiled and linked to form the application run-time. This application run-time can be coupled directly back to the development environment for graphical debugging of reasoning strategies, and knowledge and goal decompositions. In addition the run-time environment can be configured to allow editing and modification in an operational system of the knowledge and goals as required. The run-time also has a remote debugging facility that allows the graphical (i.e., pictorial) procedures to be debugged, via traces, breakpoints, and inspection directly though the pictorial formulation in the CASE front-end.

A significant advantage discovered in transitioning from the original PRS (i.e., PRS-CommonLisp) to PRS-Ada is the run-time performance of the generated application. The run-time performance of PRS-Ada procedures is approximately fifty (50) times that of the PRS-CommonLisp procedures on the identical platform (i.e., Sun-3). The Sparcstation PRS-Ada provides another factor of three (3) in performance over the Sun-3 due to a faster hardware processor. Thus, run-time performance of the knowledge based application is quite good as compared to typical CommonLisp based knowledge based systems.

4.6 Development History

PRS-Ada has been under development during the past three years in support of the AirLand Battle Management program (DARPA/BRL) for a Army Corps and Division level planning system. It is currently supporting the Survivable Adaptive Planning Experiment (SAPE) for RADC/DARPA, the Advanced Planning System (APS) for RADC/TAC, and the Real-time/Mission Planning (RT/MP) for DARPA/ASTO.

PRS-Ada is a powerful tool that satisfies the major requirements of many DoD contracts for delivering knowledge based systems in Ada. The tool provides high performance goal-direct and data-directed reasoning in an Ada run-time environment, yet retains the graphical (i.e., pictorial) interactive knowledge engineering metaphor. PRS-Ada is a pictorial CASE tool for delivering knowledge base applications in Ada.

5.0 Summary

The Cobra product family is a set of CASE products for knowledge bases systems design and development. The first products, Ada Object Base and PRS-Ada, are targeted to embedded system applications in the Ada programming language. These products provide engineering oriented tools for utilizing advanced technology in production embedded systems.

Prototypes of these products have demonstrated significant productivity gains in developing complex systems, and have allowed a typical software engineer to build systems heretofore requiring a knowledge based system developer.

TOOL NEEDS FOR A BEHAVIOR-BASED APPROACH TO DISTRIBUTED INTELLIGENT CONTROL

S. Harmon, D. Payton & D. Tseng
Hugher Research Laboratories
Malibu, CA 90265

INTRODUCTION

A distributed intelligent control system can be modelled as a collection of individual intelligent agents which coordinate their actions through communications and common knowledge. An agent in this collection can be constructed as a set of behaviors whose outputs are integrated by some form of arbitration logic. This class of designs of intelligent systems includes subsumption architectures, community models and some biological paradigms.

Each agent contains its own sensors, computing, controls, actuators, local knowledge and communications. A behavior is a control loop which determines some aspect of the agent's responses to a particular set of sensed conditions. Arriving communications simply are treated as sensor input which alter the agent's local knowledge of the task state. An agent's arbitration logic ultimately decides the appropriate set of control and communication actions to take when given the combined recommendations of the instantiated behaviors.

This approach to intelligent system design has numerous advantages. An enormous range of flexible interactions are possible between individual behaviors. The formulation of behaviors as control loops permits the design to build upon a vast body of control theory and tools. The arbitration logic takes advantage of the synergy of constructive interference between behaviors. In addition, it can effectively resolve the effects of simultaneous cooperating and competing goals. Thus, simple behavioral constructs can produce complex agent responses to complex situations. The agents in this design can exhibit reactive and opportunistic behavior when the task is not well understood at design time. This enables the agents to participate in dynamic community interactions in which roles may shift between agents as a function of task state. The flexibility of this design approach together with the ability to construct and test pieces of the system incrementally makes exploratory implementation of distributed intelligent control systems much more practical.

Despite the compelling advantages to this design approach, it is at a very early stage of development and several problems have been recognized and others undoubtedly exist which have not yet been encountered. Complex coupling between behaviors is possible which complicates the design of individual behaviors. This complexity also makes the configuration of the arbitration logic quite tricky. Furthermore, determining the stability of the complete control system is not straightforward and classical analysis techniques may not be sufficiently powerful. In general, controlling the system complexity is difficult. Faithful simulations of the agent processes may lessen the impact of these problems but verifying the accuracy of any simulation of an agent's responses or of the responses of a collection of agents may require considerable effort.

CONSTRUCTION OF DISTRIBUTED INTELLIGENT CONTROL SYSTEMS

The construction of a distributed intelligent control system consists of several steps: task description, system design and system implementation.

Task Description

The task description step should precede any aspect of design but seldom does in reality. The task description identifies what is important in the task environment and defines precisely what the distributed intelligent control system must accomplish. The task definition includes specifications of the point at which the system enters the task, the goal conditions which define termination, if any, and the envelope of constraints which exist between starting and goal conditions. If aspects of the task are not understood at this time then obtaining sufficient information about the task becomes part of the task itself. This description is formulated before anything whatsoever is said about the specific nature of the control system. No attempt should be made to include any more information about the task than is needed to specify it since overspecification only limits the design options. Presently, no formal representation and methods for task description are available even though this may be one of the most important conponents of design.

Design

This discussion of the design of distributed intelligent control systems considers the influences of the design philosophy, describes the design process, briefly reviews some decisions which must be considered in the design of a distributed intelligent system and summarizes some of the lessons which have been learned to date.

The prevailing design philosophy greatly influences the end product. Two primary philosophies exist. A design can be formulated from the top down or from the bottom up. Top-down design works splendidly when all aspects of the task are very well understood. Although this philosophy is quite appealing to the designer, rework of the design due to errors or omissions is expensive and potentially dangerous. Bottom-up design is often needed when the design team is knowledge poor (e.g., rapid prototyping of expert systems). While this philosophy is appealing to the implementor, it is often time consuming and inefficient. Most design efforts of intelligent systems use both philosophies with the hope that a clean juncture in the middle will be possible when it is needed.

First, the system performance specification must be derived from the task description. This information is used at several stages in the construction process. The actual design usually consists of decomposing the task description into system and subsystem components, mapping function into hardware and software, and designing components and component interactions. Once a system or subsystem design is complete it should be verified either through simulation or formal proof or both to be sure that the design meets the performance specification. The verification step may well reveal the need for corrections to the design or the task description. Thus, design becomes an iterative process. Only when the complete design has been verified can it be communicated to the implementor.

Several important decisions must be considered when designing any distributed intelligent system. Some of these involve the distribution of knowledge. All interacting agents must share common knowledge. Common knowledge includes overlapping task knowledge, communications protocols, information sharing strategies and interaction strategies. A tradeoff exists between the amount of communications and the amount of common knowledge in the system. Communications are expensive (in terms of communications bandwidth and actual cost) and the interactions can often be slow. However, agent communications make the system more versatile. Communications between agents are not needed if the task is well understood so that all agent interactions can be specified at design time and if the agents do not cooperate (i.e., share resources) and dynamics permit no interactions. On the other hand, common knowledge is both cheap (i.e., in terms of memory) and fast. However, a system which relies completely upon common knowledge with no communications updates is likely to be unresponsive to deviations from the expected situations. Some minimum amount of common knowledge between agents is always needed. This can be minimized if everything is open for negotiation, if a large communications bandwidth between agents is available and if the time exists for negotiation.

Several issues of system organization are also important to distributed intelligent system design. The system organization defines the communications strategy and interactions which are available to the agents. The organization can be either fixed or dynamic and either hierarchical or community-based. Agent roles never change throughout the entire task in a fixed organization while agent roles can be reassigned repeatedly in a dynamic organization. A fixed hierarchical organization is the best understood of all the possibilities. In addition, it is the most efficient organizational choice. The other choices are not as well understood. However, the community organization is desirable because it is the most flexible and responsive to new situations. Clearly, hybrid organizations also have attractive attributes. Dynamic organizations are necessary if the roles of the agents must change due to unknown task conditions or factors which might change agent capabilities (e.g., agent attrition).

In the course of gaining what little experience with intelligent control systems which exists, several design lessons have been learned which can be extended to distributed intelligent control systems. In general, design errors become expensive once they have been implemented (e.g., Hubble Space Telescope). The interactions between loosely coupled components are often not suitably appreciated and accommodated. It is in these interactions where redesign is usually necessary. The designer must use the principle of least constraints when the task is not well understood so as to maintain flexibility for himself and the implementor. Obviously, well understood tasks are not so picky. Existing tools support this domain well. However, existing software engineering tools (maybe, engineering tools altogether) do not support exploratory implementation well. In these cases, even documentation becomes a problem!

Implementation

During the implementation process, the implementor take the completed design and maps it onto actual hardware and software. Each component and, ultimately, the entire system must be debugged and repaired. The implementor also evaluates the performance of the subsystems and of the completed system against the performance specification which is derived from the task description. Failures to meet the specification may require implementation reworks, design revisions or modifications to the task description. Thus, like the design phase, the construction process (i.e., design plus implementation) is iterative. However, implementation problems are much more costly to resolve than design problems which have not been implemented. Once the completed system passes the evaluation tests, it can be delivered to the customer.

Some of the lessons learned from the implementation of intelligent control systems can be applied to distributed intelligent control systems. The implementor has the fewest options when he needs the most. Thus, the differences between the designer's verification methodology and reality show up here. Unfortunately, the implementation step of intelligent control systems is very poorly supported by software tools. Existing debuggers are barely adequate for centralized control systems and totally inept for distributed control systems. Poorly understood tasks, usually those to which intelligent control systems are applied, require much more iteration between design and implementation than tasks which are completely understood. Repeated iterations increase the cost of the construction of distributed intelligent control systems significantly. Finally, implementors are often those who are least appreciated in the construction process but they solve the really hard problems (i.e., getting the thing to work).

NEEDED TOOLS

Considerable work is needed in tool development because the state of the art of distributed intelligent control systems is so immature. Tools are needed for the foundations of design, for individual intelligent agent design, for distributed intelligent system design and for implementation.

Foundation tools support the description of the task and the specification of system performance requirements. In addition, these include any standards which may be needed. Formal methods and representations are needed for describing a distributed intelligent control task. Techniques must be developed to derive system performance specifications from the task description. Even though distributed intelligent control is in its infancy, standards are needed for communications protocols and for a common language to represent common knowledge (i.e., vocabulary and syntax). However, standards are only needed if interoperability at any point in the system's life is desired. These would enable the products from several different manufacturers to be integrated relatively painlessly.

Design of a distributed intelligent control system is quite complex. Several tools are needed to aid the designer. The designer needs assistance with the decomposition of the task description and the mapping of its components onto the individual agents. A tool which enabled the designer to easily evaluate the effects of different organizations, communications strategies and shared knowledge allocations would be invaluable. Techniques are also needed which enable the designer to reliably evaluate the performance and stability of the completed system design. These techniques should be founded upon formal methods for describing distributed intelligent control system performance.

Several tools are needed to aid in the design of the individual agents in a distributed intelligent control system as well. For instance, tools are needed to aid the designer in the efficient mapping of task requirements onto component functions, the specification and construction of individual behaviors, and the identification of cooperative and competitive coupling between behaviors. Reliable simulations and formal methods must be developed to debug and, ultimately, verify the resultant behavior of the control system within the complete task context. In the distant future, it should be possible to formally verify the robustness of the combined system behavior..

Finally, as mentioned earlier, implementation of intelligent control systems is very poorly supported and several tools are needed in this area. The implementor needs assistance with the debugging of a system and its components. These tools should aid in the identification and location of bugs and should make the distinction between hardware and software problems clear. Performance measures are needed which derive directly from the task description. These may require much more complete theoretical understanding of intelligent control systems than now exists. A tool is needed which assists the implementor in the construction of an evaluation test plan from the task description. Finally, reliable evaluation techniques and methods to analyze the evaluation results must be developed.

CONCLUSIONS

In all, the state of the art of intelligent control systems is underdeveloped. Thus, the construction of distributed intelligent control systems is at an even more primodal stage. Very little substantive and widely accepted theory is available. Almost no tools for design and implementation are available and few of those which exist for conventional systems can be adapted. Almost no experience exists in the construction of intelligent control systems and none exists in the construction of distributed intelligent control systems. Therefore, almost any results (even negative) in any of these areas would be significant.

Conceptual Programming

Roger T. Hartley
Computer Science Department and
Computing Research Laboratory
New Mexico State University
Las Cruces, NM 88003

Abstract

Conceptual Programming (CP) is a knowledge representation language based on Sowa's conceptual graphs. It provides an expressiveness at least equal to first-order logic, and extends both logic and conceptual graph theory in several aspects. CP allows representation of causality through actor nodes that accept proper epistemological categories as input and output. The concepts of state and event have been expanded to allow for both temporal and spatial representations, and both are integrated as far as reasoning is concerned. Another type actor of actor node allows functional computation with individuals which may be symbols, numbers or sets of these. The graph operations join and project allow structures to be built (called *programs*) that serve as qualitative and quantitative simulations. Constraint propagation through the actor nodes determines the outcome of the simulation.

CP serves as the representation underlying the techniques of Model Generative Reasoning (MGR) problem solving methodology that builds interpretations of data abductively through parsimonious set cover and allows for revision of these interpretations through differential deduction.

1 Languages for knowledge-based systems

The history of knowledge based systems is full of attempts to provide a user, whether novice or expert, with a set of tools that ease the task of building a working application in the shortest possible time. Early work concentrated on extensions to Lisp giving it the flavor of a deductive retrieval data-base, i.e. one based on a deductive model of problem solving. Micro-planner and Conniver can be seen as forerunners of Prolog in this sense. Other attempts to provide a tool based on the deductive model (e.g. Omega) have been made since then.

On the other hand, early expert system work lead eventually to the rash of expert system shells that are sold on the commercial market today. These are all close cousins to the logic-based systems in that a set of rules are executed by an inference procedure to effect the desired computations. Some of these are now powerful enough to be called general-purpose languages (they are Turing machine equivalent) rather than just single application packages (e.g. OPS83). Moreover, many shells incorporate a variety of symbolic knowledge representation schemes such as frames, demons, classes and so on (e.g. KEE, ART).

All of these systems share a set of common beliefs. These are:

- Problem solving can be effected through the manipulation of symbolic representations.
- A representation language can be designed to capture all relevant knowledge, in an easy-to-use yet expressive form.
- An inference procedure can be designed to allow all desired inferences to be drawn through computations on these symbolic structures.

- The representations and procedures can be made generic enough for a wide variety of problem solving activities.

Apart from a few hard-line connectionists, most people in AI stick to these principles. We are no different in our design of Conceptual Programming, although the details of how the representations and procedures come together are somewhat different. This paper presents an overview of our beliefs, how they are explicated through CP and how CP is used to support a general problem solving system.

2 Conceptual graphs

In 1984, John Sowa (*op. cit.*) published his influential book that describes his ideas of knowledge and its representation. His theory of conceptual graphs remains as the best example of the form of representation loosely called ***semantic networks***. From this book there sprung a community of like-minded people who hold regular workshops at the AI conferences, and who participate in regular discussions over an e-mail based bulletin board. Sowa himself has improved his theory since the book was published and several research groups have made other changes and additions. Conceptual Programming is one such attempt to enrich the basic theory with the aim of becoming a general purpose tool for problem solving.

2.1 A brief summary

The theory of conceptual structures can be described briefly as (see Sowa's book for more detail):

- Knowledge can be organized through a system of *concept types* that have sub/super-type relationships in a lattice. This is very much like any taxonomy of natural types, except that new types may be defined (i.e a new concept can be introduced) as having more than one sub or super-type to form a complete lattice.
- Concepts are related together by *relations* that form a disjoint set of terms having their own lattice.
- Concepts can be specialized by *individuals* that refer to distinct objects, not to a class of objects as do the concept terms themselves.
- Concepts are defined in one of three ways:
  1. As an Aristotelian definition, with a a *genus* term and a set of *differentiae*.
  2. As a *schematic* (i.e. contingent) definition, that relates the term to others.
  3. As a *prototype* that looks like a compound individual containing typical (default) values for the related terms.
- Relations can be primitive (defined canonically) but may also be defined in a schematic form that mentions the concepts involved.
- The expressiveness of first-order predicate calculus (FOPC) can be obtained by notating the forms as two-color graphs (one for concepts, one for relations) call *conceptual graphs*.
- Functional dependencies between individuals may be expressed as *actor* nodes tying concept nodes together.
- A set of starter graphs may be defined in the same way that axioms are defined in logic.
- The *canonicality* (i.e. their semantic well-formedness) of this starter set may be maintained through four operations: copy, join, project and simplify. These operations allow graphs to be combined in a variety of ways to support inference. Project is truth preserving, whereas join is not.

2.2 Expressiveness of CGs

Sowa provides an operator ϕ that translates any well-formed conceptual graph into an equivalent FOPC form. The inverse operator ϕ' reverses the transformation. Thus, FOPC and CGs are simply notational variants for a first-order logic with equality. Both have advantages, but the graphical form has a distinct advantage in that variables can very often be omitted entirely where they are merely place-holders for equal values in different sub-expressions. Overall the strengths of CGs are shown in declarative representations, matching the power of FOPC (there are improvements to make, however, notably in the area of sets). Weaknesses show up when these forms are used to express dynamic happenings i.e. causality.

2.3 Limitations

This being so, the weaknesses of CG theory are the weaknesses of FOPC. For knowledge based system work, the lack of a theory of causality (therefore of the real world) is the worst omission. Often FOPC uses material implication to capture causality, and even the modal variants are based on this. Where FOPC would write

$$A \rightarrow B$$

meaning A causes B, Sowa would be forced to draw

$$\neg \boxed{\neg \boxed{A} \; \boxed{B}}$$

because of the absence of a pictorial equivalent to disjunction. The boxes delimit conjunctive regions called *contexts*. Neither form really captures the directionality of the intended causality.

A further limitation of CGs also comes through the insistence on FOPC equivalence. Inference making in FOPC reduces to its proof theory, usually reductio ad absurdum and the use of the rules of inference like modus ponens, resolution etc. Again this is not adequate for many forms of inference, especially where hypotheses are generated, as in many forms of problem solving. Meta-rules can express these inference control relationships, but this is a cumbersome and confusing way to proceed, when a more direct method can be used. The production system, as used in almost all expert system shells, is one answer to this problem. As we shall see, CP has another answer, based on the idea of a simulation.

3 Conceptual Programming

3.1 Epistemology of CP

CP follows fairly conventional ideas, but extends the epistemology of FOPC (objects, predicates, functions) to include categories for problem solving. In particular, a duality between spatial and temporal notions has proved to be useful.

The world consists of ENTITIES and RELATIONS. Entities can be OBJECTS, ACTS, or PROPERTIES. Relations are:

- SPATIAL, between objects
- TEMPORAL, between acts
- TEMPORO/SPATIAL between objects or acts and properties
- CASE, between objects and acts

Properties are:

- CHARACTERISTICS that are intrinsic
- ATTRIBUTES that are accidental

Characteristics can be omitted (unknown), their value can change, but they cannot be negated. Attributes can be omitted (unknown), their value can change, and they can be negated.

Spatial relations can be omitted (unknown), their value can change and every one has an inverse (negation). Temporal relations can be omitted (unknown), their value can change and every one has an inverse (negation).

A temporal INTERVAL consists of an infinite number of MOMENTS. The smallest moment is the interval's START-POINT, and the largest is its END-POINT.

A spatial REGION consists of an infinite number of LOCATIONS. A region has a BOUNDARY.

An object and its properties form a STATE. A single triple (object relation property) is a PARTIAL STATE. A state is fixed at a moment.

An act and its properties form a PROCESS. A single triple (act relation property) is a PARTIAL PROCESS. It is fixed at a location.

An act and a number of objects are connected in EVENTS, through case relations. Events occupy time through the act's interval, and space through the objects' regions.

An object and a number of acts are connected in EXPERIENCES, through case relations. Experiences occupy time through the acts' intervals, and space through the object's region.

A set of objects and their spatial relations to other objects forms a SCHEMATIC. It is time-independent. A single triple (object relation object) is a PARTIAL SCHEMATIC. A schematic plus its objects' states at one point in time forms a SNAPSHOT.

A set of acts and their temporal relations forms a CHRONICLE. It is space-independent. A single triple (act relation act) is a PARTIAL CHRONICLE. A chronicle plus its acts' processes at one location forms a HISTORY.

A temporal sequence of snapshots is the same as a spatial arrangement of histories and they form a WORLD. It is a number of events, connected by temporal relations, seen from an act perspective, or a number of experiences, connected by spatial relations, seen from an object perspective.

Over a given spatial region and temporal interval objects and spatial relations persist in time, whereas acts and temporal relations do not; acts and temporal relations pervade the region, whereas objects and spatial relations do not.

A SITUATION is a partial state or schematic. Situations enable or trigger acts. Acts constrain the start/end points of situations.

A situation persists unless destroyed by one or more acts. This is causally PREDICTIVE. A situation pre-exists unless created by one or more acts. This is causally EXPLANATORY.

A partial chronicle or partial process has infinite (i.e. unknown) extent unless delimited by one or more objects. Objects constrain the boundaries of process/chronicles. Processes/chronicles constrain the regions of objects.

3.1.1 Declarative forms

In CP there is only one declarative form, the *schema*. The purpose of a schematic definition of a concept is to relate the concept to others, in a given generic context. Thus, only concept and relation nodes appear in a schema. Figure 1 shows a simple schema that can be paraphrased as "Giving involves an agent and an experiencer, both people, and a patient that is a ball.". All of the case relations used in CP are fixed and primitive. There are also a number of primitive spatial and temporal relations.

The terms defined in this way form a lattice of concept types, as explained above. GIVE, for instance might be a subtype of PHYSICAL-ACT, which in turn is a subtype of ACT. It is also possible to define a schematic cluster for a concept. This cluster contains several alternate definitions of a concept for different purposes.

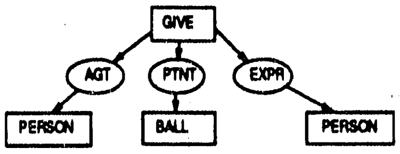


Figure 1: The graph for GIVE

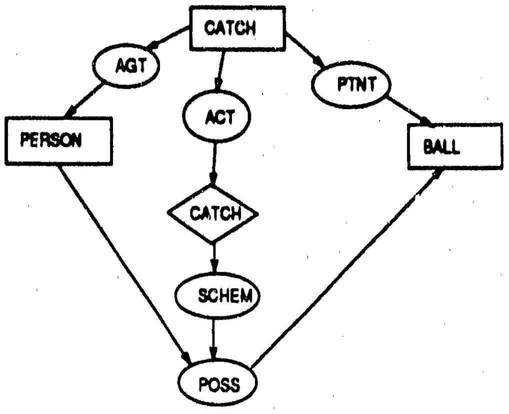


Figure 2: The temporal overlay for CATCH

3.1.2 Procedural forms

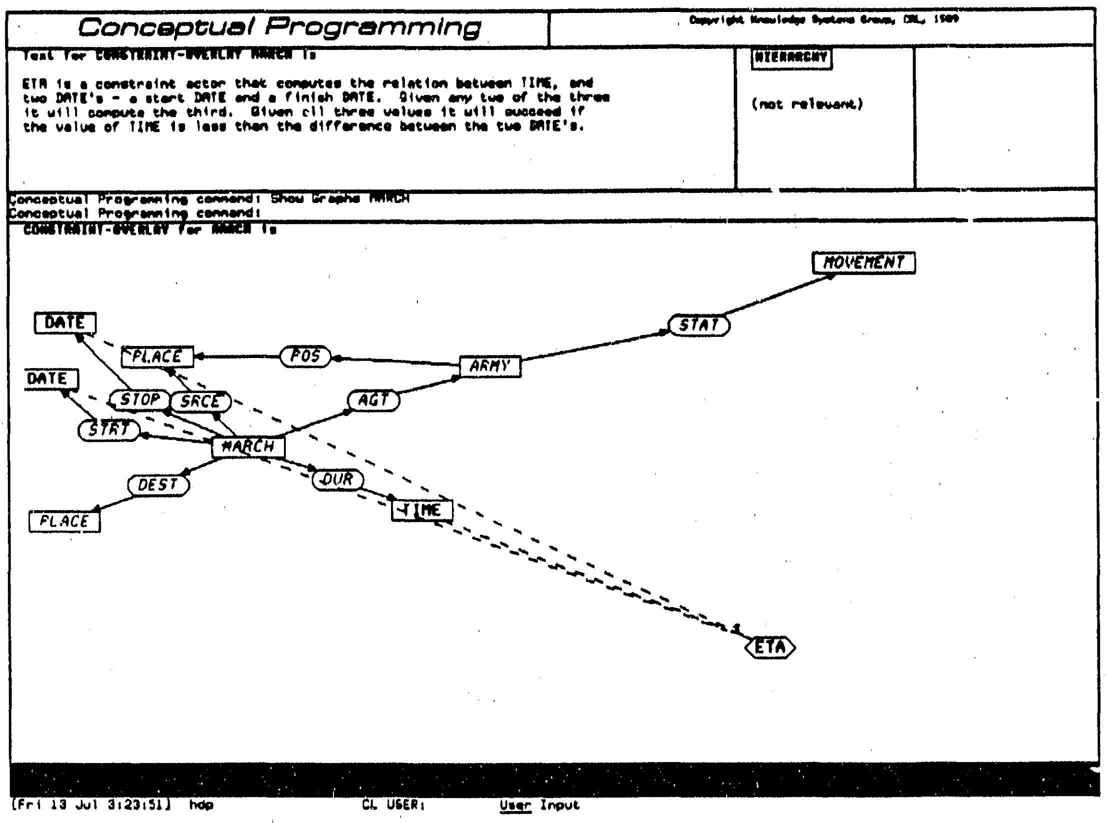
The largest change to the general philosophy of CGs comes in CP's ability to define procedural additions to the basic schema, called *overlays*. There are three kinds of overlay currently implemented:

- A feasibility overlay that performs semantic checking during graph operations. Basically it stops production of canonical but meaningless graphs, such as a coin with three sides, a pipe with three ends etc.
- A temporal overlay that enables simulations to compute the intervallic relations between acts and the states that enable them, and are altered by them. This overlay models causality according to the epistemology above.
- A constraint overlay that enables functional computations in a similar fashion to Prolog. This is very close to Sowa's functional actors, but whereas he only allows one-way computations (input-output functions), CP allows constraint checking and propagation just like Prolog.

A fourth overlay, the spatial overlay, when it is implemented, will be the dual of the temporal overlay in the spatial domain. Figure 2 shows a temporal overlay that changes in the person's eraltionship with the ball during catching. The actor node serves to draw these elements together. The relation ACT connects the temporal actor with its act; the relation SCHEM connects the actor with the concept-relation-concept triple that represents a partial schematic.

3.2 The implementation of CP

CP has been implemented on the Symbolics 3600 series under Genera 7.2 using CommonLisp, New Flavors and Symbolics Windows. It is currently undergoing a rewrite to improve speed and file-handling efficiency and will be ported to the Sun/UNIX environment in CommonLisp (Allegro), CLOS and Common Windows. After the rewrite, we plan to reimplement on the Symbolics and maintain the two versions thereafter.



```
Text for CONSTRAINT-OVERLAY MARCH is

ETA is a constraint actor that computes the relation between TIME, and
two DATE's - a start DATE and a finish DATE.  Given any two of the three
it will compute the third.  Given all three values it will succeed if
the value of TIME is less than the difference between the two DATE's.
```

(not relevant)

```
Conceptual Programming command: Show Graphs MARCH
Conceptual Programming command:
 CONSTRAINT-OVERLAY for MARCH is
```

```
[Fri 13 Jul 3:23:51]  hdp                    CL USER:          User Input
```

Figure 3: The CP screen

CP presents itself as a graphic editor for conceptual graphs. All input and editing is done graphically by using a menu/mouse interface methodology. Figure 3 shows the appearance of the Symbolics screen during a graph edit session. A window (not shown) displays the lattice of concept types, in a mouse-sensitive fashion (for browsing) and another has a hypertext sub-system for adding descriptive comments, with mouse-sensitive words, to any graph entered into the system. Functional constraints are written in Lisp into a pre-formed skeleton with parameters taken from the overlay graph. Figure 4 shows sample code for a numeric function.

4 Model Generative Reasoning

4.0.1 An operator based architecture

CP was always intended to form the representational substrate for research in problem solving. The work at CRL in the Knowledge Systems Group has attacked the brittleness problem of expert systems in two ways. Firstly, we have attempted to remove the reliance on the perfect quality of input data seen by the system. An expert system assumes that all data fed to it are accurate, relevant and true. However, data often does

```
(DefActor ETA CONSTRAINT-OVERLAY MARCH ((CONCEPT DATE1) (CONCEPT DATE) (CONCEPT TIME))
   (Declare ((TIME NUMBER)
             (DATE NAME)
             (DATE1 NAME)))
   (LET (M D Y M1 D1 Y1 START-TIME END-TIME)
     (WHEN DATE
       (MULTIPLE-VALUE-SETQ (M D Y)
         (TRANSLATE-DATE DATE))
       (SETQ END-TIME (TIME:ENCODE-UNIVERSAL-TIME 0 0 0 D M Y)))
     (WHEN DATE1
       (MULTIPLE-VALUE-SETQ (M1 D1 Y1)
         (TRANSLATE-DATE DATE1))
       (SETQ START-TIME (TIME:ENCODE-UNIVERSAL-TIME 0 0 0 D1 M1 Y1)))
     (COND ((AND DATE
                 DATE1
                 TIME)
            (IF (OR (> START-TIME END-TIME)
                    (> TIME (/ (- END-TIME START-TIME) 60 60 24)))
                (SETQ OUTPUTS 'FAILED)
                (SETQ OUTPUTS 'SUCCEEDED)))
           ((AND DATE TIME)
            (MULTIPLE-VALUE-BIND (IGNORE IGNORE IGNORE D2 M2 Y2)
                (TIME:DECODE-UNIVERSAL-TIME (- END-TIME (* TIME 24 60 60)))
              (ASSIGN-VALUE DATE1 (UNTRANSLATE-DATE M2 D2 Y2))))
           ((AND DATE1 TIME)
            (MULTIPLE-VALUE-BIND (IGNORE IGNORE IGNORE D2 M2 Y2)
                (TIME:DECODE-UNIVERSAL-TIME (+ START-TIME (* TIME 24 60 60)))
              (ASSIGN-VALUE DATE (UNTRANSLATE-DATE M2 D2 Y2))))
           (T (SETQ OUTPUTS 'FAILED))))
  )
```

Figure 4: The code for a functional constraint

not come in such a nicely packaged form. Too often data are imprecise, irrelevant or just plain wrong. Any problem solver should be able to cope with this *noise* in the input data. Secondly, most expert systems, if they are model-based, will do differential diagnosis on the data that do not conform to the norm. They cannot cope with data that are *novel* in any way, i.e where the difference goes off the scale, or cannot be measured at all.

These limitations of expert system technology led us to a system that addresses these issues head-on. We have developed the ideas in *model generative reasoning* for these purposes. The basic technique is to view the data as *interpretable*, not as correct, true, etc. Data are interpreted by generating a *model* that 'covers' it. The notion of covering data with stored knowledge is similar to the general set covering model of Reggia et al. as used in medical diagnosis. A successful cover leads to a *program*, that consists of a graph formed by merging one or more schematic definitions, complete with their overlays, with the data graph being covered. It is at this point that MGR depends on the ideas of canonicality in CP. The operation of join that is used to merge all the graphs ensures that the resultant program is well-formed. The program can be seen as a simulation (since it contains actor nodes) of the local cause and effects. Figure 5 shows the graphs for catch and throw with their temporal overlays. Below each graph is the time-chart for the actor.

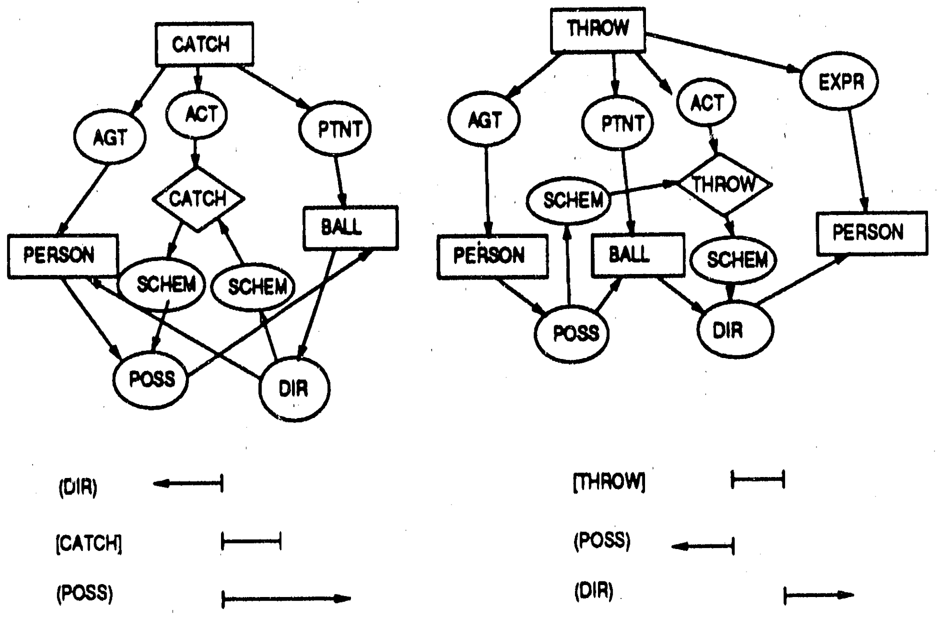


Figure 5: The graphs for CATCH and THROW

This time-chart is part of the definition of the graph, i.e. it is part of the knowledge of what it is to throw a ball to a person or to catch it. Figure 6 shows the two combinations of the individual catch and throw time-charts (the two graphs join in two ways on PERSON and BALL). It shows that if the sequence of acts is THROW–CATCH (one peson to another) then THROW must precede CATCH (hence the temporal relation computed is BFOR). However, if the sequence is CATCH–THROW (by the same person), then there is ambiguity. Either the throw comes after the catch, when the person hangs on to the ball for a period of time, or the throw immediately follows the catch. The ambiguity is show by the dashed line for the interval for POSS after CATCH. This indicates a partial ordering between the end-points of these intervals. In the CATCH–THROW case we might talk of catch and throw 'in one motion'. These time-charts are produced by constraint propagation through the actor network. In this example there are only two actors, but in general there could be a network of multiply connected actors where there are multiple events happening simultaneously.

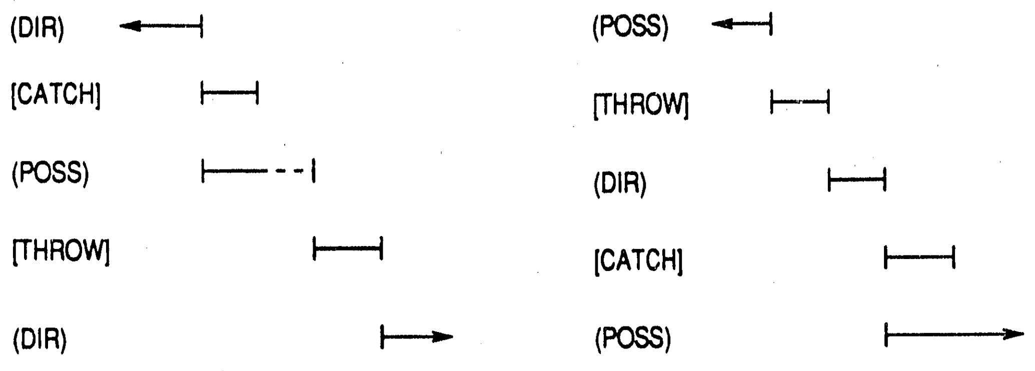


Figure 6: The time charts for CATCH and THROW

4.0.2 The logical basis of MGR

MGR consists essentially of two operations: *Specialize* and *Fragment*. Specialize is the abductive operator that takes data and covers them with schemata to form programs, which are then executed to form models. It is composed of two more primitive operators, *cover* and *join*. This leads to the slogan:

Abduction = cover + join

The functionality of specialize is:

$$specialize : 2^{\mathcal{F}} \times 2^{\mathcal{D}} \rightarrow 2^{\mathcal{H}}$$

where $\mathcal{F}$ is a set of input graphs representing data, $\mathcal{D}$ is a set of schemata (definitions) and $\mathcal{H}$ is the resultant set of hypotheses produced by cover and join.

That specialize is abductive can be seen by considering that the hypothesis generated contains a projection of each input graph, whether data or schema. When these are joined together, some concepts may be specialized by replacement by a common subtype, and new nodes may be added. Considering the truth values of the graphs, the hypothesis implies its constituents. Put another way

$$F' \cup S \rightarrow F$$

where F is the input data, F' is this data specialized through join, and S is the additional nodes added by the covering schemata. As an abductive task, this can be stated as "given F, what is the (minimal) S that can be added s.t. the result implies F".

Fragment takes models apart by breaking links such that data are preserved in each fragment. The functionality of fragment is:

$$fragment : 2^{\mathcal{F}} \times \mathcal{H} \rightarrow 2^{\mathcal{H}}$$

Here a set of fact graphs taken from $\mathcal{F}$ are projected into a single hypothesis $h \in \mathcal{H}$ to produce a set of fragments $\mathcal{H}$. It is not necessary for the subset $\mathcal{F}$ to be the set of facts that were originally covered to produce h. Fragment implements a deductive operation since each fragment is guaranteed to be a projective sub-graph of the original model.

4.0.3 The implementation of MGR

Currently MGR operates as an open architecture with the operators specialize and fragment embedded in CommonLisp on the Symbolics, together with a host of ancillary functions. Thus MGR is an embedded algorithmic language like those mentioned in the introduction. Work is underway to implement the architecture in a parallel environment (Sequent Symmetry) in order to combat the explosion of alternative hypotheses produced by cover and join, and to handle the multiple hypothesis space demanded by the architecture.

We are also engaged in a research effort to let the operators run autonomously and opportunistically and to optimize their effects through a genetic algorithm, and eventually through a dynamical systems control mechanism where the rates of execution of the operators will be important parameters.

References

[1] Coombs, M. J. and R. T. Hartley (1987) The MGR algorithm and its application to the generation of explanations for novel events. *International Journal of Man-Machine Studies* 27: 679-708.

[2] Coombs, M. J. and R. T. Hartley (1988) Explaining novel events in process control through model generative reasoning. *International Journal of Expert Systems* 1: [illegible]9-109.

[3] Coombs, M.J. and Hartley, R.T. (1987a). CP: A Programming Environment for Conceptual Interpreters. CRL Memoranda Series MCCS-87-82, New Mexico State University.

[4] Eshner, D.P. and Hartley, R.T. (1988). Conceptual Programming with Constraints. In *Proceedings Third Annual Workshop on Conceptual Graphs*, Minneapolis, MN, 3.1.2-1-3.1.2-6.

[5] Fields, C. A., M. J. Coombs and R. T. Hartley (1988). MGR: An architecture for problem solving in unstructured task environments. *Proceedings of the Third International Symposium on Methodologies for Intelligent Systems*, Elsevier, Amsterdam, 44-49.

[6] Fields, C. A., M. J. Coombs, E. S. Dietrich, and R. T. Hartley (1988b) Incorporating dynamic control into the Model Generative Reasoning system. *Proc. ECAI-88*, 439-441.

[7] Hartley, R. T. (1986). The Foundations of Conceptual Programming. In *Proceedings of First Rocky Mountain Conf. on AI*, Boulder, CO, 3-15.

[8] Hartley, R.T. (1986). An overview of conceptual programming. *Conference on Intelligent Systems and Machines*. Oakland University, Rochester, Michigan. 43-48.

[9] Hartley, R.T. (1988). CP - The Manual. Memoranda in Computer and Cognitive Science series. Computing Research Lab. MCCS-88-127, New Mexico State University, Las Cruces, NM.

[10] Hartley, R. T. and Coombs, M. J. (1989) Conceptual programming: Foundations of problem solving. In: J. Sowa, N. Foo, and P. Rao (Eds) *Conceptual Graphs for Knowledge Systems*. New York, NY: Addison-Wesley.

[11] Hartley, R.T. and Coombs, M.J. (1989) Reasoning with graph operations. *Proc. workshop on foundations of semantic networks*. Santa Catalina Island, 1989.

[12] Hartley, R.T. (to be published, 1990) A uniform representation for time and space. Invited paper, special issue on semantic networks, Journal of Mathematics and Computers with Applications.

[13] Pfeiffer, H.D. and Hartley, R.T. (1989) Semantic additions to conceptual programming. *Proc. 4th Annual Conceptual Graphs Workshop*, IJCAI89, Detroit, MI.

[14] Pfeiffer, H.D. and Hartley, R.T. (1990) Additions for SET representation and processing to Conceptual Programming. Accepted for *5th. Annual Conceptual Graphs workshop*, AAAI90, Boston, MA.

[15] Sowa, J.F. (1984). *Conceptual Structures*. Reading, MA: Addison Wesley.

Language Development Systems in Support of Software Engineering

Vincent P. Heuring
Departments of Electrical and Computer Engineering
and Computer Science
University of Colorado - Boulder
Boulder CO 80309-0425

1. ABSTRACT

An important way to enhance productivity in software engineering is to provide the means for appropriate descriptions of problems and their solutions. This paper will describe the Eli Language Development System, and its use in developing application software. The Eli system provides an integrated set of tools that translates formal specifications for the lexical, syntactic, and semantic parts of an application program into executable code. The Eli system encourages the writing of declarative specifications instead of code, allows specification reuse, partitions the specifications for the language into manageable modules, and employs an expert system to manage the details.

2. THE SOFTWARE ENGINEERING PROCESS

Software engineering can be viewed as the process of designing solutions to problems. Or, restated, as the production of code from requirements. The conventional software engineering process proceeds in stepwise fashion from requirements to coding, with the coding being done in a more-or-less appropriate programming language. Much of the code may concern itself with parsing user input, testing for errors, building tables for the storage of information and preparing data for output. Often the "heart" of the application comprises relatively little of the total code written. The basis for writing the code is, ''how can I solve this problem,'' rather than, ''how do I specify the solution to this problem.''

The latter basis seems a more natural approach to problem solution, since requirements can be viewed as specifications of *what* problem is to be solved, not *how* the problem is to be solved. So the programmer needs to translate requirements into algorithms. Contrast this approach with the specification approach, where the programmer translates specification to specification.

Unfortunately, there are presently not many formal specification languages that are appropriate to the application domain. Many of the current generation of specification languages are not general purpose, but are specialized to the areas of algebraic simplification or theorem proof. The Eli system can be viewed as a general purpose specification system where solutions are described in the form of specifications that may be created specially, or extracted from a library. The next section describes the Eli system.

2.1. THE ELI SYSTEM

The Eli† system was originally developed as a compiler construction environment that integrated off-the-shelf tools and libraries with specialized language processors to provide a system for generating complete compilers quickly and reliably. Its aim was to simplify the development of new special-purpose languages, implementation of existing languages on new hardware and extension of the constructs and features of existing languages.

Considerable experience with the Eli system has shown us that it is also useful in developing a variety of specification-to-program translators. Eli automates the solution to complex problems: problems that can be broken down into smaller problems. Thus we can easily put together complex systems from existing pieces.

The reader should understand that in a very real sense, developing a compiler is no different than developing any other piece of complex software: the user input must be collected into a sequence of basic symbols; the symbol sequence must be parsed to ascertain the structure of the user input; when certain pieces of input structure have been recognized, the program must provide processing and output. Intelligent and timely error reports must be generated. (The same process obtains whether the processor being generated is an interpreter or a compiler. The only difference is when the final interpretation process takes place.)

Eli's collection of off-the-shelf tools is controlled by an expert system whose problem domain is the management of complex user requests.[1] This expert system is discussed in the next section.

Because the expert system manages the construction process, we can interpose arbitrarily complex processing to make simple user input acceptable to off-the-shelf tools. The number of specifications can be reduced, and special-purpose languages can be used to simplify those specifications. Values for many of the tool parameters can be deduced from the specifications themselves.

The Eli system allows arbitrary processing to be carried out to match the output of one tool to the input of another, to prepare consistent input for several tools from a single specification, and to combine outputs from several tools. Thus, tools developed by different people with different conventions can be combined into an integrated system, even when only executable versions of those processors are available.

To add a new tool to Eli, or to replace an existing tool with a better one, only the expert system's knowledge base must be changed. Eli users are not concerned with the knowledge base; they are only interested in the products and parameters that Eli provides. Knowledge base changes may add new products and parameters or make existing ones disappear, but most users can continue business as usual.

† Named after Eli Whitney, who was the first US manufacturer to make extensive use of interchangeable parts.

2.1.1. Controlling the Tools A complete system specification will probably consist of several files, each containing an additional specification or part of a specification. Some of these files may be taken from a library, others may be shared among several different projects. A user needs to be able to submit this collection of files to Eli, and to specify a user request to Eli. It may only be necessary to test some aspect of the application. On the other hand, it might be necessary to obtain an executable version of the application, or a directory of source files from which an executable version could be built without the use of Eli.

Most requests will invoke a bewildering array of tools and intermediate products. Some of these products may already have been constructed to satisfy previous requests. Eli removes the burden of managing this complexity from the user by placing it upon the shoulders of an expert system called Odin,[2] whose area of expertise is the management of complex user requests. Eli's component tools and their relationships are described by a *derivation graph* that resides in Odin's knowledge base. Odin also manages a cache of derived objects. When a user makes a request of Eli, Odin's inference engine determines the sequence of operations needed to satisfy that request, re-using cached objects in the derivation wherever possible.

Eli's primary input is a text file whose name is the name of the application being generated, followed by the extension ``.specs''. (Each file name has an extension that gives the ``type'' of that file; file types are used by the expert system to determine how the file should be processed.) Eli passes the specification file through the C pre-processor, and then interprets each line as the name of a specification file. The use of the C pre-processor allows the user to group specification file names logically, control the selection of certain specifications by directives, and include specifications from libraries.

Figure 1 shows the top-level specification file for the development of a Minilax compiler, and some Eli requests that might be made during compiler development. (Minilax is a small teaching language used in our compiler construction classes.) The specification files `structure.specs`, `translate.specs` and `vax.specs` list the specifications for the three major compilation subtasks of structuring, translation, and encoding.[3] A standard module for carrying out the name analysis task is available in Eli's library, and this module's interface is made available via `environment.lib`. `Opident.oil` and `Properties.ala` are all themselves specifications, describing operator identification, and symbol table properties respectively.

Figure 1b shows examples of some user requests of the Eli system. Each request line in the figure is read from left to right. A colon (:) can be read as ``derive to''; plus (+) introduces a keyword parameter, which may or may not have an associated value. Notice that it is possible to derive an object from a derived object — ``: err'' is a general derivation that obtains error reports produced by some other derivation. Greater than (>) is a re-direction mechanism; it is used to place the object resulting from the derivation into a file or a directory.

Notice that individual tools are never invoked directly when using Eli, and their particular interfacing requirements are invisible. The user is concerned only with composing the appropriate request. Based upon that request, and the state of the cache, Eli determines what needs to be done: what tools to invoke, and what intermediate results to

`#include "structure.specs"`	Specifications for the structuring task
`#include "translate.specs"`	Attribute grammar for the translation task
`environment.lib`	Use the standard name analysis module
`Opident.oil`	Specification for the type analysis module
`Properties.ddl`	Specification for the definition table module
`Properties.ala`	Implementation of the stored information
`#include "vax.specs"`	Specifications for the encoding task

a) The content of file `Minilax.specs`, a specification file

Check whether the concrete syntax satisfies the parser generator constraints:

```
   Minilax.specs :parsable
```

Apply the compiler being developed to program `test.mla` and display its output:

```
   Minilax.specs +arg=(test.mla) :stdout
```

As above, but only display error reports:

```
   Minilax.specs +arg=(test.mla) :stdout :err
```

Put an executable version of the Minilax compiler into file `Minilax.exe`:

```
   Minilax.specs :exe > Minilax.exe
```

Put complete source text for the Minilax compiler into directory `src`:

```
   Minilax.specs :source > src
```

Obtain a version of the Minilax compiler with embedded profiling code:

```
   Minilax.specs +prof :exe > Minilax_prof.exe
```

b) Some typical Eli requests involving the specification file of (a)

Figure 1
Using Eli

produce. In order to make requests of Eli, a person must learn only a few derivation and parameter names. By hiding all of the conventions and most of the options needed to control each tool, Eli sharply reduces the number of things that must be learned.

Eli uses a hypertext-based help system that provides the entire system documentation on line. This documentation is constantly being upgraded, and presently comprises nearly 1 MB of text.[4]

3. CONCLUSIONS

Eli is a complete, flexible specification based software engineering development system. Although it was originally developed as a compiler construction environment, it is useful in many other areas of software engineering. Based on existing tools, it is an open system that is able to evolve as new tools and techniques become available. Eli does not rely on any one specification language to describe all of the subproblems that might be involved in a development project; rather it provides a coherent framework in which specifications written in a number of languages are combined to describe a complete product. A simple user interface provides a uniform method for requesting derivations from specifications. Such requests only involve product names. Eli determines the particular set of tool invocations on the basis of the state of its cache, as reflected in a knowledge base.

We have used Eli to create processors for small, special-purpose languages, standard programming languages and extensions to existing languages. It has improved our productivity and has enabled inexperienced users to undertake and complete significant software development projects. Our current research program is aimed at further simplifying the use of Eli itself and improving the performance of the generated software.

4. REFERENCES

1. W. M. Waite, V. P. Heuring and U. Kastens, 'Configuration Control in Compiler Construction', in *Proceedings of the International Workshop on Software Version and Configuration Control*, Teubner, Stuttgart, FRG, 1988.
2. G. M. Clemm and L. J. Osterweil, 'A Mechanism for Environment Integration', *ACM Transactions on Programming Languages and Systems*, **12**, 1-25 (January 1990).
3. R. W. Gray, V. P. Heuring, S. P. Krane, A. M. Sloane and W. M. Waite, 'Eli: A Complete, Flexible Compiler Construction System', Software Engineering Group Report 89-1-1, University of Colorado ECE Dept. Boulder Colorado, June 6, 1989.
4. R. M. Stallman and R. J. Chassell, *Texinfo - The GNU Documentation Format*, Free Software Foundation, 675 Massachusetts Ave. Cambridge MA 02139, May 1988.

THE G2 REAL-TIME EXPERT SYSTEM

Roland Jones
Gensym Corporation

The practical application of expert systems to dynamic domains requires a second-generation approach toward knowledge representation. In particular there is a need to represent dynamic qualitative knowledge, dynamic analytic knowledge and the structure of the object interactions in the domain. The application of inference in real-time requires paradigms which go beyond rote pattern matching, to use metaknowledge to focus the inferencing resources of the expert system. Finally the application of truth maintenance requires a temporal model of the time dependence of the truth of data and inferred results.
The G2 expert system technology was developed for real-time applications. Current installations are primarily in large chemical process plants, where the need for this technology is to advise operators for safety and economic reasons. Recently the application of the technology of real-time expert systems has been extended to robotics, and the Savannah River Laboratory of the Department of Energy has used G2 in mobile robot applications. Their purpose is to employ expert systems supervision to eliminate many of the requirements for human supervision. Additional applications in the aerospace industry include rapid prototyping and on-line manufacturing.

The underlying methodology of G2 is object oriented. Classes, frames, inheritance and other concepts are extended from heuristic reasoning to include analytic dynamic models. Expert system reasoning is extended from rules to structure using object connections. The resulting G2 application framework allows an engineer, or a cooperating group of engineers, to rapidly prototype new expert systems, control systems, decision support systems, networks, schedules and other applications where a combination of heuristics, analytic models and domain structure is needed.

Several considerations of dynamic domains impose requirements of the knowledge representation:

1. The concurrent use of analytic and heuristic models. Conventional simulation methods allow analytic models. Conventional expert systems allow heuristics, but leave the analytic part for the use to program. The combination of analytic and heuristic knowledge in an object oriented framework allows the applications to be addressed in a unified way.

2. Interaction between objects. The structure of an application is frequently important in predicting behavior, performing diagnosis or in scenario simulation. Structure is generally expressed as connectedness of objects, or proximity of objects. Structure may also be expressed in an object's attributes, especially where connections may vary in time. A framework which has the built-in capability to reason in terms of object connectedness or proximity, and to integrate analytic as well as heuristic knowledge in these terms, allows construction of the knowledge for the application.

3. Dynamic behavior and live data. Many problems have a real-time aspect, including dynamic knowledge in differential equation form, such as equations of motion. Live data may be needed for the eventual deployment, and data access and real-time processing may be important. A framework which includes these real-time considerations in the expert system design is required. The framework allows simulation to provide real-time values for prototyping and development, to be supplanted by sensor-based data at installation. Data servers provide interfaces to other systems with a minimum of user work, so the prototype can become the actual application.

In addition to the general characteristics of the applications, which call for a unified framework, the general desirability of rapid implementation calls for the use of high level interfaces. In G2 these include graphical construction of the application domain and structured natural language for expression of knowledge, models, and other information. Modern parsing techniques allow the user to express the knowledge in reasonably natural form, and G2 checks the user input as it occurs. Look-ahead menus help the user, and errors are immediately flagged. This eliminates a whole level of debugging which conventional programming requires.

Two apparently conflicting requirements dominate the inference paradigm considerations in the real-time domains. One is the need for truth maintenance. With thousands of data changing rapidly, the validity of conclusions at all levels of inference are in question. The other requirement is fr real-time performance, where real-time means fast enough to advise the human operator and/or control the robot process.

Code improvements and computer improvements can help. However a fundamentally different inference approach is appropriate for real-time problems. The approach that a human expert uses in a real-time situation is to maintain a peripheral awareness across the domain, watching for performance exceptions, and then focusing on areas of interest. The G2 inference engine operates similarly. The inference engine continually scans knowledge which the expert has specified for peripheral awareness. If a safety-threatening condition occurs in a reactor, for example, the G2 inference engine uses metaknowledge to determine which knowledge to invoke, thus focussing on the area of interest.

One benefit of the metaknowledge approach is that very large knowledge bases can be run in real time. Since many types of problems and behaviors are represented in the knowledge base, it can get quite large, with thousands of rules. However Ge does not consume computer time looking for patterns in all of this knowledge all the time. Rather it focuses attention on the knowledge needed. The concept is like the human thought process, in that a human does not use knowledge of swimming or driving when walking in the park. The human mind focuses, using the knowledge relevant to the task.

In static expert systems, truth maintenance involves changing inferences when data changes. In real-time problems there is an additional requirement to change inferences even if no new data is available, since time is a factor in validity or certainty of inference. One way to express this temporal validity information is to attach an expiration time to each value maintained by the inference engine, and propagate this when inference is carried forward. Generally, when a conclusion is based on several time sensitive variables, the earliest of their respective expiration times will be carried forward. Expiration times can be propagated forward through multiple levels of inference, but there are also ways to limit this propagation.

The real-time expert system technology described in this paper represents a departure from static expert system design, as the issues of time relationships and dynamic behavior have been addressed. The resulting expert system is capable of applying thousands of rule-frames of knowledge, and of performance in real time for reasonably complex operations.

Software Tools for Lower Echelon Systems Development

Dr. Larry U. Dworkin
Consultant
Holmdel, New Jersey 07733

Dr. Dirk R. Klose
U.S. Army Communications-Electronics Command
Center for Command, Control, and Communications Systems
Fort Monmouth, New Jersey 07703-5000

Lanny L. L. Gorr
TELOS Corporation
Shrewsbury, New Jersey 07702

ABSTRACT

The U.S. Army Communications-Electronics Command (CECOM) Center for Command, Control, and Communications (C^3) Systems has proposed an Advanced Technology Transition Demonstration (ATTD) for a Lower Echelon Command, Control, Communications, and Intelligence (LEC^3I) system.

