REPORT ON THE

EMBEDDED AI LANGUAGES WORKSHOP

Ann Arbor, Michigan
November 16-18, 1988

Richard A. Volz
Head, Dept. Computer Science
Texas A&M

Trevor Mudge
Dept. Electrical Engineering & Computer Science
University of Michigan

Gary Lindstrom
Dept. Computer Science
University of Utah

January 27, 1990

Sponsored by the Army Research Organization
Grant DAAL03-88-G-0030

1 Introduction

The past five years has seen an emerging recognition of two important issues in computer science and engineering: 1) languages and environments for distributed, embedded real-time control, and 2) languages and environments for artificial intelligence. For the past decade, people in real-time computing have been developing the basic concepts and techniques required for correct use of computers in real-time embedded systems. More recently, there have been a growing number of artificial intelligence applications involving real-time control systems. However, there has as yet been little contact between people understanding requirements and techniques for artificial intelligence and those understanding requirements and techniques for real-time systems. Yet, if many efforts currently being mounted to combine these ares are to be successful, it is essential that they come together.

This workshop was held to begin the process of cross fertilization among areas bearing upon the problems of embedding AI solutions in real-time systems. We are indebted to Dr. Ronald Green, when he was at ARO, and Dr. David Hislop of ARO for recognizing the importance of these issues. Dr. Green encouraged the preparation of a proposal to host the workshop and Dr. Hislop arranged its funding.

The workshop was originally conceived to have a language focus, and had the following principal goals:

1. To bring together leading researchers in software engineering, real-time programming languages, languages for artificial intelligence, and new computer architectures.
2. To identify the range of applications for distributed embedded real-time AI systems, e.g., the scope of the problem?
3. To identify the technical scope of the problem.
4. To determine the language requirements to support embedded AI systems, and evaluate existing languages such as Ada, Lisp and Prolog from this perspective.
5. To identify areas needing research in support of such languages.

As the workshop unfolded, the issues became less a matter of language, and more a matter of algorithms, as will be seen later in this report.

The set of the workshop participants fulfilling the first goal is given in Appendix A. The other goals were addressed in two ways. A small number of experts in key subareas relating to the overall problem were invited to give presentations providing an orientation to that subarea and issue challenge positions on issues remaining to be solved. Secondly, working groups were formed to identify and discuss further issues in each of the subareas. The broad goals given to each of the working groups were to formulate a definition of the problem area and develop prioritized recommendations concerning:

1. Major issues needing research
2. Promising directions for pursuit of these issues, and

3. Identification of problems already satisfactorily addressed.

In the remainder of this introduction, we discuss briefly what is meant by "embedded AI," describe the subareas considered, and overview the organization of the report.

1.1 What is Embedded AI?

Before proceeding, it is important to present at least a preliminary idea of what is meant by the term "embedded AI." The workshop adopted the following working definition of the term: *An* **embedded AI system** *is an AI system contained as part of some larger system that operates in real-time.* To complete this definition, one must state what is meant by "real-time system." By "real-time system," we mean one which operates under externally imposed timing constraints which must be satisfied if correct operation is to be achieved. Neither producing correct values to computations nor being "fast" is sufficient to be real-time. Obviously, the time requirements can vary widely according to the environment, and will typically be dictated by the physical constraints governing the environment with which interaction is taking place.

A few examples of embedded AI systems may help establish the context of the issues facing the workshop. First, consider the thermal control system being developed for the Space Station. This has been under development at NASA Ames Research Center and Johnson Space Center for two years. When completed, it will control the temperatures in all parts of the Space Station. It is being built as a rule-based expert system, and will operate continuously. However, the response time requirements of the system are not very stringent. Temperatures change slowly relative to the computing speeds of current computers.

As a second example, consider an expert-system-driven automatic rendezvous and docking system for the Flight Telerobotics Servicer (FTS). In this case, the physics of the environment places some rigid constraints on the time within which the system must respond. If the FTS is coasting toward a satellite it is to service and is waiting for a guidance decision from the expert control system, one can easily predict the length of time before a collision would take place. Certainly the expert control system must respond within this period. Approach velocities are generally quite slow, however, and again the expert system will typically have substantial time for computation. Nevertheless, this example does demonstrate one important principle of real-time systems: The computer system *must* be able to satisfy externally imposed timing constraints, and this is something that current AI systems do not do as a matter of practice.

As a third example, consider the Pilot's Associate that is being developed by DARPA to assist pilots in flying high performance aircraft. Again, AI systems are being employed. However, in the case of high performance avionics systems, the response times required are on the order of milliseconds rather than seconds or minutes; it is difficult to meet timing constraints in this case.