The LEC^3I system contains features such as a distributed intelligence control system to manage the network of user units and nodes (e.g., dynamic reconfiguration to address node/unit losses and acquisitions); a very high-speed distributed database management system; a near real-time remote and local access knowledge-based Decision Support System (DSS); an automatic format and protocol conversion capability to facilitate interoperability; an object-oriented, "Fact"-based approach to reduce the communication volume, state-of-the-art communication error detection and correction; and a generic software interface approach for standardizing user display hardware, software, and human interfaces.

The LEC^3I ATTD development is a rapid prototyping process that begins using modeling and simulation techniques, evolves to a mixture of prototype elements and modeled elements connected to simulators, and concludes with field demonstrations of a prototype system. The paper will identify the tools needed to support the full development cycle of the ATTD, particularly the need for consistency in approach, language, and object representation over the entire cycle.

INTRODUCTION

Significant financial resources are being focussed on Computer-Aided Software Engineering (CASE) tool development. Current tools are quickly being supplanted by ones that are far superior. Many of the tools are very specialized, choosing to deal thoroughly with only a portion of the development process and often limited to very specific development and/or target systems. The available tools typically integrate poorly with one another (if at all), and are often not usable at the level of complexity found in Government systems. Therefore, to identify the tools needed for a system development activity such as the Lower Echelon Command, Control, Communications, and Intelligence (LEC^3I) Advanced Technology Transition Demonstration (ATTD), it is critically important to define the scope of the application development being addressed. This situation is exacerbated by the complexity of the LEC^3I ATTD, in that it includes reverse engineering, traditional design and system generation, and extensive use of modeling and simulation. The situation is further exacerbated by the fact that the development work will be performed in heterogeneous development environments at a group of organizations having local variations in methodology.

As a result, the initial focus for specifically identifying needed tools is on the critical issue of clearly defining a methodology model: how is information about information organized and how is the LEC^3I system to be developed from that base? This meta model is critical in determining what CASE tools will be needed to support the ATTD methodology, handle objects and relationships, and integrate with the different modeling and simulation concepts. Having identified the

ATTD methodology and environment, tool requirements can be identified that support the desired methodology, handle objects and relationships, and use the different modeling concepts. The intent of this paper is to recommend tool concepts that can be developed to support a complex project methodology such as the LEC^{3}I ATTD, not to move to a different methodology to fit the predilections of current tool vendors.

The following paragraphs will present a brief description of the lower echelon battlefield environment and the LEC^{3}I system to be developed by the ATTD, an overview of the engineering design process over the various phases of the ATTD life cycle, a listing of the tool capabilities desired, and a summary of the required capabilities.

BATTLEFIELD CHARACTERISTICS

The lower echelon battlefield is nonlinear and contains many highly mobile units. Large facilities, such as those used in the higher echelons, are not practical because soldiers to man them are not likely to be available and the large facility size and commensurate lack of high mobility are likely to make the large facility vulnerable to enemy destruction or capture. The probability of individual unit loss at the lower echelons (coupled with the fluidity of conditions, low communications quality, and severe data volume limitations) has historically resulted in a low level of processing power and automation.

The lower echelons battlefield conditions lobby for a solution to be based on a loosely coupled network of powerful, compact processors. Since communications capacity is limited and transmission quality is often poor, any data collection process needs to correlate data as soon as feasible to minimize the volume of data to be passed onward. Personnel providing/-receiving information at the lower echelons are normally quite busy and therefore require a graphical information presentation interface.

LEC^{3}I ATTD DESCRIPTION

The Advanced Technology Transition Demonstration (ATTD) for Lower Echelon Command, Control, Communications, and Intelligence (LEC^{3}I) is an advanced development effort to transition the set of technologies needed to implement a "smart node" infrastructure that collects, processes, and distributes data among the Battlefield Functional Areas (BFAs) and representative task force elements. The "Smart" node infrastructure will be a design that transitions the Combat Vehicle Command and Control (CVC^2) and the Battalion and Below Command and Control (B^{2}C^2) concepts into LEC^{3}I capabilities and provides seamless functional connectivity to the Army Tactical Command and Control System (ATCCS). The hierarchy of these tactical command and control acronyms is shown in Figure 1. An overview of the LEC^{3}I ATTD is provided in Figure 2.

The LEC^{3}I system will contain the following segments:

1. A Combined Arms Command and Control (C^2) system providing vertical and horizontal interfacing of all BFAs at the Battalion and Below (B^2) echelons, i.e., a Lower Echelon C^2 (LEC^2) system

2. A secure, dispersed, tactical information collection, correlation, storage, and dissemination system, i.e., a Battlefield Information Management System (BIMS) and the high-speed database search/retrieval hardware

3. A near real-time Tactical Situation Status Display

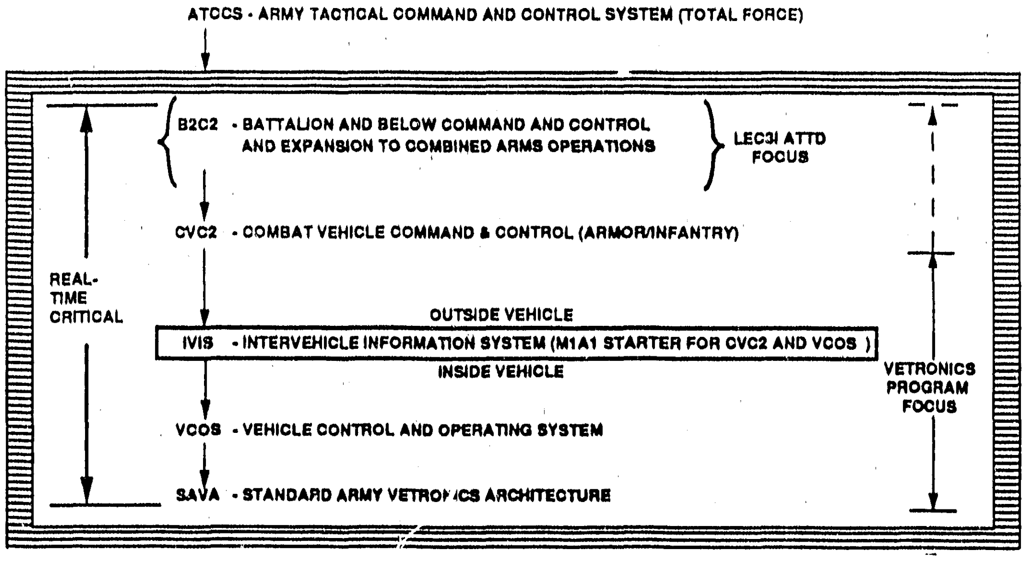

ATCCS - ARMY TACTICAL COMMAND AND CONTROL SYSTEM (TOTAL FORCE)
B2C2 - BATTALION AND BELOW COMMAND AND CONTROL AND EXPANSION TO COMBINED ARMS OPERATIONS
LEC3I ATTD FOCUS
CVC2 - COMBAT VEHICLE COMMAND & CONTROL (ARMOR/INFANTRY)
REAL-TIME CRITICAL
OUTSIDE VEHICLE
IVIS - INTERVEHICLE INFORMATION SYSTEM (M1A1 STARTER FOR CVC2 AND VCOS)
INSIDE VEHICLE
VETRONICS PROGRAM FOCUS
VCOS - VEHICLE CONTROL AND OPERATING SYSTEM
SAVA - STANDARD ARMY VETRONICS ARCHITECTURE

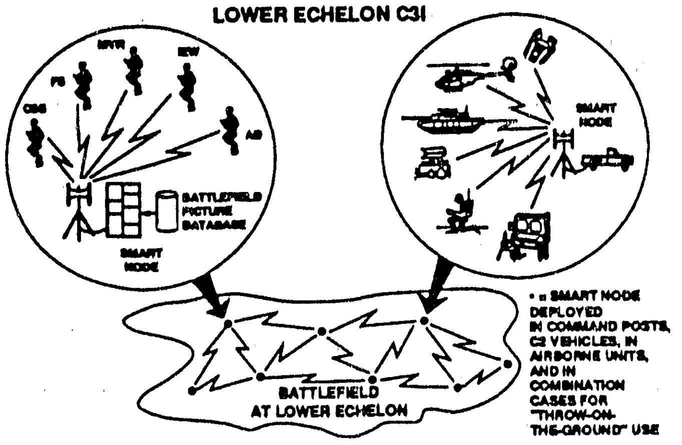
FIGURE 1. COMMAND AND CONTROL ACRONYM HIERARCHY

OBJECTIVE

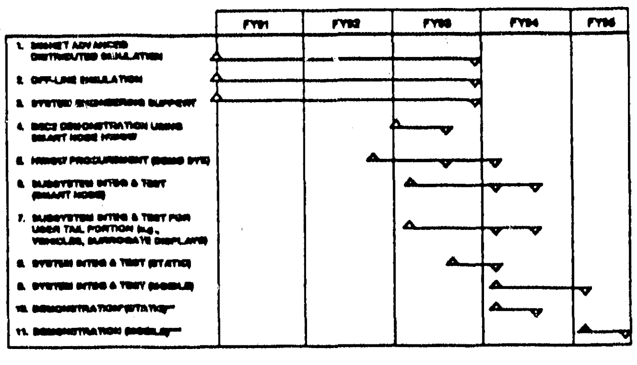
- PROVIDE CRITICAL INFORMATION IN A TIMELY MANNER THROUGH:
  - AUTOMATED INTEGRATION AND CORRELATION OF INFORMATION RESOURCES TO REDUCE COMMUNICATION VOLUME WITHOUT THE LOSS OF FACT INFORMATION
  - AUTOMATED TAILORING, DISSEMINATION, AND PRESENTATION OF INFORMATION IN A GRAPHICAL DECISION-ORIENTED FORMAT
  - REMOTE ACCESS DISTRIBUTED PROCESSING CAPABILITIES
  - BATTALION AND BELOW FORCE LEVEL CONTROL SYSTEM
  - A HIGHLY MOBILE COMMUNICATION INFRASTRUCTURE
- A COMMON POSITION AND TIME GRID FOR SYNCHRONIZATION OF FORCES
- PROVIDE INTERFACE/INTEROPERABILITY TO EXISTING COMMUNICATION AND C2 SYSTEMS:

- AREA SYSTEM, e.g., MSE, TRI-TAC	- ATCCS
- SINCGARS	- IVIS, CVC2, B2C2
- EPLRS/JTIDS	- CATC2

LOWER ECHELON C3I


MVR
IEW
FS
CSS
AD
BATTLEFIELD PICTURE DATABASE
SMART NODE
SMART NODE
BATTLEFIELD AT LOWER ECHELON
• = SMART NODE DEPLOYED IN COMMAND POSTS, C2 VEHICLES, IN AIRBORNE UNITS, AND IN COMBINATION CASES FOR "THROW-ON-THE-GROUND" USE

PROGRAM SCHEDULE



* POTENTIAL JOINT [illegible] II DEMONSTRATION
** BASED ON ESTIMATE OF 6 SMART NODES AND 20 USER TAILS
*** BASED ON ESTIMATE OF 10 SMART NODES AND 40 USER TAILS

APPROACH

- CREATE/EVOLVE TECHNOLOGY AND REQUIREMENTS BASELINE LEVERAGING OFF ATCCS CVC2, ATTDs, SBIRs, AND COMMERCIAL TECHNOLOGY
- DO SIMULATIONS AND DETAILED ANALYSES TO IDENTIFY INFORMATION NEEDLINE, INFORMATION FLOW, LINK LOADING, TIMELINESS, PERFORMANCE, AND INTEROPERABILITY CONSTRAINTS
- WITH TRADOC/USER/SUBJECT MATTER EXPERT INVOLVEMENT, ESTABLISH SYSTEMS CONCEPT AND TEST USING MODELING, SIMULATION, AND LEC3I PROTOTYPES
- DEMONSTRATE PROTOTYPE APPLICATIONS SUCH AS SMART MAP, INFORMATION FILTERING, AND DECISION SUPPORT AIDS
- RUN A SERIES OF TECHNOLOGY DEMONSTRATIONS AND FIELD TESTS TO FULLY EVALUATE PROTOTYPES, DESIGN CONCEPTS, AND DERIVE USER REQUIREMENTS
- TRANSITION LEC3I TECHNOLOGY PROTOTYPES AND FINALIZED SPECIFICATIONS TO APPROPRIATE PEO/PM COMMUNITY

FIGURE 2. LOWER ECHELON C^3I ATTD

(TSSD) generic software interface for user display devices such as heads-up displays. A near real-time, remote access Decision Support System (DSS), including the advanced processing hardware and software technology required to provide Expert System (ES) decision support aids

4. A communication system that is transparent to the dispersed and diverse combined arms user community, secure, highly flexible, and linked to existing systems such as the Mobile Subscriber Equipment (MSE) network and Combat Net Radio (CNR). The system will utilize such software and hardware tools as are needed to address the interoperability problems present among connected systems.

The LEC^{3}I ATTD proposes to combine the infrastructure needed for the BIMS, TSSD, DSS, and communications network node into a single assemblage known as a "smart node". A smart node is intended to function as a relay for point-to-point and conference communication within its own service area, a fully automated format/protocol converter to support interoperability, a transparent gateway for traffic utilizing the MSE network, a storage and processing location for the BIMS, a filtering and correlating system to provide users only the data that they wish and in the form most conducive for immediate comprehension, and a source for ES support to those users whose own computer resources are inadequate to support ES applications.

The LEC^{3}I ATTD is based on state-of-the-art commercial technology and the products developed under a number of other ATTDs, particularly the Multi-Mission Area Sensor (MMAS) ATTD and the Air-Land Battle Management (ALBM) ATTD. The LEC^{3}I ATTD will determine what capabilities are required for a LEC^2 system and identify the requirements for the communications, database management, decision support, and data display elements needed to support all of the lower echelons. A prototype set of products will be built and tested to validate the requirements analyses and better identify and quantify the benefits expected if a production version is fully deployed.

The LEC^{3}I ATTD will investigate commercial and military state-of-the-art radio communication technology and existing Army communication systems to select a technology to be utilized in the demonstrations and to make a recommendation for a production LEC^{3}I system.

The LEC^{3}I ATTD will make extensive use of modeling and simulation to evolve requirements definitions, evaluate design alternatives, and quantify expected benefits from fielding-recommended products. The LEC^{3}I ATTD expects to make significant use of the Simulation Network (SIMNET) program.

The LEC^{3}I ATTD will use rapid prototyping to determine what capabilities can be implemented and to improve the performance.

ATTD ENGINEERING DESIGN PROCESS

The ATTD for LEC^{3}I can be divided into a four-phase life cycle. During the first phase, alternative architectures for the LEC^{3}I system are reviewed and conceptual system and segment designs are made. The hardware and software for the most promising concepts are modeled for testing and evaluation using the SIMNET program. The design will evolve as dictated by results of the simulations. During this phase, the existing assets being incorporated will be reverse engineered and choices will be made among off-the-shelf components by comparative evaluation. The final conceptual design from the phase one simulation efforts will be the starting point to design and construct a few prototype units during phase two.

During phase two, hardware will be acquired and integrated and software will be written to build prototype units suitable for use with SIMNET and AIRNET. Mathematical models of the prototypes will also be produced so that the simulations can be performed with a normal deployment quantity of units. Lower echelon systems typically deploy in quantity. Therefore, the simulators can reduce the cost and time needed to evolve a lower echelon system design using a rapid prototyping approach. It should be noted that user community participation is continual throughout the ATTD life cycle. Therefore, the results of the simulations are expected to change the user requirement documents and battle strategies. The final prototype hardware and software design from phase two will be used to build the phase three prototypes.

During phase three, an adequate number of ruggedized prototypes will be built to conduct field demonstrations. The prototypes will exchange information with deployed military systems and be operated by military personnel. Based on the experience gained from the field demonstrations, the LEC^{3}I specification and requirements documents will be revised.

The fourth phase is the transition of the prototypes, testing documentation, and specifications to the appropriate Program Executive Offices (PEOs) for Full Scale Development (FSD).

The ATTD effort is expected to be organized against the five primary product development efforts; i.e., Lower Echelon Command and Control (LEC^2) System, near real-time Battlefield Information Management System (BIMS), Tactical Situation Status Display (TSSD), Decision Support System (DSS), and the communication system. Work in these areas will be coordinated to avoid duplication of effort, obtain the maximum level of synergy, and produce products on the schedule needed to meet transition goals. Metrics indicators are expected to be used to aid in meeting quality, schedule, and cost goals.

The execution of an ATTD project involves the combined efforts of Government organizations, their support contractors, and an ATTD prime contractor with subcontractors. It has been determined as a matter of national policy that restriction of competition is undesirable (e.g., the Competition in Contracting Act). Therefore, the details of the development approach and the selection of the software development tools to be used are influenced by the outcome of the competitive bidding process. This influence is particularly acute when transitioning from the ATTD development team to a PEO and, later, to the FSD contractor(s).

Because of the leverage of existing programs such as CVC^2, existing technology demonstration (e.g., ATTDs, Small Business Initiative Research [SBIR] programs, Balanced Technology Initiatives [BTIs]), and commercial efforts, considerable parallel efforts are present. Therefore, software portability and reusability are significant cost drivers. These cost drivers are negatively influenced by the heterogeneous mix of development methodologies, tools, and documentation techniques present in the DoD community. In view of the passion displayed by proponents of various design approaches, there is no reason to expect that the DoD community will adopt a single approach (e.g., object-oriented, structured). Any tool set developed will have to accommodate more than one approach.

TOOL REQUIREMENTS

A generic engineering design approach is shown in Figure 3. It should be noted that Figure 3 is generic in that the "New System" product can range from the phase one model of the LEC^{3}I system run in a computerized simulation of a battle to phase three executable code running on prototype target system.

The first activity is to reverse engineer the real assets (source code, test cases, etc.) that have been obtained from existing systems and other research activities. This activity is needed to generate a description data set. Although new, such reverse engineering tools do exist and have been proven to be effective. Once the description of the old system has been built, more familiar ground is reached. Many CASE tools exist to define requirements, create a design, and generate the resulting new system assets. As Figure 3 shows, this task is unusual in that the create design process is driven by both the traditional define requirements step and the description of the existing systems. More correctly, the old system description serves as a resource that the designer can draw on, at his option. His first concern is presumably to reflect the new ATTD requirements and to conform to a set of architectural principals that were probably part of the library products to use as components of the new system.

When the old system assets were designed, different standards and performance criteria may have been utilized. An issue raised by Figure 3 is represented by the dotted box and lines. Can any components of the old system be salvaged and retained in the new? Candidates for retention might include device drivers, database managers, or modules of source code that implement particularly complex algorithms. Retention of source code would seem to be particularly risky in that it may undermine the completeness and accuracy of the old and new system descriptions, and therefore the quality of the new system. Retention, in a sense, "short circuits" the reverse/-forward engineering process, as the diagram shows. The retention step is included in this discussion because it may be appealing on economic, schedule, and complexity grounds.

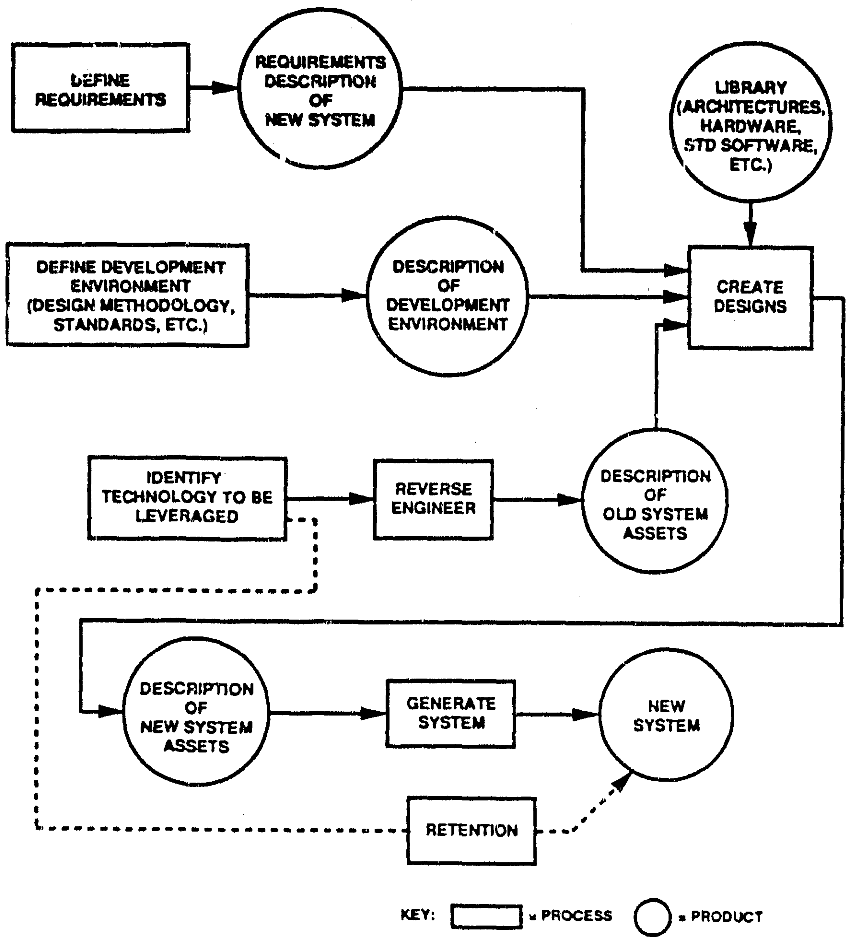

DEFINE REQUIREMENTS
REQUIREMENTS DESCRIPTION OF NEW SYSTEM
LIBRARY (ARCHITECTURES, HARDWARE, STD SOFTWARE, ETC.)
DEFINE DEVELOPMENT ENVIRONMENT (DESIGN METHODOLOGY, STANDARDS, ETC.)
DESCRIPTION OF DEVELOPMENT ENVIRONMENT
CREATE DESIGNS
IDENTIFY TECHNOLOGY TO BE LEVERAGED
REVERSE ENGINEER
DESCRIPTION OF OLD SYSTEM ASSETS
DESCRIPTION OF NEW SYSTEM ASSETS
GENERATE SYSTEM
NEW SYSTEM
RETENTION
KEY: ▭ = PROCESS ○ = PRODUCT

FIGURE 3. HYPOTHETICAL CASE-ORIENTED FRAMEWORK

The main criteria that can be used to select CASE tools to address this ATTD are the following:

a. Methodology Support
The constructs used by the CASE tool must support the concepts used to design the target system and handle information according to the methodology model.

b. Functional Span
No single CASE tool exists today that encompasses all of the processes in Figure 3, particularly the need to produce models for a simulator as well as executable code. This being so, each tool needs to be evaluated in terms of its functional span.

c. Integration with Other Tools
Since several tools will be required to supply all of the required capabilities, it is essential that they be integrated with one another in a cohesive fashion. Commonality of user interface, design approach, and data structures is a key requirement. Support for emerging standards such as EDIF would be evidence of attention to the integration issue.

d. Specificity of the Development and Execution Environment
The ATTD will have a number of participating Government and contractor organizations. The tools must deal with a variety of developer execution environments. The need to target a variety of computer systems to compare alternatives creates a heterogeneous execution environment. The Government requirements for competitive procurement ensure the persistence of heterogeneous environments.

e. Portability to Another Execution Environment
The Government's procurement policies require free and open competition. Therefore, when hardware used in the ATTD is replaced by the hardware proposed by the winning FSD prime contractor, the execution environment may change significantly. It is desirable that the system being developed with the CASE tools be portable to a new FSD execution environment, thus avoiding a need for a repeat of the development process.

f. Scalability
Many CASE tools perform well when used to address small problems and small systems. Therefore, a proven capability for use of the tool on large, complex systems is a likely prerequisite for use on the ATTD.

g. Support for Rapid Prototyping
Rapid prototyping is a valuable technique for improving the quality of the design and keeping users involved. The use of this technique has been mentioned as a requirement.

h. Level of Abstraction
The current focus of many tools is the design and generation code, i.e., an automation of the current design and coding activities for writing high order language programs. It is desired that the ATTD CASE tools document the software and target hardware at as high a level of abstraction as is technically feasible and generating executable code from that level of input. The intent is to move to the next generation of programming technology, eliminating the generation and documentation of code such as Ada or C language statements.

i. Other Criteria Include:

1. Functional richness within the span being addressed

2. Learning curve
3. Ease of use (human factors)
4. Representation of ambiguity pending resolution
5. Support for multiple iterations
6. Availability of technical support
7. Linkage to commercial marketplace to ensure an aggressive program of tool maintenance and enhancement to remain in step with the state of the art.

The tool set needs to support the system concept development. The tool set needs to permit designers to define the system and its segments conceptually by supporting first the requirements definition, then the system definition, and finally the segment definition process.

DoD systems are increasingly being built with standard pieces such as those contained in the U.S. Army Common Hardware and Software (CHS). A convenient library capability is needed to encourage reusability of data, code, and design parameters.

In the post-Cold War period, modeling and simulation will be more heavily utilized in system development and testing. The tool set must thoroughly support the modeling process, providing a convenient way to assemble known hardware and software pieces into system element models and to integrate the element models to obtain a system model. Capabilities of the tool set should make it easy to build models for custom hardware and software and integrate the individual models into the system model. The modeling system should readily facilitate swapping out modeled items to permit the direct evaluation of alternative items, e.g., a 50 MFLOP processor versus a 100 MFLOP processor, the substitution of elements to permit the use of one communications protocol versus another. The tool set should provide the capability to reverse engineer existing software to facilitate generation of the models.

As a minimum, the modeled battlefield system must be able to be utilized both on-line and off-line with SIMNET and its derivatives (e.g., AIRNET). In the on-line mode, experienced military and civilian personnel can exercise the modeled battlefield system to determine its strengths and weaknesses. In the off-line mode, the developer can replay an on-line exercise or execute an unmanned simulator scenario designed to isolate problem sources and evaluate solution alternatives.

The products of the ATTD are transitioned to one or more PEOs. To support a PEO, input to a Procurement Data Package (PDP) is provided. A prime contractor is selected by competitive bid and the ATTD materials are provided to the contractor for use in FSD. The tool set must support the documentation activities needed to produce specification materials of suitable quality for use in a solicitation.

CONCLUSIONS

In conclusion, no current tool set has all the capabilities required. Any new tool set developed should:

a. Work at the highest of level of abstraction technically possible to improve development productivity, reduce documentation requirements, and increase portability beyond that provided by current high order languages
b. Use a uniform approach and standardized human interface across all functions
c. Be capable of working in a heterogeneous development and target execution environment
d. Be capable of passing data among user sites possessing different development environments

e. Fully support rapid prototyping
f. Support, as a minimum, model and simulation for system development using SIMNET
g. Provide technical support and adequate ties to the commercial market to ensures tool maintenance and frequent updates to remain at the state of the art
h. Provide support for international and national data standards, generic device interface standards, and communication standards (e.g., International Standards Organization [ISO], Federal Information Processing Standards [FIPS])

GLOSSARY

Acronym	Definition
AD	Air Defense
ALBM	Air-Land Battle Management
ATCCS	Army Tactical Command and Control System
ATTD	Advanced Technology Transition Demonstration
B^2	Battalion and Below
B^2C^2	Battalion and Below Command and Control
BFA	Battlefield Functional Area
BIMS	Battlefield Information Management System
BTI	Balanced Technology Initiative
C^2	Command and Control
C^3	Command, Control, and Communications
CASE	Computer-Aided Software Engineering
CECOM	U.S. Army Communications-Electronics Command
CHS	Common Hardware and Software
CNR	Combat Net Radio
CSS	Combat Service Support
CVC^2	Combat Vehicle Command and Control
DSS	Decision Support System
EPLRS	Enhanced Position Location Reporting System
ES	Expert System
FIPS	Federal Information Processing Standard
FS	Fire Support
FSD	Full-Scale Development
IEW	Intelligence-Electronic Warfare
ISO	International Standards Organization
IVIS	Inter-Vehicular Information System
JTIDS	Joint Tactical Information Distribution System
LEC^2	Lower Echelon C^2
LEC^3I	Lower Echelon Command, Control, Communications, and Intelligence
MCS	Maneuver Control System
MMAS	Multi-Mission Area Sensor
MSE	Mobile Subscriber Equipment
MVR	Maneuver Control
PDP	Procurement Data Package
PEO	Program Executive Office
PM	Project Manager
SAVA	Standard Army Vetronics Architecture
SBIR	Small Business Initiative Research
SIMNET	SIMulation NETwork
SINCGARS	Single-Channel Ground and Airborne Radio Systems
TRADOC	Training and Doctrine Command
TRI-TAC	Tri-Service Tactical Communications
TSSD	Tactical Situation Status Display
VCOS	Vehicle Control and Operating System

DECLARATIVE HIERARCHICAL CONTROLLERS

Wolf Kohn

Boeing Computer Services
Scientific Computing and Analysis
Seattle, WA 98124-0346

1. INTRODUCTION

This paper presents a new general purpose feedback controller for driving a system through complex tasks. The proposed controller, termed Declarative Hierarchical Controller is based on a theory of knowledge based controllers developed by the author [1], [2], [3]. In this theory, the plant dynamics, control requirements, and goal dynamics are declared in an axiom base. The actuator commands, which are functions of the sensor and goal command signals, are generated on-line as side effects of showing whether a theorem (Task Theorem) representing the system task, logically follows from the equational axiom base. That is, the controller is an on-line mechanical theorem prover, whose inference mechanism is based on equation solving over a Variety. The Task Theorem is constructed by an on-line planner as a conjunction of primitive lemmas, each of which is a carrier of an elementary control action. The control actions are multiplexed in time so that at each instance one and only one is active.

The theorem prover of the controller operates as follows:

- At each controller sample interval, the Task Theorem and the corresponding active axioms generate a set of simultaneous equations in which the variables are Actuator commands and the Controller state. This set is referred to as the Active Set.

- The Active Set is used by the inferencer to build a procedure for computing instance values of the variables. This procedure is a locally finite automaton over the Rational variety.

- The automaton is executed to compute instances of the commanded actions.

In addition, the theory provides an algorithm for transforming the axiom base and the inferencer into a recursive hierarchy for its efficient implementation and for satisfying real time and architectural constraints.

The paper illustrates the theory of Declarative Hierarchical Controllers with a robot manipulator control under End Effector force constraints. It also gives some stability, robustness and complexity results. The general controller capabilities are illustrated with a particular robot: A planar 3 link manipulator robot controller for painting the inside surface of a 2-dimensional balloon which deforms elastically under contact force. The task will also be constrained by End Effector angular velocity and position constraints (over determined problem). The problem has no known solutions using conventional control and planning schemes.

The rest of the paper is organized into 5 sections: Section 2 presents an overview of knowlege based controllers, Section 3 presents declarative contollers, Section 4 discusses the main elements of multiplexing action in the context of declarative controllers, Section 5 illustrates the concept with an example, and in Section 6 some conclusions are established.

2. OVERVIEW OF KNOWLEDGE BASED CONTROLLERS

Declarative Rational Controllers are a class of knowledge based feedback digital controllers.

As for conventional controllers, the function of a declarative controller is to generate an action (e.g. actuator command) as a function of sensor data, stored data and goal command data. In symbols, let T be the set of natural numbers, called the time set. Let G, S, X, A be semimodules over a common semiring H. A controller is fully characterized by two functions ρ, Ψ

$$\rho\colon\ G \times S \times X \times T \rightarrow A$$

$$\Psi\colon\ G \times S \times X \times A \times T \rightarrow X \qquad (1)$$

called the control law [4].

The semimodules G, S, X, A are referred to respectively as the goal space, the sensor space, the internal storage space, and the action space.

The control law (ρ, Ψ) is determined by the requirements, the characteristics of the system to be controlled (the plant) and the characteristics of the goals the system is supposed to achieve.

In conventional controllers the control Law (ρ, Ψ) is explicitly implemented as an algorithm for generating actions as a function of time. Specifically, let $s(k)$, $g(k)$, $x(k)$, $x(k)$ be the sensor data goal data and internal storage data at the current sample time $k \in T$. Then, the control action $a(k+1)$ and the updated internal storage $x(k+1)$ are given by

$$a(k+1) = \rho(g(k), s(k), x(k), k)$$

$$x(k+1) = \Psi(g(k), s(k), x(k), a(k), k) \qquad (2)$$

In knowledge-based controllers in general, and in declarative controllers in particular, the control law is not constructed explicitly. Instead a knowledge base containing the requirements, plant dynamics representation and goal characteristics is constructed. At run time the controller generates the action signals $\{a(k), k=0,1,\ldots\}$ by an inference procedure [5] which operates on the knowledge base at each sample time to find an instantiation of the control action to be implemented in the next sample time.

Some of the potential advantages of knowledge based controllers over conventional ones are [6]: simplified design process, multiplicity of control laws, built-in adaptability, hooks and scars and dependency on computational characteristics. These are briefly discussed next.

Simplified Design Process: In principle, given an inferencing shell and a compatible specification for knowledge representation, the design process consists of collecting the rules or frames that capture the control and computational requirements and those aspects of the plant and goal that are relevant to the design and the encoding of them. The actual "algorithm" implementing the control law is inferred from the knowledge base at run time.

Multiplicity of Control Laws: Since the control requirements are explicitly declared in the knowledge base, the controller may have more than one control law if knowledge is encoded so that it can select which one to execute as a function of external (goal) commands.

Built In Adaptability: Parameters or structural characteristics of the plant can be easily declared in generic form and appropriate instances of them can be generated as a function of sensory data at run time [7].

Hooks and Scars: Typically the knowledge base of a knowledge base controller is composed of independent discrete elements (clauses, rules, frames, etc.) which are only connected at run time during inference. Therefore adding or deleting elements does not destroy the logic of the controller.

Dependency on Computational Characteristics: The knowledge base may include characteristics of the architecture in which the controller runs. These characteristics are treated in the same way as control requirements.

However, general knowledge based controllers suffer from three serious drawbacks which have damped their popularity among control designers. These are stability verification, performance verification, and computational complexity. These are briefly discussed next.

Stability Verification: This is understood as a test on the controller specification to determine whether it will drive the system to the current desired goal while satisfying the control requirements in a finite interval of time. In most of the knowledge based control schemes that have been proposed, no such test is available.

Performance Verification: Since the control law function is not explicitly given, it is very difficult and in most cases practically impossible to guarantee a performance level for all the possible inference paths.

Computational Complexity: Many knowledge based control systems are implemented on expert system shells which are not well suited for real time implementation. The theory of declarative hierarchical controllers was developed to address some of the special requirements for robot control systems. Their functionality possesses the good characteristics of knowledge based controllers discussed previously. In addition, the theory provides for effective stability and performance tests and feasible (with current hardware) computational requirements.

The central objective of knowledge based controllers in general, and of Declarative Hierarchical Controllers in particular, is the generation of autonomous feedback policies.

Autonomous feedback policies are trajectories of control actions and control action generators that are causally measurable in the Σ-algebra defined by the sensory data. This means that the control law is not completely pre-specified, as in conventional control systems, but rather it is inferred locally at run time as a function of the environment, the stated goal and the control specifications.

The degree of autonomy is a quantitative measure of the amount of knowledge that is generated at run time to complete a local instance of the control law. This measure is the central discriminator between conventional control systems and knowledge based control systems.

A descriptive overview of declarative rational controllers is given in the next section.

3. DECLARATIVE RATIONAL CONTROLLERS

This section presents an overview of Declarative Hierarchical Controllers. Space constraints prevent the detailed discussion of the mathematical basis behind some of their characteristics. These can be found in some of the references. [8], [9].

The structure of a Declarative Hierarchical Controller is illustrated in Figure 1. It is composed of four functional elements: a *Knowledge Base*, an *Inference Mechanism*, a *Theorem Planner* and an *Adapter*. In the next paragraphs those elements are briefly described.

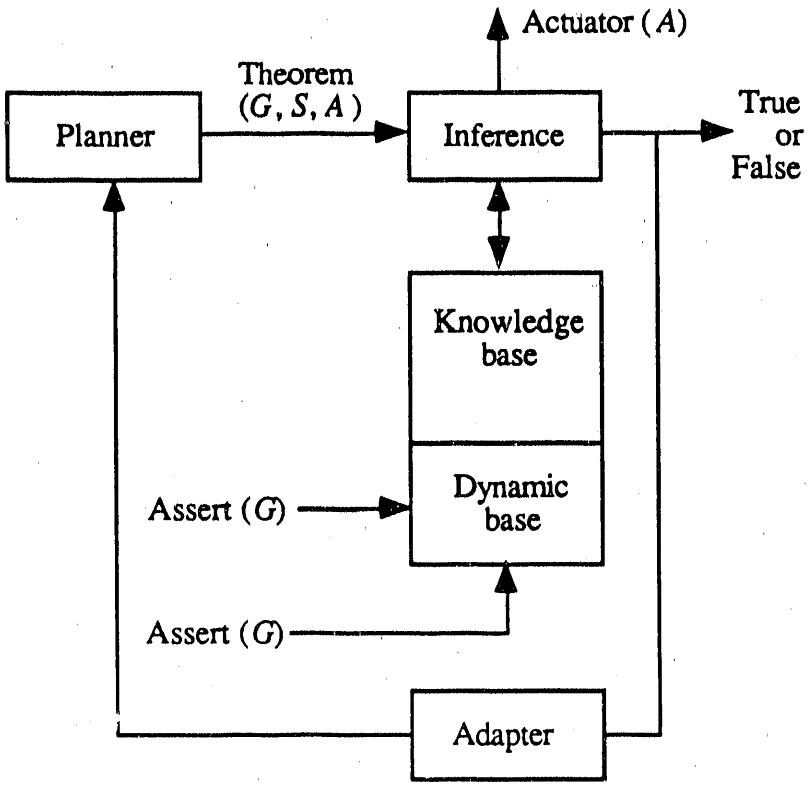


Figure 1. Declarative Rational Controller

Knowledge Base: This element consists of a set of equational, first order logic equational clauses [10] with some extension whose characteristics are described next.

A clause in the knowledge base of a declarative rational controller with sensor semimodule S, control action semimodule A, goal semimodule G, and internal storage semimodule X and time set T, is one or more Horn clauses of the form:

$$P_i\big(x(t)\big) \Leftarrow e_1{}^i\big(x(t), y(t)\big) \wedge \ldots \wedge e_s{}^i\big(x(t), y(t)\big) \bullet \tag{3}$$

where $\Leftarrow$ is the logical implication, $\wedge$ is the logical And, and $x(t), y(t)$ are sets of mappings of the form:

Let $U = G \times S \times X \times A$,

$$\underline{x}(t) = \big(x_1(t), \ldots, x_n(t)\big)$$
$$\underline{y}(t) = \big(y_1(t), \ldots, y_m(t)\big)$$

$$\text{with} \quad x_\ell : \in U \quad \ell = 1, \ldots, n$$
$$y_\ell : \in X \quad \ell = 1, \ldots, m \tag{4}$$

The $\{x_\ell\}$ are called the external variables and the $\{y_\ell\}$ are called the internal variables.

In (3), P_i is termed the clause head. The logical interpretation of (3) is $P_i(\bullet\bullet\bullet)$ is true if $e_1{}^i(\bullet\bullet\bullet)$ is true and $\bullet\bullet\bullet$ $e_s{}^i(\bullet\bullet\bullet)$ is true. The $e_j{}^i(\bullet\bullet\bullet)$ are referred to as the terms of the body of the i^{th} clause.

Each term $e_j{}^i\left(\underline{x}(t), \underline{y}(t)\right)$ is exactly one of 5 possible forms:

a) an equational term.

b) an inequational term.

c) a partial order term.

d) a clause name.

e) a frozen term.

a) An equational term is an expression of the form

$$w(\underline{x}, \underline{y}) \approx v\,(\underline{x}, \underline{y}) \tag{5}$$

where w and v are polynomic terms associated in an algebraic variety V_R defining the domain of the control system.

In an equational term, the variables and parameter sequences are subsequences of variables and parameters, respectively, the sequences appearing in the associated clause head.

The semantics of an equational term e^i_{name} is a subset E^i_{name} of a cartesian product of the universe U in which the variables and parameters take values. In symbols this subset can be expressed as

$$E^i_{name}\left(\underline{y}\right) = \left\{\underline{x} \middle|\ \text{length}(\underline{x}) = n\ ,\quad \underline{x} \in U^n\ ,\quad w\left(\underline{x}, \underline{y}\right) \approx v\left(\underline{x}, \underline{y}\right)\right\}$$

where length is a function that computes sequence length. Note that the set E^i_{name} is a function of the parameter sequence $\underline{y}$. This set is a rational set.

b) An inequational term is an expression of the form

$$w\left(\underline{x}, \underline{y}\right) \quad \neq \quad v\left(\underline{x}, \underline{y}\right) \tag{6}$$

where w, v, $\underline{x}$, and $\underline{y}$ are defined as in (5). The semantics of an inequational term e^i_{name} is a subset E^i_{name} of a cartesian product of the universe U defined as follows:

Let D^i_{name} be the following set:

$$D^i_{name}(\underline{y}) = \left\{ \underline{x} \middle| \quad \text{length}(\underline{x}) = n \ , \ \ \underline{x} \in U^n \ , \ \ w(\underline{x}, \underline{y}) \approx v(\underline{x}, \underline{y}) \right\} \tag{7}$$

Then, $E^i_{name}(\underline{y})$ is defined as the compliment of $D^i_{name}(\underline{y})$ with respect to U^N.

$$E^i_{name}(\underline{y}) \ = \ U^N \ - \ D^i_{name}(\underline{y}) \tag{8}$$

This set is rational.

c) A partial order term is an expression of the form

$$w(\underline{x}, \underline{y}) \quad \leq_\alpha \quad v(\underline{x}, \underline{y}) \tag{9}$$

where $\leq_\alpha$ is a partial order over the one or more of the polynomial algebras associated with the algebras in the rational variety.

The semantics of a term of the form of (9) is the relation that is the set $E^i_{name}(y)$ defined by

$$E^i_{name}(\underline{y}) = \left\{ \underline{x} \middle| \quad \text{length}(\underline{x}) = n \ , \ \ w(\underline{x}, \underline{y}) \leq_\alpha v(\underline{x}, \underline{y}) \right\} \tag{10}$$

If the lattice L_α corresponding to the partial order $\leq_\alpha$ is modular, the set defined by (10) is rational.

d) A clause name is the head of a clause in the knowledge base. Its semantics is the intersection of the rational sets of the terms in its body. This includes recursive clauses. For example, a clause of the form:

$$p\ (x, y) \ \Leftarrow \ w_1\ (x, y) \ \approx \ v_1\ (x, y) \ \wedge \ w_2\ (x, y) \ \approx \ v_2 \ \wedge \ p_3\ (x, y)$$

has semantics given by the set:

$$E_p\,(y) \ = \ E^1_p(y) \ \cap \ E^2_p(y) \ \cap \ E^3_p(x, y)$$

where the semantics of the third term is the rational set associated with a clause.

e) A frozen term is an expression of the form:

$$\text{rat(relation, polynomial}_1\text{, polynomial}_2\text{, variables, parameters)} \tag{11}$$

where relation is either $\approx$, or , $\not\approx$, or , $\leq_\alpha$, and polynomial$_1$ and polynomial$_2$ are polynomic expressions.

A frozen term is a term associated with a clause in which w is activated, i.e., unfrozen, as a consequence of the instantiation of another clause during an inference process. Any term in a clause can be frozen during inference. The purpose of this capability is to allow for context and covering of alternative semantics, i.e., rational sets for a concept.

The terms w, v in (5) and (6) are polynomials [10] over an algebra $B = <U, \Omega>$ where U, the universe of the algebra is given in (4), and Ω is the set of primitive operations:

$$\Omega = \left\{+, \bullet, 0, 1, \left[f_r, r \in R\right]\right\}. \tag{12}$$

These operations satisfy the following axioms:

(i) $(U, +, 0)$ is a commutative monoid, with unit 0

(ii) (U, $\bullet$,1) is a monoid with unit 1

(iii) For all a, b, c in U

$$(a+b) \bullet c = a \bullet c + b \bullet c$$
$$a \bullet (b+c) = a \bullet b + a \bullet c$$

That is $<U, +, \bullet, 0, 1>$ is a semiring.

(iv) The set $\left[f_r, r \in R\right]$ is a set of unary operations of the algebra, referred to as the Custom Operators. Their axioms, and computation values are determined for each controller by the clauses in the knowledge base. This means that each declarative controller C have associated with it a unique algebra $B_C = <U_C, \Omega_C>$. The custom operators are the basis for the local construction of the control law under composition.

(v) The algebra $<U, \Omega>$ satisfies the central factorization principle. This principle states that a term w constructed by composition from primitives in Ω is either a primitive or else can be expressed in finitely many different ways in terms of derived operations of the algebra on some of its elements.

The denotational semantics of the clauses defined in (3) are one of the following

1) A conservation principle, or
2) An invariance principle, or
3) A control constraint.

Conservation principles are logic statements about dynamic behavior of the controller, associated plant or goal. These principles serve analogous roles to the ones played by mass momentum and energy conservation principles in mechanics and thermodynamics.

Invariance Principles are logic statements establishing constants of motion in a general sense. Examples include logic formulations of stationarity principles and geodesics.

Both conservation and invariance principles are characterized by equations or inequations valid in the controller algebra and therefore can be written as clauses of the form of (3).

Finally, control constraints include actuator and sensor limitations and the control requirements. These can either be written in equational form or else written in tems of more general Horn clause forms. If a clause is not in the form of (1) it can be transferred to a set of clauses of that form using Colmerauer's construction which is compatible with the algebraic structure of elements of *V*.

The clause database is organized in a nested hierarchical structure as illustrated in Figure 2. The bottom of this hierarchy contains the equations that characterize the variety *V*, termed Laws of the Variety.

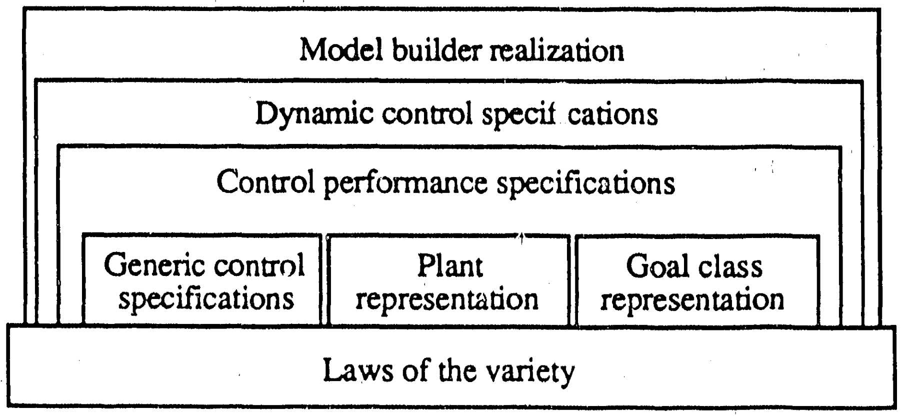

Model builder realization
Dynamic control specif cations
Control performance specifications
Generic control specifications
Plant representation
Goal class representation
Laws of the variety

Figure 2. Clause Knowledge Base of a Declarative Rational Controller

At the next level of the hierarchy, three types of clauses are stored: Generic Control Specifications, Plant Representation and Goal Class Representation.

The generic control specifications are clauses expressing general desired behavior of the system. They include statements about stability, complexity and robustness that are generic to the class of declarative rational controllers. These specifications are written by constructing clauses that combine laws of the variety using the Horn clause format described earlier.

The Plant Representation is given by clauses characterizing the dynamic behavior and structure of the plant, which includes sensors and actuators. These clauses are written as conservation principles for the dynamic behavior and as invariance principles for the structure. As for the generic control specifications they are constructed by combining variety laws in the equational Horn clause format.

The next level of the hierarchy involves the Control Performance Specifications. These are typically problem dependent criteria and constraints. They are written either in equational Horn clause format or in rational tree format [4] which can be translated into this equational form.

Dynamic control specifications are equational Horn clauses whose bodies are modified as a function of the sensor and goal commands. (See Figure 1.)

Finally model builder realization clauses constitute a recipe for building a procedural model for variable instantation and theorem proving. Their function serves as an interface to the inferencer whose operation will be discussed next.

Inferencer: This is an on-line equational theroem prover. The class of theorems it can prove are represented by clauses of the form:

$$\begin{aligned}\text{Theorem} \quad & (G(t), s(t), x(t), A(t+1), x(t+1)) \Leftarrow \\ & \text{unify}\ (G(t), s(t), x(t); y(t)) \wedge \\ & \text{unify}\ (A(t+1), x(t+1); z(t+1)) \wedge \\ & \bigwedge_{k=1}^{s} P_{i_k}\left(z(t+1), y(t)\right) \bullet \end{aligned} \tag{13}$$

In expression (13), theorem is the clause head, t is the current time, $G(t)$, $S(t)$, and $A(t+1)$ correspond to the variables representing the goal command to the controller ($G(t)$) the sensor inputs ($S(t)$) and the actuator commands ($A(t+1)$). Note that the actuator commands to be generated, are one unit of time ahead of current time.

In the right hand side of (13) unify is a special factual clause head whose function is to unify some of the external variables $X(t)$ with the input output variables $G(t)$, $S(t)$, $A(t+1)$, and $P_{i_j} \cdots P_{i_k}$ are clause heads of clauses in the knowledge base.

The theorem represents the desired behavior at the current update time. The purpose of the inferencer is to determine whether the theorem logically follows from the clauses in the knowledge base. A side effect is to find values in the universe of the controller algebra for tuples of the form ($G(t)$, $S(t)$, $x(t)$, $x(t+1)$, $A(t+1)$) where $G(t)$, $S(t)$, $X(t)$ are given.

Note that the theorem is an encoding of a system of equations and inequations from the knowledge base. So, proving the theorem is equivalent to solving this system.

The inference principle can be stated as follows: Let Σ be the set of clauses in the knowledge base. Let $\neq$ represent one or more partial orders in the universe of the controller algebra. Then proving the theorem is equivalent to show.

$$\Sigma \vdash L \tag{14}$$

In principle the proof can be accomplished by a sequence of applications of the following axioms:

a) Equality axioms

Let w, v and u be polynomic forms in an FPS algebra. The equality inference principles are:

a1 $\quad w \approx w$ identity

a2 $$\frac{w \approx v}{v \approx w}$$ commutativity

a3 $$\frac{(w \approx v) \wedge (v \approx u)}{w \approx u}$$ transitivity

a4 let $w_1, \ldots, w_n$, $v_1, \ldots, v_n$ be polynomic terms:

$$\frac{h.\ n\text{-ary operation} \wedge w_1 \approx v_1 \wedge, \cdots, \wedge w_n \approx v_n}{h(w_1, \cdots w_n) \approx h(v_1, \cdots v_n)}$$ composition

a5 $$\frac{\rho \text{ a substitution} \wedge w \approx v}{w\rho \approx v\rho}$$

a6 The equality clauses in the knowledge base are valid.

Briefly, a1 establishes that every polynomic form is equal to itself, a2 and a3 are self-explanatory, and a4 says that if h is any arbitrary derived n-ary operation in the associated algebra, then the equality of n pairs of terms is preserved by the operation. In a5 the symbol ρ, termed a substitution, is an equation of the form:

$$x \approx u(\underline{z}, \underline{y}) \tag{15}$$

where x is a variable, $\underline{z}$ is a sequence of variables <u>not</u> containing x, and $\underline{y}$ is a sequence of parameters.

As an example of the operation of the substitution inference principle, suppose that:

$$w(x_1, x_2) \approx v(x_1, x_2, x_3, y)$$

and let ρ be of the form:

$$x_1 \approx u(x_5, y')$$

then upon the application of the substitution principle,

$$w(u(x_5, y'), x_2) \approx v(u(x_5, y'), x_2, x_3, y)$$

which is denoted by

$$w\,\rho \;\approx\; v\,\rho$$

Finally, a6 establishes that there is no conflict between the knowledge base clauses and the inference principles of equality.

b) Inequation axioms

Let v and w be polynomic terms in an FPS algebra. The inequation inference principles are:

b1 $$\frac{v \not\approx w}{w \not\approx v}$$ symmetry

b2 $$\frac{v \not\approx w}{\dfrac{v \approx w}{\text{set}} \wedge \text{complement}(\text{set}, U, \text{set}^1) \wedge \dfrac{\text{set}^1}{v^1 \approx w^1}}$$ definition

b3 The inequality terms in the clauses of the knowledge base are satisfied.

b1 is clear. b2 says that if $v \not\approx w$ entails that the rational set associated with $v \not\approx w$ is set and the complement of set relative to the universe U of the semimodule is set^1 the semantics of the term $v^1 \approx w^1$.

In b2 one exploits the fact that rational sets are both mappings and sets and furthermore, if a set is rational its complement is also rational and therefore is generated by a polynomic equation, which is a member of the semimodule.

c) Partial orders axioms

Let w, v and u be polynomic terms in an FPS algebra B. For each partial order $\leq_\alpha$ defined in B, the partial order inference principles are:

c1 $w \leq_\alpha w$ identity

c2 $$\frac{w \leq_\alpha v \wedge v \leq_\alpha u}{w \leq_\alpha u}$$ transitivity

c3

$$\frac{h \text{ is an } n\text{-ary operation} \wedge h \text{ monotonic} \wedge w_1 \leq_\alpha v_1 \wedge \cdots w_n \leq_\alpha v_n}{h\left(w_1 \cdots w_n\right) \leq_\alpha h\left(v_1 \cdots v_n\right)}$$

composition

c4 $$\frac{\rho \text{ a substitution} \wedge w \leq_\alpha v}{w\rho \leq_\alpha v\rho}$$ substitution

c5 The partial order terms in the clauses of the knowledge base are valid.

In addition to c1 – c5 the following compatibility principle is given

ac1 $$\frac{w \leq_{\alpha} v \wedge v \leq_{\alpha} w}{w \leq_{\alpha} v}$$ compatibility

The meaning of these inference principles is similar to the corresponding principles for equality.

d) Convergence principles

The convergence principles are formal inference principles derived from the defining axioms of convergent sequences discussed in section 2.1. These principles, together with the limit principles are to be given in e).

Both the convergence and limit principles are needed to determine solutions of equational systems in the presence of recursion.

Let $\boldsymbol{B}$ be a rational algebra, $\boldsymbol{B}^{\mathrm{N}}$ the associated algebra of sequences with values in B, and C^N the set of convergence sequences that can be inductively built from $z(n)$ and $\wedge(n)$ using the following inference principles.

d1 $z(n) \wedge \wedge(n) \in C^N$ $\quad z(n) = 0 \ \forall n$, $\ n(n) = 1 \ \forall n$ initiality

d2 $$\frac{a \in C^N \wedge b \in C^N}{\left(a +_s b\right) \in C^N}$$ additive closure

d3 $$\frac{a \in C^N \wedge c \in B}{c \cdot a \in C^N \wedge a \cdot c \in C^N}$$ scalar closure

d4 $$\frac{a \in C^N \wedge c \in B}{a_c \in C^N}$$ shift closure

e) Limit axioms

These are given by the following inference clauses.

e1 $\lim (z) \approx 0 \wedge \lim (\wedge) \approx 1$ initiality

e2 $$\frac{a \in C^N \wedge b \in C^N \wedge \lim a \approx A \wedge \lim b \approx B}{\lim \left(a +_s b\right) \approx A +_s B}$$ additive preservation e3 $$\frac{a \in C^N \wedge c \in B \wedge \lim a \approx A}{\lim(c \bullet a) \approx c \bullet a \wedge \lim(a \bullet c) \approx A \bullet C}$$

scalar multiplication preservation

e4 $$\frac{a \in C^N \wedge c \in B \wedge \lim a \approx A}{\lim(a_c) \approx A}$$

shift preservation

Note that in d) and e) the inference clauses include membership terms. These terms can be written as equational terms by including in the custom operations of the associated base algebra, Indicator operations.

Let $a \in U$ the universe of the base algebra an indicator operation f_a on the base algebra is a function

$f_a: U \rightarrow U$ defined as follows:

$$f_a(x) = \begin{cases} 1 & a \approx x \\ 0 & \text{otherwise} \end{cases} \qquad (16)$$

Indicator operations on the associated rational algebra are defined in terms of indicator operations on the associated base algebra component wise.

Because rational sets are subadditive, each rational set can be decomposed as the union (addition) of singletons that is in fact, the structure of the semantics of the elements in the associated semimodule. Therefore, membership can be reduced to equational definitions of the form of (16).

The following theorem summarizes the inference procedure.

Theorem: Given a controller algebra B, and the corresponding knowledge base Σ then if L is the system associated with the goal theorem, then there exists an effective procedure for showing

$$\Sigma \vdash L$$

Outline of the proof: Since B satisfies the central factorization principle, a finite chain of applications of the axioms in 1-, 2-, 3- will yield the proof.

The theorem above is an extension to rational algebras of a result for finite algebras due to Evans [11].

Although theorem 1 guarantees convergence of the proof, it is a nondeterministic, highly inefficient procedure. Therefore, for practical reasons, a more efficient but equivalent procedure has been developed [12]. This approach, which involves the construction of a procedure for solving the system L, is outlined next.

The inferencer operates according to the following procedure.

Step 1. Unify the appropriate subset of the external variables $\underline{y}(t)$ with the current value of $G(t)$, $S(t)$, and $\underline{x}(t)$.

Step 2. Search the database for the equations and inequations associated with clauses $P_{i_j} \dots P_{i_k}$. The result of this step is a system of simultaneous equations and inequations in terms of the unknowns, the subset of $\underline{X}(t)$ unified with $A(t+1)$. This system is referred to as the Active Set (A_S).

Step 3. Convert the system into the active set into a system of equations in the canonical form. This system is referred to as the Linear Set (L_S).

The object of step 3 is to rewrite the Active Set in a form which allows the construction of a procedural model which in turn can be used to compute instance values of the unknown. This procedural model is a locally Finite Automaton. The canonical form of L_S is given by a system of equations of the form

$$\underline{X}(t) = \underline{E}(\underline{Y}(t))\,\underline{X}(t) + \underline{T}(\underline{Y}(t)) \qquad (17)$$

where $E(\bullet)$ is a matrix of appropriate dimensions with entries in the algebra of formal power series over the controller Algebra B: $B<M>$, M is the locally finite monoid generated by the set of primitive custom unary operations in B, Φ_R, and $\underline{T}(\bullet)$ is a vector whose entries are elements of $U<M>$.

Step 4. Given the linear set (L_S), construct a Locally Finite Automation (LFA) for solving it.

Step 5. Execute the LFA to obtain the values, if any, for $A(t+1)$.

If the theorem (13) logically follows from the knowledge base (i.e., it is true), the inferencer procedure outlined will terminate on step 5 with an actuator command value $A(t+1)$. If this is not the case, then the adapter is activated and the theorem is modified by the theorem planner according to a prespecified strategy.

Before proceeding to discuss the theorem planner, a brief outline of the theory behind steps 3-5 of the inferencer will be presented. This theory is the central element of the paper.

In general, given a knowledge base of Horn clauses and a goal, the process of proving the goal and finding instances of its variables can be carried out by a procedure, developed by Kowalsky, known as resolution [13]. The Resolution Procedure is based on constructing a tree (the Resolution tree) with the goal as its root and with the appropriate predicates of the knowledge base at its branches.

The resolution procedure consists of two subprocedures, a Navigator and a Unifier. The Navigator is a strategy for traversing the tree. Most systems currently available use depth first with backtracking as a strategy [14]. The unifier, usually a variant of Robinson's unification principle [15] is a general pattern matching recursive algorithm, which given a set of clause heads, generates the Most General Unifier (MGU) associated with them.

In contradistinction with the approach discussed above, the inferencer of a declarative controller builds a procedure for variable goal variable instantiation: a locally finite automaton.

A locally finite automaton is a non-deterministic machine with an arbitrary number of states (without loss of generality, it can be assumed that the number of its states is infinity), that satisfy the following condtions:

a) There is a finite number of states that are initial states

b) There is a finite number of states that are terminal states

c) The behavior of the automaton is the set of all paths from initial to terminal states (successful paths). This set is an element of a semimodule of formal power series over the Controller Algebra *B<M>*.

d) Every successful path involves a finite number of automaton edges (this is the locally finite condition).

e) Every successful path represents a map of the semimodule space $G \times S \times A$ into itself, where G is the space of goals, S is the space of sensor signals and A is the space of actuator commands.

f) The Automaton is provided with an input and an output function: The input function is of the form:

$$\begin{aligned} I\colon\ & G \times S \times X \rightarrow G \times S \times S \times A \\ I\,&(g,s,x) = (g,s,x,\Phi) \end{aligned} \tag{18}$$

where Φ is the additive identity in the semimodule A. The output function is of the form

$$\begin{aligned} O\colon\ & G \times S \times A \rightarrow A \\ O\,&(g,s,a) = a \end{aligned} \tag{19}$$

That is, O, is a projection function.

In synthesis, each successful path is a feasible control law.

The successful paths can be well ordered by any user defined optimization criterion for selection of the actual control law to be executed.

Now the theorem planner is described. The theorem planner generates theorems of the form of (3) according to a prespecified strategy. A theorem remains in effect as long as it has truth value true. The theorem has truth value true if the system of equations L_S have at least one solution different from the empty set.

If the theorem is not true, the adapter (see Figure 1) activates the strategy procedure in the planner. This produces a modified theorem to be proved.

This concludes the description of the concept of declarative controllers. A more detailed description of this concept will appear in [16].

4. MULTIPLEXING IN DECLARATIVE HIERARCHICAL CONTROLLERS

This section introduces the concept of multiplexing action in the context of declarative controllers and establish its use for the control of robot manipulators. A detailed description of multiplexing action in the context of a robot application can be found in [17].

Let Δ be an interval in the nonnegative real line. Let $\Delta_1,...,\Delta_n$ be subintervals of Δ such that

$$\Delta = \Delta_1 \, U ... U \, \Delta_n \tag{20}$$

A multiplexing action over Δ is a staircase function f_Δ over Δ taking values in the semimodule of control actions A such that

$$f_\Delta(t) = a_i \qquad a_i \in A \qquad t \in \Delta_i$$

A multiplexing action f over the nonnegative real line is a staircase function f_Δ over each interval Δ on it.

A feedback multiplexing action over Δ is a term of the form

$$\underset{i\,=\,1,...,n}{O\,(w_i\,(g,s,x,a))} \;:\; I \times G \times S \times X \times A \rightarrow A \tag{21}$$

Where w_i is a polynomial of the controller algebra B; O is the output function of the controller automaton introduced in the previous section and I is the set $\{1, 2, \ldots, n\}$

A feedback multiplexing action over Δ with sub intervals $\Delta_1, \ldots, \Delta_n$ generates a multiplexing action $f_\Delta(t)$:

$$f_\Delta(t)=O(w_i\,(g,\, x,\, x,\, a_i))=a_i \,,\; t \in \Delta_i \quad i = 1, ..., n \tag{22}$$

A feedback multiplexing action over Δ is termed synchronous if the subintervals Δ_i $i = 1, ..., n$ are all equal. In this paper only synchronous multiplexing is considered.

Now the concept is particularized for robot manipulators.

Suppose that a manipulator has a DC motor at each of its joints. Each motors armature voltage is driven at every instant of time by exactly one of three possible controllers: a position controller, a rate controller or a force controller further suppose that the time line is divided into intervals of duration Δ and each Δ is further subdivided into 3 subintervals Δ_1, Δ_2 Δ_3 at each Δ_i, i = 1,2, 3, and for each joint only one of the 3 possible controllers position, rate or force is active. Clearly this schema is a feedback multiplexing action over Δ if it is assumed that the three controllers for each joint are generated as functions of the form of (22).

Note that for an n-joint manipulator there are $3^{N+3} - 1$ different controllers. So, the function of a declarative controller is to select for each Δ interval and for each joint the type of length -3 control sequence to be applied and then to determine the appropriate polynomial of the controller algebra for each element in the sequence.

As discussed in the previous section, this is accomplished by generating a locally finite automaton and simulate it to generate the values of the joint commands. The LFA at each Δ interval is a representation of the proof of the theorem characterizing the task to be accomplished by the manipulator in this time interval. This is illustrated with an example int he next section

5. EXAMPLE

The concept of multiplexing declarative controllers will be illustrated with a simple manipulator. The manipulator is a plannar three-link chain with rotational joints and a rigidly attached and effector (Figure 3). The central characterisitcs of the manipulator are:

- The lists are assumed to be solid rigid long cylinders
- The joints are driven by ideal permanen-magnet DC motors
- Each joint is coupled to its driving motor by a "sloppy" gear box whose characteristics are shown in Figure 4. This gear box characteristic is similar to that in each joint of NASA'S Shuttle RMS.
- Each joint is equipped with a position, rate and force sensors
- A sensor, which detects the quadrant in which the end effector is with respect to base coordinates, is attached to the end effector.

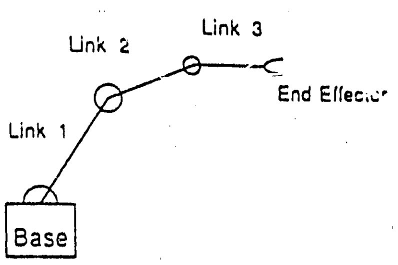


Figure 3. Simplified Robot Geometry

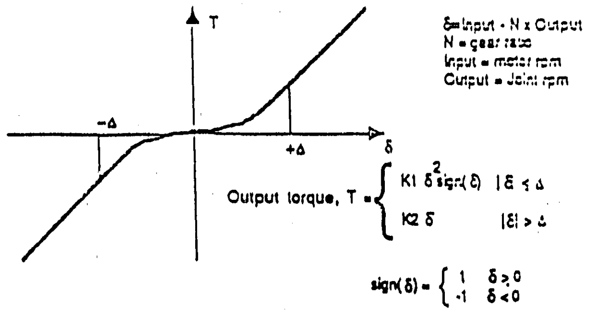


Figure 4. Joint Gearbox Characteristic

The dynamics of the manipulator is represented in the knowledge base with the following Langrangian conservation principle.

$$L = E - V - \sum_{i=0}^{3^{N+3}-1} V_i \tag{23}$$

where E is kinetic energy, V is the physical potential energy and V_i are virtual potentials representing the characteristics of the 3 types of controllers available at each joint in the multiplexing schema.

The maneuver to be accomplished by the robot is the following. The end effector is to follow a circular path exercising a constant force against it. The circular path deform's elastically under load. This is modelled as a preloaded spring distributed over the path. The stiffness of the spring is constant and uniform along the path. The base of the robot is not located at the center of the circular path.

In addition, tolerance for end effector position angular velocity and force are provided.

The controller also monitors the health of the manipulator with respect to failures such as joint runaways and tachometer failures and executes a crating maneuver if a failure is detected.

The thereom that characterizes this maneuver is given by:

$$\begin{aligned} \text{thereom}\,(\ldots) \quad \Leftarrow \quad & \text{pos}\,(r, b, c, \varepsilon_p) \wedge \\ & \text{vel}\,(r, b, c, \varepsilon_v) \wedge \\ & \text{for}\,(r, b, K_{EE}\, \varepsilon_f) \wedge \\ & \text{HMS}\,(\ldots). \end{aligned} \tag{24}$$

In (24) pos(...) is the end effector position lemma, vel(...) is the end effector velocity lemma, for(...) is the end effector lemma. r is the radius of the circular path b is the rleative position of its center with respect to the base, c is the vector of joint command (multiplexing actions) and ε_p, ε_v, ε_f are the corresponding tolerances. HMS (...) is the health monitoring lemma.

A multiplexing declarative controller for this robot was implemented in Quintus Prolog on a Sun 3-160 workstation. Following are some sample results of a typical run. First Δ was selected to be 6 milliseconds. Δ_i , i =1, 2, 3 was selected to be 2 milliseconds. The maneuver was completed in 6.48 seconds. The system ran 1.5 times faster than real time.

Figure 5a shows the angle described by the end effector with respect of the base as a function of kilo samples. Note that the path followed is nearly linear in angle (no detected backtracking). Figure 5b, 5c, and 5d, show the position velocity and force potentials of the end effector with respect to kilosamples. Note that these potentials, after an initial transient reach a steady state which is maintained throughout the maneuver in spite of the fact that the manipulator go through several singularities as can be seen in Figure 5e that shows the path as detected by the quandrant sensor.

Notice also the effects of those singularities in the magnitude of the end effector angular velocity, Figure 5f.

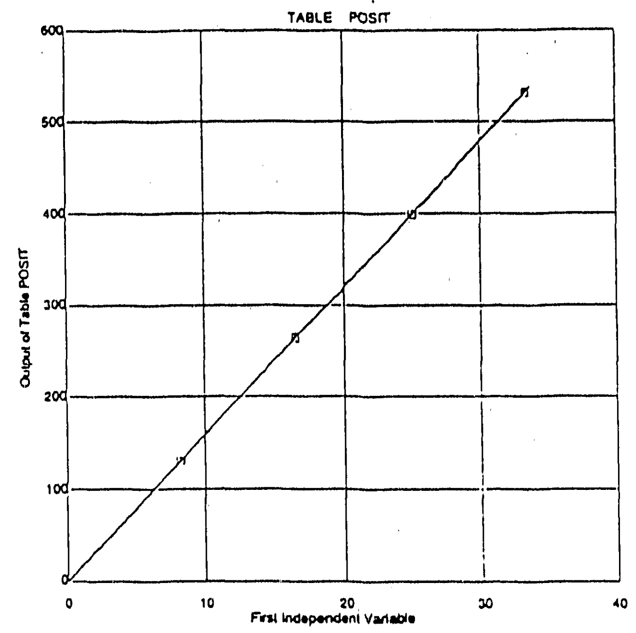


Figure 5a

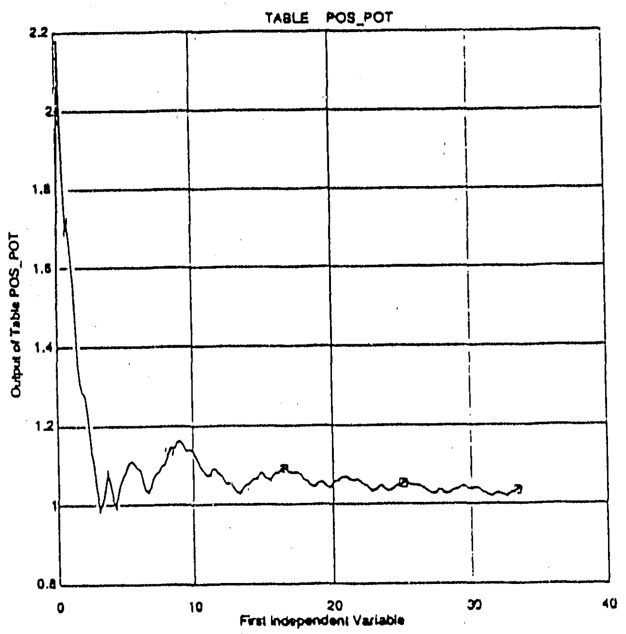


Figure 5b

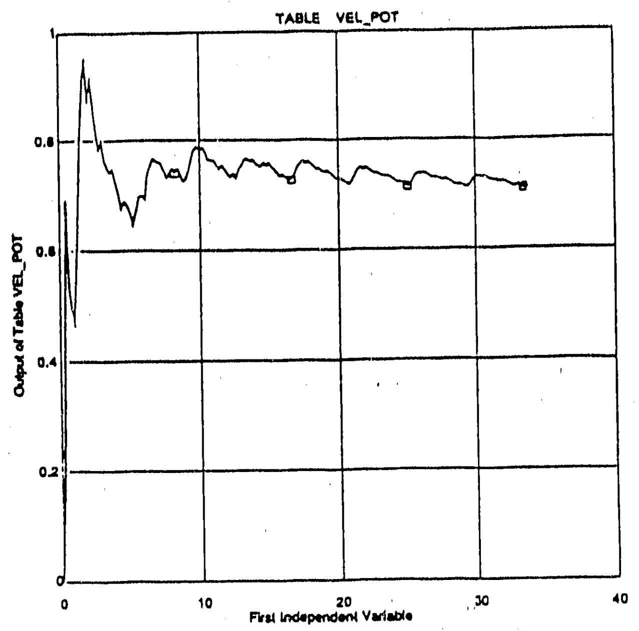


Figure 5c

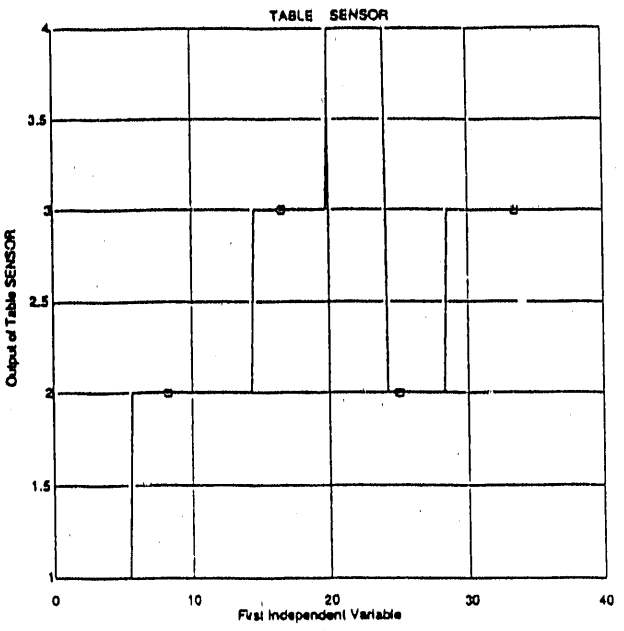


Figure 5d

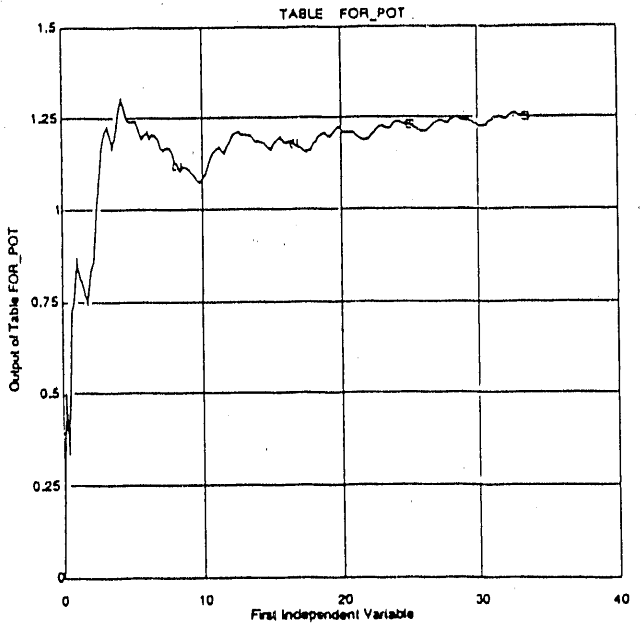


Figure 5e

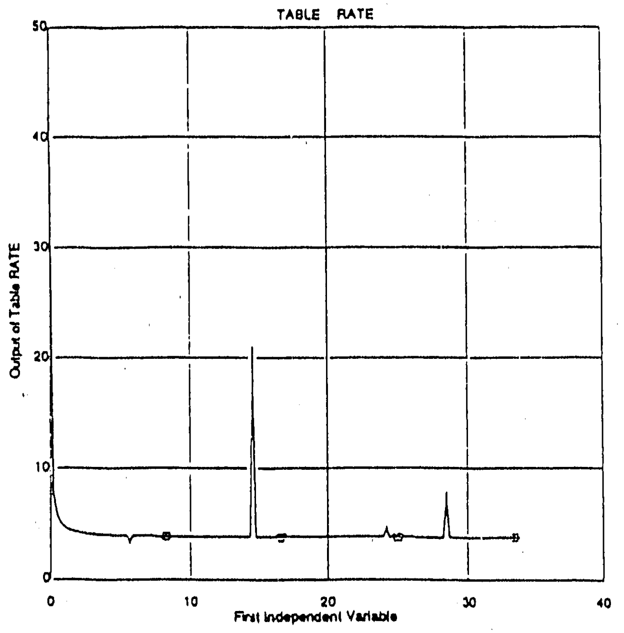


Figure 5f

In summary, the concept of declarative multiplexing controllers was used in solving a reasonably complex robot control problem and shows all the advantages predicted by the theory.

Finally the knowledge acquisition and coding took about 2 weeks.

6. CONCLUSIONS

A new class of knowledge based controllers was introduced. The class termed declarative rational multiplexing controllers shows great promise for addressing the verification issues discussed in Section 2.

In addition it was illustrated with the example, which is a representation of a wide class of robot control problems, that the concept yields to feasible implementations.

REFERENCES

[1] Kohn, W. "Declarative Theory of Rational Controllers" proc. of IEEE CDC 1988, Vol. 1, pp. 130-136, Austin, TX, Dec. 7-9, 1988

[2] Kohn, W. "Hierarchical Control Systems for Autonomous Space Robots" proc. of 1988 AIAA GN&C conf., Vol. 1, pp. 382-390, Aug. 15-17, 1988, Minneapolis, Minnesota

[3] Kohn, W. , Butter, J., Graham, R. "The Rational Tree Machine" proc. Applications of Artificial Intelligence VII SPIE Vol 1, pp. 264-274, Orlando, Fla., March 28-30, 1989

[4] Cadzow, M. "Discrete-Time and Computer Control Systems" Prentice Hall, Englewood Cliffs, NJ., 1970

[5] Morgenstern, L. "A First Theory of Planning, Knowledge and Action" proc. of the 1986 conf. on Reasoning about Knowledge: Theoretical Aspects, pp. 99-113, March 19-22, 1986

[6] Blanchard, D.C., Myers, R.M. "The Knowledge Representation Tool" proc. of Robex 85, Houston, June 27-28, 1985, pp. 137-140

[7] Kohn, W., Carlsen, K. "Symbolic Design and Analysis in Control" proc. of the 1988 Grainger lecture series at U. of Illinois, Urbana, May 23-25, 1989, pp. 40-52

[8] Kohn, W. "Declarative Control Theory" accepted for publication in AIAA GN&D Journal, Feb., 1989

[9] Skillman, T., Kohn, W. "A Class of Hierarchical Controllers and their Blackboard Implementation" accepted for publication in AIAA GN&D Journal, Sep., 1988

[10] Taylor, W. "Equational Logic" in Universal Algebra by G. Gr?atzer, 2nd edition, Appendix 4, Springer Verlag, NY., 1979

[11] Evans, T. "An Algebra Has a Solvable Word Problem iff it is Embedable in a Finitely Generate Simple Algebra" Algebra Universalis, 1978

[12] Kohn, W. Lecture Series on "Declarative Control and Rational Algebra", HTC, June 2 Aug. 28, 1988, Seattle, Wa.

[13] Lloyd, W. "Introduction to the Theory of Logic Programming" 2nd ed., Springer Verlag, NY., 1987

[14] Yashuhara, A. "Theory of Recursive Function Theory and Logic" Academic Press, 1966

[15] Kowalsky, T. "Logic Programming" Northholland, 1982

[16] Kohn, W. "Declarative Hierarchical Controllers" submitted to AIAA GN&D Journal, Feb., 1989

[17] Jurica, K., Kohn, W., Lai, D. "A Variable Configuration Controller for a Multipurpose Articulated End Effector" proc. of AIAA/NASA symposium on Automation Robotics and Advanced Computing for the National Space Program, pp. 11-18, Sept. 4-6, Washington D.C., 1985

An Expert System for Tuning Particle Beam Accelerators

Darrel L. Lager, Hal R. Brand, and William J. Maurer
Engineering Research Division
Electronics Engineering

We have developed an expert system that acts as an intelligent assistant to operators tuning a particle beam accelerator. The system incorporates three approaches to tuning:

•Duplicating within a software program the reasoning and the procedures used by an operator to tune an accelerator. This approach has been used to steer particle beams through the transport section of LLNL's Advanced Test Accelerator and through the injector section of the Experimental Test Accelerator.

•Using a model to simulate the position of a beam in an accelerator. The simulation is based on data taken directly from the accelerator while it is running. This approach will ultimately be used by operators of the Experimental Test Accelerator to first compare actual and simulated beam performance in real time, next to determine which set of parameters is optimum in terms of centering the beam, and finally to feed those parameters to the accelerator. Operators can also use the model to determine if a component has failed.

•Using a mouse to manually select and control the magnets that steer the beam. Operators on the ETA can also use the mouse to call up windows that display the horizontal and vertical positions of the beam as well as its current.

Introduction

Particle beam accelerators are members of a class of large, complex systems that must be operated by people rather than machines to be effectively controlled. When machines have been used to control such systems in the past, they have frequently failed, usually because the conventional approach of feedback control using a numerical model of the system has failed. This failure occurs because the system is strongly nonlinear, continually changes due to component failures, involves physical phenomena for which satisfactory models have not been derived, or may be so complex that it is simply too expensive to derive the model.

Despite these problems, individual operators are able to run such complex systems effectively. The operators appear to have a set of small, rule-of-thumb (heuristic) models for the various components of the system and to know how to manipulate those components to achieve the desired control.

In an attempt to duplicate the expertise of operators of large accelerators, we have developed an expert system called MAESTRO (Model and Expert System Tuning Resource for Operators) that models the procedures involved with tuning such accelerators and with fixing them when components fail (a frequent occurrence). MAESTRO is a software program that blends physics models of the system and operator heuristics. We chose the MAESTRO acronym to emphasize the metaphor of a conductor unifying and coordinating the activities of control, diagnostics, physics models, and post-run analysis — all activities that are critical to the success of physics experiments. Traditionally those activities would have been separated, and different people within different groups would have been responsible for them. However, one of the advantages of MAESTRO is that is imposes unity and consistency on the system while it is also helping operators tune the system more efficiently. As an example of the latter, operators can make much more informed control decisions because they can observe the simulation while the machine is running. Also, physicists can quickly gain insight into a particular phenomenon and make appropriate changes to the model because discrepancies between the model and the machine are more readily apparent.

Particle Beam Accelerators

A particle beam accelerator is a device that accelerates electrically charged atomic or subatomic particles, such as electrons, protons, or ions to high energies. Laboratory researchers use accelerators to study high-energy physics problems that arise in the Weapons, Beam Research, and Magnetic Fusion Programs. We have focused our efforts on two accelerators, the Advanced Test Accelerator (ATA) located at LLNL's Site 300, and the Experimental Test Accelerator (ETA) located at the Laboratory itself. The ATA is the more complex of the two and consists of an injector, an accelerator, a transport section, an emittance selector, a tuning dump, an achromatic jog (a-jog) and a wiggler. The injector produces a pulse of electrons and injects it into a beam pipe, an evacuated pipe a few inches in diameter running the length of the machine. (A beam pipe with its associated components is called a beam line.) The accelerator section increases the energy of the electrons in the beam pulse up to about 50 MeV. An alignment laser guides the beam through the accelerator section and delivers it to the transport section. The transport section steers and focuses the beam into the emittance selector which "strips off" the electrons with undesired energy. A bending magnet directs the beam either to the tuning dump which absorbs the energy in the beam while the operator is tuning, or to the a-jog which moves the beam onto the wiggler beam line. The wiggler couples energy from the electron beam into another laser beam, greatly increasing the energy in the laser beam. A typical experiment is to deliver an electron beam into the wiggler and investigate its characteristics for producing gain in the laser beam. The ETA is somewhat simpler than the ATA and consists of an injector, an accelerator, a transport section and a wiggler. One of the differences between the two is that the ETA accelerator increases the energy of the electrons in the beam pulse to only about 6 MeV, without using laser guiding. Another difference is that the wiggler couples energy into a microwave beam, increasing the energy of the beam. In a typical experiment the ETA produces high-power microwaves that heat a magnetic fusion plasma. Both accelerators have a variety of diagnostic devices that determine the beam's position within the pipe and its energy distribution. Both systems also have ancillary components (vacuum pumps, safety systems, cooling systems, and pulsed power systems) in their beam lines, further increasing their complexity.

Tuning

The goal when tuning an accelerator is to produce a beam that has the desired temporal and spatial energy and charge distributions required for a given experiment. The optimum distribution or "beam profile" is depicted in Fig.1, where the beam pulse traveling down the beam pipe is on center within a few millimeters, all the electrons have uniform axial velocity and zero transverse velocity, and all are uniformly distributed within the pulse. In reality the beam has nonuniform velocities and is subject to a variety of instabilities, causing it to disperse at the nose and tail. (A common instability is a corkscrew, named for the spiral shape assumed by the beam.) Serious instabilities cause the beam to break up or strike the wall of the beam pipe.

Figure 2 shows the devices used for tuning the beam line. The beam bugs shown at the left of the figure are diagnostic devices for determining the beam position and current profile. A beam bug produces three oscilloscope traces as the beam pulse passes through it, one for the current, one for the x-position, and one for the y-position. The operator determines if the beam has the desired axial charge distribution and total charge by observing the current waveform. The position waveforms give the beam position along the horizontal (x) and vertical (y) transverse axes of the pipe. Video cameras record the light emitted when the beam strikes foils inserted into the beamline. The resulting images give information on the transverse distribution of the beam pulse and are used to determine the beam focus.

Steering magnets change the direction of propagation of the beam. It requires a pair of steering magnets a distance apart to displace the axis of propagation without changing the direction. The first changes the angle of propagation and the second compensates for the angle introduced by the first. Quadrupole magnets focus the beam and have the usually undesirable side effect of steering the beam when it enters off-center. A general rule-of-thumb is to keep the beam on-center through the quadrupoles to suppress the unwanted steering.

Complications

The tuning task is complicated by things the operator can readily adapt to, but are difficult for computers to deal with. Since the machine is made up of a large number of components, and since many of them are highly stressed state-of-the-art designs operating at high voltage and high current, there are random failures in the hardware that occur each day. Common occurrences are power supplies failing and the alignment laser drifting out of position. Less often magnets will short- or open-circuit. The operator handles these by periodically observing the status of the machine. If he

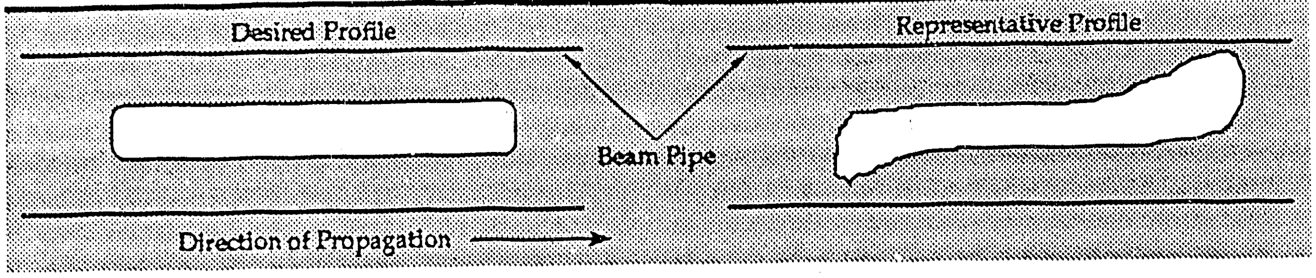


Figure 1. Beam profile while tuning.

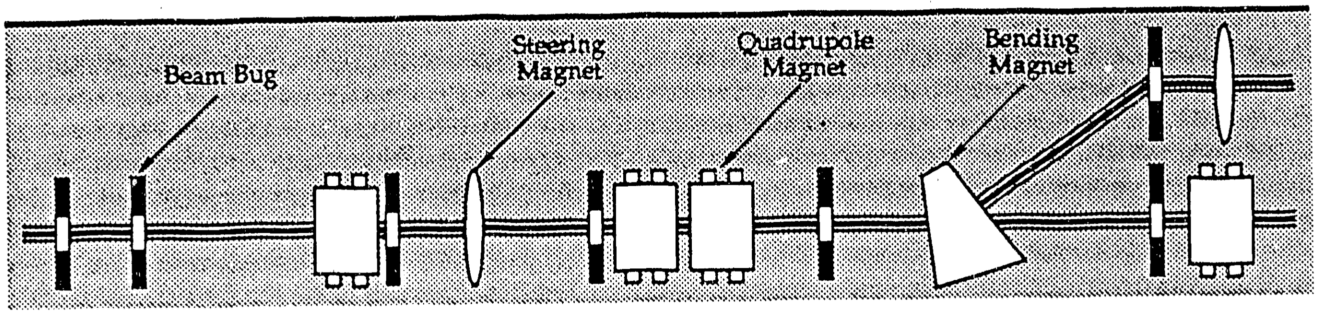


Figure 2. Components affecting tuning.

can't correct the problem, he modifies his tuning strategies and attempts to tune anyway until the problem is fixed.

The operator must also deal with unreliable diagnostics data. There is a high shot-to-shot variation in the characteristics of the trace data. There are also bad shots that must be ignored entirely when the machine misfires. As a result the operator observes several shots and determines an "average" of the beam characteristics. He will also devote one beam bug to watching the beginning of a section so he can determine if changes in the bug he is observing are caused by his tuning efforts or by changes in the beam characteristics at the input.

Since these accelerators are research tools the hardware is often reconfigured. Generally, the changes involve adding or deleting components, moving existing ones to different places or changing the relationships among them (e.g., connecting magnets to different power supplies). Another way the configuration can change is that during a run the beam may be directed down different beam pipes to do a desired experiment. The operator is usually told of the changes at the beginning of a shift and readily adapts to the differences. However, the expert system had to be designed with an interface that could tell it about the changes and could verify that the MAESTRO representation reflected the true machine.

Another complication the operator easily adapts to is that the machine is not built exactly as designed. Beam bugs may be installed with their measurement axes rotated away from perfectly vertical or horizontal. The operator simply realizes this and instead of just using the x-axis knob to change beam position he will vary both x and y to do it. He will also move a knob and if the beam steers in the wrong direction, he will just move the knob the other way. The components are also imperfect so they all need calibration curves. The operator easily deals with these problems because much of the tuning is aimed at achieving a position equal to zero (on center). However a model needs more precise knowledge than an operator of the calibrations, positions, and orientations of the devices if it is to compute a desired change in one step (an operator can determine a change experimentally).

Finally, the machine does not measure the same way from one day to the next, even when all the hardware is unchanged. Various critical aspects of the machine are impossible to measure accurately enough. For example the ion gauges that measure the vacuum profile along the beam path are not sufficiently accurate. As a result the benzene profile is not the same as before and the propagation of the beam through the benzene is slightly different. The physics of transport of a beam through benzene are not well understood, so it is extremely difficult to compensate for the measurement system by modeling. The operator has evolved tuning strategies that are relatively insensitive to these phenomena.

Expert System Techniques

MAESTRO blends three distinct tuning approaches to achieve better performance than any one alone. A trade-off among these approaches can be made as the machine is modified or grows more complex, as the operators learn more of its idiosyncrasies, or as more is understood about its physics. In the first approach, called "cloning the operator," we encode as faithfully as possible the procedures and reasoning followed by the operator. This was the approach taken for tuning the ATA because it was not possible within the allotted time to develop an accurate model for the beamline. A disadvantage of only using this approach is that operators may not be able to develop successful procedures for the far more complex machines being designed, such as the superconducting supercollider. A second approach, model-based tuning, simulates the beam propagating through the sections of the machine. The notable aspect of this approach is that the model is hooked into the actual system and is running at the same time the machine is running, so the simulation is based on measurements that are being made by the machine at the moment the machine is making them. This approach allows the operator to compare the simulation with the actual beam position, to choose an optimum set of parameters for centering the beam, and to download those parameters onto the real machine at a substantial savings in time. The disadvantage is that it may not be economically feasible to develop a sufficiently accurate simulator, either because the machine may be too complex or because it changes too often due to component failure. The third approach is to tune the system manually, but provide the operator with more powerful tools for tuning the machine. For example, this might take the form of displays derived from the raw data or different interfaces that make it easier to control the machine. The goal is to achieve a blend of these approaches that minimizes the time required to tune and maximizes the time available for performing physics experiments. Each of these approaches is discussed below.

Actually, no matter which approach we stress, we will always need the manual approach because operators will almost always have more knowledge of the machine than is economically feasible to encode into a computer program. Moreover, they can take into account information that may not be readily accessible to the computer such as the "sound" the machine makes when it is not running quite right. They may also be able to instantly diagnose a failure because they remember what happened when that same situation arose on a machine they were tuning 20 years ago.

Cloning the Operator

The operators of the Lab's accelerators go through a set of procedures when their machines are first powered up. The procedures reflect two different kinds of reasoning by the operator. In the first instance, the operator is concerned with the overall tuning of the machine ("global strategy"). Global strategy is made up of many lower-level "local strategies" that are each concerned with the tuning of a small section of the machine.

When operators are performing a local strategy, they reason about components upstream (opposite the direction of propagation) and downstream from a chosen diagnostics device. For example, as shown in Fig. 3, when steering through the transport section, the operator observes the trace data and determines there is a position error at a bug. He reasons about the devices downstream and decides whether it is desirable to correct the error at this bug or to ignore it and examine the next bug. He reasons about devices upstream to decide which can be used to correct the error. He usually chooses a steering magnet and changes its field strength until the position error is zero.

In pseudo-english form this strategy can be described as:

Walk the bugs of the transport section moving downstream from the one nearest the injector — if the position error at a bug is too large and the error can't be ignored

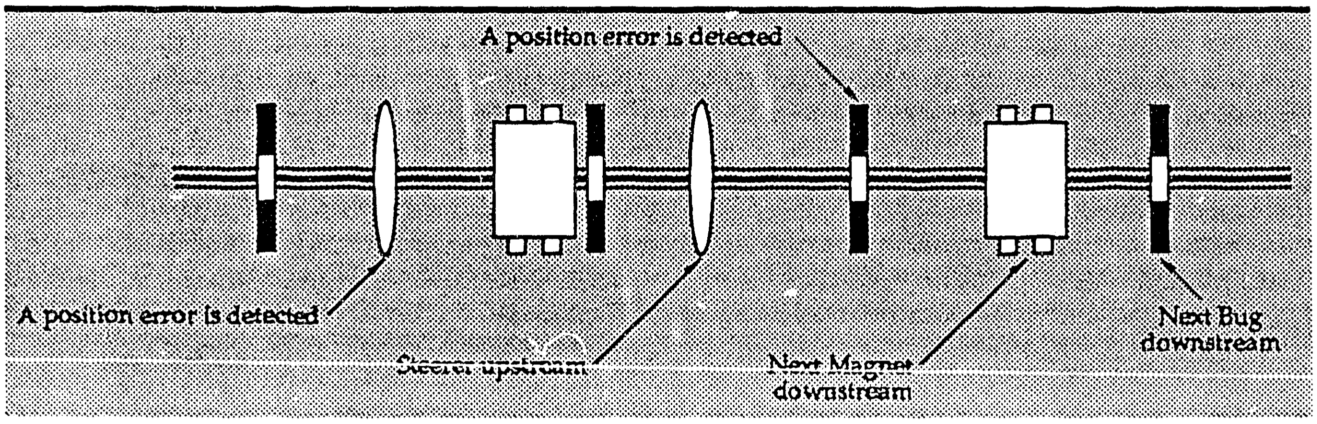


Figure 3. The operator reasoning about devices upstream and downstream from a bug.

then reduce the error to zero by tweaking the steering magnets upstream
if there are none, align the laser
if there are two with no quadrupole between them, use them as a pair
if the first upstream steerer is too close, use the second one upstream
otherwise use the first upstream steering magnet

Note that this strategy is relatively unaffected by component failures, since a failed component is simply deleted from the beamline and is no longer considered as an upstream or downstream component.

The global strategy is concerned with properly applying local strategies, based on the present state of tune of the machine. The guiding philosophy is "focus and steer the beam but don't put the beam into the wall." Given that philosophy, the operators first check each major section of the accelerator to see if the beam has arrived from the previous section. If it has not, the operators go back to the previous section and use a coarse steering technique to get the beam through the section. Once the beam has made it all the way to the tuning dump (that portion at the end of the accelerator that can absorb the full energy of the beam without disrupting the system or damaging components), the operator goes back to the begining of the transport section and meticulously centers the beam while monitoring the beam current at the tuning dump. We have encoded the local strategies followed by the operator in a representation we call a Monitored Decision Script (MDS). An MDS is an extension of the notion of scripts introduced by Schank.[1] Scripts are used to represent step-by-step procedural knowledge. The canonical example is the script for eating at a restaurant — first you are seated, then you order, then you eat, then you pay, and then you leave. These things have to be performed in order, but exactly what is done at each step is decided when you visit a particular restaurant. We use the Script part of an MDS to represent the procedures involved in performing the local strategies. The Decision part of the MDS name comes from the decisions that must be made when errors are detected (say when a magnet doesn't respond to a request to change its field because the supply connected to it has failed). The Monitor portion of the name comes from monitors that are periodically examined to check the health of the system. This emulates the behavior of the operator, for example, when he periodically checks to see if the laser alignment has drifted off position.

To implement the global strategy, we have written a number of separate MDS's that are grouped according to what state the accelerator is in; that is, whether the accelerator is in the start-up, coarse, or fine-tuning states. One of these states is listed in a so-called pre-condition field of each MDS. Each MDS also contains a post-condition field, within which is listed the result of successfully executing the MDS. For example, if a particular MDS has a pre-condition field that lists initial start-up, then the post-condition field would state that the beam has reached the tuning dump. We have also developed an AI program called an inference engine to manipulate the MDS's and perform the global strategy. The engine first checks to see which state the accelerator is in; next, it matches that state with one of the MDS's that lists the state in its pre-condition field; next, it selects one of these MDS's and executes it. Executing the MDS puts the machine in a new state (not necessarily the one listed in the post-condition), then the cycle of match, select, and execute is repeated until the accelerator is tuned.

We applied the MDS and inference engine programs to the problem of centering the beam in the transport section of the ATA accelerator. Our initial approach was to decompose the steering problem into two simpler, decoupled problems; namely, centering the beam in x, then y. We thought this was possible because our system had "knowledge" of the beam bug rotation angles and other information that would enable decoupling the x and y steering. Unfortunately this approach was unsuccessful because there was coupling between the vertical and horizontal steerers that we were unable to represent given our time constraints. Instead we modified our approach to more faithfully incorporate the strategy used by the operator. This was performed in two phases. In the first phase we diagnosed a pair of steerers and determined which more strongly influenced the x position and which the y. In the second phase we centered the beam using the steerers by repeatedly halving the error in x, then y, using an optimization algorithm.[2] This approach successfully automated the centering of the beam.

Model-Based Tuning

For the ATA discussed above, we chose to clone the operator as the best way to automate the system, given the complexity of the accelerator and the limited amount of machine time available to us. For the ETA we chose to model the system because the ETA is less complex than the ATA and there has been more of an opportunity to get onto the system and model the various components. Consequently, this fiscal year we are much closer to a tuning model of an accelerator.

Making use of such a model involves two distinct phases. The first is the "commissioning" phase[3] during which the simulator is matched with the real accelerator by measuring what effect each component has on the beam. This phase forces proper bookkeeping because the effect of each component on the beam trajectory must be calculated accurately for the model to work. Any rotations, tilts, offsets, and miscalibrations must be eliminated or incorporated into the model during this phase so the simulator and real machine produce the same beam trajectory. As part of the commissioning phase we have developed procedures to measure various beam parameters. For example, we measure the effects that steering magnets have on beam position by sweeping horizontal and vertical steering magnets through a range of values, producing the cross-shaped pattern in Fig. 4. Similarly, we measure the effects that solenoid magnets have on focus by sweeping through a range of settings, producing the spiral-shaped pattern shown in Fig. 5. We have started the commissioning phase on the injector section of the ETA accelerator and have chosen the smallest subset possible: one magnet followed immediately by a beam-bug. Once the effects of these components have been measured, we will commission the remaining injector magnets and then proceed down the accelerator beginning with the first ten-cell set. Once we're finished with the accelerator, we will tackle the remaining beam lines.

As discussed above, during the commissioning phase, we bring the simulation model and the accelerator into relative agreement. In the second, operational phase we can actually begin to tune the accelerator and diagnose failure. To tune the accelerator, the first step is to turn the system on and measure actual beam position in the accelerator. Given these measurements, the initial beam energy, and the commissioned model we can estimate the launch conditions of the beam - its initial position and transverse velocities (denoted as x, x′, y, y′). These conditions are estimated by fitting the measured data to simulated data and then varying the launch conditions to get the best fit. Finally, we can use a nonlinear parameter estimator (part of the simulation model) to select tuning parameters that will center the beam. Obviously, a very large number of tuning parameters are available. Initially, an experienced operator will select a subset of these for tuning, and iterate the process with different subsets. Eventually, the operator's knowledge and experience will be encoded into an expert system that will select the subset of tuning parameters and will decide how much iteration is necessary.

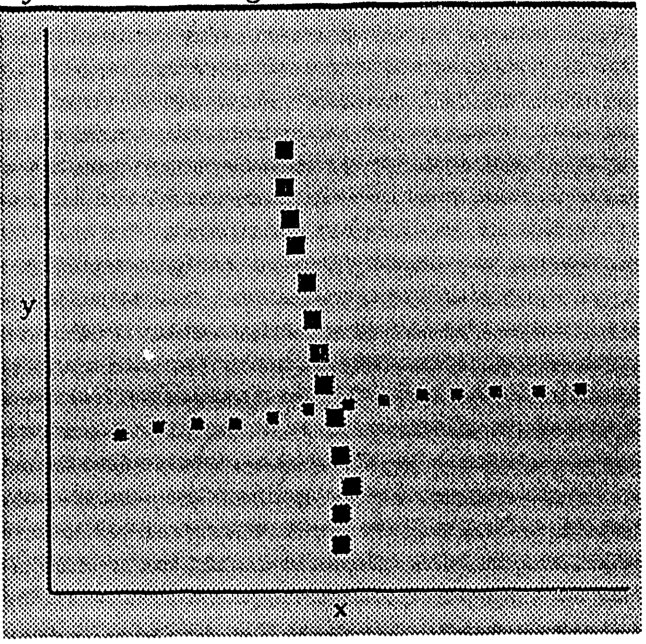


Figure 4. Variation of x and y beam positions as the vertical and horizontal steering magnets are swept through a range of values.

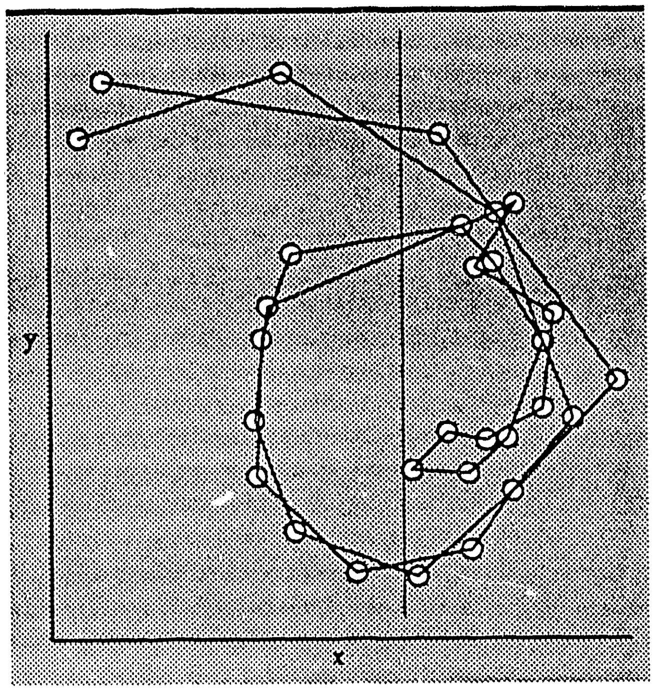


Figure 5. Variation of x and y beam positions as the solenoidal field is varied through a range of values. The solenoidal field causes the beam to spiral as it passes through the magnet. As a result, varying the field causes the locus of points representing the beam position to assume a spiral shape. The spiral becomes tighter as the field increases.

When the simulation model and the accelerator go out of relative agreement (due to component failure), the model can be used to find those regions within the accelerator where there is still agreement. For a single component failure, there will be two regions of agreement roughly surrounding the failed component. From this information, a list of suspect components can be compiled. With the aid of an expert operator (or eventually, an expert system), a number of possible failures can be proposed. The nonlinear parameter estimator is then used to estimate both the magnitude of the proposed error and the improvement in the model's predictive ability, given the proposed error and the data currently available from the accelerator sensors. With this information, many proposed errors can be rejected because either the magnitude of the error is beyond reasonable limits, or the improvement in the model by the addition of the error is too small. In the ideal case, one is left with only one reasonable error. In the other cases, one is normally left with only a few possibilities. Beam redirection experiments can then be performed to further isolate the accelerator-model discrepancy. Selection of these experiments will initially be done by experienced operators, but will eventually be included within the expert system's capabilities. When the problem is finally found, then either the accelerator can be fixed and/or the model can be updated with the (fit) error information, and tuning can proceed as before.

In conclusion, model-based tuning promises to put more science into the art of tuning and lead to a more rigorous understanding of the machines. The model enables operators to "see" the effects of their tuning in the regions between (not covered by) sensors. Given sufficiently accurate models it should be possible to determine a set of parameters that will change the present state of the machine to a desired (tuned) state in a single step. Experiments can be performed off-line using the model to determine, for example, the effects of adding new components to the beamline at far less

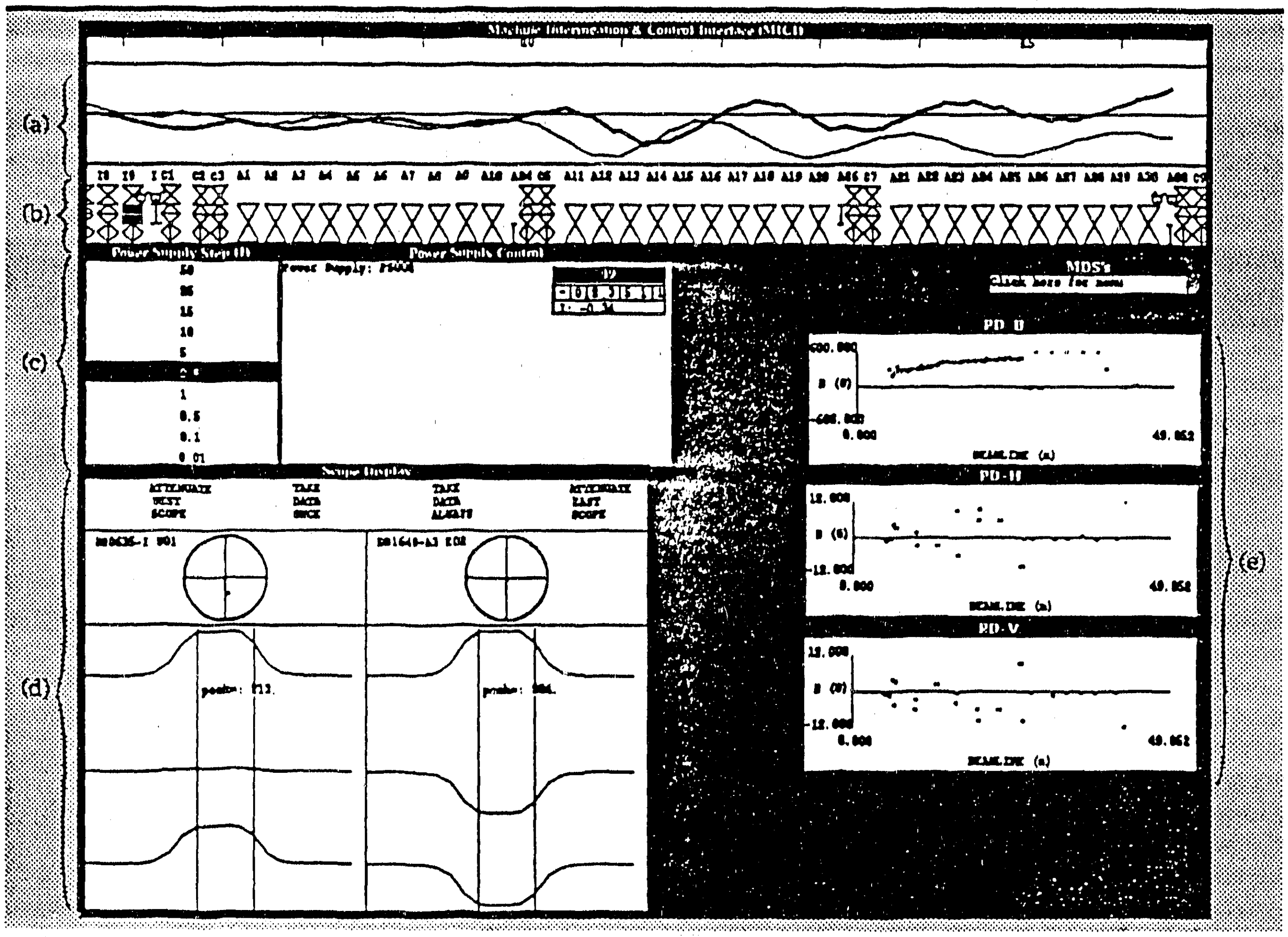


Figure 6. Screen dump of the windows associated with the manual interface to MAESTRO. The windows are: a) simulated beam position vs distance down beamline. The x position is the thin line and y is the thick line; b) icons depicting the components according to their positions in the beamline; c) power supply control window; d) scope display showing the I, x, and y waveforms from the selected beam bugs; e) field strength vs distance down the beam line for the solenoids, horizontal steerers, and vertical steerers.

expense than would be the case if the real machine were used.

Manual Interface

The manual interface for MAESTRO is shown in the screen dump in Fig. 6. The interface consists of several windows that become visible on the screen as necessary. The Machine Interrogation and Control (MICI) window is the main one for interacting with the system. It consists of two panes, one above the other. The top pane displays the output from the simulator, showing the horizontal beam position as a thick line and the vertical position as a thin line. The lower pane shows a set of icons depicting the components in the beamline. The locations of the icons and their shape are derived from the information describing the beamline in the MAESTRO knowledge base. Components added to the beamline are automatically included in the MICI display once the information has been added to the knowledge base.

The operator controls the magnetic fields by positioning a cursor over an icon with a mouse. Clicking over a vertical steering magnet icon, for example, causes power supply control windows to become visible below the MICI window (Fig. 6). By clicking the mouse as the cursor is positioned in the windows we can increase or decrease the current in the appropriate power supply by a given increment.

The Scope Display window is used to control and display data acquisition. Clicking the mouse over the appropriate label in the window causes data to be acquired and displayed as a set of three traces in the window: the time history of the beam current, x position, and y position. The actual location of the beam within the pipe is derived from the trace data after accounting for sensor misalignment and is displayed in the circular "bulls-eye" displays in the upper part of the window.

There are other windows that display additional derived data. A "position display" window shows a value vs position down the beam line. A "bug-walk," for example, makes position displays that show peak current, x position, and y position at the locations of the beam-bug position monitors along the beamline. Similarly a "magnet-walk" shows the fields at the centers of the magnets vs their location in the beamline. Three magnet walks appear in Fig. 6 showing the solenoidal fields (labeled PD-B), the horizontal steering fields (labeled PD-H), and the vertical steering fields (labeled PD-V) vs distance down the beamline.

There are also windows for displaying historical data. Clicking the mouse over a beam bug icon causes a "shot history" window to appear. By clicking the mouse over buttons on the window the operator can view all the oscilloscope traces acquired for that bug for that day's run.