Embedded AI systems, then, are part of some larger externally operating environment, and must be able to perform their computations while satisfying timing constraints externally imposed by that environment.

1.2 Subarea Selection

Initially, five subareas were targeted for study. These, and the rationale for their selection, are as follows:

1. **Closed Loop Systems and Requirements:**

   An understanding of the basic problem environment and requirements is crucial to progress in any scientific and technological areas. This area introduced examples of real-time systems, covered basic definitions of real-time, and sketched the varieties of solutions that are known. There was a heavy emphasis on task scheduling issues. Both periodic and asynchronous tasks must be dealt with correctly.

2. **Time Constraints and Algorithms:**

   This area centers on the need for time constraints. Many classical iterative search algorithms either have no finite termination time or have only large bounds on computation time. It introduced the notion that algorithms must be developed from the point of view that they must produce an acceptable answer within a prescribed time, even if that answer is sub-optimal.

3. **Languages and Implementations:**

   From its onset, the workshop was predicated on the notion that certain language features either aid or hinder real-time programming. This area explored this assumption in greater detail, identifying specific features that could cause difficulty in real-time systems and examined their occurrence in different languages.

4. **Performance Estimation and Measurement:**

   It is impossible to predict adequately the time behavior of a language feature independent of translator implementations. Accordingly, performance measurement and estimation techniques are very important. Users of a language need such measures to ascertain the characteristics of the particular translator they are using, even if a given translator is acceptable. Translator and environment vendors need them to detect weak spots in their products and to focus their development efforts.

5. **Parallel AI:**

   Parallel AI is currently a "hot" research topic. In principle, parallelism should permit faster solutions to problems, perhaps making the solutions usable in real-time systems. But does parallelism really help, or does it compound the problems of satisfying real-time constraints?

During the course of the workshop, one other area was added, and some were combined. The first two subareas did a good job of providing those with an AI background an orientation in real-time computation. It was recognized then that there was no comparable session providing an orientation on AI to those with real-time computing backgrounds. Such a session was dynamically added to the workshop.

After the initial sessions were completed, the full panel decided to combine subareas 1 and 2 and subareas 3 and 4.

1.3 Organization of Report

The remainder of the report is partitioned into three components. First, sections 2–7 describe briefly the introductory discussions in each of the subareas defined above. This is followed by section 8 presenting the recommendations of each of the working groups. In this case, the recommendations are organized by the revised arrangement of working groups described above. Finally, there is an appendix listing the participants.

2 Closed Loop Systems and Requirements

Dr. Douglas Locke, Presenter

One of the most stringent classes of real-time computer systems is that of closed-loop system control. Such systems often have a substantial number of concurrent tasks and can have timing constraints on the order of a few hundred microseconds. Consequently, closed loop systems make a good vehicle for presenting the definitions, characteristics and goals of real-time computing systems.

2.1 Definitions, Characteristics and Goals

A *closed-loop system* first of all interacts with some environment. It typically goes through the following operational steps:

1. Sensing the state of the environment.
2. Performing suitable computations.
3. Modifying the environment, usually by outputting some control values.

Typically, this is repeated on a cyclic basis, though in some cases, the action steps outlined above may only be performed in response to the occurrence of some event, such as detecting that the temperature in the Space Station has fallen below some prespecified level. All of the fundamental laws of control theory apply. One must be concerned about the dynamics of the environment, the sampling rate, stability of the system, etc. Of importance here is the fact that it is the dynamics of the environment that primarily determine when the system must respond.

Closed loop systems typically require a real-time response, which ultimately means responding within a sufficiently small time interval that the overall system, including its environment, behaves satisfactorily from a control theoretic point of view. Typical time scales might be on the order of a few seconds for chemical plants, seconds to minutes for a thermal control system, one to a few milliseconds for robot control, or as low as a few hundred microseconds for a fast avionics system.

Closed loop systems typically run continuously for long periods of time. This could range from a few hours for an aircraft control system, to months for a traffic control system, to many years

for a spacecraft system. Such systems might or might not allow or require human intervention; both situations are common.

It is also typical of closed loop tasks that though they run in the simple cycle outlined above, the computations that must be performed are often voluminous and quite complex. A complete system will generally contain several interrelated tasks of this type. Ten to twenty tasks would be common in a high performance avionics system.