Future

We are planning to improve the shot history mechanism so that it can not only manipulate past shot data but also past machine configuration data. For example, we want to have the ability to re-do the signal processing with different control parameters. We also want to be able to ask such questions as, "During the last three months what was the highest current magnitude measured when the machine had the long collimator installed?" Our approach is to develop an unstructured database based on the artificial intelligence representation scheme known as a semantic network.

We are also developing the ability to acquire image data from cameras observing the beam striking foils inserted into the beamline. We will apply image understanding techniques (the ability of an AI program to interpret or judge a visual image) to determine the position and focus of the beam.

Summary

The MAESTRO software environment was developed to function as an intelligent assistant to an operator tuning complex systems such as particle-beam accelerators. It incorporates three approaches to tuning. The "cloning the operator" approach uses an inference engine and the MDS representation to encode the strategies and reasoning followed by the operator. The model-based approach makes use of a beamline simulator and a non-linear least squares parameter estimator to first "commission" the model and then determine optimum tuning parameters. The third approach lets the operator perform tuning manually and provides him with displays that easily let him determine the machine status. Finally, a history mechanism lets the operator view past data to compare the present tune with ones previously obtained.

Acknowledgements

The authors wish to gratefully acknowledge the contributions of Doyle Rogers and John Clark, the tuning experts. Their ability to make order out of chaos is truly amazing.

Work performed jointly under auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract W-7405-ENG-48, for the Strategic Defense Initiative Organization and the U.S. Army Strategic Defense Command in support of SDIO/SDC MIPR No. W43-GBL-0-5007.

1. R. C. Schank and R. P. Abelson, "Scripts, PLans, Goals, and Understanding," Hillsdale, NJ: Lawrence Erlbaum (1977).
2. P. G. King, and S. N. Demming, "UNIPLEX: Single Factor Optimization of Response in the Presence of Error," Analytical Chemistry, Vol. 46, No. 11, pp 1476-1481 (September 1974).
3. M. Lee, and S. Clearwater, "GOLD: Integration of Model-Based Control Systems With Artificial Intelligence and Workstations," Workshop on Model-Based Accelerator Controls, Brookhaven National Laboratory, Upton, New York (August 1987).

Safety of Process-Control Software

Nancy G. Leveson
Info. & Computer Science
University of California, Irvine
Irvine, CA 92717
email: leveson@ics.uci.edu

INTRODUCTION

In recent years, advances in computer technology have gone hand-in-hand with the introduction of computers into new application areas. The problem of safety has gained importance as these applications have increasingly included computer control of systems where the consequences of failure may involve danger to human life, property, and the environment.

System-safety engineering has the goal of designing acceptable safety levels into systems by identifying and controlling potential hazards. Increasingly, system-safety engineers are finding themselves faced with the problem of ensuring safety in systems controlled by digital computers. At the same time, software engineers are being confronted with ultrahigh reliability requirements such as 10^{-9} probability of failure over some fixed time. These requirements are essentially an attempt to prevent accidents by building virtually perfect software, and they are obviously impossible to guarantee (or even to measure) with today's technology.

But perfection may not be necessary to prevent accidents. This is something that has long been recognized by hardware engineers, probably because they had no way to prevent wear-out failures in hardware and, instead, had to design to cope with failure. Although software-design perfection is theoretically possible, it may not be practical to achieve this in complex systems. An alternative is to take the hardware approach: to prepare for failures and try to minimize their consequences.

The goal of software safety is to ensure that software executes within a potentially hazardous system without causing or contributing to unacceptable risk of loss such as death, injury, property damage, environmental harm, financial ruin, and security leaks. Safety is enhanced if the risk of accidents is reduced, even though the process of risk reduction may require the (perhaps temporary) nonsatisfaction of some or all of the functional or mission requirements of the software or of the encompassing system.

Often, the design constraints needed to optimize safety conflict with those needed to optimize other qualities, including the probability of accomplishing the basic mission or functional requirements: Even if complete safety were achievable, it might cost too much in terms of resources and reduced functionality. In fact, few things in life are completely safe; how much risk is considered acceptable depends on the estimated value of the benefits of the activity compared to potential losses. Identifying and specifying the safety requirements early in the development process and establishing priorities when conflicts arise helps in the identification and evaluation of any necessary trade-offs.

Safety must be built into software starting from the earliest stages of system development and continuing through each step of software development. The goal of the Safety Project at UCI is to develop a set of techniques and tools that extend and adapt to software the methods used to control risk in the larger system within which the software is embedded. The approach combines standard software engineering techniques with proven system-safety engineering techniques and special software-safety techniques.

Because safety problems are system problems, solutions must, of necessity, involve an integrated, consistent, system-wide approach. This has resulted in our merging system engineering and software engineering (especially software requirements engineering) techniques. Our approach is an engineering approach that emphasizes modeling and analysis using both formal and informal techniques. The resulting software-safety techniques and tools now appear as recommended approaches in military standards and handbooks and are used around the world.

TECHNIQUES

Verification of safety differs from the usual verification of correctness in that the goal is to increase confidence that the code will never let an unsafe state be reached, although it may still be possible to reach an incorrect but safe state. Because the goal differs, different techniques become feasible and useful.

Safety-verification techniques often start from a hazardous output and work backward through the code or design either to demonstrate that the software cannot produce that output or to determine the conditions under which it can [LH83] One reason that backward analysis is practical for safety is that the number of unsafe states is usually much smaller than the number of incorrect states.

Safety analysis activities should span the entire software development process. This has several advantages: (1) errors are caught earlier when they are easier and less costly to fix; (2) information from the early verification activities can be used to design safety features into the code and to provide leverage for the final code verification effort; and (3) the verification effort is distributed through the development process instead of being concentrated at the end. Ideally, each step merely requires showing that newly-added detail does not violate the verification of the higher-level abstraction at the previous step. These verification activities may have both formal and informal aspects: static analysis using formal proofs and structured walkthroughs and dynamic analysis involving various types of testing to provide confidence in the models and the assumptions used in the static analysis.

Our techniques can be divided up into those for software hazard analysis, software requirements analysis, software design, code analysis, and configuration control and maintenance.

Software-hazard analysis. A hazard is a set of conditions that can lead to an accident (unacceptable loss). The goal of software-hazard analysis is to identify and categorize software-related hazards by likelihood and potential severity. For complex systems, it may be impossible to guarantee that all hazards have been identified and correctly assessed. But it is possible to make the system *safer* by optimizing the design according to the hazard assessment and then planning contingency actions in case a mistake has been made.

Determining all causes of a hazardous state is much more difficult than identifying hazards. Fortunately, this is not required to make a system acceptably safe. At worst, many systems can be designed to detect that a hazardous condition exists (without knowing why or how it occurred) and to take protective action such as failing safe. At best, analysis techniques can identify and prevent some potential causes, thus eliminating the need to fail into a safe state in those cases and increasing the system's overall reliability and effectiveness.

System-safety engineers have standard procedures for identifying system hazards. Once system hazards have been identified, fault-tree analysis or other modeling and analysis techniques can help to identify the software hazards — those actions or inactions of the software that alone or with other events can lead to the identified system hazards. We have devised algorithms for software hazard analysis and demonstrated their use on Petri-net models [LS87]. Our techniques help to determine software safety requirements directly from the system design; provide procedures to analyze a system design for safety, recoverability and fault tolerance; and provide guidance in the use of failure detection and recovery procedures. For most cases, the analysis procedures require construction of only a small part of the reachability graph.

We have recently extended this work to statechart models by adding an environmental-interaction model to statecharts and providing further analysis of the effects of failure [MEL90]. Most software requirements specifications model only the behavior of the software component. Much useful analysis can be performed on such models of the software requirements and many important types of errors detected. However, many of the most important problems in requirements specifications involve the interface between the software and the process being controlled. In order to find these, the model must include at least some aspects of that interface. We have defined a model, which we call an Environmental Interaction Model, and have illustrated it by extending the Statecharts modeling technique. Standard system engineering analysis procedures, such as Failure Modes and Effects Criticality Analysis (FMECA) and Fault Tree Analysis (FTA), can be performed on this combined model, and we have provided algorithms to accomplish this. The result provides a partial bridge of the unfortunate gap between system engineering and software engineering.

Once software hazards have been identified, they can be used to write software-safety requirements (or constraints), which then must be shown to be consistent with the software-requirements specification.

Software Requirements Analysis. Software requirements errors have been found to account for a majority of production software failures [BMU75,END75] and have been implicated in a large number of accidents. Errors introduced during the requirements phase can cost up to 200 times more to correct than errors introduced later in the life cycle [BOE81] and can have a major impact on safety. In fact, safety engineers have concluded that inadequate requirements specification and design foresight are the greatest cause of software safety problems [LEV86]. Therefore, techniques to provide adequate requirements specifications and to find errors early are of great importance.

We have recently been working on the semantic analysis of requirements specifications. In process-control systems, minor behavioral distinctions often have significant consequences. It is therefore particularly important that the requirements specifications distinguish the behavior of the desired software from that of any other, undesired program that might be designed, i.e, the software specification must be both precise (unambiguous), complete, and correct (consistent) with respect to the encompassing system requirements.

Our goal is to provide analysis procedures to help find these types of flaws (i.e., ambiguity, incompleteness, and inconsistency with system-level requirements) in the software requirements specifications for process-control systems. Special emphasis is placed on robustness and timing. The approach involves building a formal model (the requirements specification) and then analyzing it to ensure that the properties of the model match the desired behavior. Some of the analysis procedures involve the checking of consistency with criteria that must be satisfied by all such systems; these criteria often arise from the basic properties inherent in any process-control system. Other procedures rely on heuristics that can be used to improve the specification by examining, within the context of the particular process being controlled, properties that are often present in such systems.

Because our goal is to provide general semantic analysis procedures that can be applied to any black-box, behavioral requirements specification, we have devised a notation and analysis model that is independent of any specific, existing requirements language — a requirements state machine (RSM) — which is an abstraction of most state-based specification languages. The criteria and analysis techniques defined on the RSM can be easily mapped to many of the current real-time requirements specification languages. Our goal in doing this was not to provide another language for specification of requirements; the formal notation is for the purpose of providing rigor in defining the analysis procedures and criteria while requiring only a small number of primitives that are easily mapped to existing specification languages.

To date, we have defined the formal criteria that imply the types of specification correctness important in process-control software [JLHM91]. We are currently working on designing a real-time specification language that supports the type of modeling and analysis required to ensure these criteria. This work is being done in the context of a real-life, safety-critical avionics system. Because this work is being performed for a government agency, the resulting system requirements specification must be readable and reviewable, with very little instruction, by employees of the government and industrial representatives world-wide who are not computer scientists but rather application experts. At the same time, it must be usable as a software requirements specification and be expressed in a formal language that is amenable to safety analysis. Our approach to these constraints is to take a two-tier approach with a readable but precise and graphical top-level requirements specification that is tightly coupled with a lower-level specification amenable to formal analysis. The language will incorporate the best features of existing languages such as Statecharts [HAR87] and the A-7 specification language [HEN80] with new features added when existing languages do not fulfill our needs. Once the specification is completed, formal safety analysis procedures will be devised based on our previous work on safety analysis and semantic analysis of requirements.

Software Design for Safety. Once the hazardous states are identified and the software-safety requirements determined, the software can be built to minimize risk and to satisfy these requirements. Although safety-analysis techniques are necessary and useful, system safety cannot be ensured by analysis and verification alone: The analysis techniques may be so complex that they are themselves error-prone, their cost may be prohibitive, and high-confidence elimination of all hazards may require too severe a performance penalty.

Therefore, hazards will need to be controlled during the software's operation. A safe software design includes not only standard software-engineering techniques to enhance reliability but also special safety features such as interlocks, fail-safe procedures, and safety monitoring or assertions to control potential hazards.

At the high-level design stage, information about the software hazards and safety constraints can be used to identify safety-critical items (processes, data, and states). The identification process might involve backward flow analysis from hazardous outputs or other types of analysis procedures. We are working on procedures to derive safety invariants for each safety-critical module from the safety constraints [CHA90]. This information is important in the later verification steps and also in the design of assertions or other execution-time protection mechanisms.

Once the critical items have been identified, they can be subjected to special treatment in the design. For example, safety will be enhanced and later safety analysis simplified if the safety-critical code, variables, and states are minimized, isolated and protected. Isolation may, for example, be useful in satisfying certain types of safety constraints that involve enforcing separation such as ensuring that a safety-critical function is not inadvertently activated. Safety-critical data also needs to be protected from accidental alteration. Security techniques might be used to accomplish these goals. Rushby has shown how an idea from security, i.e., the encapsulation kernel, can be used to enforce certain types of safety constraints such as the isolation of critical modules [RUS86].

The safety analysis during high-level design results in a set of design constraints or assumptions that imply the overall software safety constraints. The low-level design must be shown to preserve these high-level design constraints. The analysis is also used to tailor the high-level design in order to reduce the necessary safety verification in the later stages of development.

The amount and type of safety analysis that can be accomplished during low-level design depends on how much and what kind of information is included in the low-level design specification. Not only must it be shown that the specified behavior of the individual modules preserves the individual module safety invariants (which were derived in the high-level design analysis), but it must also be demonstrated that the modules executing together preserve the high-level design constraints if this has not already been accomplished in a previous step [CHA90]. In addition, it may be possible to demonstrate that safety-critical variables and modules are adequately protected from errors in other parts of the software, at least at this level of abstraction. If a formal design language has been used, then formal analysis is possible. Again, information from this analysis can be used to design protection against hazards into the software.

I have laid out some general approaches to relating safety and software design [LEV86]. Our first attempt at investigating this relationship will be completed soon [CHA90]. We plan further work on this topic.

Code Analysis. Finally, after the coding has been completed, formal and informal verification is needed to ensure that the actual code is consistent with the assumptions made in the low-level design analysis, e.g., that the code preserves the module safety invariants, that the protection devices have been implemented correctly, and that the safety-critical functions have been properly isolated. In general, the goal at each of the analysis levels — requirements, high-level design, low-level design, and code — is to move the assurance of safety to the highest level of abstraction possible and then to show that the assumptions of this analysis are preserved throughout each of the levels of mapping down to the code.

The code-level analysis will probably involve a combination of techniques, including testing, formal proofs, and informal verification techniques such as Software Fault Tree Analysis [LH83, CLS88]. We have developed and demonstrated such techniques, and they are now used in industry. Currently, we are working on a tool to aid the analyst in performing software fault tree analysis.

Configuration Control and Maintenance. Whenever any changes occur to the software, either because of detected faults or because of functional enhancements, a safety analysis is needed to ensure that the changes are safe. This analysis starts at the highest level involved in the change: It may be necessary to start from the requirements analysis if the change involves a system safety constraint or the basic software functional requirements; in other cases, it may be necessary only to redo aspects of the design and/or code analysis. Some types of changes may not be allowed due to their potential decrease in the safety of the software or because they are deemed not worth the effort of recertification of safety.

Planning for such changes can help to minimize the re-analysis that is necessary. For example, one of the reasons to isolate safety-critical functions and data is to minimize the re-analysis resulting from changes in both safety-critical and non-safety-critical modules. Our methodology incorporates such concerns throughout.

CONCLUSIONS

Software safety is only recently beginning to be considered a unique and important software quality. Focusing on safety separately from other qualities allows conflict resolution and careful decision-making about trade-offs, allows differential handling of erroneous states, provides discipline and procedures to deal with errors, focuses attention and provides the possibility of assigning responsibility, and allows measuring and ensuring safety separately from other goals — a requirement of most regulatory agencies.

The US Defense Dept. now has strict standards (such as Mil-Std-882B) requiring special procedures for safety-critical system software. With the increasing number of accidents, interest in software safety is rising among other regulatory agencies, and new standards for safety-critical software are beginning to appear around the world.

The goal of the Safety Project at UCI is to provide support for development of safety-critical software that will allow software with acceptable risk to be built. Considering that there are no current practical software engineering techniques that alone allow such a high level of assurance as is required in most of these systems, our approach is to provide layers of protection while not depending on any one to ensure low risk. This approach augments good software engineering practice with (1) analysis procedures to identify hazards, (2) elimination and control of these hazards through various types of hardware and software interlocks and other protective design features using several layers of protection and backups, (3) application of various types of safety analysis techniques during the software development to provide confidence in the safety of the software and to aid in the design of hazard protection features, and (4) evaluation of the effectiveness of the analysis and design procedures to assess the level of confidence they merit.

Whether this approach is adequate depends upon the acceptable level of risk and how effective the software safety measures and external protection against software errors are judged for a particular application.

REFERENCES

[BOE81]
Boehm, B.W. *Software Engineering Economics*, Prentice-Hall, 1981.

[BMU75]
Boehm, B.W., McClean, R.L., and Urfig, D.B. "Some experiences with automated aids to the design of large-scale reliable software," *IEEE Transactions on Software Engineering,* SE-1(2), February 1975.

[CHA90]
Cha, S.S. *Safety Verification On Software Design*, Ph.D. Dissertation, ICS Dept., University of California, Irvine, December 1990 (expected).

[CLS88]
Cha, S.S., Leveson, N.G., and Shimeall, T.J. "Verification of Safety in Ada Programs," Proc. 10th International Conference on Software Engineering," Singapore, April 1988, pp. 377-386.

[END75]
Endres, A. "An analysis of errors and their causes in system programs," *IEEE Transaction on Software Engineering*, SE-1(6):140-149, June 1975.

[HAR87]
Harel, D. "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, 8:231-274, 1987.

[HEN80]
Heninger, K.L. "Specifying software requirements for complex systems: New techniques and their applications," *IEEE Transactions on Software Engineering*, SE-6(1):2-12, January 1980.

[JLHM91]
Jaffe, M.S., Leveson, N.G., Heimdahl, M., and Melhart, B. "Software Requirements Analysis for Real-Time Process-Control Systems," *IEEE Transactions on Software Engineering*, March 1991 (in press).

[LEV86]
Leveson, N.G. "Software Safety: Why, What, and How," *ACM Computing Surveys*, 18(2):25-69, June 1986.

[LH83]
Leveson, N.G. and Harvey, P.R. "Analyzing Software Safety," *IEEE Transactions on Software Engineering*, SE-9(5):569-579, Sept. 1983.

[LS87]
Leveson, N.G. and Stolzy, J.L. "Safety Analysis Using Petri Nets," *IEEE Transactions on Software Engineering*, SE-13(3):386-397, March 1987.

[MEL90]
Melhart, B.E. *Specification and Analysis of the Requirements for Embedded Software with an External Interaction Model*, Ph.D. Dissertation, ICS Dept., University of California, Irvine, July 1990.

[RUS86]
Rushby, J. "Kernels for Safety?," *Proc. CSR Workshop on Safety and Security*, Glascow, Scotland, October 1986. Also printed in T. Anderson (ed.), *Safe and Secure Computing Systems*, Blackwell Scientific Publications, 1989, pp. 210-220.

Real-Time Responsiveness in Distributed Operating Systems and Databases

Jane W. S. Liu
Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Introduction

The sample rate and control computations of a digital control system are typically selected to achieve the best tradeoff among many costs and benefits. Once the sample rate is chosen, the tasks of data transmissions and control computations in each sample period are considered to be time-critical. Every *time-critical* task must meet its timing constraint, which is typically specified in terms of its *deadline*. It is essential for the task to complete and produce its result by its deadline. A timing fault occurs when the result is produced too late; such a result is of little or no use. In other words, the embedded computer system which carries out the computations and supervises the data communication is a *hard real-time systems*. (Hereafter, by a real-time system, we mean a hard real-time computing system, unless it is stated otherwise.) A primary design objective of the operating system or application in a real-time system is to guarantee that all critical timing constraints are met at all times. Other optimization criteria, such as throughput and resource utilization, are typically of minor importance.

The rapid advances in computing and communication hardware, distributed and parallel algorithms, and artificial-intelligence techniques have accelerated the progress in real-time computing. The next generation real-time systems are likely to be based on parallel and distributed architectures, use highly parallel algorithms, and perform complex and intelligent functions. This paper discusses several critical problem areas in distributed, real-time operating systems and databases, together with our recent research efforts and future directions†. Specifically, Section II discusses the problems of end-to-end scheduling to meet deadlines in distributed systems, concurrency control to maintain temporal coherence of shared data, and network access to ensure timely message delivery.

An approach taken to provide flexibility in scheduling and resource management and to enhance fault tolerance in real-time systems is the use of the *imprecise computation technique*. Many factors, such as variations in processing times of dynamic algorithms and congestion or failures of the communication network, make it difficult to meet all the timing constraints in a dynamic environment. The imprecise computation technique minimizes this difficulty by trading the result quality for the amount of system resources and time. In particular, this technique prevents timing faults and achieves graceful degradation by making sure that an approximate result of an acceptable quality is available when the exact result of the desired quality cannot be obtained in time. Section III describes this technique in detail. The applicability of this technique to intelligent control is discussed.

II. Critical Problems in Distributed Real-Time Systems

As the applications of real-time systems become more critical and the underlying systems become more complex, one can no longer rely on the ad hoc methods that have been used traditionally for the

† This work has been partially supported by the U.S. Navy ONR Contract Nos. NVY N00014 87-K-0827 and No. NVY N00014 89-J-1181.

design and construction of real-time systems and for the specification, prediction and enforcement of their timing behavior. This fact has motivated the recent U.S. Navy ONR research initiative on real-time computing. In particular, the major objectives of the research efforts supported by this initiative are (1) to establish theoretical bases that support systematic and rigorous methods for the design, specification, and implementation of the next generation real-time computing systems and (2) to develop a set of basic building blocks of predictable and robust schedulers and resource managers for such systems. Despite the significant progress in recent years [1], many critical problems in the design and construction of distributed, real-time systems remain to be solved. Examples of these problem areas are end-to-end scheduling, real-time concurrency control and time-critical data communications.

End-to-End Scheduling

In a multiprocessor or distributed system, tasks may need to be executed on more than one processor. While there are simple, optimal and approximate algorithms for scheduling tasks on a uniprocessor to meet deadlines [2-5], such algorithms for task assignment and scheduling in distributed environments do not exist. One is forced to rely on either enumerative algorithms or simple heuristic algorithms. The former are typically too expensive to run for complex systems. The latter do not have bounded worst-case (that is, guaranteed) performance.

To illustrate the complexity of the distributed scheduling problems, we consider the simplest one among them: the *flow-shop problem* [6-9]. A flow shop models a multiprocessor or distributed system in which processors and devices (also modeled as processors) are functionally dedicated. Each task executes on the processors in turn, following the same order. For example, a control system consisting of an input processor, a computation processor, and an output processor can be modeled as a three-processor flow shop. The input processor reads all sensors; the computation processor processes the sensor inputs and generates commands; the output processor delivers the commands to the actuators controlled by the system. Each task, modeling a closed-loop tracker and controller, must be executed first on the input processor, then on the computation processor, and finally on the output processor. Alternatively, we can use the slightly more complex, flow-shop-with-recurrence model to characterize systems that have limited resources and hence do not have a dedicated processor for every function. In a *flow shop with recurrence*, all tasks execute on different processors in the same order; however, each task executes more than once on one or more processors. As an example, suppose that the three processors mentioned earlier are connected by a bus. We can model the bus as a "processor" and the system as a flow shop with recurrence. Each task executes first on the input processor, then on the bus, the computation processor, the bus again, and finally on the output processor. Each task executes on the bus twice.

Many real-time systems can be modeled as flow shops in which every task has a *deadline*. A scheduling algorithm is said to be *optimal* if it always finds a feasible schedule in which all tasks complete before their deadlines whenever such schedules exist. Past efforts in flow-shop scheduling have focused on minimization of completion time, that is, the total time required to complete a given set of tasks [6-8]. Unfortunately, almost every flow-shop problem that goes beyond the two-processor flow-shop problem turns out to be NP-complete. (For example, the general problem of scheduling to minimize completion time on three processors is strongly NP-hard.) We have examined several special cases that are tractable; we have developed (1) optimal algorithms for scheduling tasks in flow shops when the tasks have identical processing times on all processors and when the tasks have identical processing times on each of the processors but different processing times on different processors, (2) an optimal algorithm for scheduling tasks with identical processing times in a special class of simple flow shops with recurrence, and (3) a heuristic algorithm for scheduling tasks with arbitrary processing times in flow shops [6]. Many extensions of the flow-shop scheduling problem are of practical interest. Examples include periodic flow shops, as well as the general job shops in which tasks execute on processors in arbitrary sequences. We need to find effective ways to schedule tasks to meet deadlines for these more general models of distributed systems.

Concurrency Control to Maintain Temporal Coherence

Real-time tasks often share data. A concurrency control mechanism must be used to ensure data consistency, making it difficult to schedule such tasks to meet timing constraints. Traditionally, one approaches this scheduling problem by analyzing the schedulability of conflicting tasks and finding suboptimal heuristic algorithms to schedule them [10-17]. In general terms, this is the problem of scheduling tasks subject to resource constraints, which is NP-complete [10,11]. Several heuristic algorithms for scheduling tasks subject to resource constraints have been developed. These algorithms are mostly for the case when there are a small number of resources (for example, 10 or 20) and are not suitable when the resources, such as data, are numerous (for example, 1,000 or more). Recently, several new concurrency control algorithms have been proposed for scheduling database transactions with deadlines (e.g., [14-16]).

Alternatively, one can adopt a new approach by introducing a set of continuous criteria for temporal consistency that is more appropriate for many time-critical applications, such as intelligent control. In these applications, the notion of state- and view-consistency traditionally used in concurrency control studies can sometimes be replaced or supplemented by the notion of temporal consistency [17]. We consider data objects in a real-time database as models of real-world objects. A set of database objects is said to be *relatively temporally consistent* if it represents a valid snapshot of the state of the real-world objects modeled by it. It is said to be *absolutely temporally consistent* if the snapshot is sufficiently up to date. An objective of a concurrency control algorithm is to keep data temporally consistent in addition to maintaining state-consistency or view-consistency whenever it is necessary. We need to evaluate well-known concurrency control algorithms that ensure serializability in order to determine their average and worst-case performance in term of their ability to maintain temporal consistency [17]. An integrated scheduling and concurrency control strategy must keep the data as temporally consistent as required by the application in addition to maintaining data integrity whenever it is necessary. Basic algorithms and protocols are needed to make such a strategy feasible.

Real-Time Data Communication

It is likely that many existing network architectures and protocols are well suited for real-time applications. Unfortunately, existing performance data on them are inadequate to support the design and synthesis of real-time communication networks. Past work on performance evaluation of networks and protocols has been concerned primarily with average performance measures, such as the expected throughput and delay. In a distributed real-time system, the timely completion of distributed tasks can be ensured only when message transmission delay is reasonably predictable. It is not sufficient for a real-time communication network to have a small average delay and a large average throughput. Variations in message delay in such networks must also be sufficiently small. We need to have sufficiently accurate information on the probabilistic distribution of message delay.

III. Imprecise Computation

One effective way to avoid timing faults is to leave less important tasks unfinished if necessary. In other words, rather than treating all tasks equally, the system views important tasks as mandatory and less important tasks as optional. It ensures that all mandatory tasks are scheduled and executed to completion before their deadlines. Optional tasks may be left unfinished during transient overload when it is not feasible to complete all the tasks. The imprecise computation technique [18-27] uses this basic strategy but carries it one-step further. In a system that supports imprecise computations, every time-critical task is structured in such a way that it can be logically decomposed into two subtasks: a mandatory subtask and an optional subtask. The mandatory subtask is the portion of the computation that must be done in order to produce a result of acceptable quality. This subtask must be completed before the deadline of the task. The optional subtask is the portion of the computation that refines the result. The optional subtask, or a portion of it, can be left unfinished if necessary at the expense of the quality of the result produced by the task. An optional task that is not completed when its deadline is reached is terminated at that time.

To provide maximum flexibility in scheduling, it is ideal to design time-critical tasks so that they are *monotone*; a task is said to be monotone if the quality of the intermediate result produced by it is non-decreasing as it executes longer, that is, as more time is spent to obtain the result. The result produced by a monotone task when it completes is the desired result; this result is said to be *precise* or exact. If the task is terminated before it is completed, the intermediate result produced by it at the time of its termination is the best among all intermediate results produced within the available time. This result may be usable to the user; it is said to be *imprecise* or approximate. One way to return imprecise results is to record the intermediate results produced by each time-critical task at appropriate instances of the task's execution. Programming language primitives are provided [18-20] so that the programmer can specify the intermediate result variables to be recorded and the time instants to record them. In addition to the intermediate result variables, the programmer can also specify a set of error indicators. The latest recorded values of the intermediate result variables and error indicators are made available to the user upon premature termination of the task. By examining these error indicators, the user can decide whether an imprecise result is acceptable when the desired, precise result cannot be obtained in time. Therefore, a monotone task is logically composed of a mandatory subtask followed by an optional subtask.

The imprecise computation approach makes meeting timing constraints in real-time computing systems significantly easier for the following reason. To guarantee that all timing constraints are met, the scheduler only needs to guarantee that all mandatory subtasks are allocated sufficient processor time to complete by their deadlines; it then uses the leftover processor time to complete as many optional subtasks as possible. Only the mandatory subtasks are restricted to have bounded execution time and resource requirements. It is not necessary to eliminate non-determinism in the timing requirements of optional subtasks. A conservative scheduling discipline with guaranteed performance and predictable behavior (such as the rate-monotone algorithm [2,3]) can be used to schedule the mandatory subtasks. More dynamic disciplines (such as the earliest-deadline-first algorithm), that are capable of achieving optimal processor utilization but may have unpredictable behavior, can be used to schedule optional subtasks. In particular, when tasks are monotone, the decision on which optional subtask and how much of the optional subtask to execute can be made dynamically. Because the scheduler can terminate a task any time after it has produced an acceptable result, scheduling monotone tasks can be done on-line or nearly on-line.

Monotone Computational Algorithms

Monotone algorithms exist in problem domains such as numerical computation, statistical estimation and prediction, sorting and searching. We have been concerned with the design of monotone computational algorithms in those application domains where there are no monotone algorithms.

An example of the recent results is the monotone query processing algorithm that produces improving approximate answers to queries posed in standard relational algebra [21,22]. An answer to such a query is set-valued; it is a relation. For a set-valued query, a meaningful and useful set of approximate answers can be defined in terms of subsets and supersets of the exact answer. We have developed an approximate relational model to formally capture this semantics of approximation. Specifically, this model defines the approximations of any standard relation in terms of supersets and subsets of the relation, a partial-order relation over the set of all approximate relations for comparing them, and a complete set of new relational algebra operations on approximate operands. Every one of these relational algebra operations is shown to be monotone in the sense that the result of the operation is better when its operand(s) becomes better. Thus, an improvement in the operands of an expression containing these operators as primitives will lead to an improvement in the result of the expression. As read requests to retrieve the base relations of the query are completed or partially completed, such improvements are realized. The monotone query processing algorithm differs from the traditional query processing algorithms in an important aspect: a series of approximate answers are produced, each integrating the effect of additional data retrieved to answer the query. None but the final, exact answer requires the read requests for all base relations be completed before it can be produced. The final answer is the exact answer obtained by traditional algorithms. If query processing is prematurely terminated (due to a deadline for instance), some approximate answer will be produced, and the quality of this answer improves monotonically with the amount of base relation data retrieved and processed.

The semantics of approximation defined by the approximate relational model is not suitable for single-valued queries, for example, queries for which the exact answers are "yes" or "no". Approximate data models and monotone query processing strategies to produce approximate answers with other semantic meanings need to be developed.

Scheduling Imprecise Computations

We have developed several algorithms for scheduling imprecise computations. These algorithms are based on workload models and optimality criteria that explicitly account for the cost and benefit incurred when optional subtasks are left incompleted.

A general hard real-time scheduling problem is that of scheduling n tasks each of which has arbitrary rational ready time, deadline and processing time. The *ready time* of a task is the time instant before which its execution cannot begin. The *processing time* of a task is the amount of processor time required to complete the task. Tasks may be dependent. A task T_i is dependent on a task T_j if the execution of T_i cannot begin until T_j is completed; such dependencies are specified by a set of precedence constraints. Tasks may have different weights; weights of tasks measure their relative importance. In the corresponding new workload model of imprecise computations, each task is decomposed into a mandatory subtask followed an optional subtask; these subtasks have the same ready time and deadline as the task. A scheduling algorithm is *optimal* in the following sense: it determines whether feasible schedules that meet timing constraints and precedence constraints of all tasks exist, and when such schedules exist it finds one that minimizes the total length of the unfinished portions of optional subtasks. We have developed two algorithms for finding optimal preemptive schedules of dependent tasks on a uniprocessor system [23,24]. The algorithm for optimally scheduling tasks with identical weights has time complexity $O(n \log n)$. It can be easily modified and used to schedule independent tasks with identical weights on v identical processors. The time complexity is $O(n \log n + nv)$ in the multiprocessor case. The algorithm for optimally scheduling tasks with different weights on a uniprocessor system has a time complexity of $O(n^2)$.

Some applications may require that every task is executed satisfying the 0/1 constraint. The execution of a task is said to satisfy the *0/1 constraint* if its optional subtask is either completed before its deadline or discarded entirely. The problem of scheduling tasks with primary and alternate versions can also be formulated as one of scheduling with 0/1 constraint. The alternate version, with shorter processing time, is modeled as a mandatory subtask. The primary version is modeled as a mandatory subtask, with processing time equal to that of the alternate version, and an optional subtask, with processing time equal to the difference between the processing times of the two versions; the latter must be either completed or discarded entirely. A schedule satisfies the 0/1 constraint when the execution of every task according to the schedule satisfies the 0/1 constraint. Unfortunately, the problem of finding optimal preemptive schedules satisfying the 0/1 constraint, meeting timing constraints and minimizing the total error is NP-complete in general. The two efficient algorithms described in [24] for scheduling tasks with the 0/1 constraint can be used to schedule dependent tasks on a uniprocessor system when the optional subtasks have equal processing time. One of the algorithms is optimal when tasks have equal ready time and has time complexity $O(n \log n)$. The other algorithm is optimal even when tasks have arbitrary ready times and has time complexity $O(n^2)$. We have also found approximate algorithms with good worst-case performance bounds for scheduling tasks whose optional subtasks have arbitrary processing times to meet the 0/1 constraint and timing constraints.

A workload model commonly used in studies on hard real-time scheduling is the periodic-job model [25,26]. In this model, there is a set of periodic jobs to be scheduled. Each job consists of a periodic sequence of requests for the same computation. The *period* of a job is the time interval between two consecutive requests in the job. In scheduling theoretical terms, each request is a *task*. The ready time and the deadline of the task in each period is the beginning and the end of the period, respectively. We have extended this workload model to characterize imprecise computations for two different types of applications. Depending on the kind of undesirable effect caused by errors, we classify applications as *error-noncumulative* or *error-cumulative*. For the former type of application, only the average effect of errors is observable and relevant. Examples of this type of application include image enhancement and speech processing. In the workload model characterizing this type of application, the overall quality of the result of each periodic job is measured in terms of the average error in the results produced in several consecutive periods. The optional subtasks need not ever be completed. We evaluated several heuristic algorithms designed for scheduling error-noncumulative jobs. These algorithms not only ensure that deadlines are missed in a predictable manner as the load increases, but also make almost full use of the processor. Many of these algorithms can be used for on-line scheduling and generate feasible schedules with small average error. Detailed performance data can be found in [25,26]. Their most serious disadvantage is that they may fail to achieve zero error even when the processor is not overloaded. When it is known that the overload condition never occurs, classical algorithms should be used. The new algorithms are suitable when the overload condition occurs frequently or when the variations in the actual processing times of tasks are large.

For error-cumulative applications, the effect of errors in different periods is cumulative. Examples of this type of application include tracking and control. In the workload model characterizing error-cumulative applications, for every job, the optional subtask in one period among several consecutive periods must be completed within that period and, hence, is no longer optional. Thus far, we have considered only the simple case when the periods of the jobs are the same [26]. Schedulability criteria of jobs with different periods and error cumulation rates (that is, how often the optional subtasks are required to be completed by their deadlines) remain to be determined. Good heuristic algorithms for scheduling workloads consisting of different mixtures of jobs are needed.

The Applicability of The Imprecise Computation Technique

The applicability of imprecise computation technique in digital control systems needs to be investigated. Imprecision in the result of a control computation due to the premature termination of the computation introduces a new source of error, in addition to other types of error, such as quantization error and truncation error. The characteristics and effect of this new type of error must be determined for different types of systems. How this error affects the stability and performance of typical control systems remains to be studied.

IV. References

[1] See the proceedings of the IEEE Real-Time Systems Symposiums, 1987, 1988, and 1989.

[2] Liu, C. L. and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. Assoc. Comput. Mach.*, vol. 20, pp. 46-61, 1973.

[3] Dhall, S. K. and C. L. Liu, "On a real-time scheduling problem," *Operations Research*, Vol. 26, No. 1, pp. 127-140, 1978.

[4] Lehoczky, J. P., Sha, L., and J. K. Strosnider, "Enhanced aperiodic responsiveness in hard real-time environments," *Proc. Eighth Real-Time Systems Symposium*, pp. 261-270, San Jose, CA, Dec. 1987.

[5] Sha, L., R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols — an approach to real-time synchronization," Technical Report No. CMU-CS-87-181, Carnegie Mellon University, November 1987.

[6] Garey, M. R. and D. S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness*, W. H. Freeman and Company, New York, 1979.

[7] Garey, M. R., D. S. Johnson, and R. Sethi, "The complexity of flowshop and jobshop scheduling", *Math. Oper. Res.* 1976 vol. 1, pp. 117-129.

[8] Gonzalez, T. and S. Sahni, "Flowshop and jobshop schedule: complexity and approximation", *Operation Research* (1978) vol. 26-1, pp. 37-52.

[9] Bettati, R. and J. W. S. Liu, "Algorithms for end-to-end scheduling to meet deadlines," Technical Report No. UIUCDCS-R-90-1594, April, 1990, Department of Computer Science, University of Illinois, Urbana, Ill.

[10] Blazewics, J., J. K. Lenstra, and A. H. G. Rinnooy Kan, "Scheduling subject to resource constraints: Classification and complexity," *Disc. Applied Math.*, Vol. 5, pp. 11-24, 1983.

[11] Garey, M. R., R. L. Graham, D. S. Johnson, and A. C-C Yao. "Resource constrained scheduling as generalized bin packing", *J. Combinatorial Theory*, V. 21, pp. 257-298, 1976.

[12] Sha, L., R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols — an approach to real-time synchronization," Technical eport No. CMU-CS-87-181, Carnegie Mellon University, November 1987.

[13] Zhao, W., K. Ramamritham, and J. A. Stankovic, "Scheduling tasks with resource requirements in hard real-time systems," *IEEE Transaction on Software Engineering*, April 1985.

[14] Haritsa, J. R. , M. J. Carey and M. Livny, "Dynamic Real-Time Optimistic Concurrency Control", to appear in the Proceedings of the 11th Real-Time Systems Symposium, December 1990.

[15] Y. Lin and S. H. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjusıment of Serialization Order", to appear in the Proceedings of the 11th Real-Time Systems Symposium, December 1990.

[16] Korth, H. F., N. Soparkar, and S. Silberschatz, "Triggered real-time databases with consistency constraints," to appear in the *Proceedings of VLDB*, August 1990.

[17] Song, X. and J. W. S. Liu, "Performance of multiversion concurrency control algorithms in maintaining temporal consistency," to appear in *Proceedings of IEEE 1990 COMPSAC*, Chicago, Illinois, September, 1990.

[18] Lin, K. J., J. W. S. Liu and S. Natarajan, "Concord: a system of imprecise computations," *Proceedings of the 1987 IEEE Compsac*, pp. 75-81, Tokyo, Japan, October 7-9, 1987.

[19] Lin, K. J., S. Natarajan, J. W. S. Liu, "Imprecise results: utilizing partial computations in real-time systems," *Proceedings of the IEEE 8th Real-Time Systems Symposium*, San Jose, California, December 1987.

[20] Natarajan, S. and K. J. Lin, "FLEX: towards flexible real-time programs," *Proceedings of IEEE International Conference on Computer Languages*, October 1988.

[21] Smith, K. P., and J. W. S. Liu, "Monotonically improving Approximate answers to relational algebra queries," *Proceedings of COMPSAC '89*, Orlando, Florida, October 1989.

[22] Vrbsky, S., K. P. Smith, and J. W. S. Liu, "An object-oriented query processor that return monotonically improving approximate answers," *Proceedings of IFIP Workshop on Object-Oriented Systems*, July 1990.

[23] Shih, W. K., J. W. S. Liu, J. Y. Chung and D. W. Gillies, "Scheduling tasks with ready times and deadlines to minimize average error," *ACM Operating Systems Review*, July 1989.

[24] Shih, W. K., J. W. S. Liu, and J. Y. Chung, "Fast algorithms for scheduling tasks with ready times and deadlines to minimize total error," *Proceedings of the 10th IEEE on Real-Time Systems Symposium*, December 1989.

[25] Liu, J. W. S., K. J. Lin and S. Natarajan, "Scheduling real-time, periodic jobs using imprecise results," *Proceedings of the 8th Real-Time Systems Symposium*, pp. 252-260, San Jose, California, December 1-3, 1987.

[26] Chung, J. Y., J. W. S. Liu, and K. J. Lin, "Scheduling periodic jobs that allow imprecise results," to appear in *IEEE Transactions on Computers*, August 1990.

[27] Zhao, W and M. Berger, "An Analytical Model for Imprecise Distributed Systems," *Proceedings of International Conference on Computational Techniques and Applications*, July 1989.

LARGE-SCALE DISTRIBUTED COMPUTATION STRUCTURES FOR CONTROL SYSTEMS

J. Maitan,
Lockheed Missiles & Space Company, Inc., Research & Development Division,
3251 Hanover Street, Palo Alto, CA 94304-1191
e-mail: jmaitan@a.isi.edu

Abstract
The emergence of high-speed and low-cost communication components combined with the improved capabilities of a new generation of microprocessors creates a challenging opportunity in the area of large-scale distributed control systems. These systems may involve hundreds of nodes and will be based on the use of unreliable components and communications links. Thus, technologies that foster adaptability and scaleability, and that support self-organizing mechanisms are needed to integrate the working fabric of a large-scale distributed system. The challenge is to fuse the theory, technology, and development methodologies into a unified framework that cost-effectively satisfies the needs of the users of such large systems.

Recent reintroduction of analog computation performed in the continuous domain adds to the complexity of the programming of distributed systems. Analog computation used either as "neural networks" or "smart sensors" can be described in declarative form, but the impact of the full semantics of such an approach must be further evaluated. The paper discusses, based on descriptive examples, issues associated with languages used to describe control systems and their relevance to the implementation of distributed control systems.

1. Introduction

Any attempt to develop an environment to support the design of software for distributed intelligent systems implies that the methodologies used to develop such systems are well defined and can be cast into an effective set of software tools. In reality, such designs, especially those implemented in new technologies, will often violate existing assumptions and rules. Declarative description of control rules and perhaps the use of a hierarchy of constraints will help the development and product support of distributed systems. At the same time, the proposed environment must not alienate designers who use more traditional technologies, e.g., an all simulation approach.

To cost-effectively implement a design environment to support design methodologies ranging from the fully analytical models to those relying solely on simulations, one must be able to generate automatically tools that can accommodate widely diversified design styles. In this paper, we will use examples to highlight some idiosyncrasies related to the implementation of declarative stated control rules. Based on the discussion of three projects, we will indicate how important it is to define the the proper syntax and semantics of the system implementation.

The tenet of our presentation is the belief that the key to the success of complex projects like distributed control systems is effective communication among the designers. The issues are the semantics of the languages used to describe various design modules and access to the design database. We believe that proper understanding of the issues associated with languages used to describe designs and data-sharing required to validate designs will help in the construction of syntax-driven editors and associated databases. In the future, a domain-related knowledge incorporated in the language itself will help to incorporate the elements of system description into the system to enable it to reason about itself. Thus, the same mechanism that allows the designer to reflect about the design will be perhaps used to help the designed system reflect about its own state. The latter is needed to support self-diagnosis and fault recovery (independently at each node of the distributed system), and they are both necessary in any autonomous (and perhaps intelligent) system.

The paper is partitioned into three parts. The first section illustrates, with an example, how to generate a software environment using a well known attribute-grammar technology. The second section, describes two projects containing a distributed control component element. The last section,summarizes selected issues associated with the language-based approach used to describe control systems.

2. Language-driven support of system design

Designing a system is a process of accommodating an ever-tightening set of constraints. As will be later illustrated by the examples, the choice of syntax and semantics describing the desired system domain plays a very important role in the automatic generation of design-supporting software or implementation mechanisms. In this section, using an example of a hierarchical evaluation of a VLSI circuit, we will show that one can apply compiler technology to generate the design development environment for a physical system [Maitan 87].

First, we will introduce the concept of attribute grammars (AG). Next, we will illustrate how AG can be applied to the analysis of VLSI circuit.

2.1 Attribute Grammars. The methodology presented in this section is applicable to systems that use some form of structure description language(s) and support multiple-views from a single database. It is a subject of ongoing debate as to whether a single language can effectively support a description of a design across all views. In this paper, a hierarchy of languages is used to annotate the semantics of a hierarchy of views or design constraints.

This section presents a methodology consisting of a metalanguage to define a database transformation and its domain, and a method of constraint evaluation based on the analysis or the semantics of a system description. Both design databases and evaluation of design semantics are at the kernel of any design envirnment. This methodology is based on the theory of attribute grammars and can be implemented using existing tools.

The purpose of AG is to map from derivation trees on context-free grammars to corresponding semantic objects. The semantics is described by functions defined over a set of attributes. Each attribute is defined as a type (domain). Describing objects using the syntactic structure of a language builds a network of relations among those objects. Thus, a resulting decorated parse tree not only annotates objects, but also contains semantics of their composition described by the network of constrains. A fixed-point solution of these constraint functions yields a desired meaning of the objects described in a given language.

2.2 Hierarchical analysis of VLSI circuits using AG. Performance-driven synthesis is one of the most challenging tasks in designing commercial VLSI systems. A design is described from a general specification down to the layout level at increasing levels of accuracy. At each level, an exhaustive test of compliance with higher level specifications is made. In this section, a circuit-level model of interconnections and a higher level combinatorial logic model is briefly described. These models are applied later to analysis of the interaction between desi knowledge representation modules.

In terms of VLSI concepts, a simplified delay model derived from extracted electrical parameters is used to describe the semantics of interconnections, and symbolic simulation or a combinatorial circuit using decision graphs carries the semantics of the combinatorial logic level. For each of these models, the corresponding grammar and its attributes are described.

2.3 Summary. A prototype of a system capable of describing and analyzing the semantics of interconnection analysis and gate-level analysis based on an extension of symbolic manipulation has been implemented. Combined together, both layers provided a simple hierarchical timing analysis system with an automatically generated editor (Fig. 1). Semantics of both layers is described using AG. The uniform treatment of constraints enables use of the same evaluator for both of them. In addition to the semantics, a tree-oriented database structure is specified when the AG mechanism is used.

Fig. 1 Two levels of hierarchy used in the prototyped VLSI design environment.

Model	(G1) Circuit-level model	(G2) Combinatorial logic model.
Grammar	**Interconnection ::= port net; net ::= contact bunch; bunch ::= segment I bunch segment; segment ::= section I section contact I section contact port I section contact (bunch);**	**network ::= b_exps; b_exps ::= b_exps b_exp; b_exp ::= arg "=" operation args; args ::= args arg; arg ::= b_exp I name; operation ::+ not I nand I nor I mux;** *interconnection (args @ g1_code) ::: extension to include G1*
Semantics	● extract electrical parameters ● perform elementary delay computations	● symbolically evaluate reachability tree ● annotate tree and propagate constraints to functional level

The major point in assessing the applicability of AG for VLSI CAD systems is the issue of the uniform design knowledge specification environment. AGs have been used in compiler technology to generate compilers for languages and to specify editors to manipulate programs written in these languages. In terms of knowledge systems, it means that from a single declarative specification one can generate a transformation system (compiler), a knowledge representation (compiler tables), and a system to manipulate this representation (editor). An experiment verifying the applicability of this technology for VLSI CAD software development was performed and proved the technology.

It has been shown that:

- using nested hierarchies or knowledge described by formal languages results in a well-defined modularization of the semantics for a simple VLSI CAD problem,
- context-dependent constraints are generated automatically as a result of evaluation of the semantics of objects,
- as a result of using well-defined knowledge models, a run-time model can manage memory better by means of recomputing semantics on demand and saving only a minimal set of information abstracted therefrom.

The approach was limited to noncircular and nonmonotonic semantic functions. The experiment described in this paper demonstrates that knowledge about VLSI semantics can be expressed in the concise form of an AG description. The ultimate benefit of this methodology is a well-documented development environment. Based on a sequence of context-free grammars it is possible to augment such an environment with multiple domain semantics. AG, thereby, constitutes a specification metalanguage.

3. Implementation Issues in Nontraditional System Semantics

This section discusses two examples of declaratively formulated distributed control problems that were implemented using continuous fixed-point computation. The key advantage of such implementation was an asynchronuous implementation of computation schemes. The proposed use of continuous relaxation schemes, since they are not dependent on the use of finite state machines and discrete arithmetics, implies different programming schemes. These two examples were selected to highlight the diversity which exists in the domain of system design and implementation.

3.1 Integrated solutions to the early vision problems.[Maitan 89]. Various promising new algorithms for solving early-vision problems have been developed over the last few years. These are defined as a set of algorithms to recover properties of the visible 3-D surfaces from the 2-D intensity arrays on retinae or cameras, as well as several new concepts in architecture for vision machines. Two examples of the latter are the Connection Machine, conceived by Danny Hillis and build by TMC, and the analog VLSI retina by Carver Mead at Caltech. These two approaches represent the two extremes of programmability, and dedicated hardware. Many more intermediate machine architectures exist.

We proposed a new parallel architecture, Torus Integrated Machine (TIM), which will incorporate physical compactness, dedicated analog hardware, and programmability. The machine is an ideal testbed and implementational vehicle for early-vision algorithms (e.g., edge detection, binocular stereo, motion, structure from motion), since any typical short-range algorithm maps onto our proposed hardware.

3.1.1 Foundations of early vision. The reconstruction of 3-D scenes from their 2-D images is the major purpose of early-vision processing. The task includes: edge detection, optical flow, surface reconstruction, shape from shading, and stereo. All these vision reconstruction problems can be precisely formulated as *ill-posed problems;* that is, they either

- have no solution at all;
- do not have unique solution; or
- do not depend continuously on the initial data.

The technique of *regularization* has been developed to solve ill-posed problems. This regularization method turns a vision-processing problem into a variational problem. Examples of vision algorithms based on regularization are presented in Fig. 2.

Fig. 2 Examples of early vision algorithms.

Problem	**Regularized variational formulations**
Edge detection	$\int [(Sf - i)^2 + \lambda(f_{xx})^2] \, dx$
Optical flow	$\iint [\lambda(u_x{}^2+u_y{}^2+v_x{}^2+v_y{}^2) + (I_x u+I_y v+I_t)^2] \, dx \, dy$
Surface reconstruction	$\iint [(Sf - i)^2 + \lambda(f_{xx}{}^2 + 2f_{xy}{}^2 + f_{yy}{}^2)] dx \, dy$
Stereo	$\int \{[(\nabla^2 G * (L(x, y) - R(x + d(x, y), y))]^2 + \lambda(\nabla d)^2\} \, dx \, dy$

Ill-posed problems can be also characterized as having more unknowns than equations. Regularization provides additional equations needed to solve the problem. Additional constraints are introduced in the form of stabilizing functionals, which restrict possible solutions to smooth functions. After regularization, the vision problem can be compiled into some form of computation structure (Fig. 3).

3.1.2 The outline of the proposed approach. Image processing can be described, using generalized notions of smoothness and "best fit," as a system of ordinary differential equations (ODE) or an approximation problem. For a quadratic variational problem, the resulting computation task can be described as the iterative solution of the matrix equation

$$\mathbf{A}(\mathbf{v}^i) = \mathbf{f}$$

A well-founded problem can be described as a system of *n* equations with *n* unknowns, $\mathbf{F}(\mathbf{v}) = \mathbf{b}$, where $\mathbf{v}, \mathbf{b} \in R^n$. In the case of nonlinear $\mathbf{F}(\mathbf{v})$, the fixed-point solution of the equation can be found by solving its linearized form $\mathbf{A}\mathbf{v} = \mathbf{b}$, where the matrix $\mathbf{A}$ is a linearized $\mathbf{F}(\mathbf{v})$. The iterative solution process generates a sequence whose nth element $\mathbf{v}^n$ is generated by a formula

$$\mathbf{v}^n = \mathbf{v}^{n-1} + \mathbf{H}^n[\mathbf{b} - \mathbf{F}(\mathbf{v}^{n-1})] = \mathbf{v}^{n-1} + \varepsilon$$

where ε is an iteration error of the nth step. The choice of matrix $\mathbf{H}$ determines the type of an iterative technique (Seidel, Jacobi, etc.). If $\mathbf{H}$ is random, as in the analog case, the relaxation process is referred to as *chaotic* or *stochastic relaxation.*

Relaxation processes to solve vision problems are iterative. Two variational methods, Euler-Lagrange and direct solution, can be applied. The basis of both methods is a relaxation solution of an equation $\mathbf{F}(\mathbf{v}) = \mathbf{b}$, where $\mathbf{F}(\mathbf{v})$ is a vector function of $\mathbf{v} \in R^n$, and $\mathbf{b}$ is an independent variable. Many attempts were made to explore the parallelism involved in solving this equation. The limited communication structure of digital computers makes it difficult to implement these processes in parallel fashion. In the case of distributed parallel hardware, the problem is compounded by the need for processing synchronization.

Since relaxation techniques compute the locations of discontinuities in color, depth, or motion, they can be employed to solve one of the basic vision problems, the object *segmentation problem*.

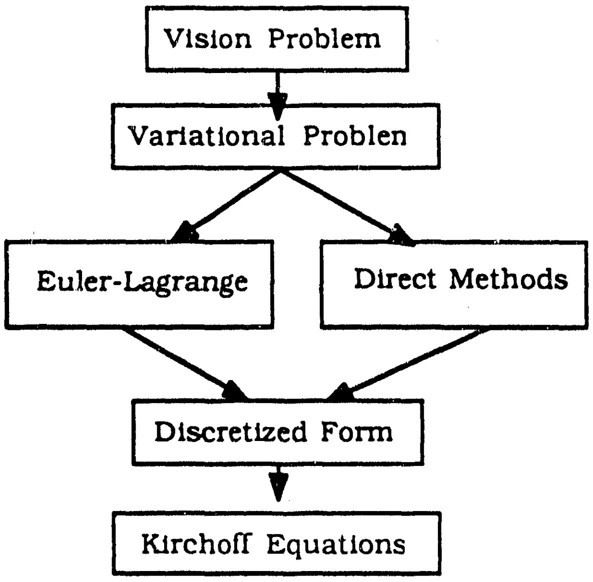


Fig. 3 General flow of the proposed analysis of a vision problem.

The iterative computation of TIM is a relaxation process and is similar to relaxation algorithms used in digital computers. In both TIM and digital architectures, components with the largest error,[1] $\max(\| \mathbf{b} - \mathbf{A}\mathbf{v}^{n-1} \|)$, change until the process converges. In the analog domain, this can be observed as a slow change of all v_i, which satisfies the constraints.

To solve algebraic equations, one can map a discrete problem into a network of analog components. In doing this, we are interested only in steady-state responses. In terms of vision, one can describe each algorithm as a dynamic problem and solve it using analog technology. In many practical applications, there is no need for very accurate solution (5 to 7 bits is satisfactory), and this accuracy matches the accuracy of existing focal planes. Recent advances in reducing the sizes of electronic devices have also scaled down the duration of the transient state, allowing for high-density memories with nsec access time and very high speed processors. The same high-speed devices can be applied to construct high-speed analog solvers of computer vision problems.

3.1.4 TIM - an architecture. The idea of TIM presented in this paper is a technique to organize the space allocated to processing elements of the network. Fig. 4 illustrates the idealized structure of TIM.

To retain a 2-D field of tightly packed photosensors, a cylinder (tube) is formed. Photosensors are on the face of the cylinder, while all other space-consuming elements are shifted to the interior of the cylinder. Instead of using a set of 2-D relaxation fields to implement several concurrent vision algorithms, a sequence of sets of parallel planes is used within a cylinder. Each plane consists of a set of 1-D relaxation processes linked by buses used to share and propagate data between these processes.

Both a local communication along the bases and concurrent 1-D relaxation of several algorithms can be easily implemented on a single plane. This process is repeated on each of sequentially connected sets of planes. Together they are equivalent to a 2-D relaxation decomposed into several axes of relaxation.

[1] Since solving $F(x) = b$ is equivalent to finding an $x = \min(\| F(x) - b \|)$, iterative techniques are comparable to minimization.

Due to the iterative character of the discussed algorithms (relaxation towards fixed-point), the signal must be made reentrant; i.e., the cylinder containing connected groups of planes must be closed, self-feeding, leading finally to the toroidal topology of TIM.

The output from each single column of the 2-D array of the focal plane is directed onto separate planes contained in appropriate processing chips. Increasing the number or/and complexity of the algorithms results in elongating the processing tube - the increase in the number of relaxation planes - while keeping the "retina" size fixed.

3.1.4 Structural assembly of TIM. As described above, each 2-D relaxation can be decomposed into a set of 1-D relaxation processes. Each 1-D process is built out of simple cells. Furthermore, node sharing/coupling among cooperating relaxation processes can be implemented using an analog bus to connect them, as illustrated in Fig. 4. Cells and the connecting buses define the processing pipeline. Each cell is a stage of the pipeline and it cooperates with any other stages to which it has access.

In the simplest case, a cell/stage contains an equivalent of a **single** resistor. More complex algorithms are constructed by linking additional functional units.

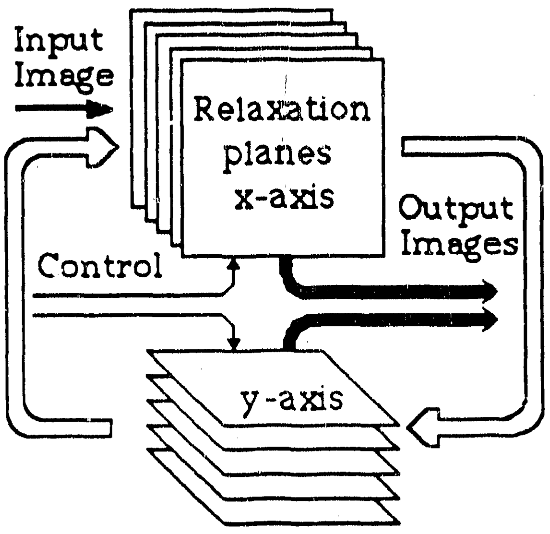


Fig. 4 Simple TIM.

Data on the bus must be multiplexed, so one can reduce the number of pins to transfer data in and out from the processor. The combined structure, a plane, involving the cooperation of several pipelines, is presented in Fig. 5. Each 2-D relaxation field was decomposed into 1-D relaxation processes and is represented here by a single 1-D pipeline stage corresponding to the selected axis of decomposition. In order to interface to the system, the compiler allocates a bus as a variable shared by the external and internal processes. For example, a focal plane, like other input devices, can broadcast data to all relaxation planes through an allocated bus. As a result, data can be processed at high speed and can be used with several cooperating vision algorithms without the need of 3-D VLSI structure.

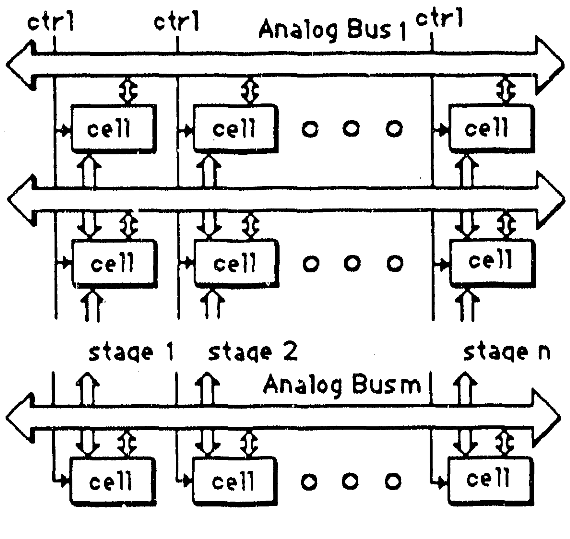


Fig. 5 Connectivity structure in a plane consisting of cooperating 1-D relaxation processes.

TIM is in principle an analog computer. Programing TIM is reduced to interconnecting available functional modules so they can be equivalent to the required algebraic or differential constraints.

There is only one rule for building electric analog circuits, the Kirchoff equations. Thus, one can build circuits based on current or voltage analogy. Discrete forms of variational formulation of vision are built as a sum of components. This can be obtained by chaining functional modules, where each module is an analog equivalent of the required component. The process of translating an algorithm into analog modules can be simply described as:

$$\text{Algorithm} \rightarrow \text{component}_1 + \ldots + \text{component}_m \rightarrow \text{analog-module}_1 + \ldots + \text{analog-module}_n + \textbf{residual}$$

where the number of components is usually different from the number of analog modules, and a residual is a component for which an analog-module cannot be found.

The simplest approach to TIM programming is to define a set of analog modules available within a cell and a technique to connect them into circuits emulating a desired vision algorithm. We propose to use a two-level approach in which

- a digital circuit controls connectivity
- analog circuits perform computation.

TIM's programs are stored as lists of interconnected analog components. Writing a program for TIM is equivalent to reconfiguring its structure. The need for additional interconnections makes programming capabilities strongly dependent on the type of available technology.[2] In many aspects the idea is similar to semicustom, personalized VLSI circuits. In a prototyped structural compiler for TIM, symbolic manipulations needed to derive the discrete structure of Kirhoff equations were performed using Macsyma symbolic manipulation package. The process of TIM programming is simply equivalent to configuring arrays of processing cells and can be performed off-line (Fig. 6).

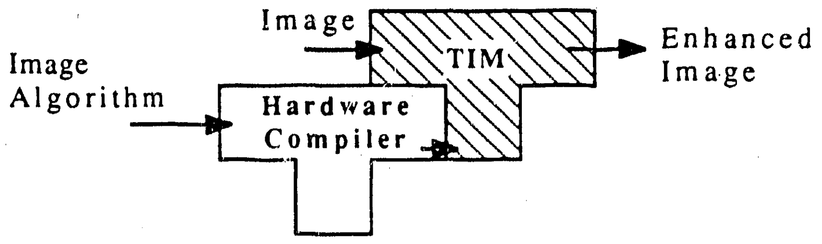


Fig. 6 Simplified structure of TIM's hardware compiler.

There are two scenarios for using TIM as a design platform. In the first scenario, debugged and fine tuned algorithms are packaged into a vision black box. The resulting machine is not programmable, but is simple and small. In the second scenario, the machine features full reconfigurability and requires larger, more complex chips.

3.2 Fixed- point solvers for concurrency control and resource allocation problems [Maitan 90]. We developed an approach for distributed control based on a formulation of the cost function for system state changes. This formulation led to a direct analog implementation of cost-minimization-based constraint solvers. Based on the same energy minimization principle, we have also developed a fast, parallel algorithm for an assignment problem with potential application to distributed control. In addition, we investigated potential application of the technique to a large class of distributed systems -- high-speed fiberoptic networks.

[2] A technology supporting multiple layers of interconnections will substantially reduce the necessary wiring problems for dynamically reconfigurable systems. Hardwired systems, due to the great regularity of interconnecting structures, can be easily built using existing technologies supporting two metal layers plus one polysilicon layers.

We are interested in supporting a concurrent programming methodology based on a sequence of correctness-preserving transformations. In this methodology, a concurrent processing application is implemented starting from high-level specifications. The direct mapping from high-level specifications into high-speed distributed solvers may reduce the number of levels of abstractions into one equivalent level and results in faster execution. This is especially important for implementing advanced languages designed to be executed in high-performance, parallel or distributed computation structures. We explored a link between the language used to code an application and its parallel operational semantics. To demonstrate the possibility of direct high-speed implementation of such primitives, we studied the use of cost or energy formulations to solve the control problem in distributed structures. The results of this research indicate the possibility of building solvers for cost functions as high-speed analog computation structures.

3.2.1 Distributed control primitives. Our model of distributed processing is simply a model in which cooperating processes are bidding for shared resources. (This model was proposed by C. Tomlinson, MCC.) For this type of processing, we construct a simple cost function for each interaction. An example of such interaction involving synchronization and resource allocation can be illustrated by the Dining Philosophers Problem (Dijkstra).

In the simplest Dining Philosophers Problem problem, two philosophers, **a** and **b**, share forks **f1** and **f2**. Each philosopher "bids" for a use of both forks. A philosopher can eat only if he can use two forks.

Each interaction between resources can be represented as a variable. For example, philosopher **b's** interaction with fork **f2** is labelled by b_f2 . Each variable can be either "1" or "0." Forks belong to philosopher **a** if both a_f1 and a_f2 equal "1", and similarly, they belong to philosopher **b** if both b_f1 and b_f2 equal "1," (Fig. 7). We proposed the method to map this problem into a minimization problem in which solutions are either 1's or 0's.

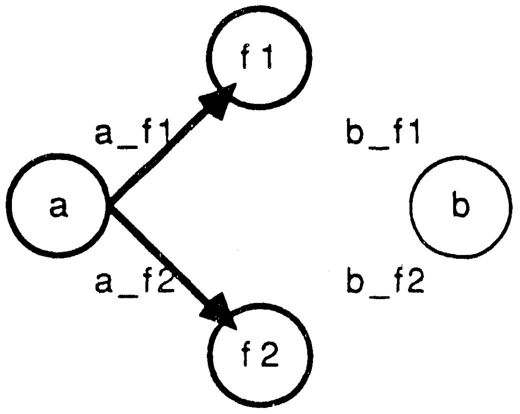


Fig. 7 Simple graph describing two philosophers, a and b, sharing forks f1 and f2.

Simulation shows that after initial perturbation, a stable state is reached and total cost is minimum and equal to zero (Fig. 8). An analog circuit can be built to find the minimum cost; at minimum, forks belong to one philosopher, i.e., a_fi=1 or b_fi=1, i=1,2.

3.2.2 Synchronization and resource allocation in distributed systems and high-speed networks. We chose the problem of control of the computer network as a realistic distributed computation problem. This problem involves the design of both software and hardware. General topology fiber-optic networks use links in which data are propagated at rates of Gbps[Maitan90]. This speed forces the use of a short routing widow (μs per data packet). In the case of several concurrently arriving packets, routing at these rates requires high-speed solutions of resource allocation problems and cannot be made using existing technologies.

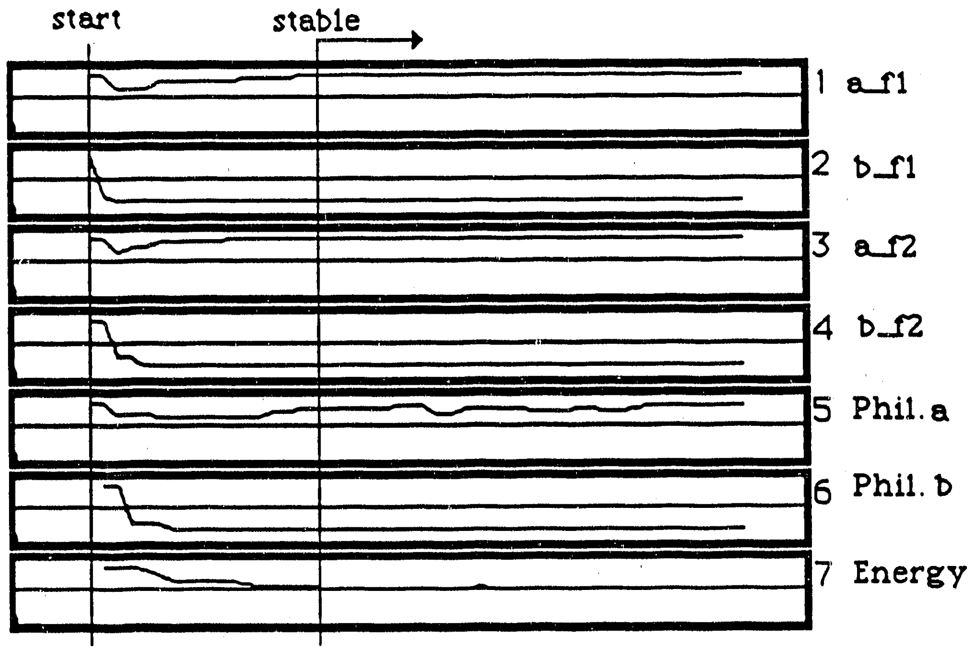


Fig. 8 Simulated cost minimization for a Dining Philosophers Problem.

High-speed fiber-optic networks have been the subject of great interest recently. The technology is relatively new. The existing network topologies are simple and do not utilize the full potential of optical processing. A key obstacle in building large-scale networks is the lack of algorithms for control and management of high-speed networks.

A simple network is built out of interconnected routing nodes. The main purpose of a routing node is to control an exchange of data with other routing nodes. In case of congestion, routing nodes are able to reroute packets using different routing paths.

Each node has two types of information: static information localizing a node in relation to other nodes, and dynamic information estimating the cost of routing to other nodes. Necessary data coercion of cost estimates associated with access to remote nodes is made by each node locally. Internal maps are continuously exchanged between neighborhood nodes using some of the broad-bandwidth subchannels.

Based on this shared local routing information, nodes bid for resource allocation by setting appropriate parameters in the cost functions. Minima of these functions computed by node solvers are the desired allocation of channels. One or more subchannels, at separate nodes, are needed to build path-connecting nodes.

To solve synchronization and resource allocation tasks within such a short time, we will formulate these tasks as cost minimization problems. We propose the use of hardware minima finders. It can be shown that by using the proper mapping, coordinates of the minima of cost function correspond to the desired network control rule. In order to find minima of cost function, one can apply a relaxation process. This process can be based on continuous analog signals.

4. Conclusions

In this paper we attempted to identify the approach(es) leading to the design of an environment for the development of software for distributed intelligent systems. In searching for solutions, we evaluated three simple projects in the areas of software environments and distributed systems. We chose projects which extend beyond the traditional real-time processing domain to include the approaches which may be followed in the future. In particular, we emphasize the use of declarative, language-based definitions of control rules and other system constraints. Our interest in these technologies is dictated by the emerging analog computation or "neural network" schemes characterized by the use of asynchronous and declaratively defined constraint solvers. Adaptive capabilities of these approaches expand the notion of intelligence in large scale distributed systems.

From the design and maintenance point of view, two major topics of interest in the analysis and management of distributed systems are:

- computation on a structured representation of knowledge,
- structure of knowledge evaluation.

Issues associated with computation guide the selection of architecture attributes such as data flow, distribution of databases, fault-recovery etc. Whereas a good understanding of the structure of knowledge evaluation controls the choice of implementation mechanisms (analog vs. digital, synchronous vs. asynchronous). Both topics complement each other and must be simultaneously evaluated to achieve an optimal (if exist) solution to the required computation problem.

Based on the presented analysis, the following are the selected findings that are relevant to the design of software environments for distributed intelligent systems:

1. **Simplicity.** Complex behavior can emerge from the interaction of relatively simple components.
2. **Flexibility.** The decription of design knowledge must be modularized. A single mechanism to evaluate and maintain its consistency must be developed.
3. **Adaptability.** Functional and declarative specification of relations may allow for design portability from one technology to the other. Many context-dependent constraints should not be entered by a designer. The user must be able to control the analysis of mixed models (analytic and simulation forms) of complex system.
4. **User friendliness.** Integrated distributed control+analog+digital systems are hard to debug and test. New techniques must be developed to specify these systems.
5. **Traceability.** Adaptive knowledge required in intelligent distributed systems is hard to document, and assessing completeness of the knowledge (with regard to a given problem) is almost impossible.

Bibliography

Extended bibliography related to the subjects described in this paper can be found in:

[Maitan90] Maitan, J., L. Walichiewicz, and B. Wealand: "Integrated Communication and Information Fabric for Space Applications," paper accepted for AIAA/NASA Second International Symposium on Space Information Systems, Pasadena, CA, September, 1990.

[Maitan 89] Maitan, J., "TIM: A Compact Architecture for Real-Time Vision System," SPIE Proceedings *Advances in Image Compression and Automatic Target Recognition*, March 1989.

[Maitan87] Maitan, J., "Syntax-Driven Management of Constraints in VLSI Data Bases," MCC, Technical Report, PP-195-87, 1987.

A Hardware/Software Environment to Support R&D in Intelligent Machines and Mobile Robotic Systems*

Reinhold C. Mann
Center for Engineering Systems Advanced Research
Oak Ridge National Laboratory
P.O.Box 2008
Oak Ridge, TN 37831-6364
mnn@ornl.gov

Abstract

The Center for Engineering Systems Advanced Research (CESAR) serves as a focal point at the Oak Ridge National Laboratory (ORNL) for basic and applied research in intelligent machines. R&D at CESAR addresses issues related to autonomous systems, unstructured (i.e. incompletely known) operational environments, and multiple performing agents. Two mobile robot prototypes (HERMIES-IIB and HERMIES-III) are being used to test new developments in several robot component technologies.

This paper briefly introduces the computing environment at CESAR which includes three hypercube concurrent computers (two on-board the mobile robots), a graphics workstation, VAX, and multiple VME-based systems (several on-board the mobile robots). The current software environment at CESAR is intended to satisfy several goals, e.g.: code portability, re-usability in different experimental scenarios, modularity, concurrent computer hardware transparent to applications programmer, future support for multiple mobile robots, support human-machine interface modules, and support for integration of software from other, geographically disparate laboratories with different hardware set-ups.

Introduction

CESAR at ORNL focuses its research on the development and experimental validation of intelligent control techniques for autonomous mobile robots able to plan and perform a variety of tasks in unstructured environments. The purpose of this paper is to provide a brief description of the hardware and software environment at CESAR which has been evolving in order to support research in several robot component technologies. The material presented in this paper is excerpted from several reports and articles published previously by CESAR staff. Selected references are given in this paper. A full CESAR publication list if available from the author upon request.

Assignments for the robot(s) originate with the human supervisors in a remote control station, and the robot then performs detailed implementation

* Research sponsored by the Engineering Research Program of the Office of Basic Energy Sciences and by the Office of Nuclear Energy of the Department of Energy, under contract No. DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc.

planning and executes the tasks. Since the operational environment is generally dynamic, the robot must be in sensory contact with its surroundings to capture and recognize changes which bear on its task objectives and , if necessary, replan its behavior. These capabilities imply that the robot has cognitive capabilities that enable it to form and modify a model of the world around it and relate this world model to the task objectives. Research is also conducted to enable the robot to learn from its past experience, and thus improve its performance.

CESAR's principle current objectives are: (a) to achieve a level of technology that enables the autonomous performance of classes of navigation and manipulation tasks of human scale in a spatially complex environment; (b) to use these performance tasks to focus research objectives. Application drivers for this basic research effort include, among others, robotics for advanced nuclear power stations, and environmental restoration and waste management activities.

CESAR is developing a series of mobile autonomous robot vehicles named HERMIES (Hostile Environment Robotic Machine Intelligence Experiment Series) as experimental testbeds for validation and demonstration of research results. The newest research robot, HERMIES-III, includes the functional capabilities that permit research in combined mobility/manipulation, and allows us to experiment with cooperative control of multiple robots having different capabilities.

Hardware Environment

HERMIES-IIB and HERMIES-III are the currently operational mobile robots at ORNL/CESAR. HERMIES-IIB stands 1m high and weighs 91kg. Rechargeable batteries supply 20W of power providing about 20 minutes of untethered running time. Peak movement speed is 0.7m/s. Sensors include four Sony CCD cameras and a number of Polaroid sonar transceivers mounted on a rotatable turret. The computer architecture consists of a VME rack housing a Motorola 68020 CPU and a variety of I/O boards interfaced via a BIT-3 communication link to an NCUBE (NCUBE, Inc., Beaverton, OR) hypercube computer. The hypercube consists of 16 nodes with 512 Kbytes RAM each and an Intel 80286 I/O processor, which also serves as host for the hypercube. Each node processor is a 32 bit microcomputer with on-chip floating point and communications hardware. This gives HERMIES-IIB roughly 16 MIPS in the on-board hypercube. HERMIES-IIB is equipped with two Zenith/Heathkit five degress-of-freedom arms which give the robot extremely limited manipulative capability. This has not been a drawback, however, since the robot was not intended for research in manipulation.

The HERMIES-III mobile robot consists of an omni-directional wheel-driven chassis, a seven degrees-of-freedom manipulator (CESARm), an Odetics laser range camera, multiple CCD cameras (two stereo pan and tilt mechanisms), an array of sonar transceivers, and an on-board computer system that includes five Motorola 68020 CPUs in four VME racks, and an NCUBE hypercube concurrent computer. CESARm is a compliant, high capacity-to-weight ratio (~ 1/10) robot manipulator, with an adjustable gripper, which is equipped with a JR^3 force-torque sensor, and a LORD tactile sensor pad.

Both robots can be operated completely autonomously, in which case they can communicate via RS-232 wireless modems to an off-board computer. They can also be interfaced through ethernet to a local are network of computers, as schematically shown in Figure 1. This network includes a Silicon Graphics IRIS 4D workstation, a microvax, and an NCUBE hypercube computer with 64 processors.

Software Environment

The computer programs that control HERMIES-IIB's behavior are mostly written in C and can be organized into four classes: the HERMIES primitives (i.e., functions that directly control platform motion, activate sensors, etc.), the expert system and associated routines for navigation and multi-sensor integration (error propagation and conflict resolution), the image analysis routines (a complete library that makes the concurrent hypercube hardware transparent), and the control and integration routines. An expert system may be executed from the NCUBE host processor; however, all of the image analysis routines and control and integration programs have been developed for execution on the NCUBE concurrent computer. A computer program that emulates the response of HERMIES-IIB is used for off-board development of the expert system rule base prior to implementation on the robot.

The expert system makes high level decisions and diagnoses unexpected occurrences. When a standard procedure is required, such as avoiding or removing an obstacle, or mapping an area, the expert system calls the appropriate routine which executes until completed or until an unexpected event generates an interrupt which returns control to the expert system. The rule base controls high level decisions and can call C-compiled navigation procedures. The rule base is loaded in an expert system shell, CLIPS, and linked to the navigation procedures. CLIPS and the navigation code run on one of the NCUBE nodes. Messages are passed from the NCUBE node to a host program which is linked to the HERMIES-IIB primitives on the VME rack.

Part of the CESAR effort is aimed at addressing crucial issues in systems integration so that research results can be integrated into the HERMIES prototypes. Recent experiments also included modules developed by groups at four university laboratories (Florida, Michigan, Tennessee, Texas) as part of a collaborative techbase development effort for which the HERMIES robots serve as user facilities. Detailed accounts of experiments with HERMIES-IIB can be found in the references.

The following material summarizes software development strategies in support of the latest experiment in which HERMIES-III was used to clean up a simulated chemical spill in the CESAR laboratory. The demonstration featured the capability to make smooth transitions between tele-operation and robot autonomy, the reconciliation of information in an a priori world model with information derived from sonars and CCD cameras, and the combined use of platform and manipulator degrees of freedom.

It is assumed that an a priori model of the environment surrounding the spill is known. The system uses this knowledge to create a path from the robot's current location to a location close to the spill. The robot then navigates to the spill, automatically avoiding unexpected obstacles en route. Once it has arrived, it senses the debris and uses a vacuum cleaner mounted on a manipulator to remove it. This process iterates until the sensing process can find no more debris. There are three main subtasks: path planning, path execution, and debris removal. An additional subtask permits operator intervention with the autonomous system. In every task, the operator is provided with a rich graphical description of the current state of the robot.

Software to operate and control the HERMIES-III robot in various experimental scenarios was developed around a simulated shared memory data structure. The shared memory model of interprocess communication was adopted because of its conceptual simplicity. This design decision made communication between various groups involved in the implementation effort relatively easy -- it was necessary only to define the format and the interpretation of the data structures written by each process without having to describe mechanisms by which these structures were communicated. Communications were assumed to be transparent by the authors of each module.

Some structure was imposed on the shared memory in addition to a simple list of variable names, types, and locations. Specifically, shared memory was divided into a number of blocks of contiguous memory, with one or more blocks associated with individual processes. The shared memory model is not without its problems. Perhaps the most obvious is that two processes may try to write to the same data item. In this situation, either of the processes may be correct depending on the state of the system or the time. Other problems include synchronization between processes and processes attempting to read a variable whose value has been only partially determined (e.g. only 4 bytes of an 8 byte record have been written). We avoided the first problem by specifying the system so that each process "owned" an area of shared memory to which only it could write. We solved the second problem by implementing a simple semaphore mechanism which guarded each area during updates.

Processes in this system communicate by accessing the shared memory, which is a replicated distributed data structure divided into exclusive-write areas (EWA). Each process making entries into the shared memory has associated with it one or more EWAs which only that process should change. In the event that multiple processes determine the values of single variables at different times, the "official" value is determined by a filtering process. Each EWA is a contiguous sequence of bytes. Shared memory is allocated by a special allocation process and is permanently memory resident. It has no internal structure at allocation time, rather, structure is imposed upon it at compile time through the use of compiler definitions. Addresses become available at run-time, through a call to the mem_attach() routine. Structure definitions and the relevant function prototypes are available by using included definitions.

The entire system is controlled by a single "state variable", and there is one (and only one) process in the system which determines the value of this variable. Decisions on the change from state to state are made by this process based on the current value of the state variable, and a state-dependent inspection of the contents of (possibly many) shared data areas. Individual processes inspect the value of this variable and respond in an appropriate manner.

Conclusions

Research and development at ORNL/CESAR centers on autonomous systems, unstructured environments, and multiple performing agents. A number of projects make use of the HERMIES mobile robot facilities at CESAR, and provide application focus for the R&D activities. Hardware and software environments have evolved to support the experimental part of the research. They facilitate software portability among systems, and re-use of applications software in different experimental scenarios. Message-passing concurrent computers have been incorporated successfully in our systems. Recent experiments show that the robot systems can perform robustly a variety of tasks of considerable complexity.

Selected References

C. R. Weisbin, "Intelligent Machine Research at CESAR", AI Magazine 4, Spring 1987, Vol. 8, No.1, (1987)

B. L. Burks et al., "Autonomous Navigation, Exploration, and Recognition", IEEE Expert, Winter 1987, pp 18-27, (1987)

C. R. Weisbin et al., "HERMIES-III: A Step Toward Autonomous Mobility, Manipulation and Perception", Robotica, Vol 8, pp 7-12, (1990)

J. P. Jones, "A Concurrent On-Board Vision System for a Mobile Robot", Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Pasadena, CA, (1988), CESAR-88/06

J. P. Jones, "On the Design of a Concurrent Image Analysis System", Proceedings of the Third International Symposium on Robotics and Manufacturing, ISRAM '90, Vancouver, Canada, 1990

F. J. Sweeney et al., "DOE/NE University Program in Robotics for Advanced Reactors", Program Plan, DOE/OR-884 Rev.2, May 1990

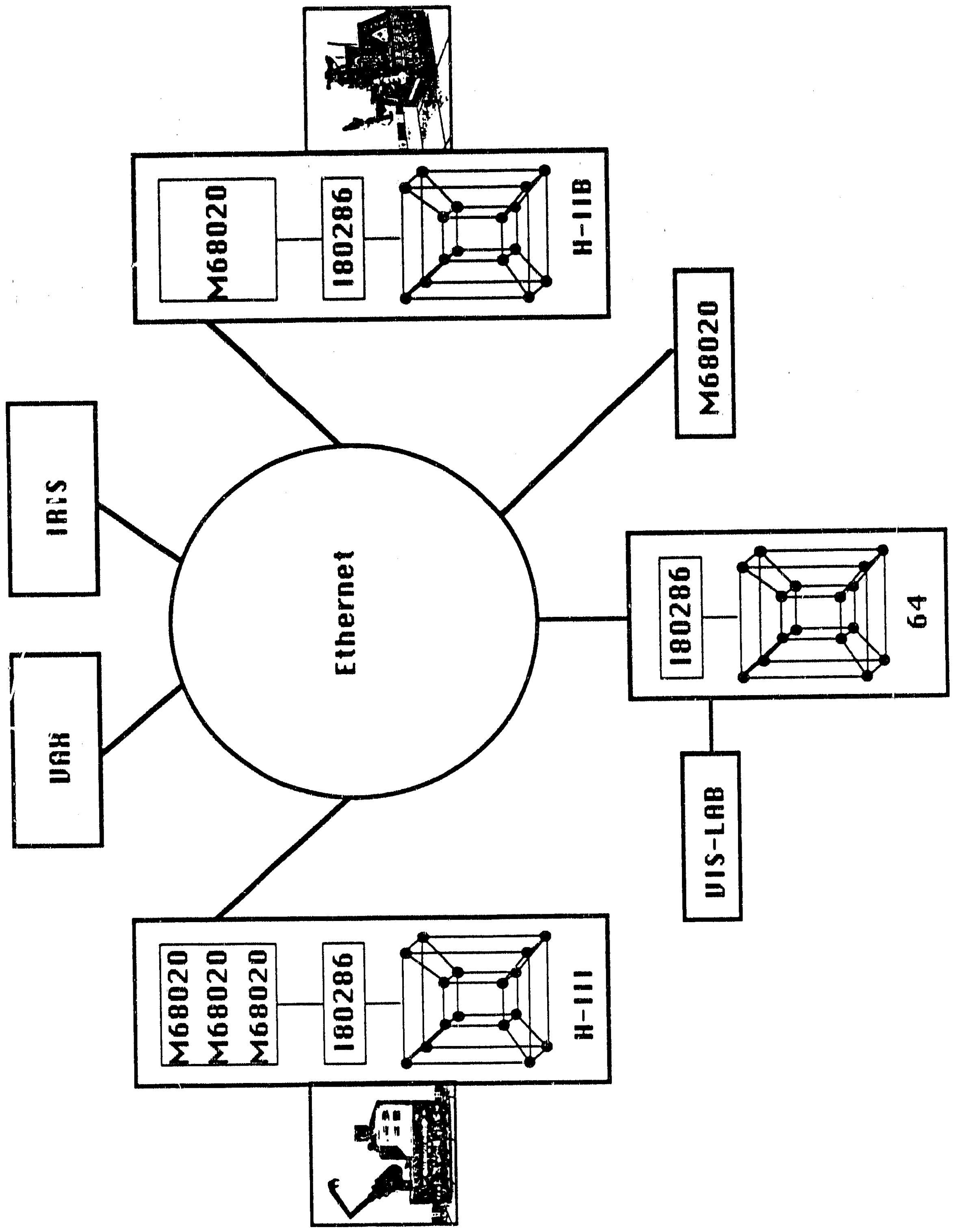


Figure 1: The CESAR robot/computer network.

Requirements for Intelligent Real-Time Control Systems

Swaminathan Natarajan and John Yen
Department of Computer Science, Texas A&M University
College Station, TX 77843-3112

Abstract

We first present some requirements which must be met by intelligent real-time control systems. A crucial aspect of these systems is that they must address the issue of resource constraints. Therefore, a tool for designing and implementing such systems must enable designers to obtain and express resource information, and provide a set of control strategies that the designer may specify for resource allocation. One such strategy, resource-based selectivity, is based on the concept of imprecise computation, which has been developed in the real-time community. This strategy can adapt the resource needs of computations by selecting and using subsets of the inputs, problem solving strategies, or outputs, thus producing imprecise results if necessary. Our work examines the application of this technique in the context of rule-based systems, and the development of a shell for building rule-based real-time control systems. Research issues to be addressed in order to build such a shell include acquisition and expression of resource information, development of a software architecture to support resource-based selectivity, and acceptability criteria for validating results obtained.

1 Introduction

This paper aims to set out some of the requirements for intelligent real-time control systems, present some techniques for meeting these requirements, and identify issues where further research is needed to solve the problems. Intelligent real-time control systems acquire input data from sensors that monitor an external environment, perform complex computations, and produce control commands that change the environment. These systems are *intelligent* because they need to perform complex reasoning to determine the appropriate actions. They must also cope with computational resource constraints and produce timely responses, which makes them *real-time* applications.

We illustrate our discussion of the requirements for intelligent control using an example of a robot which picks up objects off a conveyor belt and moves them into bins. Each object must be categorized using size and shape information (provided by the vision system of the robot) and placed in the bin for that category. Objects not picked up in time fall off the end of the conveyor belt. Some types of objects are fragile and sustain damage if they fall off. The objects may be unevenly spaced on the belt, and several may be visible at any given time. The robot can therefore concurrently perform several tasks, such as computing the commands needed to pick up the nearest object and categorizing some of the other objects visible on the belt.

2 Requirements for Intelligent Real-Time Control Systems

The need for interaction with and timely control of multiple ongoing processes imposes several requirements on the algorithms and techniques for the design and implementation of intelligent real-time control systems, as we discuss below:

1. **Timeliness:** Since the external process is ongoing, *failures* may occur if appropriate control commands are not received in time. In our example, objects may fall off and possibly be damaged. Thus the system has *deadlines* which must be met for correct operation. In our example, the deadline is dictated by the distance between the object and the end of the conveyor belt. Deadlines are the most important of the various constraints on resource availability.

2. **Resource Predictability:** Once the resource availability is known, ensuring timely responses requires knowledge of the resources needed to generate the response. In our example, in order to plan our activities, we must have some knowledge of the time needed for each step: recognizing objects, picking them up, and moving them to their bin. Therefore, when selecting algorithms for performing tasks, we prefer those whose resource needs are more predictable.

3. **Flexibility:** In situations where resource availability varies from one execution of a task to the next, it is necessary to adapt the resource requirements to match the availability. In our example, the uneven spacing of objects implies that the time available for recognizing and moving objects varies. Thus we need problem solving techniques which are flexible in the amount of resources consumed to produce a response. Typically, the only parameter against which we can make tradeoffs to achieve this is the result quality, hence we may need to employ techniques such as *anytime algorithms* [3] which make this tradeoff. In our example, there may be a "miscellaneous" bin into which the robot places objects if it cannot recognize them before they must be picked up.

4. **Robustness:** Sometimes, in a system with multiple tasks, we may encounter situations where the available resources are not sufficient to generate appropriate responses for all tasks. Under these *overload* conditions, we would still like our system to degrade gracefully, and at least provide responses to the most important tasks. In the example, we must ensure that the fragile objects at least are recognized and picked up in time. Robustness thus includes a notion of the *criticality* of requests, i.e. the importance of generating an acceptable response. The techniques we use for scheduling and resource allocation must take criticality information and the robustness requirement into account.

5. **Focus of attention:** When a system controls several processes, it may have several tasks to perform at any given time. Rather than divide its resources equally among all tasks, it may choose to devote most of the resources to some subset of the tasks, complete them and move on to the others. Thus there is a notion of the current focus of attention, which shifts as responses are generated and new requests arrive. In our example, the robot may

devote most resources to picking up the nearest object and to recognizing the next one, and less to recognizing objects further away. Criticality is a factor also in determining the focus of attention, e.g. the robot may devote more resources to the task of picking up a fragile object, even if another object is closer. This selectivity in resource allocation is necessary to ensure that timely responses are generated for the tasks with the closest deadlines.

6. **Responsivity:** When emergencies occur, they must be identified and responded to as quickly as possible. For instance, if an object leaves the surface of the belt, the robot may have to move away quickly to avoid damage to itself. This requires the ability to modify predetermined schedules and resource allocations and devote most or all of the resources to the new emergency tasks.

7. **Asynchrony:** Sometimes, some additional inputs may be received which affect the response from some computation in progress. For instance, as an object comes closer, the vision system may realize that its shape is different from what had been perceived earlier. If categorization is already in progress, it is desirable to have the ability to consider this new information without having to start over. This actually requires that we develop a mechanism for interruptible reasoning, hence it is a very complex research problem.

8. **Coherence:** Not only must the system generate responses to individual requests, but it must also ensure that the overall pattern of responses fit some criteria. In our example, if the objects are spaced very close together, we may find that we never have quite time enough to complete recognition of any of the objects before we must pick them up, hence we end up moving them all into the miscellaneous bin! To avoid such undesirable results, the scheme used for resource allocation must have some knowledge of the type of responses being generated, so that it may fit these into an overall plan of action.

9. **Performance:** Last but not least, performance is important to any real-time application. In this context, the performance criterion translates to generating the best possible quality of result given the available resources.

Further discussion of several of these requirements can be found in [7, 2].

It is interesting to notice that most of these requirements are imposed by the resource constraints, and impact resource allocation and scheduling. This indicates that resource considerations should be an integral part of any approach to addressing these requirements. In the rest of this paper, we propose an approach that enables response generation to be adapted explicitly to resource constraints. We present our approach in the context of rule-based expert systems, but we believe that the underlying ideas are applicable to a variety of intelligent real-system applications.

3 An Architecture for Real-time Rule-based Systems

3.1 Overview and Rationale

The flexibility requirement identified above involves adapting the resource requirement of the computation to match the availability. Our approach of ***resource-based selectivity*** enables the specification of several alternative ways to generate responses which vary in quality and resource needs. A particular method to generate the response is selected based on the resource constraints. This approach is based on the techniques of ***imprecise computation*** [4, 6] developed in the real-time community, which provide two ways to generate approximate, partial results from computations when they cannot be completed in time. The ***reactive*** technique of imprecise computation is similar to the notion of anytime algorithms: periodically the computation explicitly generates and saves partial results; if the deadline is reached before the computation completes, the latest partial result is returned as an approximate result from the computation. The ***predictive*** technique estimates the resource requirements of computational steps, and if the available resources are insufficient, opts to skip some steps (previously identified as optional) so that the rest of the computation may complete in time; the quality of the result is reduced by the degree that the skipped steps contribute to it.

Our approach enables the application of these techniques to provide flexibility in real-time rule-based systems. We extend the production system architecture to incorporate a task-oriented reasoning model. The tasks and subtasks serve as units for resource allocation and selection of alternative methods of generating responses. We describe a hierarchical resource allocation technique which enables adapting the resource needs to the availability, and optimizing the quality of the overall result. The architecture also contains several control modules to facilitate the application of selection strategies to several of its components and provide better control over resource needs and response quality. The proposed architecture, shown in Figure 1, is actually an extension of an existing architecture called CLASP [9].

3.2 A Task-oriented Reasoning Model

A task-oriented reasoning model structures problem solving knowledge based on the notion of *tasks*, which are generic functions that the system can perform. The proposed architecture distinguishes two types of rules: task-triggering rules and task-accomplishment rules. Task-triggering rules detect situations that require top-level tasks, while task-accomplishment rules describe various ways to achieve a given task. For the convenience of our discussion, we will refer to the former rules as *productions*, the latter rules as *methods*. Using methods, a complicated task can be decomposed into subtasks. Methods of a task may vary in their applicability, resource requirements, and result quality. Top-level tasks are posted to an *agenda* by production rules. Selecting a task from the agenda causes certain methods associated with the task to be selected and executed. The execution of a method may post additional subtasks to the agenda.

The notion of task has been used in several expert systems as a means to *focus* the attention

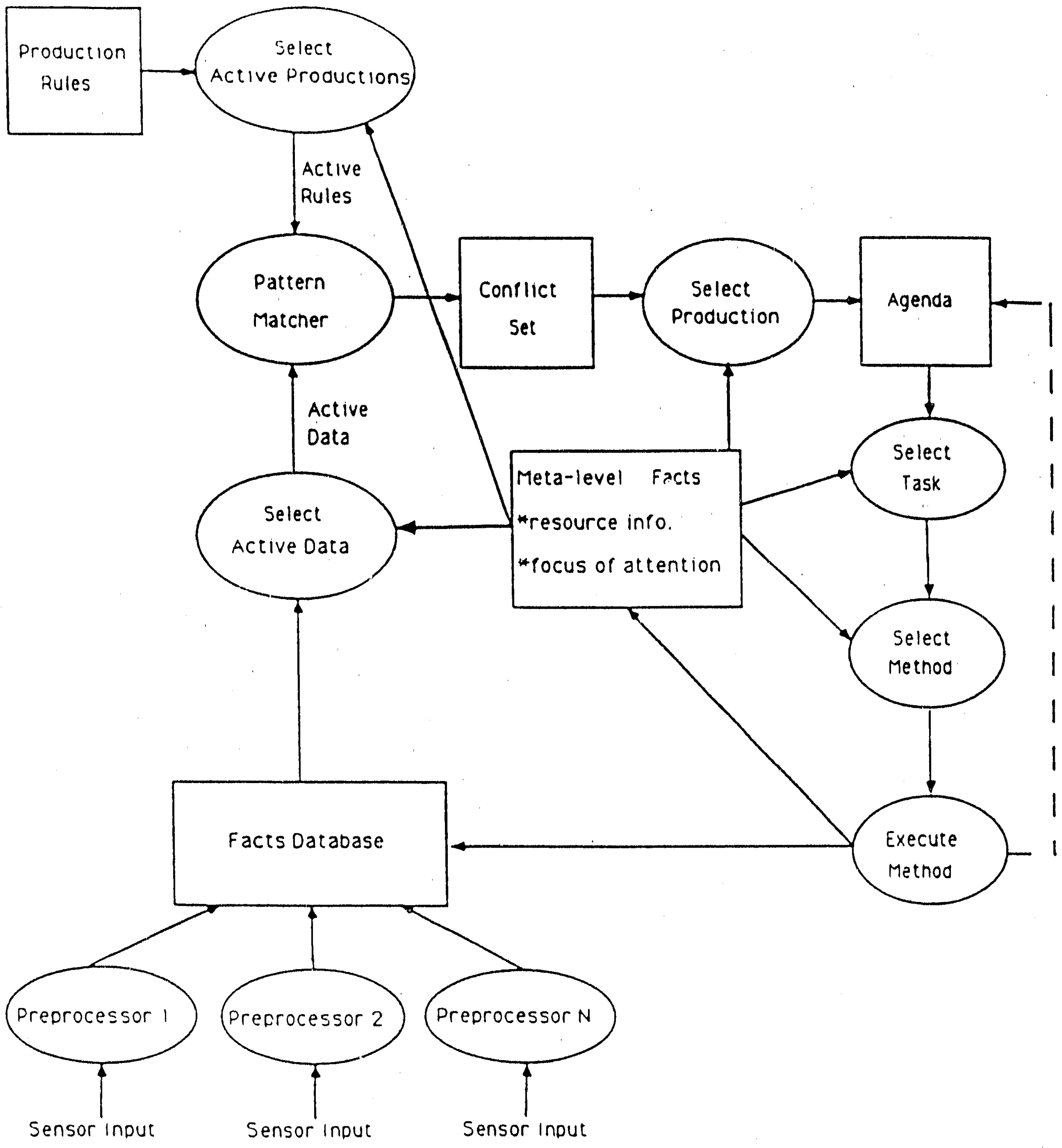


Figure 1: A Real-time AI Architecture that Supports Imprecise Computation

of the system to a subset of rules [8, 5, 1]. Here, we further extend their notion of tasks into the notion of a basic computation unit that exhibits imprecise computation behavior. Tasks and methods together form a task/subtask hierarchy where the children of a task are methods, and the children of a method are its subtasks. The terminal nodes of the hierarchy are executable methods.

The model enables the architecture to reason about tradeoffs between resource requirements and result qualities among various problem solving strategies, for it improves the availability of partial solutions, facilitates the estimation of resource requirements, and captures the inter-dependency of rules.

The task-oriented reasoning model improves the availability of partial solutions in two ways. First, solutions of varying quality can be generated by various methods of a task. Second, by explicitly representing the criticality of subtasks, the system can generate partial solutions by skipping non-critical subtasks.

Resource requirements in a task-oriented model can be propagated bottom up in a task/subtask hierarchy. First, we assume that resource requirements of executable methods can be obtained. Resource requirements of a non-executable method is just the sum of the resource requirements of its subtasks. Resource requirements of a task is an interval $[minR\ , maxR]$ where $minR$ and $maxR$ are the minimum and maximum resources, respectively, required by its methods. In general, resource requirements can be represented either numerically or qualitatively (e.g., small, medium, large, very large, etc). Qualitative description is useful whenever exact numeric resource requirement is not available.

Inter-dependency of rules is explicitly captured in task-oriented reasoning: methods of a given task are highly independent, while subtasks of a method usually are highly dependent. From a logic programming point of view, methods of a task are connected through disjuncts; while subtasks of a method are connected through conjuncts. Thus, the reactive imprecision technique usually applies to methods, but not subtasks.

3.2.1 A Hierarchical Resource Allocation Technique

Resource is allocated to tasks and methods in a top-down fashion. When a top-level task is created, the system allocates some amount of resources (e.g., the deadline for a moving robot to react to avoid hitting an obstacle) to the task. These resources are divided and allocated to applicable methods of the task, which further divide and allocate the resources to subtasks of the methods. When resources are finally allocated to an executable method (i.e., a terminal node), the method is executed. The execution can be interrupted when the allocated resources are used up.

At each level of the task/subtask hierarchy, resources can be allocated to child nodes using the predictive, reactive, or a hybrid imprecision technique. The choice of the technique often depends on the accuracy of the resource requirement estimates. For instance, using a pure predictive approach to allocate time among subtasks is advisable only if we have accurate knowledge of the resource requirements of each subtask. If we use the predictive approach to allocate time based on underestimated measures, the system may not generate any partial solutions within the deadline, even though the resource available is sufficient for a reactive approach to produce some acceptable solutions. To avoid this contingency, it is often preferable to first use the reactive approach to obtain some minimal acceptable result which can serve as a fallback, then use the predictive approach to generate the best possible result with the resources available. We call this combination of the reactive and predictive approaches a *hybrid* approach.

The choice of imprecise technique used at each levels of the hierarchical resource allocation involves a tradeoff between the timeliness and the quality of the solution. In the case that there is uncertainty about resource requirements, the tradeoff can also be viewed as between the probability of generating an acceptable solution and the quality of the solution. Reactive or hybrid approaches guarantees the generation of an acceptable solution at an earliest possible time; however, they tend to increase the total resource required for generating a high quality solution. This is due to the fact that not only there is an overhead for doing resource allocation, but also the resources spent on producing an approximate result are wasted if subsequent computation does run to completion and produces a better result. Whether we choose to incur this overhead depends on several factors: the cost relative to the total time available for the subtask, the criticality of the subtask, and the uncertainty about the estimate of the subtask's resource requirement.

3.3 Other Features of the Architecture

The architecture contains five control modules for active rule selection, active data selection, selecting rules from conflict set, selecting tasks from agenda, and selecting methods for a task. Active data/rules are data/rules that the system attends to. Only active data and active rules are considered during the pattern matching phase. The selection of tasks and the selection of methods are separated to facilitate the application of different imprecision techniques to the two control problems.

Information about resource requirements and availability is stored in a global meta-level data base, which is accessible to all control modules. These meta-level information can be updated by rules as well as by system calls (e.g., clocks). Each control module can consider the resource information, perform some selection function and update the resource requirement information in the data base accordingly. Thus the different control modules operate together at various granularities to adapt the resource requirements to the availability.

The architecture is general in that all control modules could modify its control strategy using information about resources. A particular real-time AI application, however, may only need to consider resource information using a subset of these control modules.

4 Discussion

This architecture addresses the requirements identified earlier as follows:

1. Timeliness: The architecture interrupts the execution of methods when their resources are used up. A partial solution of the task is generated either using other successfully executed methods or using the interrupted method.
2. Predictability: The use of hierarchical resource allocation ensures that the resource consumption of tasks is bounded.
3. Flexibility: The predictive and reactive imprecision techniques are used to adapt the result quality and resource needs to the availability.
4. Robustness: Criticality information is incorporated into task selection and resource allocation strategies to ensure graceful degradation under overload.
5. Focus of attention: Selective data activation and rule activation enable the system to focus on data and rules that are relevant to the current top-level task.
6. Responsivity: In an emergency, resources can be withdrawn from other tasks by updating the meta-level data base and using the reactive technique to generate partial results from current tasks. The task responding to the emergency can adapt its computation to the resource constraints to provide a quick response.
7. Asynchrony: This is a research issue yet to be addressed in the current architecture.
8. Performance: Rule and data activation can be used to reduce resource requirements for pattern matching, particularly when in conjunction with focus of attention. The techniques of imprecise computation are qualitative and hence involve relatively low overhead.

There are several research issues which must be addressed in this approach:

1. Acceptability Criteria: The reactive approach can generate partial results. However, whether these partial results are useful depends on the particular application and current situation. It is necessary to define some criteria that indicate whether a particular imprecise result is acceptable. These would have the effect of restricting the methods which may be selected.
2. Interruptibility: Asynchronous events can modify the inputs to the pattern matcher during its operation. Techniques need to be developed to avoid having to discard the partial results of the pattern matching process.
3. Obtaining resource information: Primitives and tools to specify resource requirements of rules and methods need to be developed.

References

[1] J. S. Aikins. Prototypes and production rules: A knowledge representation for computer consultations. Technical Report STAN-CS-80-814, Department of Computer Science, Stanford University, 1980.

[2] P. R. Cohen, A. E. Howe, and D. M. Hart. Intelligent real-time problem solving: Issues and examples. In *Intelligent Rcal-Time Problem Solving: Workshop Report.* Cimflex Teknowledge Corp., January 1990.

[3] Thomas Dean. An analysis of time-dependent planning. In *Proceedings of AAAI-88*, pages 49–54, 1988.

[4] K. J. Lin, S. Natarajan, and J.W.S. Liu. Imprecise results: Utilizing partial computations in real-time systems. In *Proceedings of the 8th Real-Time Systems Symposium*, pages 210–217, San Jose, CA, 1987.

[5] John McDermott. R1: A rule-based configurer of computer systems. *Artificial Intelligence*, 19(1):39–88, 1982.

[6] Swaminathan Natarajan. *Building flexible real-time systems.* PhD thesis, Univ. of Illinois, Urbana, Department of Computer Science, 1990.

[7] S. J. Rosenschein, M. Fehling, M. Ginsberg, E. Horvitz, and B. D'Ambrosio. Irtps workshop interim team report. In *Intelligent Real-Time Problem Solving: Workshop Report.* Cimflex Teknowledge Corp., January 1990.

[8] Edward H. Shortliffe. *Computer-Based Medical Consultation: MYCIN.* American Elsevier, 1976.

[9] John Yen, Robert Neches, and Robert MacGregor. Using terminological models to enhance the rule-based paradigm. In *Proceedings of the Second International Symposium on Artificial Intelligence*, Monterrey, Mexico, October 25-27 1989.

Modelling Intelligent Control

Anil Nerode
Director
Mathematical Sciences Institute
An Army Research Office Center of Excellence
Cornell University, Ithaca, NY 14853
e–mail: nerode@mssun7.msi.cornell.edu

Introduction

My point of view is conditioned by thirty–six years of academic research in mathematical logic and computer science, and thirty–six years of quite separate consultancies for design and evaluation of military systems for many agencies. These interests seemed quite separate until about 1980, when expert systems, AI, and intelligent control of military physical systems came together as a practical combination. My services for combined problems involving both areas at once suddenly became popular. When I started research in logic and computer science, and separately in weapons systems, thirty–six years ago, who would have guessed that they would overlap and then come together? It is an amazing period.

A General Concern

I want to begin by mentioning a general problem beyond the scope of this conference. Despite claims to the contrary, present and developing software tools are woefully inadequate for specifying, designing, writing, verifying, documenting, and maintaining software using distributed and concurrent computing which meets prespecified system performance requirements. This is not the fault of programmers or software architects. They need to rely on branches of computer science, science, and engineering not yet mature enough to serve as a backbone for designing a concurrent program development environment of long term use in an era of changing technology, changing computer architectures, and newly discovered concurrent algorithms. These inadequacies are present in every area of science and every area of engineering in which paralle! ɹm must be exploited, or in which distributed systems interact. Addressing this problem successfully will affect future national productivity and competitiveness, in addition to meeting military requirements. Training the scientists necessary to advance this problem will contribute to the development of intelligent control as a scientific discipline. The underlying mathematical and algorithmic science for parallel and distributed computing must be developed

systematically in and for all prospective fields of application in science and engineering. The present structure of university departments is generally not responsive to these needs. Computer science Ph. D's usually know little science and little applied mathematics. Applied mathematics Ph. D.'s usually know little about scientific computing. Hard science and engineering Ph. D.'s have little interest in developing new algorithms for parallel machines; they want off--the--shelf programs and subroutines so they can go about their business of end applications without fuss. Academic computational science, as opposed to academic computer science divorced from the rest of science and engineering, is rather thin. People learn on the job what should be part of their prior training. There should be a bigger university--trained pool from which research and development organizations can draw their talent. The Cornell Mathematical Sciences Institute, and the new Minnesota High Performance Computing Institute, and the MIT--Harvard--Brown Center for Intelligent Control represent part of the US Army Research Office response to this need. Much more educational infrastructure is needed nationwide.
Summary: **We lack educational infrastructure for producing enough scientists well acquainted with applied mathematics, parallel and distributed systems, and science or engineering simultaneously.**

What is Intelligent Control?

The term "intelligent control system" means different things to different people. What I mean is a message--passing network, with one class of nodes representing decision elements based on Logic, AI, and OR inference engines and optimizers, with another class of nodes representing effectors or sensors based on mixed discrete--continuous engineering systems, and finally with a third class of nodes which are human beings. All these nodes interact via message passing with feedback. The effectors and sensors and computers may be lumped or distributed, sequential or concurrent, synchronous or asynchronous. Intelligent Control adds to the separate subjects of traditional AI--OR--Computer Science and traditional Systems and Control Theory a unique additional complication: understanding and modelling, mathematically, and algorithmically, engineering systems in which there is constant feedback between OR--based and AI-- based and logic--based inference engines and physical devices obeying differential equations. So discrete logic or AI or OR decisions generated by inference engines and optimizers interact with human users and exert control over continuous physical processes normally modelled by discretized differential equations, and in turn the changes in the physical system are sensed and fed back into the inference engine databases. I believe that mathematical analysis and design of mixed AI--OR--logic and differential equation systems is feasible, but this is an uncharted area, a newly emerging territory of the highest order of importance in both systems based on the Von Neumann sequential architectures and those based on concurrency and distributed systems. It is hard enough to analyse cases where there is a single continuous physical device, be it sensor, effector, or control, interacting in feedback with one inference engine and database. The challenge of utililizing concurrent processing and interacting with distributed devices adds spice to what is already a leading edge area. This will require a real expansion of the areas of research for distributed and concurrent systems in computer science, which do not usually address interaction with distributed physical devices other than computers. It is well worth pursuing as a significant research initiative.

Foundations of Intelligent Control

The basic requirement on software tools for concurrent intelligent control should be that they enhance production of readable, modular, upgradable, maintainable, provably correct software. To achieve this goal we need a broad precise mathematics–computer science high–level model encompassing on common mathematical grounds <u>both</u> AI– , logic–based inference engines <u>and</u> differential equation–based models of physical systems controlled with feedback in a concurrent distributed environment. This is not satisfied by constructing mere <u>interfaces</u> between logic AI software and FORTRAN or ADA code controlling diverse devices. Instead, this refers to the development of integral models involving both physical devices and inference engines where control of those devices is exercised by functions computed from conclusions made continuously by AI–OR–logic inference engines. I believe that rational design of intelligent control requires the development of this subject. Of one thing I have no doubt. To validate that a concurrent program in this context is provably correct (or even convincingly correct) in satisfying high level program specifications, it is simply not possible to proceed without such a model.