By applying *intelligence* in the control system, one hopes to improve the control system by allowing it to adapt to the environment. For example, consider a robot working on the construction of the Space Station. The robot is itself a complex mechanical structure of links and joints, and its control system must reflect the configuration of the robot. Suppose that the robot is anchored to the Station while working and grasps a strut that is to be attached to the rest of the Station structure. As soon as the strut the robot is holding touches the Space Station structure, which it must do in order to attach it, new kinematic constraints are introduced. This generally leads to a significant change in the control computations that should be performed for the robot control. Most current robot controllers are unable to handle this kind of change. An intelligent controller could sense the contact, determine the new kinematic constraints, and shift the mode of the controller.

At a higher level, intelligence might be used to determine the inputs to be given to a standard control system. For example, an automatic driving system for a rover vehicle on the moon or Mars would sense the environment (in this case the surface of the moon or Mars), detect obstacles, determine a safe path for the vehicle to travel etc., and give those commands to the vehicle. Complex AI planners to do exactly that are presently under development.

Applying intelligence, however, can adversely affect closed loop control in a poorly understood domain. One must remember that real-time is the key issue. The *correctness* of a real-time computer system is based not only upon the correctness of the computations performed (in the usual program correctness sense), but *upon the satisfaction of timing constraints*. Indeed, most people working on real-time computer systems are primarily concerned with this latter requirement.

A consequence of the need to satisfy timing constraints is that the *usefulness* of a computed result is a function of the time at which it is obtained. One can define an abstract *usefulness function* to describe this relation. Figures 1.a and 1.b show two typical value functions. Figure 1.a shows a situation in which the computation has zero value after some point in time is reached. For example, this might occur in the docking situation described in the previous section if the computation is not completed until after a collision has occurred. Figure 1.b shows a softer situation in which the value of the computation degrades exponentially after some point in time is reached. These value curves depend upon the environment, and can change dynamically. Unfortunately, analytical methods for determining these curves do not, in general, exist.

It is important to note that time constraints can be associated with starting time of a computation, finishing time, some event within a computation, or any combination of these. Clearly, a *methodology* is needed for defining system response time requirements.

The general goal of real-time computing is resource management (cpu, memory, specialized devices, etc.) to attain correct performance in the presence of response time requirements imposed by the environment. Within this general goal, three specific subgoals have been defined:

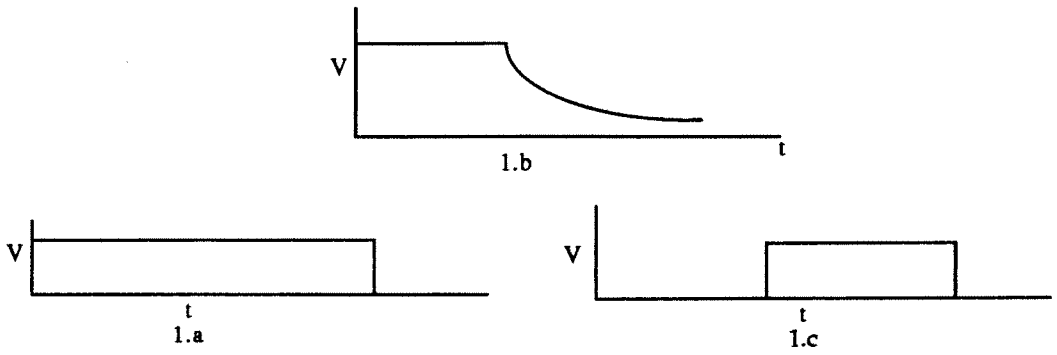

Figure 1: Typical usefulness functions corresponding to a real-time computation.

1. Meet all time constraints, if possible.
2. Meet all *important* time constraints at all times.
3. In any case, be able to predict how well 1 and 2 are met for any given process load.

These goals supercede the more traditional goals of fairness, starvation avoidance, load balancing and deadlock avoidance in multiprocessor systems.

2.2 Scheduling

The principal concern in real-time computing, then, becomes one of scheduling. There are a number of relevant tasking models that have appeared in the literature. These include:

- periodic tasks with static, preemptive, or independent execution times,
- transient (schedule on demand, independent job arrival),
- dependent tasks (with individual time precedence constraints), and
- homogeneous or heterogeneous multi-processor architectures and configurations.

In developing scheduling algorithms, there are a number of issues that must be considered. These include:

- global scheduling,
- synchronization and data dependencies,
- multi-resource dependencies,
- guaranteed vs. stochastic response time, and

- deadlock detection vs. prevention.

During the past two decades, several different scheduling strategies have been proposed. They emphasize different criteria and are based upon different assumptions. We review only the more common ones briefly.