Here, in a nutshell, is my personal conception of the required mathematical modelling. A concurrent logic program can be thought of as having a set of possible logical states, each describing the total state of the inference engine at a single moment of computation; in the logic programming model, these states are usually described by current bindings of all variables of all clauses in all processors. A single execution sequence for the concurrent logic program is a sequence of states compatible with progam execution (that is, a sequence of states which could happen). There are many execution sequences for any really concurrent program, including any concurrent logic program. To prove such a program is correct is to show by some means, formal or informal, that every possible execution sequence satisfies the program specification. For a concurrent program, this usually breaks up into a horrendous number of cases, and has to be proved by some sort of induction, either in a formal language of programs such as the Hoare systems, or by induction on the number of execution steps. In proofs that programs are correct, there are so many cases that computing machines are required in order to keep track of what cases have already been completely verified and what cases are left to verify. Similarly, for physical devices, if we look at control theory, the analog of the program specification for a physical device is its governing system of differential equations and its required accuracy and robustness of behavior. The set of allowed solutions is crudely (but not exactly) the set of allowed "execution sequences". To prove that the specification is satisfied is to prove that the implemented device satisfies the equations within the desired degree of accuracy and robustness. When this is modelled on digital machines, the equations and the solutions are modelled in discrete time steps as sequences solving difference equations. For this simplified explanation, we think of these time steps as the same time steps as for the logic program, and of the set of all the discrete sequences satisfying the physical device program specification as the set of all execution sequences for the physical device program.

But when concurrent logical programs and physical devices are connected in feedback and used in single concurrent systems, the current state of the whole system is described by the joint current states of the logic programs and the physical device. So the execution sequences of this joint program are those sequences of joint states satisfying the final program specification. To prove that all possible execution sequences satisfy the program specification, we have to perform inductions on the length of execution sequences to see that the specifications, including all accuracy and robustness requirements for all physical devices, are satisfied. We could also invent a formal system for this purpose. To get an informal correctness proof, it seems to me that some such methodology for classifying and resolving cases as to what behaviors are possible has to be carried through. This requires a high level of precision about what the specification says, what the physical models are, what approximations have been made, and what execution sequences are concurrently possible. This is my own personal analysis, highly simplified. I think that this is what underlies modelling the mathematics and algorithms of concurrent intelligent control.

I would like to see the development of mathematical and computer science models of distributed and concurrent devices involving AI–OR–logic inference engines as controllers with continuous physical devices as driven devices. With such models, computer scientists and applied logicians will be able to define useful semantics and syntax to describe execution and program specifications for hybrid programs, and a new generation of applied mathematicians and computer scientists will be able to develop provably accurate, provably robust, mixed logic–continuous model algorithms meeting program specifications. There are glimmers in research results in areas such as robot motion, machine vision, data fusion for sensors, discrete and continuous dynamical systems, concurrent algorithms for distributed systems, dataflow

models for LISP–FORTRAN, concurrent logic programming, discrete event simulation, distributed knowledge and belief, parallel implementations of combat modelling and battle management, etc. which have convinced me that this is a viable paradigm to organize future research.
Summary: Software development for intelligent control should be advanced by developing theoretical models of hybrid AI–OR–logic and differential equations systems, where results of inference engine deductions and of differential equation solving for physical devices are in mutual feedback.

With a basic understanding of the structure of these models, the features of the needed software tool environment should be much clearer. What we would like is a modelling fully understandable to scientists and design engineers and also to computer scientists or programmers. Hybrid logic–physical models are on one side of a fence, algorithm and software development are on the other. The scientists and system designers on one side of the fence and the algorithm and software implementors on the other need a common window. At present, the intelligent control system engineer–designer is much at the mercy of his programmers, simply due to the fact that hybrid systems don't have a high level mathematical model or specification language which is mutually understood on both sides of the fence. Difficult problems of correct modelling and design to meet robustness and accuracy requirements due to nonlinear interactions lurk behind every new code module for hybrid systems. A lesson from contemporary dynamical systems, which have no OR–AI–Logic subsystems, is that it becomes rapidly intractable to determine what singular behaviors violate intended specifications as system size goes up, and that much mathematical and engineering sophistication, in addition to software engineering skill, is required.

Where are we?

Intelligent control melds historically separate elements: physical networks, as studied by systems and mechanical and electrical engineers; computer networks and real time operating systems, as studied by computer scientists; inference engines, as studied in Operations Research, Artificial Intelligence, and Applied Logic; sensors and effector systems obeying mixed discrete–continuous differential equations, as studied by control engineers and dynamicists; and, lastly, human beings interacting with the systems as studied by specialists in human factors, artificial intelligence, and cognitive science. Here are personal views on the state of development of these subjects.

I. Modelling and algorithms for design of networks for communicating digital information between computers is a solid branch of computer science stemming from the study of operating systems and communication networks. Modelling and algorithms for interacting networks of inference engines and sensors and effectors is a mostly undeveloped research area. Research is needed to develop models, datatypes, and efficient, robust, accurate algorithms.

II. Understanding the behavior of networks of interacting physical devices, in the absence of logical control, requires already that we develop models and algorithms for determining the exceptional, as well as intended ordinary, behavior of very complicated dynamical systems. Tools available for this purpose are the mathematical models and algorithms of dynamical systems, and implementations of these models as software tools for simulation and design and evaluation. Mathematical algorithms and software permitting simulation of <u>small</u> dimensional problems are being developed. (One implementation is the Kaos system of Guckenheimer at Cornell, in experimental use at 100 sites, and based on the latest algorithms and theorems.) Such software, for low dimensional systems, will eventually allow the design engineer to run simulations with high assurance that there are no unobserved singular behaviors in the range of parameters likely to be encountered which would violate the program specifications. Similarly, such programs can help assess robustness, accuracy, efficiency, and correctness of code. But <u>standard</u> new military systems such as tanks, autonomous underwater vehicles, and distributed intelligent artillery, are systems with <u>many</u> degrees of freedom. At present such large systems cannot at present be simulated in full with all degrees of freedom. They rather must be simulated in simplified form with a much smaller number of degrees of freedom, even on the largest parallel supercomputers. It takes a lot of scientific and engineering skill to assure that the simplified systems, as simulated, rule out unwanted exceptional behavior and assure robustness and accuracy for the full system. Further research on mathematics of dynamical systems and corresponding algorithms is required if successful general purpose software tools are desired for such systems, even without intelligent control. Single applications often present major, but surmountable, algorithmic and modelling challenges. Modelling and simulation of large dimensional conventional dynamical systems is at the frontier of current research, without the added dimensions due to logic control. When we add AI–OR–logic elements to a network of conventional physical devices, we get a yet more challenging class of dynamical systems. This area requires simultaneous attention from experts in automated reasoning, OR optimization techniques, numerical analysis, dynamical systems, algorithm development, and control system engineering.

III. Now we discuss pure distributed logical decision networks, leaving out any controlled physical devices. Such systems are built to face continuously changing new information from many sources and must automate decision making based on fact and belief of many agents. The decisions are local for single agents and global for whole systems. Cooperating databases and cooperating expert systems are examples of this; so are command and control systems. This is a rapidly advancing logic–AI–computer science area, needing much more research.

IV. When we put physical devices to be controlled in feedback networks with decision devices, we are surely at the frontiers of knowledge. Basic modelling in this area can be developed, but is very primitive.

Models in these and other closely related areas of AI, OR, Systems, and Control, need further mathematical and algorithmic development focused on intelligent systems. Such developments, models and algorithms, will furnish the necessary datatypes and datatype transformations for implementation of future software tools for hybrid system design, simulation, and testing.

I advise:
– A short–term research program tied to a couple of military testbed projects developing models and special purpose software tools.
– A simultaneous long term research program, tied to the short–term program for inspiration and applications, but independent of the short term program. This program should develop models and algorithms needed by software engineers to produce general purpose software tools for intelligent control which can insure writing accurate, robust, correct, software.

Postscript

At the ARO Mathematical Sciences Institute at Cornell we have a concentration on algorithmic mathematics in such diverse areas as partial differential equations and dynamical systems, symbolic computation in algebra and logic and combinatorics, robust motion planning, geometric modelling, dataflow models incorporating LISP and FORTRAN, program logics and program development algorithms for concurrency, etc. Many of these areas enter into intelligent control. I hope that we will be able to contribute mathematical, computer science, and engineering tools for this emerging area as well.

Distributed Computing Research at MCC/STP

Colin Potts

MCC Software Technology Program

Abstract: The Software Technology Program at MCC is addressing techniques for the early phases of distributed systems design. We have introduced language abstractions, the interaction and the team, that permit concise, high-level descriptions of concurrent behavior. We have implemented a visual language environment, VERDI, that supports execution and performance simulation of models expressed in terms of these abstractions. VERDI has been applied by MCC and several of its member companies to practical distributed systems. We are developing a transformation-based methodology for VERDI. We are investigating several approaches to fault-tolerance, including self-stability. We are also working on techniques for validating real-time constraints.

The Software Technology Program (STP) at MCC is addressing the early stages of large-scale system development. Research into distributed computing at MCC/STP, therefore, has focused on tools and techniques to assist in the early design and assessment of distributed systems.

Concepts

During the early stages of the development process it is desirable to abstract away from architectural concerns. In centralized systems two classes of abstractions have become widely accepted: structured programming constructs for control and data abstractions. In distributed systems an additional class of abstraction is necessary: communication abstractions. Communication abstractions are preferable to the low-level communication constructs currently in use, such as message-passing and shared memory, because they require no assumptions about the underlying communication medium.

Our work has focused on the expression of synchronization and communication relations among distributed processes in terms of *multi-party interactions*. The interaction is synchronous, multi-party, and symmetric. In addition, we have developed an encapsulation mechanism, the *team*, for collections of interacting processes. Teams can be used to represent subsystems or data abstractions. Their interfaces are procedures that may be called by any process. Only processes in the same team may interact directly; those in different teams communicate by calls to interface procedures. In our experience, communication and contention constructs, such as buffers, blackboards, and shared resources, can easily be modeled by teams, and therefore need not be treated as primitives. Interactions and teams form the basis of the Raddle language model (Attie, 1987; Evangelist, Shen, Forman and Graf, 1988).

Visualization and Modeling

Concurrent systems are difficult to understand because the concurrent execution of multiple processes leads to an explosion in the size of the state space. Our experience has confirmed that program visualization techniques can be used to clarify the behavior of practical distributed systems long before they are implemented. We have developed a visual language that incorporates processes, interactions, and teams, and we have implemented a visual language environment, VERDI, for editing and executing designs (Graf, 1990; Shen, Richter, Graf and Brumfield, 1990). In VERDI we distinguish the control and communication constructs, which are represented visually, from the details of the actions and interactions, which are specified in an embedded textual language (a subset of C).

Figure 1 shows a high-level model of an electronic-funds transfer system in the form of a VERDI diagram. The large enclosing box denotes the only team in this design. The three processes are aligned vertically, and their flow of control is read from left-to-right. They are cyclical, so each process includes an implicit iteration. Choices are shown by parallel branches. Simple boxes denote local actions, whereas doubly-barred boxes denote interactions. The parts of an interaction have the same name. Thus, processes POS, c_bank, and m_bank participate in the interaction named 'transaction'.

Execution proceeds on two levels, which correspond to the visually and textually specified parts of the model. The high-level control and communication skeletons of the processes, which are expressed graphically, are interpreted by an interaction and process-scheduling mechanism that implements the operational semantics of Raddle. Local actions, interactions, and the evaluation of guards are implemented by interpreting the embedded C code.

VERDI also supports interactive performance modeling. Local actions and interactions may be assigned durations. VERDI uses these durations and a virtual clock when scheduling events. Performance statistics may be gathered by including instrumentation teams for timing and throughput analysis. Because these are just VERDI teams, it is easy to extend the performance-gathering facilities without reprogramming VERDI itself.

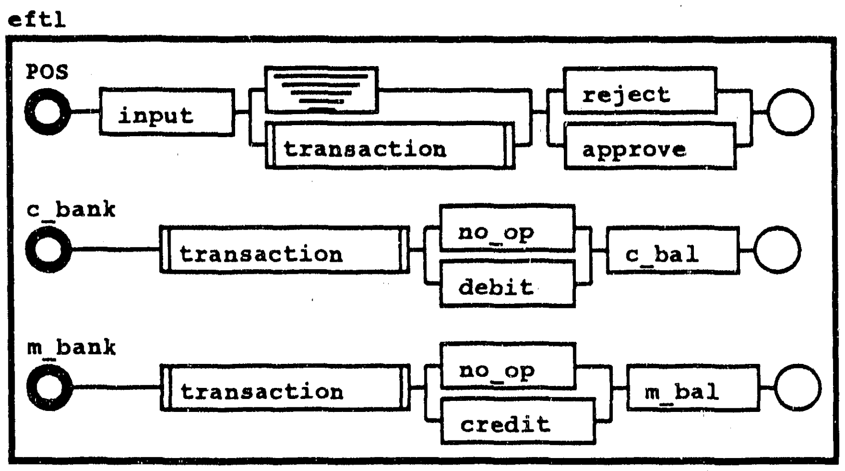


Figure 1: A high-level model of an EFT system

Methodology

We have developed a methodology for using VERDI. We recommend starting with a tightly synchronized design that can be shown to have the desired qualitative properties, and then refining the design by a series of synchrony-loosening transformations. Performance simulation is best delayed until the transformations have progressed to a point where the teams and processes in the model correspond to architectural components. Figure 2 depicts the EFT design at a point in its refinement history when the original 3-party interaction has been replaced by several 2-party interactions. A new switch process has also been introduced as an intermediary between the POS and the bank processes. Figure 2 is concrete enough for simulation to be useful for rough estimation of performance.

In practice the methodology is applied rather informally (Forman and Evangelist, 1987). We have, however, specified several correctness-preserving transformations and have developed several synthesizing transformations for obtaining finite-state models from temporal formulae (Attie and Emerson, 1989). We are planning to implement support for these transformational techniques and demonstrate their effectiveness in practical designs. To support this transformational methodology, we have integrated VERDI with the MCC/STP hypertext browser GERM (Bruns, 1988) to form a prototype VERDI information management environment (VIM) for recording design history, design alternatives, rationale, and performance data.

Prototype code can be generated from VERDI. A CSIM translator has been implemented (Ojukwu, 1990). Guidelines exist for mapping Raddle constructs to Ada (Attie, 1988), and an Ada translator and run-time system has also been implemented for a variant of VERDI, in which the embedded language is Ada (Attie, Bruns, Evangelist, Richter and Shen, 1989).

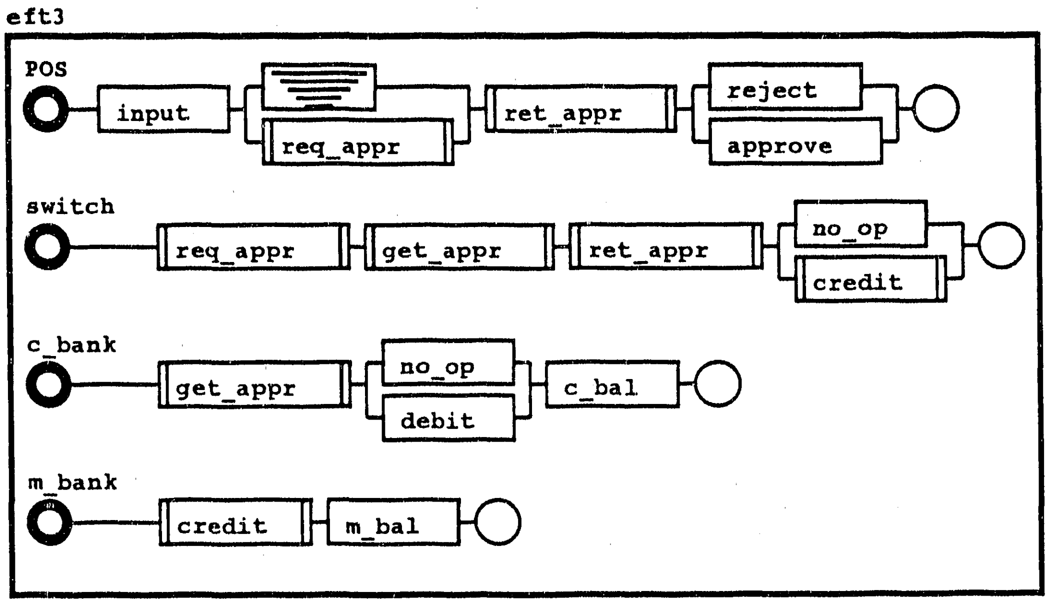


Figure 2: A refined model of the EFT system

Practical Experience

VERDI has been used to model a wide variety of applications within MCC and its shareholder companies. Applications include transaction processing (including EFT), application-specific protocols (e.g., cellular-radio handoff), distributed energy management, missile-launching procedures, post-facto tool integration, and distributed artificial intelligence.

The VERDI model for the distributed AI application was done by David Bridgeland, Natraj Arni, and Michael Huhns of MCC's Advanced Computer Technology Program. They are developing a distributed truth maintenance system, in which autonomous heterogeneous reasoners cooperate in solving a common problem. Each reasoner has an unintelligent communication aide that deals with the logistics of communicating with and delegating tasks to the others. The protocol employed is non-trivial, and an initial demonstration prototype deadlocked frequently. The VERDI model, completed in two weeks, helped uncover three problems, two of which led to unexpected classes of deadlock, and all of which were easily corrected at this stage of the design. We believe that the VERDI visual presentation will help future efforts to propose a reference model for inter-agent coordination in DAI systems.

Fault Tolerance

We are taking several approaches to fault tolerance: superimposition, quorum interactions, and self-stability.

One of the problems in understanding fault tolerant systems is that the logic and clarity of the desired behavior is cluttered by error detection and recovery details. Classical modularity is not a solution, as there is usually no single point where fault tolerance must be introduced. Superimposition, a layering mechanism whereby an underlying computation is embellished by additional functionality, is more promising. For example, a superimposition may monitor the underlying computation for faults. The attractiveness of superimposition for fault-tolerance is that the specifications of the underlying computation and the superimposition are separable.

Superimposition may be accomplished by transformations or by composition. In transformational superimposition (Katz, 1987), the superimposition is a delta on the underlying program. This technique has been applied to the EFT case study (Shen, 1988). Unfortunately, the resulting fault-tolerant program is difficult to understand, because it is specified in separate places in different languages: the design language, and the tran.formation language. More recently, Forman and Francez (1990) have proposed a composition-based form of superimposition for an extension of Raddle. The superimposition operator conjoins an underlying and a superimposed process. The superimposed process may regulate the behavior of the underlying process by inhibiting or delaying interactions in which it participates but may not change the values of any of its variables.

A Raddle interaction may only be enabled if all of its participants are ready. If a participating process fails, the interaction cannot occur. In a quorum interaction (Evangelist, Francez and Katz, 1989), the enablement conditions are relaxed, so that only a quorum of the participants needs to be ready. Its clearest application is to allow distributed agreements to be implemented when only a subset of the processes are active.

In the preceding methods, as with all classical approaches to fault tolerance, there is no notion of the completeness of the recovery procedures. It is generally possible to invent additional failure modes that a model does not handle. In self-stabilizing systems, by contrast, the state space of the system is partitioned into safe an: unsafe sub-spaces by specifying a safety predicate. A system is said to be self-stabilizing if it is guaranteed to converge to a safe state from any state within a finite number of steps. The partitioning of the state space and the self-stability condition guarantees completeness.

An arbitrary program can be made self-stabilizing by superimposition (Katz and Perry, 1989) but at the cost of great conceptual and computational overhead. We believe that it is preferable to design self-stability into a system from the start, and that it is possible to discover some useful design heuristics that lead to self-stability. We have developed several self-stabilizing algorithms and protocols.

Iteration systems provide an abstract model for self-stability. We have demonstrated the tradeoffs between resources and the convergence rate of an iteration system (Gouda and Evangelist, 1989) and have shown how to reduce proof obligations for the convergence of discrete iteration systems to fixpoints (Arora, Attie, Evangelist and Gouda, 1990).

The most interesting application of self-stability is likely to be in the augmentation of classical recovery-based fault-tolerance procedures. Many subtle system fault modes arise when a recovery procedure itself fails. It would be valuable to guarantee the eventual success of a recovery procedure in the presence of faults. There are also applications of self-stability other than fault-tolerance. For example, it may be a useful concept in discrete-event control systems and in the formalization of some classes of adaptive systems (e.g., neural nets). Self-stability is currently a subject of great interest. In 1989, MCC/STP organized the first of what is intended to become a regular series of workshops in the field (Evangelist and Katz, 1989).

Real-Time Systems

VERDI has been applied to real-time systems. We are currently collaborating with Rockwell, Collins Avionics Division, to model a real-time executive (RTE) in VERDI. It includes standard real-time facilities, such as timeouts, interrupts, and priority-based scheduling. The RTE is an ongoing development project, and is our most significant and sustained technology-transfer project to date. The model will be used by applications developers to predict the performance of their applications given alternative scheduling strategies. It reduces the risk of having to wait until a prototype is implemented on the target hardware. The current model is specific to one executive, but we intend to generalize our work into a set of standard real-time facilities that can be used to validate application models.

For this generalization to be possible, it will be necessary to extend VERDI and its language model. Although VERDI contains facilities for modeling the passage of virtual time, it is not easy to model hard real-time constraints and use them in scheduling events. We also plan to investigate the feasibility of model-checking in VERDI, or a VERDI-like real-time design language. We envisage real-time constraints being specified in a specification language based on a computationally tractable real-time logic such as RTL (Jahanian and Mok, 1986).

References

Arora, A., P. Attie, M. Evangelist and M. Gouda, "Convergence of iteration systems", *Proc. CONCUR'90*, Amsterdam, 1990.

Attie, P., "A guide to Raddle-87 semantics", MCC Technical Report, STP-340-87, MCC, 1987.

Attie, P., "The Generation of Ada Code from Raddle/VERDI Designs", MCC Technical Report, STP-236-88, MCC, 1988.

Attie, P., G. Bruns, M. Evangelist, C. Richter, and V.Y. Shen, "Vanna: A visual environment for the design of distributed systems", MCC Technical Report, STP-250-89, MCC, 1989.

Attie, P. and E.A. Emerson, "Synthesis of concurrent systems with many similar sequential processes", *Proc. 16th Ann. Symp. Princ. Programming Languages*, ACM Press, 1989.

Bruns, G., "Germ: A metasystem for browsing and editing", MCC Technical Report STP-122-88, MCC, 1988.

Evangelist, M., N. Francez, and S. Katz, "Multiparty interactions for interprocess communication and synchronization", *IEEE Trans. Software Eng., 15(11)*: 1417-1426, 1989.

Evangelist, M. and S. Katz (eds.), *Proceedings of the MCC Workshop on Self-Stabilizing Systems*, MCC Technical Report, STP-379-89, MCC, 1989.

Evangelist, M., V.Y. Shen, I. Forman, and M.L. Graf, "Using Raddle to design distributed systems", *Proc. 10th Int. Conf. Software Eng.*, IEEE Comp. Soc. Press, 1988.

Forman, I. and M. Evangelist, "EFT: A Case Study in Design using Raddle", MCC Technical Report, STP-121-87, MCC, 1987.

Francez, N. and I. Forman, "Superimposition for Interacting Processes", MCC Technical Report, STP-124-90, MCC, 1990.

Gouda, M. and M. Evangelist, "Convergence response tradeoffs in concurrent systems", MCC Technical Report, STP-124-89, MCC, 1989. (Submitted to ACM TOPLAS).

Graf, M.L., "Building a visual designer's environment", in S. Chang (ed.), *Principles of Visual Language Systems*, Prentice-Hall, 1990.

Jahanian, F, and A. Mok, "Safety analysis of timing properties in real-time systems", *IEEE Trans. Software Eng., SE-12(9)*: 890-904, 1986.

Katz, S., "A Superimposition Control Construct for Distributed Systems", MCC Technical Report, STP-340-87, MCC, 1987.

Katz, S. and K.J. Perry, "Self-stabilizing extensions for message-passing systems", in Evangelist and Katz (1989).

Ojukwu, M., "Performance Measurement and Evaluation of VERDI Designs", MCC Technical Report, STP-024-90, MCC, 1990.

Shen, V.Y., "Using Superimposition to Add Fault Handling to an EFT System", MCC Technical Report, STP-386-87, MCC, 1988.

Shen, V.Y., C. Richter, M.L. Graf and J.A. Brumfield, "VERDI: A visual environment for designing distributed systems", *J. Parallel Distib. Comp., 9*: 128-137, 1990.

Integrating Controls System Design with Systems and Software Engineering

Magnus Rimvall
Control System Laboratory
GE Corporate Research and Development Center
P.O. Box 8, Schenectady, NY 12302

Abstract

The tasks of control system requirements analysis, control system analysis and design, and controller software/hardware implementation have traditionally been viewed as three distinct activities. In large projects these activities are often handled by different groups using incompatible CAD tools. Consequently, the overall design cost is inflated by the need for manual data conversion between the different phases. Also, it limits the traceability of individual requirements through the process, and it prevents efficient design iteration involving more than one phase. The use of an integrated set of software tools would bridge the transition of information between the different project stages, and eliminate the risk of having errors introduced during any manual specification or design conversion.

Introduction

The 1980's was an important decade for control engineering. New theories and methods in such areas as multivariable and robust control, as well as better numerical algorithms (e.g. for higher-order systems), extended design capabilities and increased the operational reliability of the designed controllers. Methods for decomposing algorithms onto parallel processing architectures increased potential real-time performance. Most importantly, the introduction of powerful and easy-to-use interactive Computer Aided Controls Engineering (CACE) tools enabled the average control engineer to efficiently and confidently put all these new methods and algorithms to productive use.

Most control engineers now have access to good CACE tools. A class of extendable programs based on Matlab (Moler, 1980), the so called "matrix environments", became very popular during the last decade. A few robust and well-supported matrix environments have come to dominate the CACE market, with other tools covering different niche markets. Lead time between the development of new control algorithms/methods and their incorporation into some extendable matrix environment has become very short. Thus, the average control engineer is well equipped to perform his core engineering tasks.

These theoretical and computational advances have given industrial controls engineering and its practitioners a significant productivity boost. Further developments will undoubtable result in even better algorithms and even simpler-to-use tools, but such improvements will become more and more incremental in nature. To make another *significant* impact on industrial controls practices and cost, we need to look beyond the traditional domains of automatic control. We must examine how controls interfaces with other engineering disciplines in the overall process, and develop methods and tools to better integrate control engineering with these other disciplines.

In the next chapter we will introduce two disciplines closely coupled with controls engineering. Thereafter we will show how CACE methods/tools are best interfaced/integrated with tools/methods of these other disciplines, and how this will increase the overall project productivity. Finally, we will discuss to which extent todays tools can be integrated to accomplish this and identify technology areas where additional theoretical work and/or tools development is necessary.

Systems and Software Engineering interactions with Controls

Control Engineering is the primary, but far from the only, technical discipline involved in the overall process of designing and building a product with an embedded controller. Some of the ubiquitous activities in *systems* and *software* engineering are very closely coupled with traditional control engineering tasks (individual projects may require the use of further disciplines ranging from astronomy to zero-sum games). However, current practices suffer from duplication of engineering efforts among these disciplines, and from ineffective information/data exchanges between them. Considerable productivity and reliability gains could be attained by integrating methods and tools better, and by streamlining tool interfaces.

Systems and software engineering are established disciplines with their own set of methods and tools. Although this paper primarily deals with the ***control** system* engineering and ***control** software* engineering *sub-domains*, the methods and tools of these sub-disciplines typically remain the same:

- Discipline: *System Engineering.*
  Overall activities: Initial system requirements analysis; Overall system specification, system design and implementation; Administrative tasks such as project timing, resource allocation, and task coordination.
  Controls oriented activities: Initial feasibility studies of the proposed system/controller; Controller requirements analysis; Higher-level system/controller decomposition (hardware vs. software, in-house vs. subcontractor, etc).
  Tools: General engineering analysis programs (e.g. modeling and simulation tools); project planning tools

- Discipline: *Controls Engineering*
  Activities: System modeling and identification; Controller architecture design; System analysis and detailed controller design.
  Tools: CACE tools (matrix environments), Simulation packages

- Discipline: *Software (Hardware) Engineering*
  Overall activities: Implementation of the system; Hardware and software design of the embedded system and its peripherals; Operating system development/customization; Final validation and testing.
  Control-related activities: Implementation of control laws; Integration of the resulting code into operating software and hardware environments; Final control law validation and testing.
  Tools: Computer Aided Software Engineering (CASE) tools, compilers, debuggers; CAD tools for Analog/Digital hardware designs.

The considerable overlap in controls-related tasks performed within these different disciplines is shown in Figure 1. For example, good and reliable models of the system are necessary both during initial control systems feasibility studies and during the detailed controller design. Also, the controller architecture must be known both to the control engineer and to the control software team responsible for implementing the real-time code. Even the control code itself could be reused if the same code is included both in simulation models for controls validation and simulation, and in the production real-time code (this can for example be achieved through an automatic code generation facility with multiple target-language capabilities).

Despite the overlaps, these tasks are mostly viewed as separate activities to be performed by different teams or group of teams. Team members are typically selected by their core engineering discipline, and each discipline use its own set of engineering tools. This makes technical interaction between the related groups difficult and the overall design cost is inflated by the need for manual data conversion between the different phases. Also, incompatible tools limit the traceability of individual requirements through the process, and prevents efficient design iterations.

A tighter integration of controls, systems and software engineering is particularly important during the design and subsequent implementation of *distributed* control systems, as the number of interfaces between the three disciplines and their relative importance grow with the increased overall complexity and a finer granularity of the controller architecture. For example, the systems engineering decomposition must identify and define the distributed architecture. On the software engineering side, the binding of distributed control laws onto a distributed processor architecture substantially increase the complexity of the operating system interface.

In the next two chapters we will see how tools from these three disciplines can be combined to form an integrated software environment.

An integrated Controls Engineering environment

An integrated controls/system/software environment must incorporate functional capabilities from all three disciplines: control **system** requirements analysis, control system specification and decomposition, **control** system modeling and simulation, system and parameter identification, control system analysis and design, **software** specification and design, automatic control law code generation, computer assisted code testing and validation, etc. Ideally all these areas should be supported by a single program with a uniform user interface. However, this is not a very realistic approach:

- The cost of developing and maintaining such an all-encompassing system would be prohibitively high compared to the cost of buying existing state-of-the-art components and integrate them.
- Many of the present market-dominating tools have already achieved a high degree of acceptance in their respective domains, and the average engineer will not be very eager to trade in the effective and reliable tool he is already using for some mandated tool.
- It makes little sense to, for example, introduce a control-law CASE tool to make it compatible with the used CACE tools if this then makes the control law CASE activities incompatible with other CASE activities.

A more realistic approach would be to integrate a set of well-established state-of-the-art tools into a single system with automatic information and data conversion between the different tools. This would allow the continued use of established tools, while software development may be concentrated to specific areas where no good tools are available yet.

A smooth and seamless coupling between the different components is *the key factor* to a successful integration. This coupling must be complete automatic, if any manual conversion is necessary the integration has brought little compared to the existing situation. Figure 2 shows primary flows of information between the different controls related activities. The System Specification document and the Control System Definition constitute the main interfaces between the three disciplines. However, it is important to note that the whole process is iterative, with adjustments and re-designs potentially going all the way back to the original requirements. Thus, in order to maintain a traceability of individual requirements through the process and to ensure consistency the coupling between the different phases the interfaces must function in both directions.

The use of an integrated set of software tools would not only give productivity gains by automating the transition of information between different project stages, it would also eliminate the risk of having errors introduced during otherwise manual conversions, and provide cross-checking during any iterative re-design.

The GE integration effort

GE is currently defining, implementing and integrating a comprehensive software suite to be used throughout all phases of controller planning, specification, design and implementation. The suite will tie the control systems, control software and control design Engineering worlds together. It is based on both commercial and proprietary individual tools and runs on Unix™-based engineering workstations. All components are graphics oriented, with simple-to-use "point-and-click" interfaces and powerful display graphics. Individual workstations may be licensed and configured to contain only a subset of the component tools. Platform-independent

™ Unix is a trademark of AT&T Bell Laboratories yet well supported by presently available tools. There are several areas where further research and basic development is necessary:

- *Control design interchange format.* Some CASE tools support de-facto standards such as EDIF (Electronic Data Interchange Format), but no commercially available CACE package interface with these data interchange formats. This prevents a complete integration of the CACE and CASE world, forces the engineers to rely on manual or custom-implemented transformations, and prevents a complete consistency checking between corresponding engineering designs in the two disciplines. Basic development of a mapping scheme between control design architectures, controller implementations, and CASE interchange standards is necessary.

- *Optimized code generators.* Presently available control-oriented code generators operate from block-diagrams and/or equation sets on a block-by-block or line-by-line basis. This results in inefficient *and* unstructured code as block interdependencies and hierarchies are not utilized. The efficiency of simulation code is not mission-critical, however, the same code may create serious problems when down-loaded into real-time microprocessor(s). As many cross-compilers do not perform good code optimization (this is particularly true for most Ada compilers), the source code itself must be optimized for memory and speed.
  Source code optimization is a non-trivial task. The principles behind general variable/register optimization are known, but this is really a reverse-engineering process and either very complex or not very efficient. If the control code specification is in block-diagram form (or some other canonical representation), with well-defined control flows and algebraic operations, a much better code optimization could be done as we already have "answers to all re-engineering questions" such as dynamic data dependencies and scaling information . Special-purpose code-optimization (or, rather, code-generator optimization) could therefore result in much faster code.

- *Automatic validation and testing.* Software validation and testing is still largely a manual process. Electronic interconnection of the design and code implementation phases opens up the feasibility of automatic validation and testing of the produced code both on the module and on the integration level. As with the code optimization, contextual information from the control specification (block diagram or canonical form) could be used to automatically produce test cases and execution orders for these cases. The goal should be a fully automated generation and execution of test cases, with automatic or computer-assisted evaluation of the test results.

Conclusions

With the advent of modern CACE tools the productivity of the individual control engineer has improved significantly. The same is true in other, related disciplines such as System and Software Engineering. Yet, on a project level further overall productivity gain is hampered by tools incompatibility and the necessity of performing manual translations of information and data between different controls-related tasks. This prevents a smooth integration of the different activities and makes the

ASCII files form the primary coupling between the different individual tools. These files can be exchanged between any networked workstations.

The GE system is a joint development between different GE Aircraft Engine, GE Aerospace, and GE Corporate Research and Development departments. It presently contains the following components:

- The Cadre Teamwork™ program has been selected and installed as a comprehensive CASE tool. This program is used for general software specification and design, it is also increasingly being used for system specification and overall system decomposition.

- Beacon, a graphical editor for specifying engineering block diagrams. Block diagrams have a long tradition in controls engineering, and this editor gives the control engineer a natural tool for specifying models, controller dynamics, and controller logic. It can handle large, hierarchical diagrams. It lets the user perform either top-down or bottom-up design, or a combination of the two. Figure 3 shows a screen-dump of the editor. Some of the available blocks are shown on the palettes to the left, new ones may be added through a built in icon editor. This proprietary tool is a customization of the generic block-oriented Design/OA™ software system from Meta Software

- GE-MEAD, a proprietary CACE environment (Taylor et. al, 1990) which features an integrated engineering data base manager, a built-in expert system shell, and a flexible user-friendly user interface. The user may arbitrarily switch between an easy-to-use graphical "point-and-click" mode (Figure 4), and more versatile/demanding command modes. MEAD uses the Pro-Matlab™ package enhanced with commercial and proprietary toolboxes for its numerical control algorithms, and the ACSL™ package for nonlinear simulation.

- Proprietary automatic code generators with the capability of generating code in different languages. These generators are able to transform the output from the block-diagram editor to simulation and/or real-time code. The generators could also serve as an interface between Beacon and MEAD, and tie in with Teamwork so that the top-level Teamwork software specification/generation could be automatically coupled with the lower level control code.

Proposed Future Research Areas

The GE Controls Environment constitutes a comprehensive suite of high-quality software tools, integrating the overall controls engineering process involving Controls, Systems and Software engineering groups. These tools already find important use in the operating departments. On the research side, the evaluation of existing tools and the integration of these tools into a system have revealed technology areas not yet well understood and/or not

™ Teamwork is a registered trademarks of Cadre Technologies, Inc.
™ Design/OA is a registered trademark of Meta Software Corporation
™ Pro-Matlab is a trademark of the MathWorks, Inc.
™ ACSL is a trademark of Mitchell and Gauthier Associates

natural iteration and re-engineering a tedious process. Such an integration is particularly important for *distributed* control systems with their increased complexity and the tight coupling between controller and overall system architecture.

In this paper we have proposed that existing tools from the different disciplines should be tightly integrated to form a single, unified control design/system/software engineering environment. We have shown how GE have applied these ideas, and we have suggested some areas of further research.

References

Moler, C. (1980) MATLAB User's Guide. Dept. of Computer Science, University of Albuquerque, NM.

Taylor, J. H., Frederick, D. K., Rimvall, C. M. and Sutherland, H. (1989), "A Computer Aided Control Engineering Environment with Expert Aiding and Data-Base Management". Proc. IEEE Workshop on Computer-Aided Control System Design, Tampa, FL.

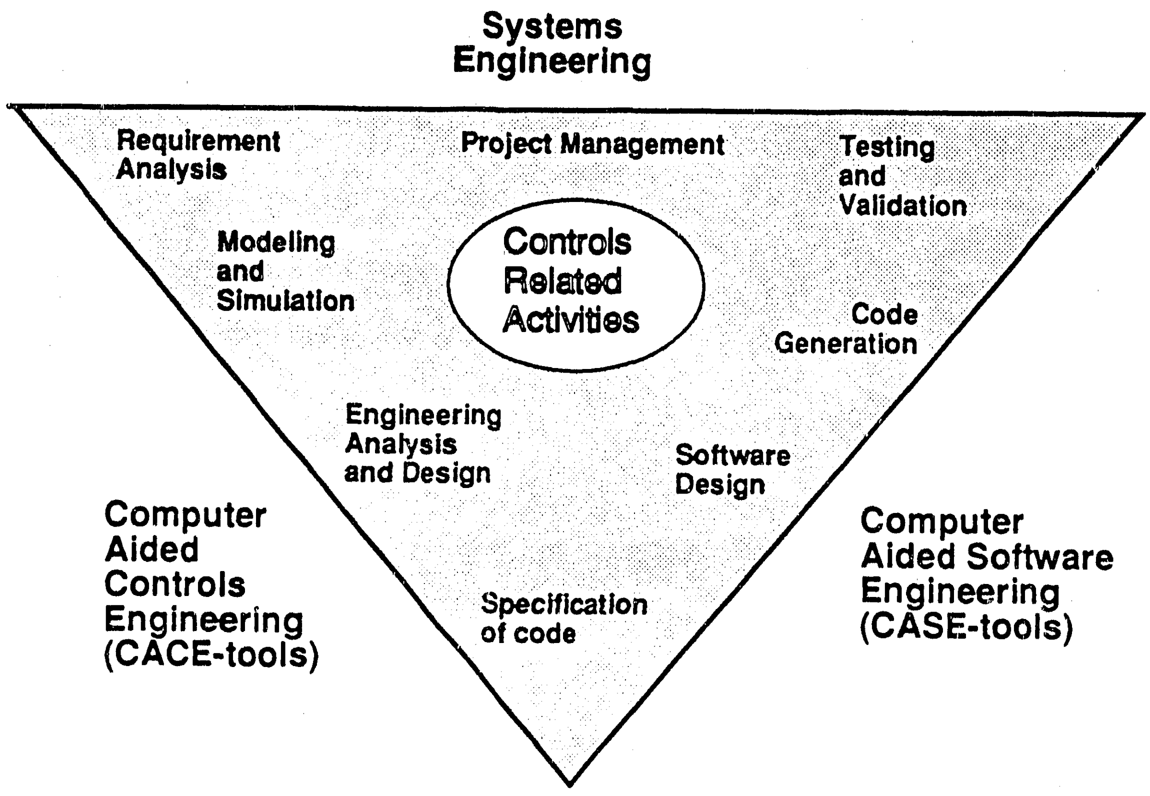

Systems Engineering
Requirement Analysis
Project Management
Testing and Validation
Modeling and Simulation
Controls Related Activities
Code Generation
Engineering Analysis and Design
Software Design
Specification of code
Computer Aided Controls Engineering (CACE-tools)
Computer Aided Software Engineering (CASE-tools)

Figure 1. The relationship between Control/System/Software Engineering

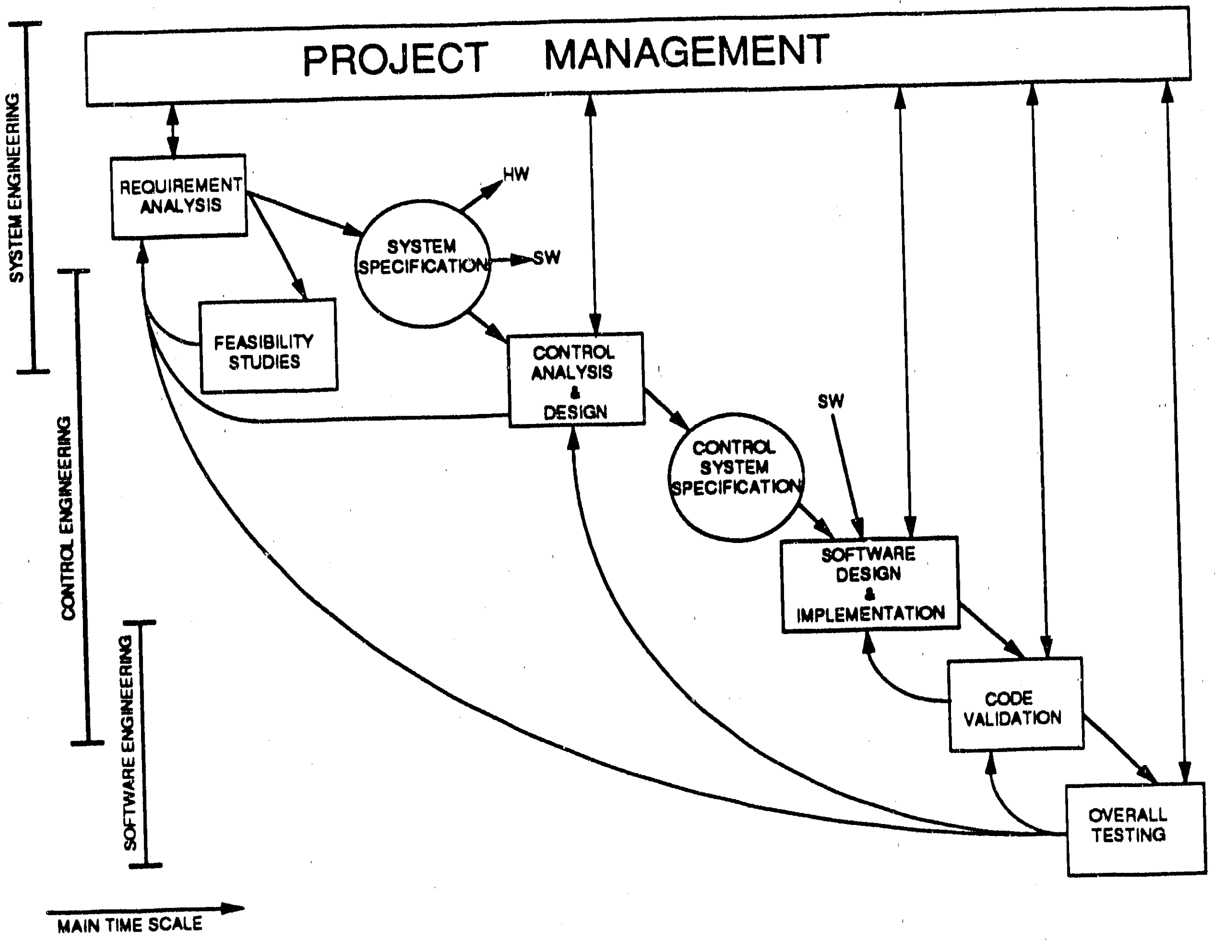


Figure 2. Main flow of information in the controller design cycle.

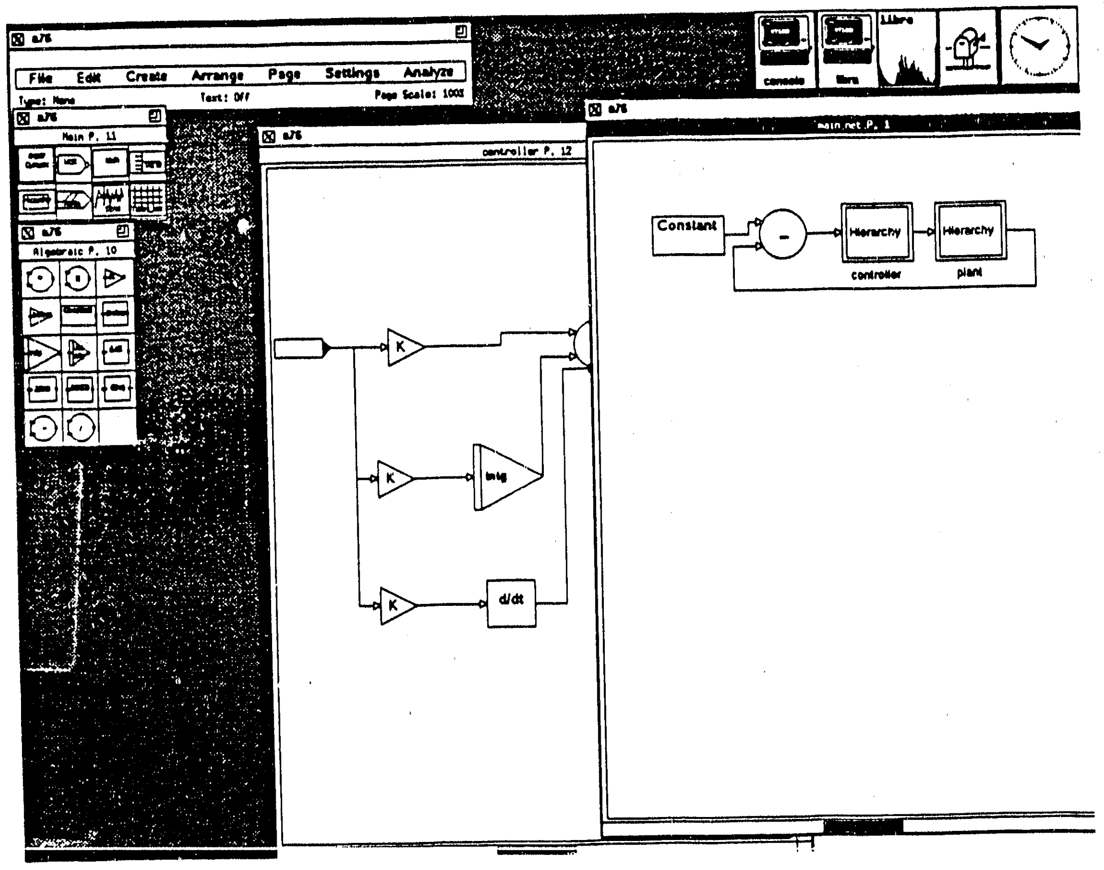
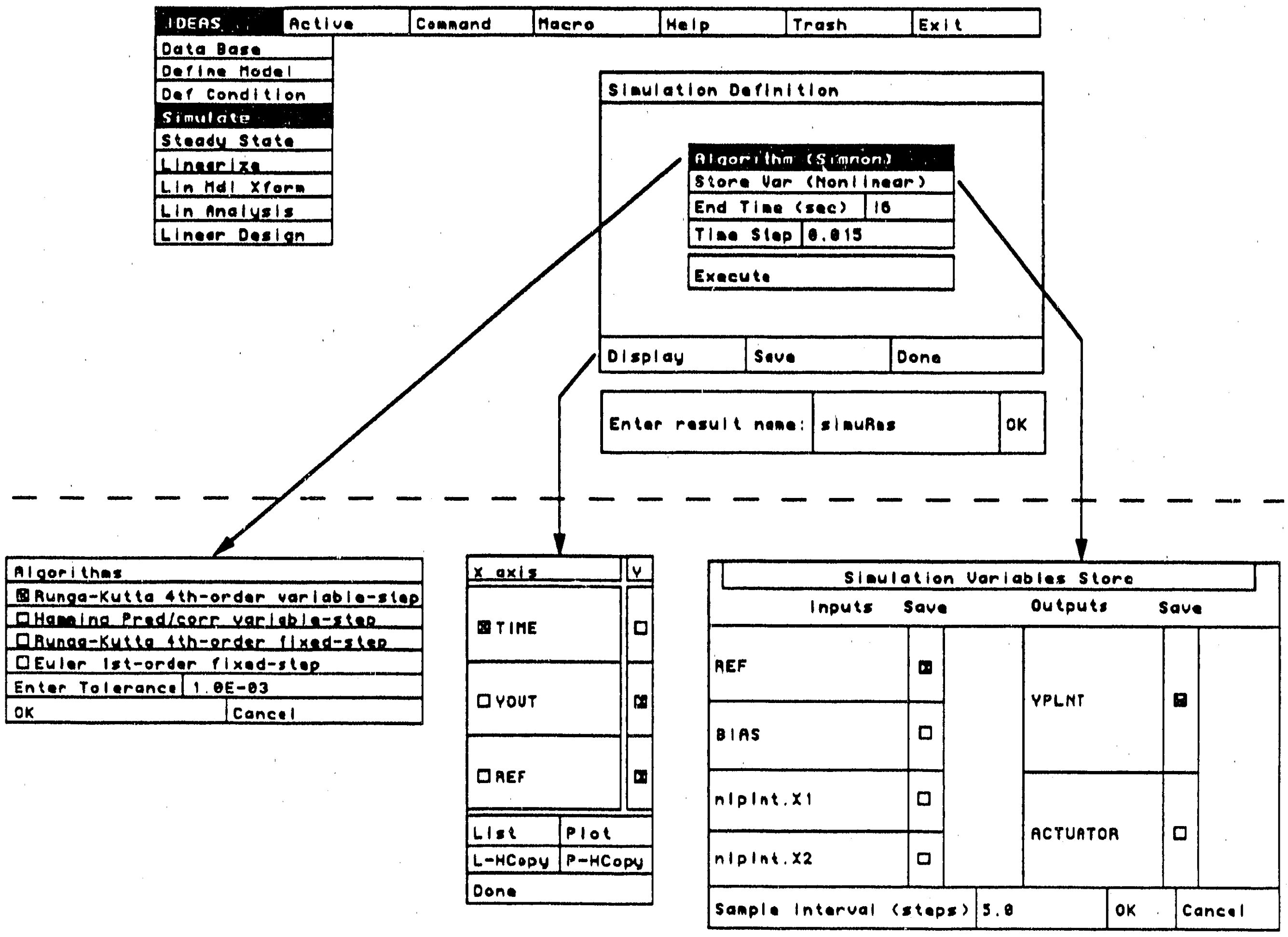


Figure 3. A Beacon screen with two hierarchical pages



End Time (sec)	10
Time Step	0.015

Enter result name:	simuRes	OK

Algorithms	
☒ Runga-Kutta 4th-order variable-step	
☐ Hamming Pred/corr variable-step	
☐ Runga-Kutta 4th-order fixed-step	
☐ Euler 1st-order fixed-step	
Enter Tolerance	1.0E-03

X axis	Y
☒ TIME	☐
☐ YOUT	☒
☐ REF	☒

Inputs	Save	Outputs	Save
REF	☒	VPLNT	☒
BIAS	☐		
nlpInt.X1	☐	ACTUATOR	☐
nlpInt.X2	☐		
Sample Interval (steps)	5.0		

Figure 4. The GE-MEAD simulation screen and affiliated forms

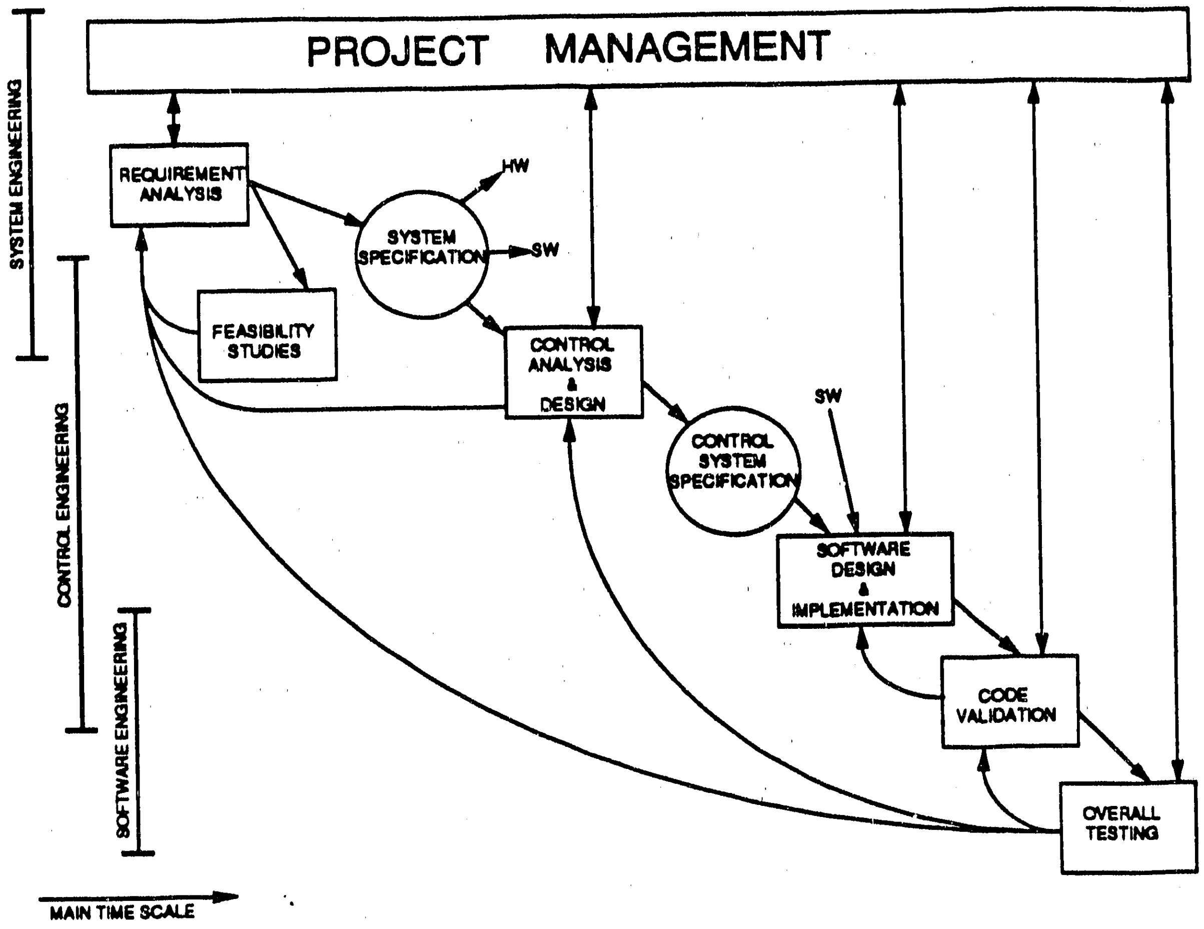


Figure 2. Main flow of information in the controller design cycle.

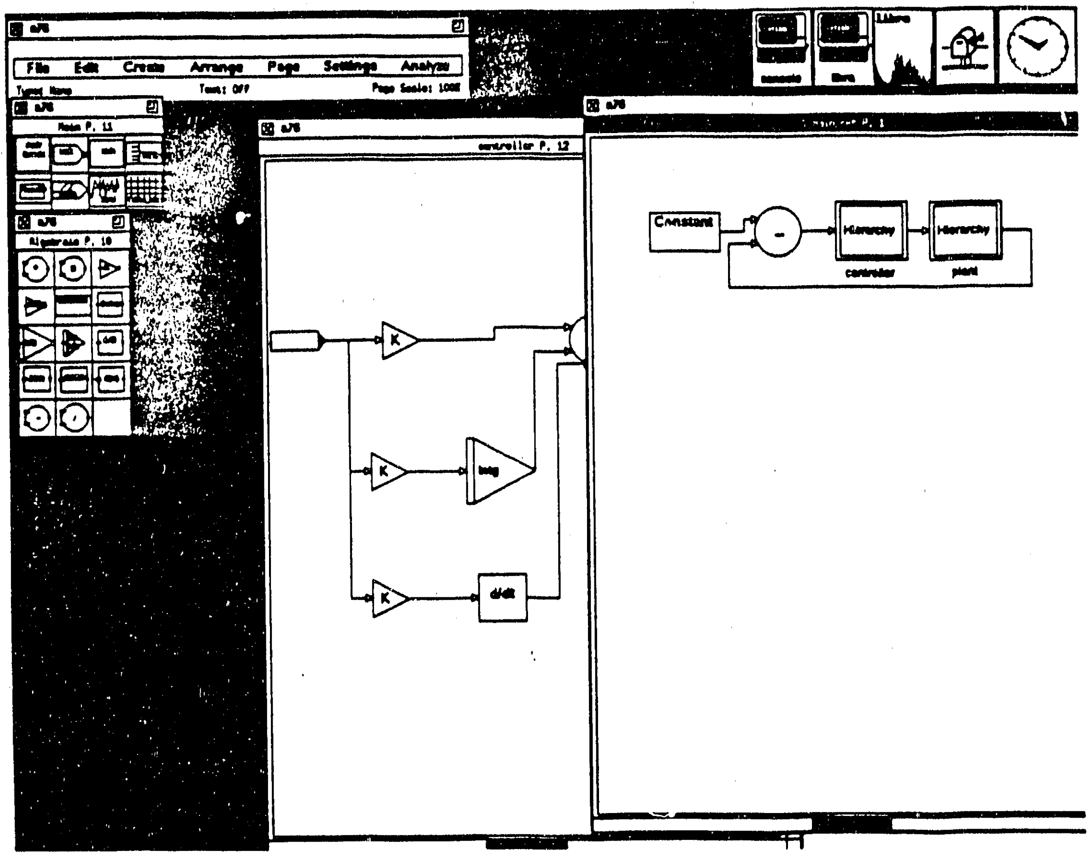
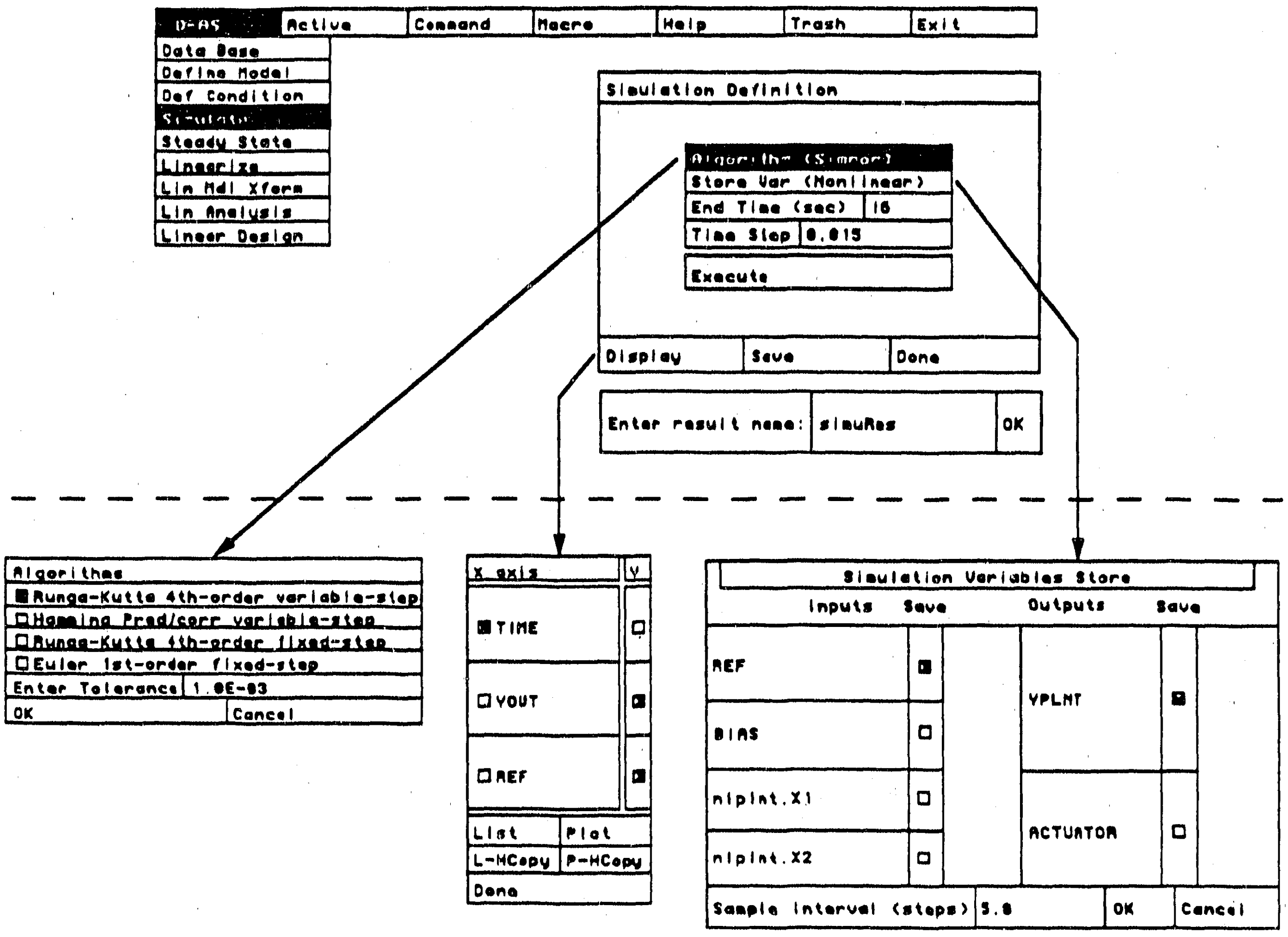


Figure 3. A Beacon screen with two hierarchical pages



Simulation Definition

Algorithm (Simnon)	
Store Var (Nonlinear)	
End Time (sec)	16
Time Step	0.015

Enter result name:	simuRes	OK

Algorithms

- ■ Runga-Kutta 4th-order variable-step
- □ Hamming Pred/corr variable-step
- □ Runga-Kutta 4th-order fixed-step
- □ Euler 1st-order fixed-step

Enter Tolerance	1.0E-03
OK	Cancel

X axis	Y
■ TIME	□
□ VOUT	■
□ REF	■

Simulation Variables Store

Inputs	Save	Outputs	Save
REF	■	YPLNT	■
BIAS	□		
nlplnt.X1	□	ACTUATOR	□
nlplnt.X2	□		

Sample interval (steps)	5.0	OK	Cancel

Figure 4. The GE-MEAD simulation screen and affiliated forms

Spatial Operator Software for Intelligent Control

G. Rodriguez
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109

Summary

A new spatial operator algebra [1,2,3] provides a model-based architecture for software prototyping and development for mobile, articulated robotic systems. The algebra is a world representation and computing framework which greatly simplifies the difficult process of modeling, simulation, control, motion planning, and coordination. It provides a modeling and computational architecture for high-level planning and intelligent control. It also significantly simplifies software development. The algebra consists of a set of symbolic operators which at a very high level of abstraction summarize the very complex time and space relationships inherent in mobile, articulated multibody systems. Once set up and understood, the operations in the algebra are almost as easy to use as addition, multiplication and division in elementary high-school algebra. Symbolic complexity visible to the user is reduced by 2 to 3 orders of magnitude because each operator summarizes about 100 to 1000 more detailed symbols and operations. Each operator leads to a fast spatially recursive algorithm that performs the corresponding control computations. Computational bottlenecks are avoided because the number of arithmetic operations required grows only linearly with the number of degrees of freedom. The algebra handles configuration changes easily. Target applications include autonomous vehicle guidance and control, vehicle-assisted cargo-handling, warehouse tasks, robotic excavation, handling of unexploded ordnance, surveillance, pointing and control of platform-mounted articulated assemblies, and robotic vehicle field operations.

1 Problem Statement

Management of complexity is one of the most critical problems in software development for intelligent control of multiple robotic systems. Such systems can undergo a vast number and variety of motion and force interactions among themselves and with the environment. The use of more than one system adds new problems that are typically not present in a single system. Some operations require: dynamic configuration transients during hand-off and transportation; intermittent contact and force interactions; and mechanical changes in the objects being handled. Transitions between operating regimes require corresponding changes to the motion planning and control software.

These software changes are typically difficult to make. A significant effort is required to make software development for control and motion planning a more efficient engineering process. Approaches to control software development typically do not lead to easy software transfer. Each installation develops its own control software. This leads to a significant amount of duplication. It also leads to control software systems that are not easily operated and maintained by users not involved in the software development.

2 Features of the Spatial Operator Algebra

The spatial operator algebra provides a useful model-based architecture for software prototyping and development because: (1) it greatly reduces the symbolic complexity visible to the user; (2) it is easy to convert high-level operator expressions to computationally fast algorithms; (3) it allows algorithm and software reconfigurability over a wide range of operating regimes; (4) it provides a wide range of models for motion planning and control; (5) it provides the foundation for a task-level programming language; (6) it can be embedded in a high-level AI-based planning system; (7) it represents the world in a modular, hierarchical framework which leads to modular and reuseable software. The software allows easy transitions from planning and control algorithm design, to simulation, and to experimental hardware evaluation. Some of these features are illustrated in the figures.

Spatial operators reduce symbolic complexity visible to user

Management of complexity using spatial operators is illustrated in Figure 1. Concise operator expressions for several robot quantities are compared to the corresponding symbolic expression for the Jacobian, perhaps the simplest robot quantity typical of a robot arm. A small number of spatial operators implement not only a variety of such familiar computations as forward kinematics, Jacobian, and the mass matrix, but also other useful ones such as the Jacobian inverse, the mass matrix inverse and the operational space inertia matrix. The symbolic complexity using four different methods for inverse dynamics of a typical robot arm is compared in Figure 2. The use of spatial operators leads to a large reduction in complexity over other trigonometric and recursive descriptions.

Operator expressions map easily to fast recursive algorithms

Spatial operator equations can be easily converted to efficient numerically robust computational algorithms. The algorithms are in the same spirit as the familiar Newton-Euler algorithms [4] for inverse dynamics and for the manipulator mass matrix. However, they go further by performing a much wider range of robot computations [5,6,7,8,9]. The number of arithmetical operations grows only linearly in the number of degrees of freedom leading to efficiency for large order systems. Conversion of an operator expression into an outward recursive algorithm that goes from the base of the robot to its tip is illustrated in Figure 3.

The spatial operator algebra also enables development of new motion planning and control algorithms which represent basic state-of-the-art advances in system control. One such trajectory design algorithm for a robot manipulator is illustrated in Figure 4. With a simple set of mathematical statements, no more complex than shown in the blocks forming the algorithm diagram, the trajectory design algorithm can be developed and tested in predictive simulation. Conversion to modular software is also immediate and can be done by visual inspection.

Software editing in response to system reconfiguration is easy

The spatial operator algebra leads to modular, reconfigurable software. The spatial operator algebra algorithms are modular [7] and map to an object–oriented modular software architecture, resulting in computational algorithms ideally suited for event-driven systems. Software can be smoothly reconfigured in response to transitory changes in system configuration. Algorithm and software editing during design, or in response to transitory events, is illustrated in Figure 5.

A model-based compiler and an automated real-time code generator facilitates robot hardware implementation.

The spatial operator algebra has a built-in hierarchical architecture [3] that leads to what can be viewed as a model-driven compilation process. The model-driven compiler translates high-level spatial operators into low–level algorithms and programs. Figure 6 illustrates operation of the model-driven compiler. The compiler takes advantage of the natural map that exists between spatial operators and efficient, spatially recursive, computational algorithms. Real- time software generation is under investigation to convert the recursive computational algorithms into real–time code for hardware implementation. This is in the same spirit as available tools [10] for real-time control system automated code generation.

Computations are embedded in the highly developed filtering architecture

The spatial operator algebra algorithms perform all robot computations within the highly developed filtering and smoothing architecture, which is very easy to understand, program and debug [1]. This leads to numerical stability and robustness so that the effects of truncation and round-off errors tend to remain at a very low level. Some examples of the applicable techniques are: square–root filtering algorithms for the numerical stability of Riccati equations; monitoring of the statistical whiteness of residuals to monitor numerical error buildup, and new [11] computing architectures for concurrent processing.

3 Software Development

The spatial operator algebra has been used to develop new recursive algorithms and software for motion planning and control of single and multi-arm robotic systems. Software implementation of a large portion of the algebra has been carried out in Ada as well as in MATHEMATICA. The software has been embedded into part of an integrated telerobotic system, consisting of multiple-arms, a vision system, an operator station, and an AI planner. An object–oriented language and interpreter Thread , has been implemented [12] for interactive program development. It is used to rapidly prototype various robot task sequences as well as to plan, execute and monitor high-level operator commands.

4 Concluding Remarks

The spatial operator algebra provides a model-based architecture for fast software prototyping and development. It applies to the domain of multibody systems formed by interconnected articulated elements and subassemblies. It provides a unique hierarchical computational engine to conduct all motion planning and control computations.

References

[1] G. Rodriguez, "Kalman Filtering, Smoothing and Recursive Robot Arm Forward and Inverse Dynamics," *IEEE Journal of Robotics and Automation*, vol. 3, Dec. 1987. (See also JPL Publication 86-48, 1986).

[2] G. Rodriguez and K. Kreutz, "Recursive Mass Matrix Factorization and Inversion: An Operator Approach to Manipulator Forward Dynamics," *IEEE Transactions on Robotics and Automation*, 1990. (See also JPL Publication 88-11, 1988).

[3] G. Rodriguez, K. Kreutz, and A. Jain, "A Spatial Operator Algebra for Manipulator Modeling and Control," *The International Journal of Robotics Research*, 1990. (See also Proceedings of 1989 IEEE Conf. on Robotics and Automation).

[4] J. Luh, M. Walker, and R. Paul, "On-line Computational Scheme for Mechanical Manipulators," *ASME Journal of Dynamic Systems, Measurement, and Control*, vol. 102, no. 2, pp. 69–76, 1980.

[5] G. Rodriguez, "Random Field Estimation Approach to Robot Dynamics," *IEEE Transactions on Systems, Man and Cybernetics*, Sept. 1990.

[6] G. Rodriguez, "Statistical Mechanics Models for Motion and Force Planning," in *SPIE Conference on Intelligent Control and Adaptive Systems*, Philadelphia, PA, Nov. 1989.

[7] G. Rodriguez, "Recursive Forward Dynamics for Multiple Robot Arms Moving a Common Task Object," *IEEE Transactions on Robotics and Automation*, vol. 5, Aug. 1989. (JPL Publication 88-6, 1988).

[8] A. Jain and G. Rodriguez, "Recursive Linearization of Manipulator Dynamics Models," in *IEEE Conference on Systems, Man and Cybernetics*, Los Angeles, CA, Nov. 1990.

[9] G. Rodriguez, "Spatial Operator Approach to Flexible Multibody Manipulator Inverse and Forward Dynamics," in *IEEE International Conference on Robotics and Automation*, Cincinnati, OH, May 1990.

[10] "MATRIXx/SYSTEM_BUILD Software Package, Integrated Systems Inc., Santa Clara, CA."

[11] J. Charlier and P. Van Dooren, "A Systolic Algorithm for Riccati and Lyapunov Equations," *Mathematics of Control, Signals, and Systems*, vol. 2, no. 2, pp. 109–136, 1989.

[12] J. Beahan, "Thread: A Programming Environment for Interactive Planning–Level Robotics Applications," in *NASA Conference on Space Telerobotics* (G. Rodriguez and H. Seraji, eds.), Pasadena, CA, Jan. 1989.

Figure 1. SPATIAL OPERATORS PROVIDE A RICH VOCABULARY FOR COMPLEXITY MANAGEMENT

Jacobian

$$J = B^* \phi^* H^*$$

Jacobian Inverse

$$J^{-1} = [I - H\psi_1 K_1] D_1^{-1} H \psi_1 B$$

Manipulability Measure

$$\mu = Det[D_1]$$

Mass Matrix

$$\mathcal{M} = H \phi M \phi^* H^*$$

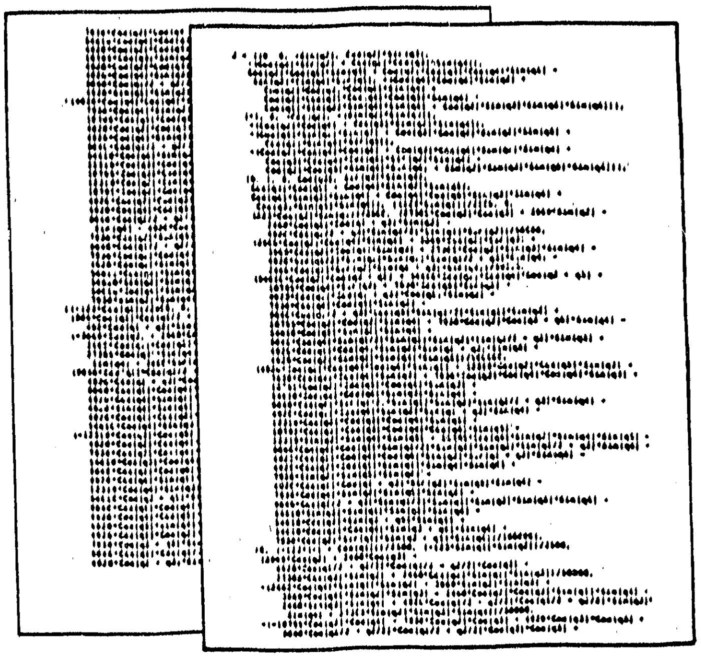
Mass Matrix Inverse

$$\mathcal{M}^{-1} = [I - H\psi K]^* D^{-1} [I - H\psi K]$$

Operational Space Inertia Matrix Inverse

$$\Lambda^{-1} = \psi^* H^* D^{-1} H \psi$$

.
.
.
.



Spatial operators provide a rich vocabulary for managing the complexity of robotics systems. This is evident from comparing their concise descriptions of most robotics quantities of interest (as illustrated on the left), with the symbolic expression (on the right) for just one of the simpler quantities such as the Jacobian for a typical robot arm.

FIGURE 2: COMPARISON OF MODELING COMPLEXITY VISIBLE TO THE USER

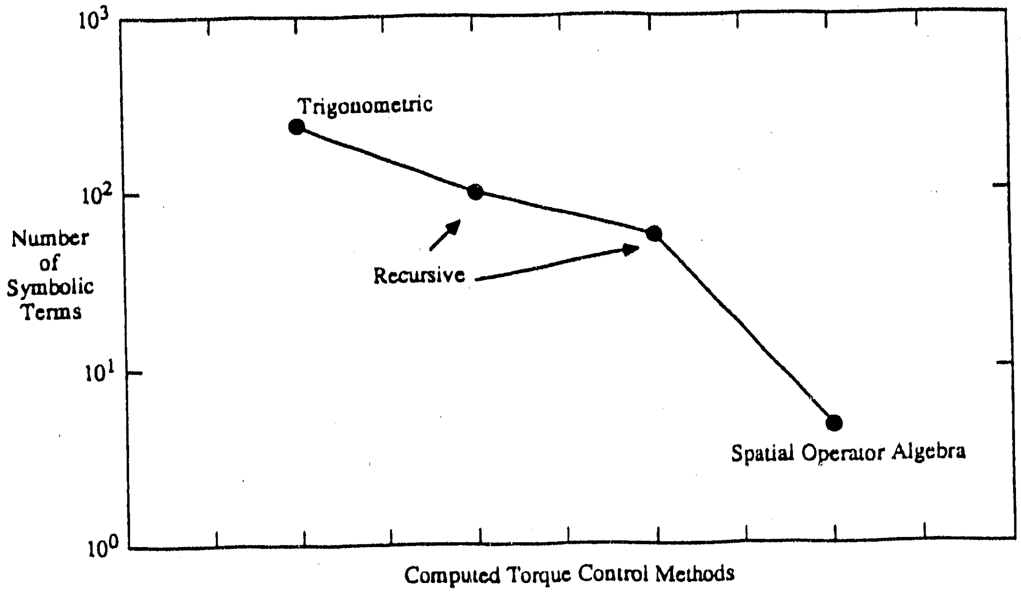
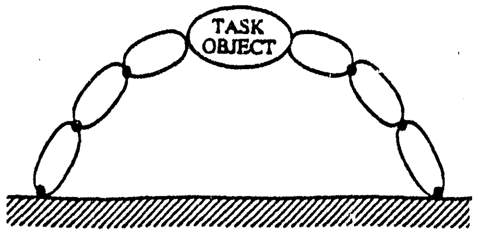


FIGURE 3: MAPPING BETWEEN OPERATOR EXPRESSIONS AND RECURSIVE ALGORITHMS

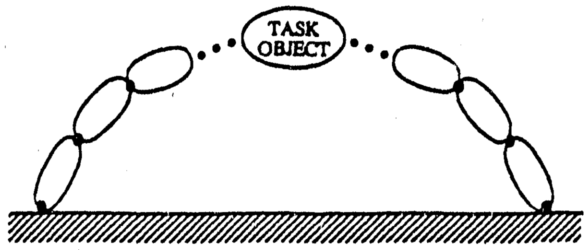
$$V = \phi^* H^* \dot{\theta} \quad \Longleftrightarrow \quad \begin{cases} \quad\quad V(n+1) = 0 \\ \textbf{for k = n} \cdots \textbf{1} \\ \quad V(k) = \phi^*(k+1,k)V(k+1) + H^*(k)\dot{\theta}(k) \\ \textbf{end loop} \end{cases}$$

Operator Expression *Computational Algorithm*

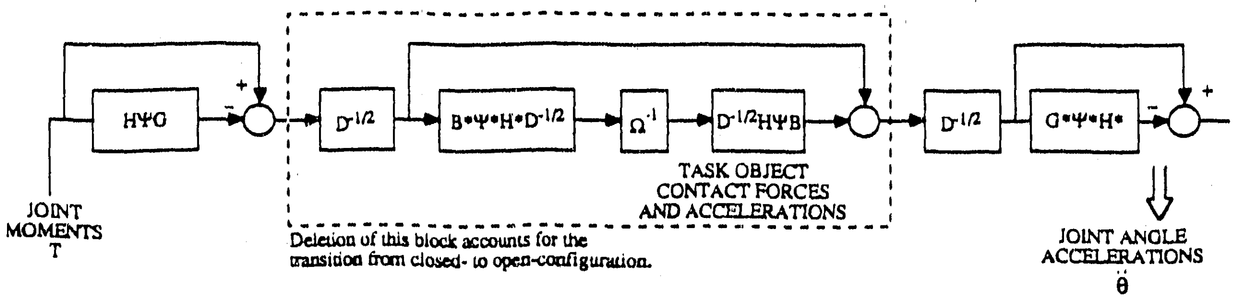
Figure 4. EASY MODEL SETUP/EDITING USING SPATIAL OPERATORS



The articulated task object together with the two robot arms represented by means of an object tree/graph. Each link of the robot is a link of the tree/graph and each node of the tree/graph represents a constrained spatial relationship at a joint. The overall graph encodes the highly complex non-linear dynamics of the overall system.

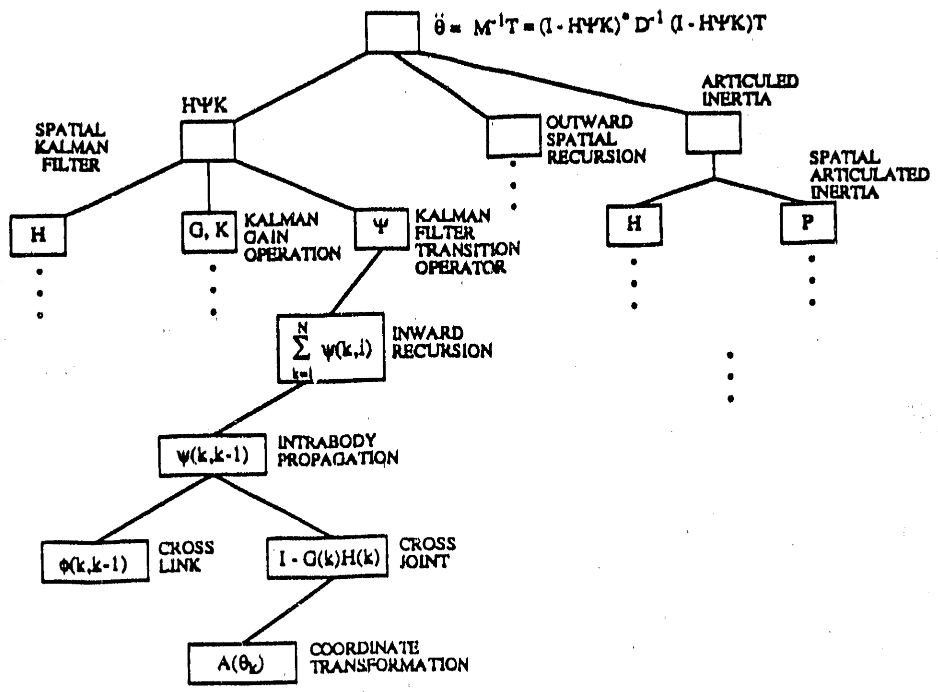


Some of the constrained relationships are transitory and evolve with the task. These transitions represent planned events as well as unexpected events. It is important that the system remain stable during all such transitions and that the underlying motion planning and control software accommodate the changes.


HΨG
+
−
$D^{1/2}$
$B^* \Psi^* H^* D^{-1/2}$
Ω^{-1}
$D^{-1/2} H \Psi B$
TASK OBJECT CONTACT FORCES AND ACCELERATIONS
$D^{1/2}$
$G^* \Psi^* H^*$
JOINT MOMENTS T
Deletion of this block accounts for the transition from closed- to open-configuration.
JOINT ANGLE ACCELERATIONS $\ddot{\theta}$

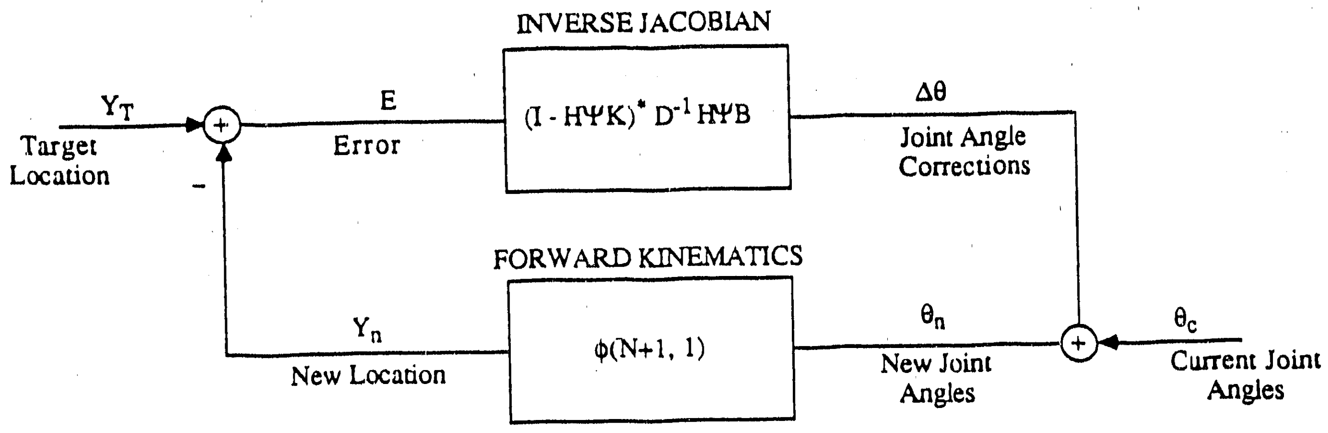
The spatial operator algebra system allows the study of these transitions and provides the basis for quick software/system reconfiguration or editing via a graphical user interface. This is useful for modeling, motion planning and control of robot systems.

Figure 5. SMART COMPILATION OF SPATIAL OPERATOR EXPRESSIONS TO EFFICIENT COMPUTATIONAL ALGORITHMS


$\ddot{\theta} = M^{-1}T = (I - H\Psi K)^* D^{-1} (I - H\Psi K)T$
HΨK
SPATIAL KALMAN FILTER
OUTWARD SPATIAL RECURSION
ARTICULED INERTIA
SPATIAL ARTICULATED INERTIA
H
G, K
KALMAN GAIN OPERATION
Ψ
KALMAN FILTER TRANSITION OPERATOR
H
P
$\sum_{k=1}^{N} \psi(k,l)$
INWARD RECURSION
ψ(k,k-1)
INTRABODY PROPAGATION
φ(k,k-1)
CROSS LINK
I - G(k)H(k)
CROSS JOINT
A(θk)
COORDINATE TRANSFORMATION

Efficient real-time code generation is possible because of the smart compilation features of spatial operators. Fast computational algorithms are easily embedded in these operators.

Figure 6. EASY SETUP OF COMPLEX MOTION PLANNING AND CONTROL STRATEGIES



The use of spatial operators to define individual blocks allows the easy set up of complex models, motion planning and control strategies. Illustrated above is the simple implementation of a trajectory design algorithm. Spatial operators are used to define individual blocks via the graphical user interface, which are then tied together in an iterative feedback loop to compute a joint and task space trajectory for the robot arm.

Single Board System for Fuzzy Inference

James R. Symon Hiroyuki Watanabe
Department of Computer Science
CB# 3175 Sitterson Hall
University of North Carolina
Chapel Hill, NC 27514-3175
TEL. (919) 962-1817, 962-1893

Abstract

The VLSI implementation of a fuzzy logic inference mechanism allows the use of rule-based control and decision making in demanding real-time applications such as robot control and in the area of command and control. We have designed a full custom VLSI inference engine. The chip is fabricated using 1.0 μ CMOS technology. The chip consists of 688,000 transistors of which 476,000 are used for RAM memory.

The fuzzy logic inference engine board system incorporates the custom designed integrated circuit into a standard VMEbus environment. The Fuzzy Logic system board uses TTL logic parts to provide the interface between the Fuzzy chip and a standard, double height VMEbus backplane allowing the chip to perform application process control through the VMEbus host. High level C language functions hide details of the hardware system interface from the applications level programmer. The first version of the board was installed on a robot at Oak Ridge National Laboratory in January of 1990.

1 Introduction

Fuzzy logic based control uses a rule-based expert system paradigm in the area of real-time process control [4]. It has been used successfully in numerous areas including train control [12], cement kiln control [2], robot navigation [6], and auto-focus camera [5]. In order to use this paradigm of a fuzzy rule-based controller in demanding real-time applications, the VLSI implementation of the inference mechanism has been an active research topic [1, 11]. Potential applications of such a VLSI inference processor include real-time decision-making in the area of command and control [3], and control of precision machinery.

An original prototype experimental chip designed at AT&T Bell Labs [7] was the precursor to the fuzzy logic inference engine IC that is the heart of our hardware system. The current chip was designed at the University of North Carolina in cooperation with engineers at the Microelectronics Center of North Carolina (MCNC) [8]. MCNC fabricated and tested fully functional chips.

The new architecture of the inference processor has the following important improvements compared to previous work:

1. programmable rule set memory
2. on-chip fuzzifying operation by table lookup
3. on-chip defuzzifying operation by centroid algorithm
4. reconfigurable architecture
5. RAM redundancy for higher yield

The fuzzy chips are now incorporated in VMEbus circuit boards. One of the boards was designed for NASA Ames Research Center and another board was designed for Oak Ridge National Laboratory (ORNL). The latter board has been installed and is currently performing navigational tasks on experimental autonomous robots [9].

ORNL will soon receive the second version of the board system featuring seven Fuzzy chips in a software reconfigurable interconnection network. The network provides host and inter-chip I/O in any logical configuration of the seven chips.

2 Fuzzy Inference

The inference mechanism implemented is based on the compositional rule of inference for approximate reasoning proposed by Zadeh [13]. Suppose we have two rules with two fuzzy clauses in the IF-part and one clause in the THEN-part:

Rule 1: If (x is A_1) and (y is B_1) then (z is C_1),
Rule 2: If (x is A_2) and (y is B_2) then (z is C_2).

We can combine the inference of the multiple rules by assuming the rules are connected by OR connective, that is Rule 1 OR Rule 2 [7]. Given fuzzy proposition (x is A') and (y is B'), weights α_i^A and α_i^B of clauses of premises are calculated by :

$$\begin{aligned} \alpha_i^A &= \max_x(A', A_i), \\ \alpha_i^B &= \max_y(B', B_i), \quad for \quad i = 1, 2. \end{aligned}$$

Then, weights w_1 and w_2 of the premises are calculated by :

$$\begin{aligned} w_1 &= \min(\alpha_1^A, \alpha_1^B), \\ w_2 &= \min(\alpha_2^A, \alpha_2^B), \end{aligned}$$

Weight α_i^A represents the closeness of proposition (x is A_i) and proposition (x is A'). Weight w_i represents similar measure for the entire premise for the i^{th} rule. The conclusion of each rule is

$$C_i' = \min(w_i, C_i), \quad for \ i = 1, 2.$$

The overall conclusion C' is obtained by

$$C' = \max(C_1', C_2').$$

This inference process is shown in Figure 1. In this example, $\alpha_1^A = 0.5$ and $\alpha_1^B = 0.25$, therefore $w_1 = 0.25$. $\alpha_2^A = 0.85$ and $\alpha_2^B = 0.5$, therefore $w_2 = 0.5$.

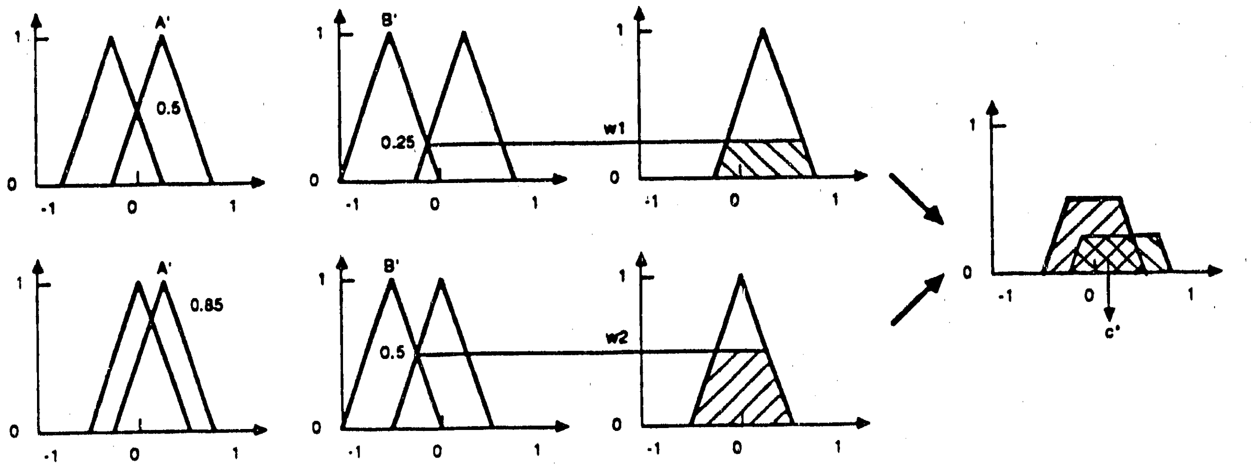


Figure 1: Inference.

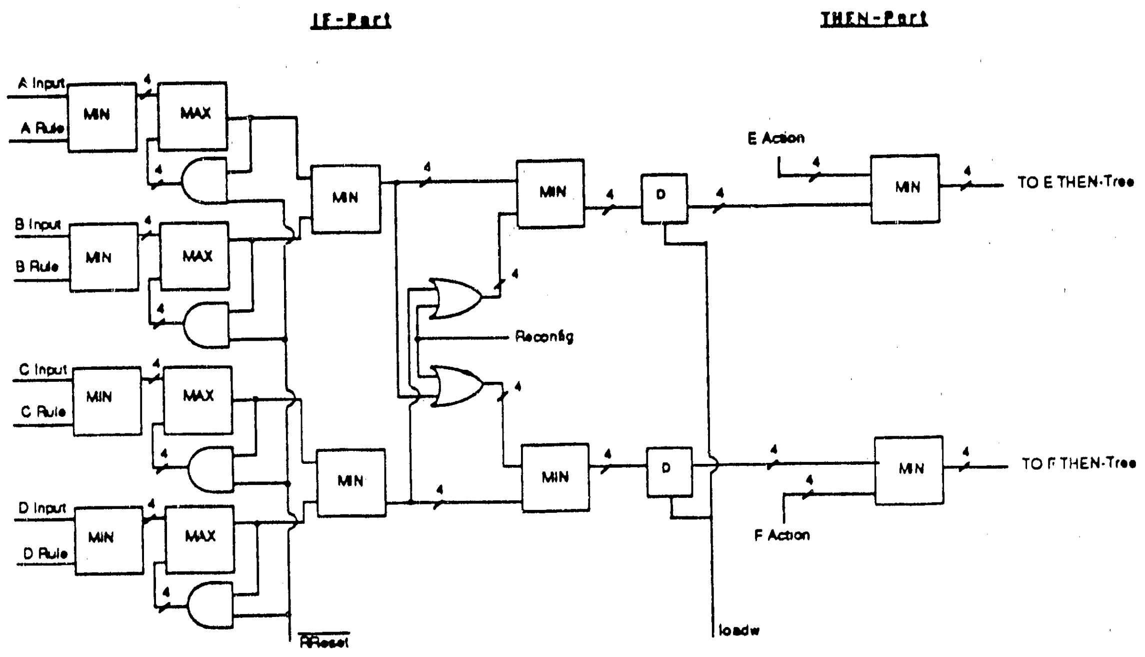


Figure 2: Fuzzy Chip Datapath.

3 Fuzzy Chip

The fuzzy logic inference engine is a fully custom designed 1.0 micron CMOS VLSI circuit of 688,000 transistors implementing a fuzzy logic based rule system. Included on chip are a programmable rule set memory, an optional input fuzzifying operation by table lookup, a minimax paradigm fuzzy inference processor, and an optional output defuzzifying operation using a centroid algorithm. The standard data path configuration is shown in Figure 2. The design has a reconfigurable architecture implementing either 50 rules with 4 inputs and 2 outputs, or 100 rules with 2 inputs and 1 output. Separately addressed status registers allow programmed control of the fuzzy inference processing and chip configuration. All the rules operate in parallel generating new outputs over 150,000 times per second.

The chip has 12 bidirectional data pins and 7 address pins for rule memory I/O. For process-control I/O, each of 4 inputs and 2 outputs has 6 pins. Each of 4 inputs has a corresponding **load** pin. The chip also has several control signals. Control signals **RW**(read high write low) and **CEN** (chip enable) are similar to that of a memory chip.

4 The System Boards

4.1 Single Chip Systems

The Fuzzy Logic system boards place the Fuzzy chip into a VMEbus environment to provide application process control through a VMEbus host. The single chip system designed for NASA Ames Research Center uses an off-the-self VMEbus prototyping board [10]. The overall configuration of the design is shown in Figure 3. In this design, the VMEbus interface is provided by the prototyping board system and needed a minimum of design for integration of the fuzzy chip. The fuzzy chip interface to the board is realized using discrete TTL parts and wire-wrapping. In the board system for ORNL, the VMEbus interface was designed by the first author and realized using a programmable logic device (PLD) and TTL parts. More robust printed circuit board (PCB) technology was used. The PCB architectural concept is shown in Figure 4. The UNIX device driver interfaces of these two boards are quite similar.

The ORNL board is designed to standard VMEbus specifications for a 24 bit address, 16 bit data, slave module as found in *The VMEbus Specification*, Revision C.1, 1985. It provides digital communication between the host and the Fuzzy chip. A large, UV erasable PLD generates the board control signals. VMEbus interface is through TTL parts. One Fuzzy Inference IC processes four 6-bit inputs to generate two 6-bit outputs. The interface with the host computer uses memory mapping to include the Fuzzy chip's I/O addresses in the application process storage space. All of the chip's memory as well as its inputs and outputs are accessed through addresses on the VMEbus so that the entire Fuzzy Logic board system responds like a section of memory.

The board's address space is 1024 bytes or 512 16-bit words in length. Most of the addresses in that space are not used by the board. The lower 128 word addresses of the board are mapped into the fuzzy chip. One hundred addresses are for rule memory. Another six addresses are mapped to four *fuzzification* tables and two status registers. The board has six addresses for I/O for the fuzzy chip, and addresses for hardware reset and board ID. On-board dip switches and signal jumpers allow the user to select the board base address comprised of the upper 14 bits of the 24 bit address, and the board's user privilege response characteristic determined by the VMEbus *address modifier* bits. Further design details are shown in Figure 5.

4.2 Multiple Fuzzy Chip System

The second version of the system board keeps the standard VMEbus interface of the first version but adds significant new capabilities. Seven Fuzzy chips communicate with each other and the host through a software reconfigurable interconnection network. Two Texas Instruments digital crossbar switch IC's

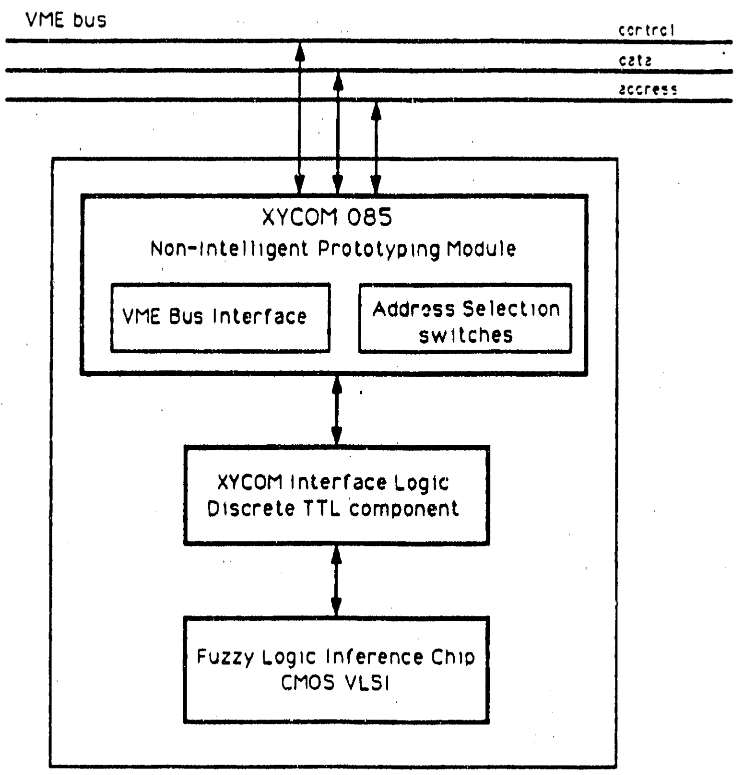


Figure 3: Single Chip System Based On Prototyping Board.

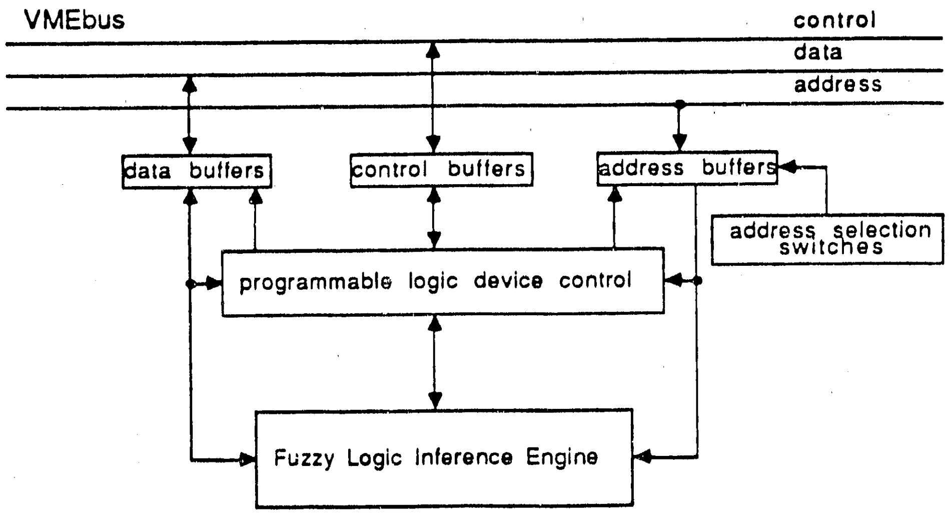

VMEbus
control
data
address
data buffers
control buffers
address buffers
address selection switches
programmable logic device control
Fuzzy Logic Inference Engine

Figure 4: Single Chip System Based On Custom PCB.

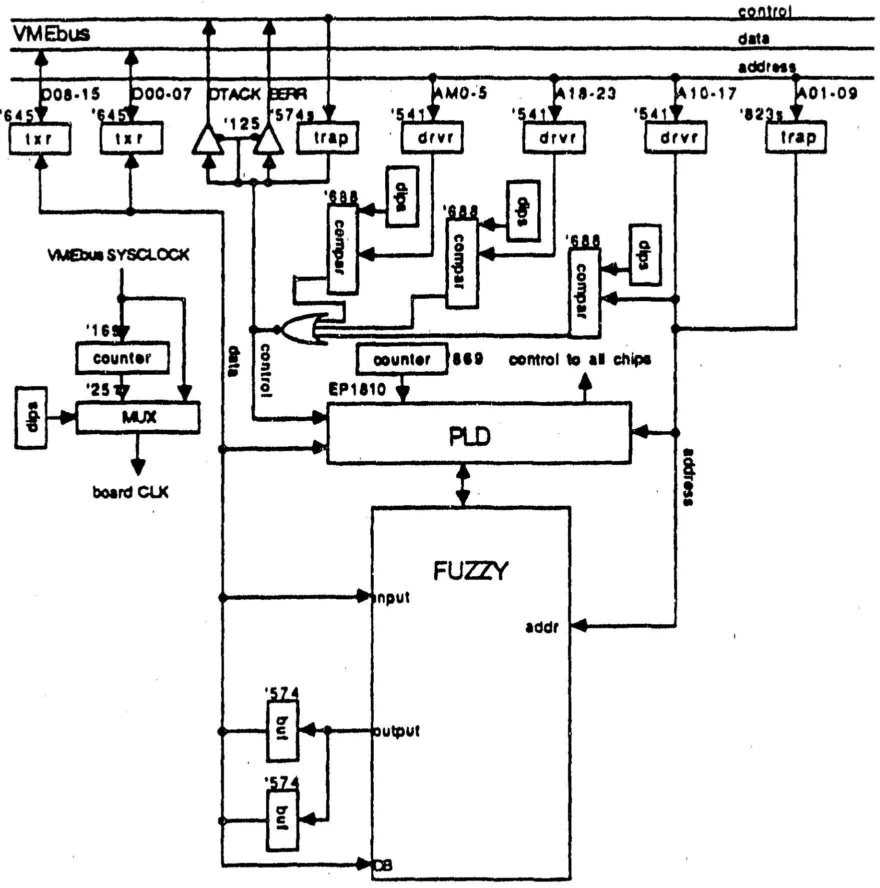


Figure 5: Details of PCB Architecture

implement the network. Any logical configuration of the seven chips may be specified in software, e.g. seven in parallel, 4-2-1 binary tree, etc. Any fuzzy output may be routed to any input. With the new board more inputs may be processed and hierarchies of rule sets may be explored. We can simulate rules with up to 16 conditions in the IF-part by using three layers of Fuzzy chips. Another application is to load multiple rule sets for different tasks in a single board. This is done by configuring multiple chips in parallel. The new printed circuit board architectural concept is shown in Figure 6.

This arrangement exploits an important feature of the Fuzzy chip. Normal input to the chip is by 6-bit integers which the chip *fuzzifies* into 64-value membership functions to be fed into the processing pipeline. The final output membership function is *defuzzified* into a 6-bit output integer. However, the chip has another mode of operation. Any input or output can bypass the [de]fuzzification process so that I/O occurs in *streaming* mode. The full 64-value input or output membership function is placed on the pins, one value per clock cycle. When an output of one chip is connected to an input of another chip (or itself), communication can be done in streaming mode without the loss of information inherent in the [de]fuzzification operations. On this system board, all inter-chip communication is done in streaming mode.

The new board also has four 64-value FIFO queues which allow final output to the host to be done in streaming mode. The application process is then free to perform its own custom operations on the full output membership functions. The final defuzzification is no longer limited to a centroid method. One can, also, generate the result in higher precision than 6 bits if necessary.

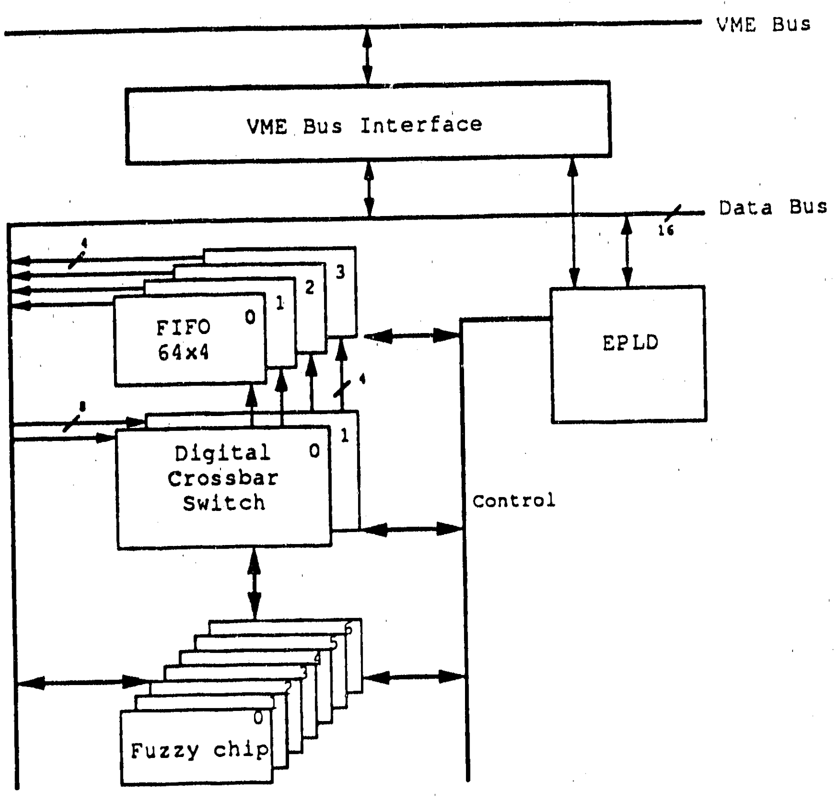


Figure 6: Seven Chip System Architecture.

The new board will be installed at ORNL in August, 1990. In addition to navigational tasks the system will be used to explore fuzzy logic control of manipulator arm functions.

5 Software Interface

High level C language functions can hide the operational details of the board from the applications programmer. The programmer treats rule memories and fuzzification function memories as local program structures passed as parameters to the C functions. Similarly, local input variables pass values to the system and outputs return in local variable function parameters. Programmers are only required to know the library procedures. Some procedures provided for the version 1 board are described in the following table.

1. *WriteRule(rulenum, ruledata)* - The rule data structure pointed to by *ruledata* is written to the board.
2. *ReadRule(rulenum, ruledata)* - Reads back into *ruledata* the rule identified by *rulenum* currently stored in the chip.
3. *WriteFuzz(fuzznum, fuzzdata)* - Fuzzification table is written to the board.
4. *StartFZIAC(inpA, inpB, inpC, inpD)* - Four inputs are sent to the fuzzy board and inference processing will be started.
5. *ReadOut(outE, outF)* - Both outputs are read from the board. Inference process will be continued.

6. *StopFZIAC(outE, outF)* – Both outputs are read from the board. Inference process will be halted.

6 Summary

We have described the architecture and associated high level software of two VME bus board systems based on a VLSI fuzzy logic chip. In addition to operating in the robot at ORNL, the single chip board is installed on a Sun-3 workstation at the University of North Carolina for further research and software development. For example, it is useful to provide an X-w.ndow based user interface to this fuzzy inference board. The complex and flexible architecture of the multiple chip board will require more sophisticated support software to facilitate exploration of various hierarchical interconnection schemes.

7 Acknowledgements

The research reported here is supported in part by Oak Ridge National Laboratory, by MCNC Design Initiative Program, and by NASA Ames Research Center.

References

[1] Corder, R. J., "A High-Speed Fuzzy Processor," *Proc. of 3nd IFSA Congress*, pp. 379-381, August 1989.

[2] Holmblad, L. P. and Ostergaard, J. J., "Control of a Cement Kiln by Fuzzy Logic," *Fuzzy Information and Decision Processes* (eds. M. M. Gupta and E. Sanchez) pp. 389-399, 1982.

[3] Kawano, K., M. Kosaka, and S. Miyamoto, "An Algorithm Selection Method Using Fuzzy Decision-Making Approach," *Trans. Society of Instrument and Control Engineers*, Vol. 20, No. 12, pp. 42-49, 1984. (in Japanese)

[4] King, P. J. and E. H. Mamdani, "The Application of Fuzzy Control Systems to Industrial Processes," *Automatica*, Vol. 13, No. 3, pp. 235-242, 1977.

[5] Maeda, Y., "Fuzzy Obstacle Avoidance Method for a Mobile Robot Based on the Degree of Danger," *Proc. of NAFIPS'90*, pp.169-172, June 1990.

[6] Shingu, T. and E. Nishimori, "Fuzzy-based Automatic Focusing System for Compact Camera," *Proc. of 3nd IFSA Congress*, pp. 436-439, August 1989.

[7] Togai. M. and H. Watanabe, "An Inference Engine for Real-time Approximate Reasoning: Toward an Expert on a Chip," *IEEE EXPERT*, Vol. 1, No. 3, pp. 55-62, August 1986.

[8] Watanabe, H., W. Dettloff and E. Yount "A VLSI Fuzzy Logic Inference Engine for Real-Time Process Control," *IEEE Journal of Solid-State Circuits,* Vol.25, No.2, pp.376-382, April 1990.

[9] Weisbin, C.R., G. de Saussure, J.R. Einstein, and F.G. Pin, "Autonomous Mobile Robot Navigation and Learning," *Computer*, Vol.22, No.6, June 1989.

[10] XYCOM, *XVME-85 Prototyping Module Preliminary Manual*, 1984.

[11] Yamakawa, T. and T. Miki, "The Current Mode Fuzzy Logic Integrated Circuits Fabricated by the Standard CMOS Process," *IEEE Transactions on Computers*, Vol. C-35, No. 2, pp. 161-167, February 1986.

[12] Yasunobu, S. and S. Miyamoto, "Automatic Train Operation System by Predictive Fuzzy Control," in *Industrial Applications Of Fuzzy Control*, M. Sugeno (Ed), pp. 1-18, 1985.

[13] Zadeh, L. A., "Outline of a New Approach to the Analysis of Complex Systems and Decision-Making Approach," *IEEE Transactions on Systems, Man and Cybernetics*, Vol. SME-3, pp. No. 1, pp. 28-45, January 1973.

On Computational Requirements for Design of Large Order, Complex Control Systems

Matthew R. Wette
M/S 198-326
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109
mwette@csi.jpl.nasa.gov

Abstract

A position presenting computational software needs for the design and test of guidance and control (G&C) systems is presented along with a description of current and proposed approaches to solving these problems. The objectives of this work are to provide advanced computational methods and prototype tools for G&C systems.

1 Introduction

Control system engineers' design problems are quickly approaching the computational and functional limits of current computer-aided control system design (CACSD) packages. In the course of designing guidance and control systems (G&C) systems at the Jet Propulsion Laboratory (JPL), we have recognized a need for advances in control system design and verification tools and techniques. This recognition has been influenced by the evidence that performance requirements for design and simulation are quickly outpacing current design and verification capabilities. This paper outlines objectives and justification for research in the development of design and verification tools and provides an overview of current and proposed approaches for developing the required capabilities.

2 Objectives

The chief objective of this work is to develop computational methods and prototype tools for the design and testing of G&C systems. These methods and tools are intended to support several needs. First, we wish to relieve the user from the burden of handling large order, complex systems in the design process. Second, we wish to bring advanced control system design techniques to the users' disposal. Third, we wish to provide a real time simulation capability for the verification of control system hardware.

The design of large order, complex control systems will require advanced control system design and analysis methodologies. To bring these methodologies to the user, a flexible environment will be needed which can support the advanced methodologies and the associated complex control system models. With this in mind, we intend to provide a design environment in which complex system representations and other information associated with advanced design techniques can be handled in a structured and efficient manner. This should be done with the user in mind. That is, however complex the system or design process may be, the designer should be isolated as much as possible from the complexities of the system representation as well as the implementation of the design algorithms.

The existence of a suitable environment for control system design of large order, complex systems would provide an opportunity to apply advanced methodologies for control system design. Based on such an implementation, we wish to provide reliable implementations of some state of the art control system design methodologies. Since efficient and reliable tools for large order systems are of interest, the algorithms and software implemented will provide robust solution techniques on advanced computational hardware.

No control system design is complete without verification tests. Hence, with the capability of advanced control system design would comes the desire for a capability for verification. Based on this need we wish to provide a capability for real time simulation of multiple flexible body systems. This would provide a means for verification of control system hardware operating in closed loop with the real time simulation.

3 Justification

Justification of the work is supported by the lack of current tools to handle current and anticipated needs. Current computer-aided control system design (CACSD) packages typically handle only simple models and simple techniques and their tools can be very inefficient and even break down for large order systems. In addition, current simulation software is inefficient and does not provide the capability for current real time simulation needs.

Current control system design tools provide only simple modeling capabilities. Typically, these models are based on polynomial expressions or are represented as regular state space models. With advances in control system design technology, system structure is becoming more complex and these simple models are becoming too cumbersome to work with. Current CACSD packages do not, in general, provide the capability to model structured uncertainty, discrete states, and other features associated with more complex methodologies. For example, a tool recently developed at JPL for designing optical control systems had to be recoded into Fortran after the Matlab version was found to be too cumbersome and inefficient due to the need for packing information into arrays. Packing and unpacking of data into two dimensional arrays has become a too-often used workaround.

Another important point to make is that currently available commercial CACSD packages are not designed to handle large order systems. The analysis and synthesis tools provided by these packages are often inefficient and even break down for large order systems. For example, engineers at JPL have experienced CACSD packages which provided, without warning, completely unreliable results for several analyses. Other packages were able to provide more reliable results, but at a substantial cost in time. As an example, a frequency response calculation for a system of order two hundred required five hours of user time on our VAX. Other inefficiencies abound. Graduate students we interviewed at Caltech working in H_∞ and μ synthesis find that typical design problems of order fifty can require several hours of computational time with current software. In addition, they often experience sensitivity problems. In general, we have found that all packages have problems. The end effect is that the user must be a master of the design process as well as its implementation.

Current CACSD packages in general do not provide efficient storage mechanisms or computational algorithms for large order systems. Often, large order systems require have special structure for which special storage methods and computational algorithms can be used to provide very efficient and accurate solutions. Examples include systems modeled with sparse or banded matrices. Sensitivity is also an issue. With the increase in order and system dynamics, specialized algorithms and system descriptions may be required which current CACSD packages cannot handle. For example, consider the typical state space system realization for a time-invariant linear system

$$\dot{x} = Ax + Bu\,, \quad y = Cx + Du$$

As the system dynamics cover a larger range the eigenvalues of A (and hence its norm) will grow, leading to large relative computational error in the modeled slow dynamics of the system. This situation can be overcome by using, for example, generalized state space realizations of the form

$$E\dot{x} = Ax + Bu\,, \quad y = Cx + Du$$

Here a wide range in dynamics (i.e., eigenvalue magnitudes) does not imply large magnitude coefficients in the system matrices as for the regular state space representation.

Development of advanced CACSD tools for design and analysis of large order systems is justified not only by its need but also by the introduction of a soon-to-be-released a FORTRAN77 subroutine library (*LAPACK*) targeted for solving large order matrix problems on vector and parallel machines. *LAPACK* has been developed by leaders in the field of scientific computing and promises to receive much support from the scientific computing community in the future. A natural front end for such a software library would be much like current Matlab-based CACSD packages. The success of these packages as an environment for the design and application of algorithms for control system design implies that a new tool geared toward large order, complex problems should include capabilities provided by Matlab-type environments.

In the area of control system hardware verification, JPL has experienced a great lack of capability in the current technology. Currently, real time simulation of spacecraft dynamics must be limited to rigid body models. Models which provide simulation of flexibility effect are too inefficient. Since control system verification is limited, designers are forced to make control system performance conservative to compensate for the uncertainty in behavior. With the emerging parallel computer hardware technologies, the pursuit of specialized algorithms for parallel architectures seems a good pursuit.

4 Approach

Our proposed approach to providing the needed capabilities requires selected research efforts in three areas. The first area involves the development of a prototype environment for handling advanced control design techniques for complex systems. The second area involves selecting a number of advanced control system design methodologies and providing efficient and reliable algorithms and software for implementing these methods. The third area involves the development of multibody simulation algorithms and software for use on state of the art computer hardware architectures.

In response to the need to provide advanced environments for development of CACSD tools we propose to work on a new version of a Matlab-type tool. The new tool would provide an environment for the development, test and application of advanced algorithms and software for control system design. The purpose for undertaking this work is to "bootstrap" efforts at producing "next-generation" environments for design of complex, large ordercontrol systems. Our hopes are that this effort will encourage further development through technology transfer.

The proposed new tool would provide some features necessary for large order, complex design problems which current Matlab-type CACSD packages do not support (to our knowledge). Possible added features include the support of user-defined data types, the use of the soon-to-be-released LAPACK Fortran subroutine library for matrix computations and the support of compiled "toolboxes" for efficiency. We feel that a well designed "core" implementation would provide a sound base for the development of advanced algorithms and software for engineering applications. The tools would be developed with the intention of allowing possible other features (graphics, expert-system front end, etc.) to be added at a later time. The development of such a tool would provide a sound base for the production of advanced algorithms and software for many engineering applications.

In order to provide users advanced control system design technologies to users, we propose to survey the state of the art in control system design. This would allow us to determine which methodologies can provide the necessary functionality and practicality for complex, large order systems. Necessary functionality includes the ability to model system uncertainties and nonlinearities. Practicality issues include the need to be reasonably implemented in a computer-aided environment and that the computer implementation be reliable and have a reasonable design turnaround time for large order systems. Once a set of methodologies has been selected, work will concentrate on collecting and developing algorithms and software for these methodologies. Importance will be placed on efficiency and reliability (e.g., numerical stability) of the algorithms and software.

Work in the development of efficient algorithms for real time simulation of spacecraft systems with multiple flexible bodies is currently underway. Current state of the art algorithms based on $\mathcal{O}(n)$, $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$ formulations are being evaluated for efficient computation on various (e.g., parallel) computer architectures. Amoung the more promising formulations is that based on the spatial algebra developed by G. Rogriguez and colleges [1].

5 Conclusion

Experience is now showing that the need for new tools for G&C design has hit a critical level. Current CACSD packages cannot satisfy current needs and effectively limit design capabilities. To satisfy the need for these capabilities we see the need to develop a *new environment* for the designing theses tools for complex, large order systems. In addition, we feel a real time simulation capability is essential to designing and testing high performance control systems. Lastly, support of research and advanced development in the proposed areas will on the long run save costs, enhance reliability and improve performance.

References

[1] G. Rodriquez, K. Kreutz, and A. Jain. A spatial operator algebra for manipulator modeling and control. In *IEEE Conf. Rob. and Aut.*, May 1989.

TRACS: An Experimental Multiagent Robotic System

Xiaoping Yun, Eric Paljug, and Ruzena Bajcsy
Department of Computer and Information Science
University of Pennsylvania
200 South 33rd Street
Philadelphia, PA 19104-6389

Abstract

TRACS (Two Robotic Arm Coordination System), developed at the GRASP Laboratory at University of Pennsylvania, is an experimental system for studying dynamically coordinated control of multiple robotic manipulators. The systems is used to investigate such issues as the design of controller architectures, development of real-time control and coordination programming environments, integration of sensory devices, and implementation of dynamic coordination algorithms. The system consists two PUMA 250 robot arms and custom-made end effectors for manipulation and grasping. The controller is based an IBM PC/AT for its simplicity in I/O interface, ease of real-time programming, and availability of low-cost supporting devices. The Intel 286 in the PC is aided by a high speed AMD 29000 based floating point processor board. They are pipelined in such a way that the AMD 29000 processor performs real-time computations and the Intel 286 carries out I/O operations. The system is capable of implementing dynamic coordinated control of the two manipulators at 200 Hz.

TRACS utilizes a C library called MO to provide the real-time programming environment. An effort has been made to separate hardware-dependent code from hardware-independent code. As such, MO is used in the laboratory to control different robots on different operating systems (MS-DOS and Unix) with minimal changes in hardware-dependent code such as reading encoders and setting joint torques.

TRACS utilizes all off-the-shelf hardware components. Further, the adoption of MS-DOS instead of Unix or Unix-based real-time operating systems makes the real-time programming simple and minimizes the interrupt latencies. The feasibility of the system is demonstrated by a series of experiments of grasping and manipulating common objects by two manipulators.

1 Introduction

An intelligent robotic system consists of multiple agents such as manipulators, end effectors, sensory devices, controllers, environments, etc. The operation of such a system requires real-time coordination of multiple agents. Coordination involves many research issues and can be interpreted differently from different perspectives. A recent NSF sponsored workshop on coordination and coordination theory [8, 5] suggested five definitions of coordination. This lack of a unique definition of coordination signals a need for understanding and investigation of coordination problems. With respect to a multiagent robotic system, those problems include real-time dynamic coordinated control of multiple manipulators and real-time communication among agents of the multiagent robotic system. Underlying those two problems is the development of a real-time programming environment and hardware architecture for design and implementation of coordinated control algorithms.

This paper describes an experimental multiagent robotic system, called TRACS, developed at University of Pennsylvania. TRACS is aimed to primarily address real-time coordinated control of multiagent systems. Two PUMA 250 manipulators, together with end effectors and sensors, are utilized in a testbed system. An IBM PC/AT is chosen as the host computer for the development of real-time programming environment.

Other real-time robotic systems include Chimera II [10], Condor [7], SPARTA [3], SAGE [9]. TRACS has the advantage of being cost-effective, simple to use, free from staff support, and easy to transfer to other labs and field applications.

2 System Description

As discussed in the introduction section, we are interested in two problems related to coordination of a multiagent system: control and communication. Realizing the difficulty of the problems, two independent yet integrated approaches are taken towards understanding of coordination. In the first approach, the effort is focused on coordinated control algorithms while the effect of communication delays on coordination is intentionally minimized. This is made possible by considering fewer agents so that one processor provides adequate computational needs. Interprocessor communications are thus eliminated. TRACS mentioned early is a result of this approach and it will be addressed in detail in this paper. In the second approach, the emphasis is placed on real-time communication of a distributed system. Towards this end, a real-time kernel for distributed system, called TIMIX, has been developed in the GRASP laboratory [4].

2.1 Hardware

The principal agents of TRACS are two PUMA 250 robot manipulators. The goal of the system is to dynamically coordinate motions of the two manipulators to perform cooperative tasks in their common workspace. Due to their primitive computing resources, the original Unimate controllers of PUMA robots are not capable of implementing dynamic coordinated control algorithms. A new controller is built. However, to eliminate unnecessary hardware constructions, the power amplifiers, D/A converters, and encoder decoding/counter circuits of the Unimate controllers are retained.

The following considerations are taken into account before choosing a controller architecture. Firstly, distributed systems should be avoided to eliminate communication delays as the emphasis of this study is on coordinated control. Single processor and coprocessor architectures are preferred. Of course, the choice is constrained by the current technology of microprocessors (speed) as well as available budget. Secondly, a simple operating system is preferred which provides such basic functions as file server, editing, compiling, and debugging and which can provide, after minimal effort, a real-time programming environment. With this regard, Unix should be on the low end of preference list. Thirdly, the interface to manipulators and sensors should be simple and standard.

After a thorough evaluation, an IBM PC/AT with the MS-DOS operating system is chosen as the host computer. This choice satisfies the above requirements, namely the PC-AT, even with a coprocessor, is not a distributed system, MS-DOS is a simple operating simple that provides basic functionality and a real-time environment, and off-the-shelf technology is readily available for sensor interfaces.

The real-time computational burden of the coordinated dynamic control algorithms requires the addition of a coprocessor to the host computer. An AMD 29000 high speed floating point coprocessor board by YARC, INC. was chosen to satisfy this requirement. It is a single processor board with shared memory and I/O space on the PC-AT system. It comes with a C compiler.

The hardware architecture of this system is shown in Figure 1. The PC-AT communicates with the Unimate controllers through a parallel interface. The exchanged information includes angular encoder readings from the joints to the host computer and control commands from the host computer to the Unimate controller. The end-effector control information is communicated through an analog interface. The host computer can also communicate with other machines, if needed, through an Ethernet interface.

The end-effectors are custom-made for the experiments. Currently, two end-effectors have been made, a multi-configurable gripper and an open palm. The open palm is now being used for experiments in coordinated control where the two arms must manipulate large objects by maintaining a specified internal force between the palms.

2.2 Software

The development of new control algorithms is the purpose of this system. The system's software environment must allow the programmer to easily incorporate existing software, such as kinematics and gravity compensation. It must also provide easy access to information from the agents and sensors, and to easily communicate the algorithm's results. It should be based on a portable language. The system's underlying operation should be transparent to the normal user. However, if a programmer must make changes to the system, because, for instance, by the addition of a new sensor board, then the underlying system software should be simple enough to allow this without a major effort.

These requirements are satisfied by using the MO control structure which is written in the C language [2]. MO is a library of control package which provides a real-time scheduler and a virtual agent interface. Because of the virtual interface, MO is used throughout the laboratory for real-time control applications, independent of hardware.

The PC-AT host computer is configured as a real-time system by use of its clock interrupt. Each interrupt will cause the PC-AT to execute the code that gathers new data from the manipulators and sensors into shared memory, signals the AMD 29000 coprocessor to begin calculations with this new data, and output the latest control result from the AMD 29000 to the manipulator actuators. The time between interrupts is used by the PC-AT to perform user interface tasks. The AMD 29000 executes its code, the control algorithm in the MO control structure, at the servo rate. Figure 2 shows the pipeline timing table of the Intel 80286 and AMD 29000. Thus the underlying system is not formidable to the programmer who must reconfigure it.

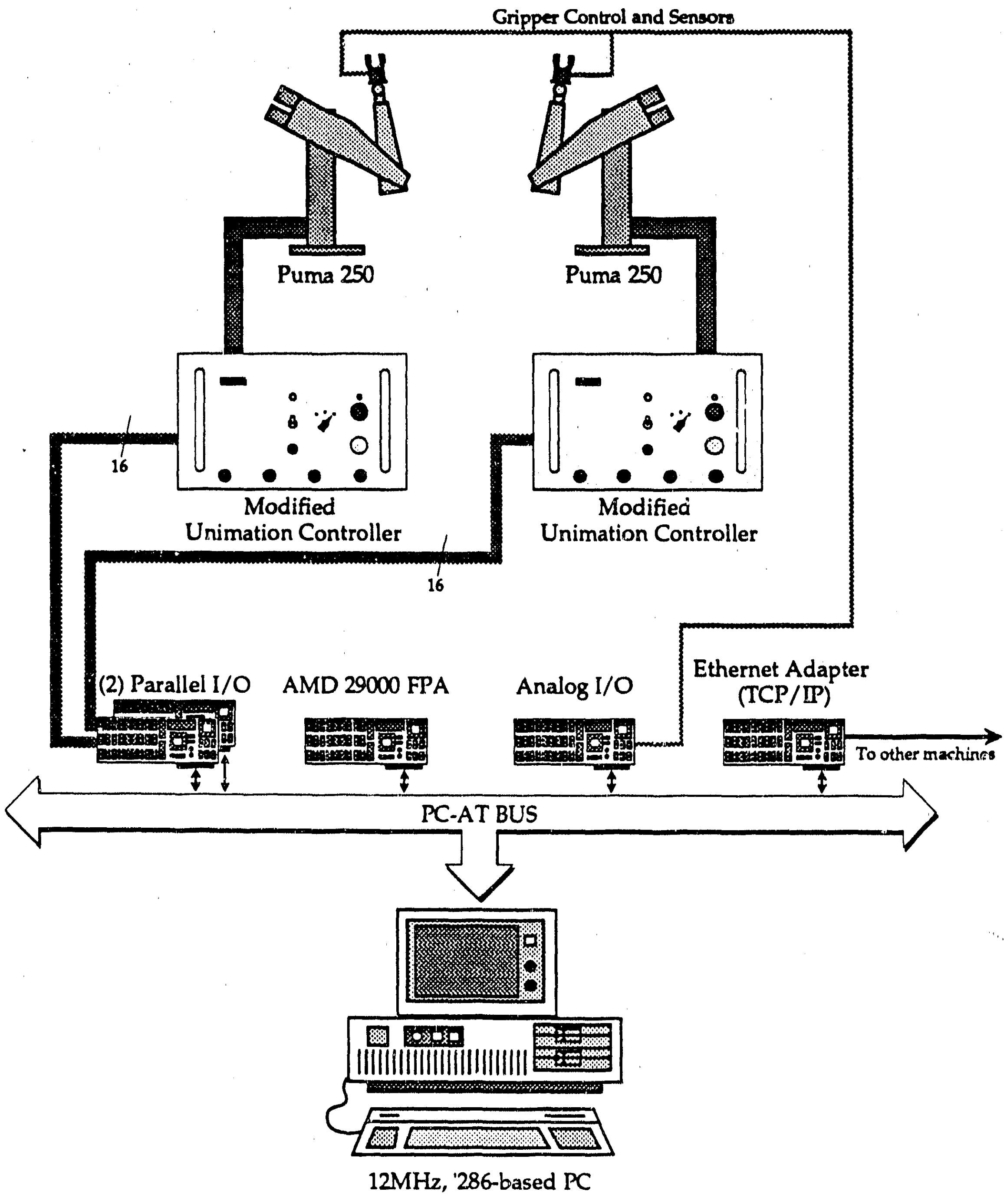

Gripper Control and Sensors
Puma 250
Puma 250
16
Modified Unimation Controller
Modified Unimation Controller
16
(2) Parallel I/O
AMD 29000 FPA
Analog I/O
Ethernet Adapter (TCP/IP)
To other machines
PC-AT BUS
12MHz, '286-based PC

Figure 1: TRACS Hardware Architecture

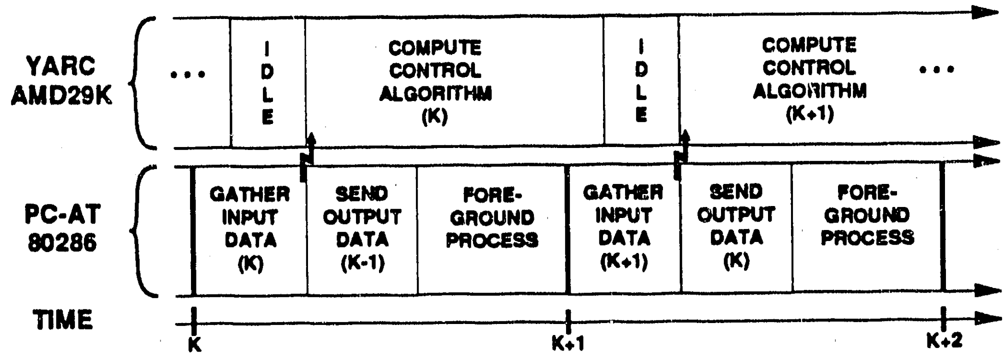


Figure 2: Timing Table of the Intel 80286 and AMD 29000

3 Implementation of Two-Arm Coordination

A series of experiments are conducted to test the functioning of the TRACS system. The tasks of the experiments are grasping, lifting and transporting objects by using two manipulators. The performance of those tasks requires dynamic coordination of the two manipulators, which in turn requires the understanding of two handed grasping and coordinated control of two manipulators.

Two handed grasping is concerned with the process of approaching the object, detecting contacts, evaluating the grasping configuration, determining the grasping force, and applying the grasping force to the object. An important step is the evaluation of the initial grasping configuration. Due to the dependence on friction, certain hands/fingers configurations can not guarantee stable grasping. In experiments, the end effectors (or hands) are simply flat surface palms instrumented by force sensors. Using such palms greatly reduces the uncertainty of contact normals. Regardless of the local geometry of the object at the contact point, the contact normal is that of the palm which is known to the controller. Given the coefficient of friction, an algorithm has been established to evaluate grasping configuration based on the relative position and orientation of the two palms. In the 2-dimensional case, the algorithm is based on the relative offsets of the two palms in x and y directions, and the relative angle between the two palms. It is noted that this algorithm does not dependent on the shape of the object and the exact contact points on the palms.

Having stably grasped the object, the next issue is coordinated control of the two manipulators. This problem has been theoretically studied by various groups including [13, 6, 1, 11]. Nevertheless, there is little experimental work in this area. Using TRACS, it is straightforward to implement and test any control algorithms. Coordination of two manipulators requires the simultaneous control of the Cartesian position and the interaction force [12]. The challenging problem here is the development of force control methods and the associated stability analysis. In the TRACS experiments, the problem is further complicated by the unilateral constraints, namely, each palm can push but can not pull the object. A solution that utilizes a nonlinear decoupling coordination algorithm which allows independent control of force and position is being studied. The experiments show that TRACS is capable of implementing dynamic coordinated control algorithms.

4 Conclusion

This paper described an experimental real-time control system, TRACS. Though the system was developed to control a multiagent robotic system, it is applicable to other real-time control systems. In particular, it is attractive to control of mobile platforms such as vehicles due to the portability of the system.

TRACS uses all off-the-shelf hardware components to reduce the cost and to expedite the technology transfer. The adoption of the simple MS-DOS operating system simplifies real-time programming, minimizes interrupt latencies, and reduces response overheads. Using C library to support the real-time programming environment and to implement real-time control algorithms makes the system portable and easy for distribution.

5 Acknowledgement

The authors wish to thank Filip Fuma and John Bradley for technical design support. This work was in part supported by: Airforce grant AFOSR F49620-85-K-0018, Army/DAAG-29-84-K-0061, NSF-CER/DCR82-19196 Ao2, NASA NAG5-1045, ONR SB-35923-0, NIH grant NS-10939 -11 as part of Cerebro Vascular Research Center, NIH 1-RO1-NS-23636-01, NSF INT85-14199, NSF DMC85-17315, ARPA N0014-88-K-0632, NATO grant No. 0224/85, DEC Corp., IBM Corp., and LORD Corp.

References

[1] Samad A. Hayati. Position and force control of coordinated multiple arms. *IEEE Transactions on Aerospace and Electronic Systems*, 24(5):584–590, September 1988.

[2] Gaylord Holder. Mo robot control software. December 1989.

[3] J. Ish-Shalom and P. Kazanzides. SPARTA: multiple signal processors for high-performance robot control. *IEEE Transactions on Robotics and Automation*, 5(5):628–640, October 1989.

[4] Robert B. King. *Design, Implementation, and Evaluation of a Distributed Real-Time Kernel for Distributed Robotics*. Technical Report MS-CIS-90-40, GRASP LAB 220, Dept. of Computer and Information Science, University of Pennsylvania, 1990.

[5] Joshua Lederberg and Keith Uncapher. *Towards a National Collaboratory*. Technical Report, National Science Foundation, The Rockefeller University, March 17-18, 1989. Report of an Invitational Workshop.

[6] Y. Nakamura, K. Nagai, and T. Yoshikawa. Mechanics of coordinative manipulation by multiple robotic mechanisms. In *Proceedings of 1987 International Conference on Robotics and Automation*, pages 991–998, Raleigh, North Carolina, 1987.

[7] S. Narasimhan, D. M. Siegel, and J. M. Hollerbach. CONDOR: an architecture for controlling the utah-MIT dexterous hand. *IEEE Transactions on Robotics and Automation*, 5(5):616–627, October 1989.

[8] NSF-IRIS Review Panel. *A Report by the NSF-IRIS Review Panel for Research on Coordination Theory and Technology, June 26, 1989.* Technical Report, National Science Foundation, 1989.

[9] Lou Salkind. The SAGE operating system. In *Proceedings of 1989 International Conference on Robotics and Automation*, pages 860–865, Scottsdale, Arizona, May 1989.

[10] David Stewart, D. Schmitz, and P. Khosla. Implementing real-time robotic systems using CHIMERA II. In *Proceedings of 1990 International Conference on Robotics and Automation*, pages 598–603, Cincinnati, Ohio, May 1990.

[11] T. J. Tarn, A. K. Bejczy, and X. Yun. New nonlinear control algorithms for multiple robot arms. *IEEE Transactions on Aerospace and Electronic Systems*, 24(5):571–583, September 1988.

[12] Xiaoping Yun, T.J. Tarn, and A.K. Bejczy. Dynamic coordinated control of two robot manipulators. In *The 28th IEEE Conference on Decision and Control*, Tampa, Florida, December 1989.

[13] Y. F. Zheng and J. Y. S. Luh. Control of two coordinated robots in motion. In *Proceedings of 24th IEEE Conference on Decision and Control*, pages 1761–1765, Ft. Lauderdale, Florida, December 1985.

Appendix I

AGENDA

Workshop on Software Tools for Distributed Intelligent Contol

July 17-19, 1990

Monday (July 16, 1990)

Arrival at Workshop hotel

Tuesday (July 17, 1990)

0730	Breakfast and registration in the South Bounty room	
0830	Welcome M. Barbee	C. Herget
0845	DARPA Program on Domain Specific Software Architectures	E. Mettala
0900	Workshop Objectives	J. James
0915	Architectures/Environments to Support Advanced Weapon Crew Station Automation	N. Coleman
0930		R. Shumacher
0945		J. Chandra
1000	Break	
1030	A Software/Hardware Environment to Support R&D in Intelligent Machines and Mobile Robotic Systems	R. Mann
1045	AI Software Architecture and Tools for Process Control	D. Lager
1100	Concept for a Reference Model Architecture for Real-Time Intelligent Control Systems (ARTICS)	J. Albus
1115		K. Baheti
1130	Discussion	
1200	Lunch	
1330	Declarative Hierarchical Controllers	W. Kohn
1345	The Rocky Road to "Standardized" Software Engineering Environments for Military Vehicle Management Systems	C. Hall
1400	Challenges of Providing a General-Purpose Environment for Building Intelligent Control Systems	M. Fehling
1415	Modeling Intelligent Concurrent Control	A. Nerode
1430	TRACS: An Experimental Multiagent Robotic System	X. Yun
1445	Conceptual Programming	R. Hartley
1500	Break	
1530		R. Schappell
1545	Hierarchical Heterogeneous Symbolic Control: Lessons Learned From TEXSYS	B. Glass

Tuesday (continued)

1600	Tool Needs For A Behavior-Based Approach To Distributed Intelligent Control	S. Harmon
1615	A Performance Based Methodology and Tool Set to Support the Engineering of Real-time Intelligent Systems	T. Hester
1630	CASE Products for Knowledge Based Systems Design and Development in Ada	J. Greenwood
1645		R. Jones
1700	Discussion	
1730	Break	
1830	Dinner	
2000	Reception	

Wednesday (July 18, 1990)

0730	Breakfast	
0830	CACE-III Expert System	D. Frederick
0845	Concurrent Processing Environments for Distributed Intelligent Control Systems	D. Birdwell
0900	Computer-Aided Control Systems Engineering	D. Gavel
0915	On Computational Requirements for Design of Large Order, Complex Control Systems	M. Wette
0930	Spatial Operator Software for Modeling, Motion Planning and Control	G. Rodriguez
0945	Integrating Controls System Design with Systems and Software Engineering	M. Rimvall
1000	Break	
1030	Safety and Reliability of Process Control Software	N. Leveson
1045	Requirements of Intelligent Control Systems	S. Natarajan
1100	VERDI: A Visual Environment for Distributed Systems Design and Simulation	C. Potts
1115	Real-Time Issues in Distributed Operating Systemsand Databases	J. Liu
1130	Discussion	
1200	Lunch	
1330	Language Development Systems in Support of Software Engineering	V. Heuring
1345	A Hierarchical Approach to Specification and Fault-Tolerant Operating Systems	J. Caldwell
1400		R. Hayes-Roth
1415	Large-Scale Distributed Control Systems	J. Maitan
1430	VLSI Fuzzy Inference Chip and Single Board System	H. Watanabe
1445	Applying a Computer Aided Prototyping System to the Software of an Autonomous Underwater Vehicle	T. Bihari
1500	Break	
1530		C. Barrett
1545	Software Tools For Lower Echelon Systems Development	D. Klose

Wednesday (continued)

1600	UAV JPO Interest	K. Thurman
1615	Research at Army Research Office	D. Hislop
1630	Discussion	
1645	Break into working groups, instructions for Thursday's meeting	M. Barbee
1730	Break	

Thursday (July 19, 1990)

0730	Breakfast
0830	Working Groups
1200	Lunch
1330	Recommendations of the Working Groups
1730	Break
1830	Dinner

Appendix II

<u>Group Summaries:</u>
This appendix contains brief summaries of each of the five working groups as recorded on flip charts.

Group 1

Jim Greenwood
Reinhold Mann
David Lager
Swami Natarajan
Norm Coleman
Dean Frederick
Donald Sasseman

NEAR TERM DEVELOPMENT ITEMS

<u>D.S.S.A.</u> <u>S.E.E.</u>

- C^2
  - Process control
  - Simulation

- Vehicle Management
- Arch Framework/Delivery
- Framework for tools to develop

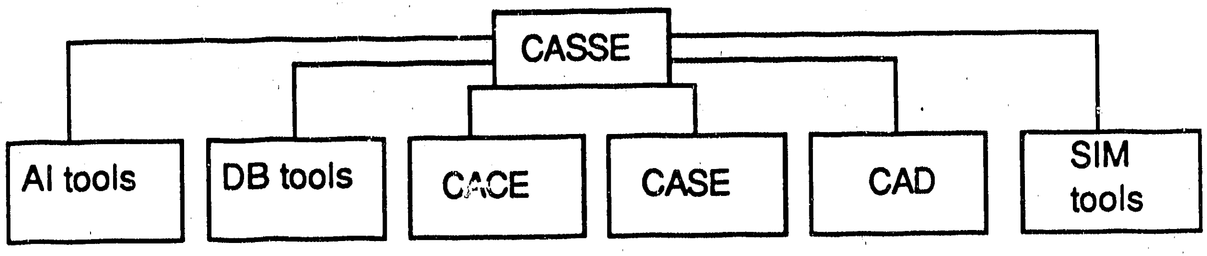
<u>Tools</u> - Softwear System Engineering

- Engineering Process
- Targeted to develop for a domain

<u>Common Architecture</u>

1) Tools for Engineering Process
   - Software Systems Engineering
   - Case - SIM
   - A I - DB
   - CACE - Management

2) Delivery System
- Candidate Architecture
  - Vehicle Sys
  - C^2



Tool Integration

Components
- O/S
- DB
- Develop libraries

- Tools need to be driven
- Tool integration - pick some and apply in integrated fashion
- Incremental expansion of tools
- New tool development

Priorities - Tools Builds

1) Build CASSE
Computer Aided Software System Engineering
- MegaProgramming?
- S.E.E.

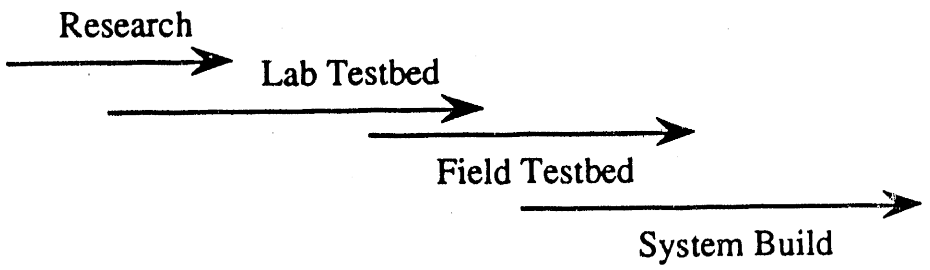
2) Tools for Exploring Delivery Architectures
- Design
- Research
- Test/Eval/Simulate
- Measure

- CASyE
- Terrain Reasoning
- Matrix_x
- Real-time O/S

- O.O/OS - Choices
  - Posix
  - VxWorks
  - OS/O$_1$/PSOS
- G^2/ABE
- PRS/AOB
- <u>VERDI</u> - Distributed Systems
- <u>Simulation Tools</u> - SIMSCRIPT
- <u>Managment Tools</u> - MRP
- <u>CASE</u>
- <u>CACE</u>
- Methodology Tools
- AI tools - KEE ART, OPS5
- Communication Tools

<u>What is the testbed?</u>

- End item target for testbed'ed systems
- Development Testbed (Lab)
  - feeds a test bed
  - feeds real system build



Group seems <u>SYSTEMS</u> oriented

<u>Candidate Drivers - Target Systems</u>

- AFAS - Mobile artillary
- Blk3 Tank -
- C^2 -
- U$_x$V -

<u>AFAS</u>

- Semi-autonomous artillary piece

- Men-machine are system
- Intelligent in support of C^2 and operations
- Reactive
- Real-time -
- Integral-distributed/Control/Intelligence
- Hierarchical

<u>Test Requirements for AFAS</u>

- Wargame simulator
- Simulation drives for system
- SIMNET-like (mand & system)
- Evaluate system and software architecture
- Evaluate reasearch and ideas - prototype
- <u>Tools</u> for gen/eval

<u>Field Tested</u>

- Real platform
- Evaluate architecture

<u>Long-Term Research</u>

1) Knowledge representations
   - fusion
   - machine perception
2) Languages
   - real-time
   - parallel __?___
3) Compiler Tec.
   - parallel detection ______
   - visual to code
4) Data Bases Systems - of Population
   - object oriented
   - distributed
5) Expert Systems - Reasoning Strategies
6) Human Computer Interactions - HCI
   - voice recognition
   - virtual reality
7) Control Theory
   - other than linear
   - stability

Group 2

Ken Thurman
Scott Harmon
Roland Jones
Vince Heuring
Randy Shumaker
Mike Fehling
Colin Potts
Fred Hadaegh
Guillermo Rodriguez

<u>Positions</u>

I. At least three tiers: (languages different at each tier)
Top - global planner/assessment
Intermediate - housekeeping/system integrity. Monitoring and integrity management for robustness.
Low order on board:
control system (auto-pilot)

- *levels of abstraction*
  functional task
- *interface between tiers*
- *languages at each level*

II. Focus on specific end product.
Unmanned system
But related to the concept as a whole

III. Critical design features such as time have to be incorporated
Time as a specification issue

IV. Human interfaces must be integral to whatever we produce (at least a 2-agent *[man/machine]* problem requiring distinctly different technologies)

V. Prototyping test bed is necessary *(and available in some form).* Configurations control mandatory

Main Issues

1) Bound the problem: 40-45 briefs. *Probably 300 others out there.*
2) Est. functional relationship of a specific entity *(unmanned tgt system)*
3) What is technology transfer potential with other areas?
4) Product plan - including technology and industry plan *(flow from R&D to user)*
5) Way to specify tasks which incorporate notions of time, function, events.

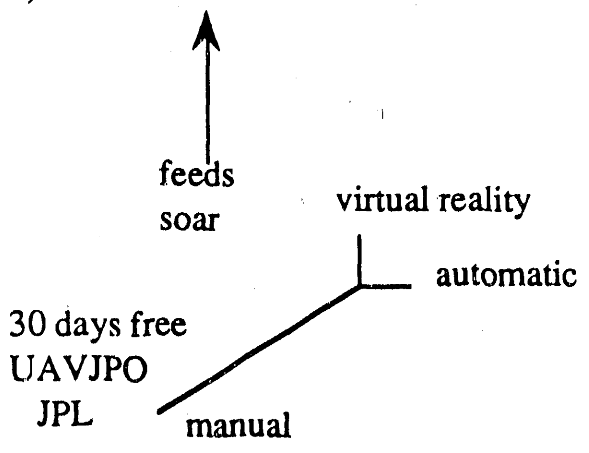
Recommendations

1) State-of-the-art review of distributed intelligent control theory

$200-300K for six months
Tools/database
standards
Analyse/characterize

Goal - whats needed
Experience - what's been done
Deficit - what is missing
Assets - what do we have available?
- what can we do?

2)



Est. functional relationship for unmanned systems

3) Review technology transfer potential
   other programs
   existing tools

$200-300K for six months

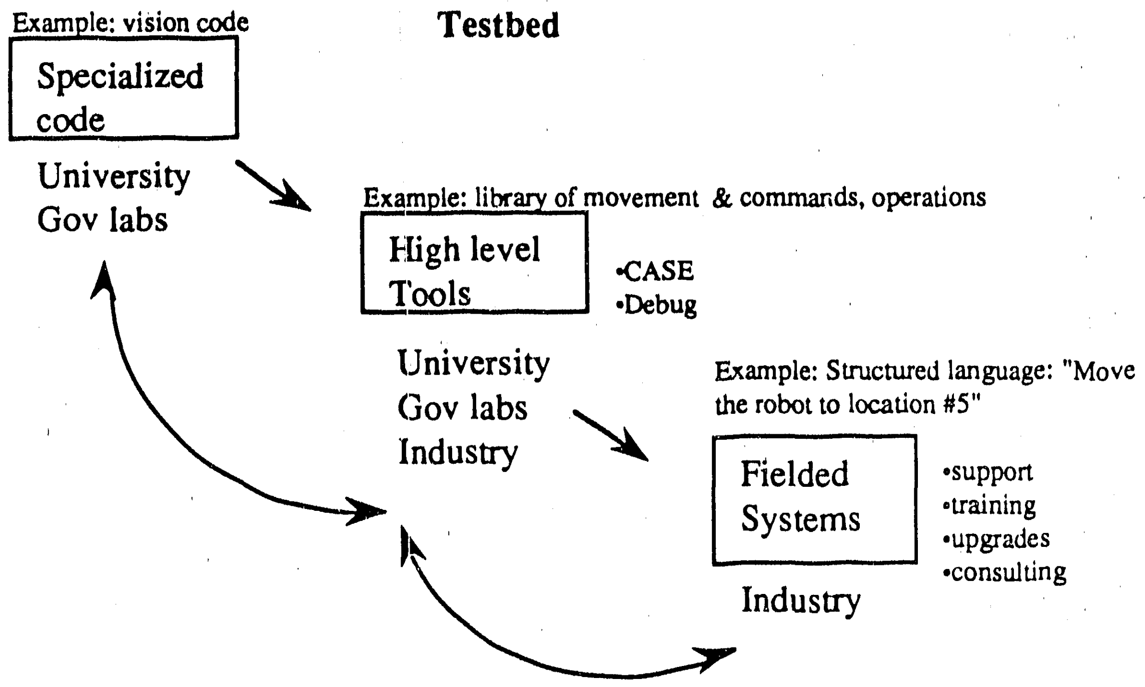
4) Languages to specify tasks which incorporate notions of time, function and events at each level

MEGA $K

<u>Distributed Test Beds</u>

5) Prototype should not be one specific hardware at one location. Establish agent to manage configuration control.
   - To include stimulation, simulation, emulation . . .animation

6) Project Plan



7) Next meeting
   - report out on soar
   - develop "accurate"budget and program

End Product

UNMANNED AIR VEHICLE:

1) Systems run by 20-300 people and want to reduce to 2 people (4-5 people in aircraft, reduce to 1)
2) Totally autonomous vehicle with various degrees of smarts: i.e.
   - (A) Give general directions, when can't find within parameters, opens up and finds anything and asks permission
   - (B) virtual reality system: Man operates, thinks he is in vehicle (using goggles) - telepresence

Group 3

Doug Birdwell
Thomas Bihari
Jagdish Chandra
Hiroyuki Watanabe
Matthew Emerson
Jacek Maitan
Guillermo Rodriquez

Modeling And Simulation Environment

1) Standardized and integrated w/previous work - 40 man years
   - communications
   - databases
2) Required features of implemntation - 100-1000 man yrs.
   1/4 billion $$
   - Equivalent to writing good operation system and support system
   - Soft target architecture
     like META-compilers
   - User definable multiple views
   - Hierarchiacal executable specification languages
     full life cycle (Like VLSI CAD systems)
   - Given to everybody supported long term - 5 man years per year
3. Validation tools
4. Module techniques to support reusability

UAV
Block 3 Tank
AFAS

- Representation should focus on test bed condidates
- Programmable interface so that existing tools can be used in intelligent design
- Two test beds avoids inbreeding
- Two different problems

- Goal is to get commercially viable product win-win for University and Industry
- Vehicle with incremental improvements and supporting test be dor software tools
- Must be rich enough - have really distributed

<u>Two Test Beds</u>

<u>Equal Priority and Funding</u>

1. UAV - Automatic landing system - Reusability - demonstration $5 million
2. UUV - Demonstrate ability of multiple groups to develop Cooperating modules for the same platform(s) $5 million

<u>Long Term Research</u>

Policy: Decouple long term research from short term deliverables

- More theory/algorithms of Distributed Intelligent control
- More work in large distributed real-time computing systems (full life cycle - specifications to debugging)
- Fund interdisciplinary education
  CS, systems and controls
- Fund Technology Transfer - tight coupling between University and Industry
- Encourage interaction
- Cleaning house for information of ideas on DIC

Group 4

Wolf Kohn
James Caldwell
Jane Liu
John James
Magnus Rimvall
Anil Nerode
Alan Laub
Matthew Wette
Chris Barrett

Long Term

$5.5 M/5 yrs. (inc. manpower, software, hardware)

Short Term

$14.5 M (manpower only) all year
$ 3.5 M hardware, software
$? miliatry equipment

Priority (3)
    At least one team for each item
    Near term is minimum plan
    Time (1) 2 years for initial system

Long term

Incorporate real-time research -
Co-ordinate, form bond with real-time

National Labs should be included long term
Funding for #3a - 500K/year, 100K/project
    3b - 600K/year, 300K/project
        (incl manpower, software, hardware)
    5 years for 1st results from grad students

Short Term

Find representation - $1 M/year 2 teams
Software house to do it

Develop, code programmable translator or integrate prog. trans.
10 man yrs @ $300K/year

Field engineers interface to university 1 man/yr

Meetings, travel, incidentals 3 years @ 50K/year
doesn't include capital cost-software, hardware etc.

COSTS

<u>Hardware</u> $30K/per man 10 sets

Controls tool	30K each
Simulation	
Database	
Case tool	
System engineering	
Specification	
Language env. (ADA)	
Cross compiler	
Networking	
Symbolic tools	

Group 5

Thomas Hester
Richard Quintero
Xiaoping Yun
Charles Hall
Roger Hartley
Dirk Klose
Brian Glass
David Hislop

TESTBED RECOMMENDATIONS:

Multidomained

- More than 1 testbed
- Perhaps one per program
- Should already be in being at some level
- Take ongoing program and pull out a problem
- Testbed available on a reasonable basis to DARPA
- Look at taxonomy of domain features
- Identify common set and testbed and problem in set
  - Examples:
  - AFATADS
  - NAVLAB
  - AUV AT DRAPER

SHORT TERM OPPORTUNITIES
YEARLY MILESTONES

Year one demonstrate adaptive interfaces
- The look and feel of the user interface for a controls engineer
- Interface a software development tool to a controls development tool

Year two demonstrate encapsulated tools
- Supports prototype standard representing functional requirements for standards

- •Adaptive interface is controlling the encapsulated tools
- •Info passing among tools

Year three demonstrate an integrated tool set
- •Supports a standard methodology
- •For specified problem
- •In chosen testbed

TASKS

Year one: develop taxonomies
- •Domain features (1 man year)
- •Tool features (3 to 4 man years)
- •Taxonomy of standards (1 man year)
  - Interface standards
  - Reference architecture
  - Methodologies
  - Leverage existing standards

Needs assessment (1 man year)
Develop demonstration (5 man years)

Year two: develop system design plan (1 man year)
Define standard (2 man years)
Acquire/develop testbed (GFE)
Encapsulate tools (10 man years)
Demonstrate encapsulated tools (2 man years)

Year three (11 man years): apply tool set
Demonstrate tools set
Evaluate
Iterate
Demonstrate

LONG TERM RESEARCH ISSUES

Not general software engineering but software for control systems

Therefore for control and system engineers designing and integrating control systems

Levels:
- •Systems: C3, tactics, hardware
- •Languages for descriptions within a level
- •Languages for descriptions between levels

Formality of description languages
- •Devices
- •Processes
- •Control characteristics
- •Communications
- •Command

Software build according to these requirements will:
- •Be modular (interoperable)
- •Have standard interfaces
- •Conceptually integrated

Tools work:
- •Within a level
- •Between levels
- •Use language for interchange and HCI

Automation should increase formality with help
all tools should emphasize evolutionary design
all tools should emphasize dynamic nature of systems
(animation)

Appendix III

Participants in the Workshop on Software Tools for Distributed Intelligent Control

July 17-19, 1990

Dr. James S. Albus
National Institute of Standards and Technology
Robot Systems Division
Bldg. 220, Rm. B124
Gaithersburg, MD 20899
PH: 301/975-3418

Dr. Kishan Baheti
National Science Foundation
1800 G Street NW Rm. 1151
Washington, DC 20550
PH: 202/357-9618 FAX 202/357-7636

Dr. Chris Barrett
Los Alamos National Laboratory
M/S 605 Group A5
Los Alamos, NM 87544 email: cbarrett@LANL.gov
PH: 505/665-0733 FTS 855-0733 FAX 505/665-2017

Dr. Thomas Bihari
Adaptive Machine Technologies
1218 Kinnear Road
Columbus, OH 43212 email: amt@eagle.eng.OHIO-STATE.edu
PH: 614/486-7741

Prof. J. Douglas Birdwell
University of Tennessee
Dept. of Elec. & Comp. Engr.
Knoxville, TN 37996-2100 email: birdwell@cascade.engr.utk.edu
PH: 615/974-5468

Mr. James Caldwell
NASA Langley Research Center
Mail Stop 130
Hampton, VA 23665-5225 email: jlc@air12.larc.nasa.gov
PH: 804/864-6214

Dr. Jagdish Chandra
U.S. Army Research Office
SLCRO-MA (Dr. Jagdish Chandra)
P.O. Box 12211
Research Triangle Park, N.C. 27709 email: Chandra@brl.arpa
PH: 919/549-0641 FAX 919/549-9399

Dr. Norman Coleman
U.S. Army ARDEC
Attn: SMCAR-FSF-RC (Dr. Norman Coleman)
Bldg. 95 North email: ncoleman@pica.mil
Picatinny Arsenal, NJ 07806-5000 FAX 201/724-5597
PH: 201/724-6279 conf # 724-2124

Mr. Matthew Emerson
Naval Avionics Center
Indianapolis, IN 46219-2189
PH: 317/353-7825 FAX 317/353-3583

Prof. Michael Fehling
Stanford University
Dept. of Engineering Economic Systems
321 Terman Engineering Center
Stanford, CA 94305-4025
PH: 415/723-0344 FAX 415/723-1614

Prof. Dean Frederick
Rensselaer Polytechnic Institute
7032 JEC
Troy, NY 12181 email: dean frederick@mts.rpi.edu
PH: 518/276-6080

Dr. Donald T. Gavel
University of California
Lawrence Livermore National Laboratory
P.O. Box 808, L-496
Livermore, CA 94550
PH: 415/422-8539

Dr. Brian Glass
NASA Ames Research Center
Mail Stop - 244-18
Mt. View, CA 94035 email: glass@pluto.arc.nasa.gov
PH: 415/604-3379 FAX 415/604-6997

Dr. James Greenwood
Advanced Decision Systems
1500 Plymouth Street
Mountain View, CA 94043-1230 email: greenwood@ADS.COM
PH: 415/960-7551

Dr. Fred Y. Hadaegh
Jet Propulsion Laboratory
Guidance and Control Section
4800 Oak Grove Dr.
Pasadena, CA 91109-8099
PH: 818/354-8777 FTS 792-8777

Dr. Charles Hall
Lockheed Missile & Space
Organization 259
3251 Hanover Street
Palo Alto, CA 94304
PH: 415/354-5260

Prof. Roger Hartley
Computing Research Lab.
New Mexico State University
P.O. Box 3CU/30001
Las Cruces, NM 88003 email: rth@nmsu.edu
PH: 505/646-1218 FAX 505/646-6218

Dr. Scott Harmon
Hughes Research Laboratory
3011 Malibu Canyon Rd.
Malibu, CA 90265
PH: 213/317-5140

Dr. Frederick Hayes-Roth
Cimflex Teknowledge Corp.
P.O. Box 10119
Palo Alto, CA 94303
PH: 415/424-0500 ext. 410 FAX 415/493-2645

Dr. Charles Herget
University of California
Lawrence Livermore National Laboratory
P.O. Box 808, L-194
Livermore, CA 94550 email: herget@icdc.llnl.gov
PH: 415/422-7786

Dr. Thomas Hester
Electronic Engineering & Comp Sci.
FMC Corporate Technology Ctr.
1205 Coleman Ave.
Santa Clara, CA 95052 email: hester@ctc.fmc.com
PH: 408/289-0461

Prof. Vincent Heuring
University of Colorado at Boulder
ECE Department
Campus Box 425
Boulder, CO 80309-0425 email: heuring@boulder.colorado.edu
PH: 303/492-8751

Dr. David W. Hislop
U.S. Army Research Office
SLCRO-EL (Dr. David W. Hislop)
P.O. Box 12211
Research Triangle Park, N.C. 27709 email: hislop@aro-emh1.mil
PH: 919/549-0641

Col. John R. James
HQ Training and Doctrine Command
ATRM-K
Fort Monroe, VA 23651
PH: 804/727-3945 or 727-3948 FAX 800-365-5181

Mr. Roland A. Jones
Gensym
125 Cambridge Park Dr.
Cambridge, MA 02140
PH: 617/547-9606 FAX 617/547-1962

Dr. Dirk Klose
USACECOM C3 System Center
Chief C2 System Development Division
Attn: AMSEL-RD-C3-IR (Dr. Klose)
Fort Monmouth, NJ 07703-5000
PH: 201/544-2213 FAX 201/544-4084

Dr. Wolf Kohn
Boeing Company
Chief Researcher in Artifical Intelligence
Science Computing and Analysis
P.O. Box 24346
Mail Stop 7L-23
Seattle, WA 98124-0346 email: wolfk@boeing.com
PH: 206/865-3598 FAX 206/865-2996

Mr. Darrel L. Lager
University of California
Lawrence Livermore National Laboratory
P.O. Box 808, L-156
Livermore, CA 94550 email: lager@lucky.llnl.gov
PH: 415/422-8526

Prof. Alan Laub
Department of Electrical & Computer Engr
University of California
Santa Barbara, CA 93106 email: laub%lanczos@hub.ucsb.edu
PH: 805/893-3616 FAX 805/893-3262

Prof. Nancy Leveson
Department of Info. & Comp. Science
University of California
Irvine, CA 92717 leveson@ics.uci.edu
PH: 714/856-5517

Prof. Jane Liu
University of Illinois
Department of Computer Science
1304 W. Springfield Avenue
Urbana, IL 61801
PH: 217/333-0135

Dr. Jacek Maitan
Lockheed
Research & Development Division
0/97-40 B/201
3529 Hanover St.
Palo Alto, CA 94304 email: jmaitan@a.isi.edu
PH: 415/424-2742

Dr. Reinhold Mann
Martin Marietta Energy Systems
P.O. Box 2008
Bldg 6025
Oak Ridge, TN 37830
PH: 615/574-0834

LTC Erik Mettala
DARPA/ISTO
1400 Wilson Blvd.
Arlington, VA 22209-2308
PH: 202/694-5037

Dr. David Morganthaler
Martin Marietta
Information and Communications System
P.O. Box 1260
Mail Stop XL4370
Denver, CO 80201
PH: 303/977-4200 FAX 977-7946

Prof. Swaminathan Natarajan
Texas A&M University
Dept. of Computer Science
College Station, TX 77843-3112 email: swami@cssun.tama.edu
PH: 409/845-8287

Prof. Anil Nerode
Mathematical Science Institute
Cornell University
Ithaca, NY 14853-7901 email: nerode@mssun7.msi.Cornell.edu
PH: 607/255-3577)

Dr. Colin Potts
MCC
Software Technology Program
P.O. Box 200195
Austin, TX 78720 email: potts@mcc.com
PH: 512/338-3629 FAX 512/338-3899

Mr. Richard Quintero
National Institute of Standards and Technology
Robot Systems Division
Bldg. 220 B124
Gaithersburg, MD 20899
PH: 301/975-3456 FAX 302/990-9688

Dr. Magnus Rimvall
General Electric Corporate R &D
Mail Stop KWD213
P.O. Box 8
Schenectady, NY 12301 email: rimvall@crd.ge.com
PH: 518/387-5698 FAX 518/387-5164

Dr. Guillermo Rodriguez
Jet Propulsion Laboratory
4800 Oak Grove Dr. MS 198-219
Pasadena, CA 91109
PH: 818/354-4057

Mr. Donald Sassaman
Vitro Corporation
2121 Crystal Dr.
Arlington, VA 22202
PH: 703/418-8038 FAX 703/418-9028

Mr. Roger Schappell
Martin Marietta
Information and Communications System
P.O. Box 1260
Mail Stop XL4370
Denver, CO 80201
PH: 303/977-4474 FAX 977-7946

Dr. Randall Shumaker
Naval Research Laboratory, Code 5500
4555 Overlook Avenue S.W.
Washington, D.C. 20375-5000 email: shumaker@itd.nrl.navy.mil
PH: 202/767-2903

Major Ken Thurman
UAV Joint Program
PEO (CU) - UTMR
Crystal Gateway #4 Room 304
Washington, DC 20361-1014
PH: 703/692-3423 FAX 202/746-5682

Dr. David Tseng
Hughes Research Laboratory
3011 Malibu Canyon Rd.
Malibu, CA 90265
PH: 213/317-5677 FAX 213/317-5484

Prof. Hiroyuki Watanabe
University of North Carolina
Sitterson Hall
Campus Box 3175
Department of Computer Science
Chapel Hill, NC 27599
PH: 919/962-1817 FAX 919/962-1799

Dr. Matthew R. Wette
Guidance and Control Section
Jet Propulsion Laboratory
4800 Oak Grove Dr. M/S 198-326 email: mwette@jpl-gnc-gw.jpl.nasa.gov
Pasadena, CA 91109

Prof. Xiaoping Yun
University of Pennsylvania
Computer Science Department
200 South 33rd Street
Philadelphia, PA 19104-6389 email: yun@central.cis.upenn.edu
PH: 215/898-6783

END

DATE FILMED

11 / 21 / 90