Basic priority scheduling: One of the simplest methods is to distinguish among the urgencies of the tasks through the use of priorities. It uses a very simple scheduling algorithm: It shall not be the case that a lower priority task is executing while a higher priority task that is ready to run is not. Often, this scheme will be used in combination with other scheduling algorithms. Indeed, many of the schemes listed below are priority scheduling schemes with different choices for priorities.

However, one must be careful to avoid priority inversion. Priority inversion can occur if a high priority task requires the resources that are already allocated to a lower priority task and is blocked until that lower priority task finishes. During the execution of the lower priority task, it may be interrupted by a task whose priority is between the original high priority task and the low priority task. By itself, priority scheduling is generally inadequate.

Cyclic Executive: This is by far the scheduling method most widely used by industry. It consists of the repeated sequential execution of a set of tasks in some order. The worst-case execution time of each task must be known. If some tasks execute less frequently than others, then they will at times simply skip their turn. This system is easy to understand, at least conceptually, and one feels that one has a firm control on the timing of the system. However, the system is fragile, static and very difficult to maintain. Tuning to effectively utilize the time available for each major cycle is difficult. Any change to any task can cause severe timing problems. Timing is often determined by placing an oscilloscope on the bus and looking for the occurrence of specific bit patterns to identify the beginning and ending of a task.

Shortest Processing Time First: This is a well known, but rarely used scheduling algorithm. It attempts to minimize mean lateness, and is optimal for subgoals 1 and 3, stochastically. It has $n \log n$ complexity.

Earliest Deadline First: This attempts to minimize maximum lateness. It has $\log n$ complexity. However, it fails disastrously on overload.

Smallest Slack Time: This algorithm also attempts to optimize for subgoals 1 and 3 (if preempting overhead is free). It has a $\log n$ complexity. However, it again fails disastrously on overload.

Best Effort Scheduling: This method attempts to maximize the total value from the value curves. However, it is computationally too demanding to be used today.

Rate Monotonic Scheduling: This method deals primarily with a set of cyclic tasks. Aperiodic tasks are essentially converted to cyclic tasks by requiring a minimum time between occurrences. For this method, as well as many others, the system is assumed to have satisfied a deadline if the task has completed its computations before the next time that it is scheduled. The system is

non-optimal with respect to cpu utilization, but fulfills all of the subgoals for real-time scheduling and actually achieves a relatively high cpu utilization rate. The scheduling is static in that the priorities of the jobs are fixed throughout execution. Use of this algorithm is beginning to grow.

During one of the panel discussions, it was noted that the definition of deadline that is used by some of the scheduling algorithms is inadequate for two important classes of problems. Many data processing algorithms depend upon exactly periodic data, and can degrade very substantially if the data sampling points deviate very much from the nominal. Yet, all deadlines can be satisfied and a jitter in the sampling times equal to nearly the nominal sampling period can still occur. Control is a second example. A delay of nearly one full sample period can occur while still satisfying the definition of deadline scheduling. Yet, a delay of one sample period is enough to cause some control systems to go unstable. These effects are largely independent of the speed of the computer; they are related to the dynamics of the environment.

2.3 Summary of CL Systems and Requirements

In summary, it was noted that real-time computing is **not** synonymous with fast. There are many efforts today that focus on building faster systems for the purpose of real-time computations. Fast is important, but predictability of the time of computations and operations is essential.

The view was expressed that the major issues that need to be addressed for achieving embedded AI systems include the following:

- Bounded predictable execution times,
- Dynamic resource scheduling,
- Garbage collection techniques, and
- AI language support for the above.

3 Timing Constraints and Algorithms

Prof. Jane Liu, Presenter

The discussion in this session continued the emphasis on the importance of timing constraints, the need for predictability and the difficulty of solving these problems. It emphasized the multiple processor situation more than the earlier sessions. It tended to identify and describe unsolved scheduling problems. Also, a new view on algorithm development was introduced, one which could have great importance for embedded AI systems.

3.1 Constraints and Scheduling

Timing constraints can be expressed in several different ways, e.g., computational deadlines, frequencies at which tasks must be executed, the required response time to the occurrence of some event, or allowed lateness or tardiness in completing some action.

One method of enforcing these constraints is through the use of exception handling. For example, if an exception is generated upon the expiration of a deadline, any of the following actions might be taken, depending upon the nature of the deadline.

- Sound an alarm and terminate if a hard deadline was missed.
- Resume the task whose deadline expired, regardless of relative priority.
- Retry the entire function that was being performed to assure functional consistency.

An alternative approach is possible if sufficient information is statically determinable. In this case, a priori scheduling algorithms can be developed that will guarantee enforcement of timing constraints and make effective use of system resources. Both optimal and good heuristic algorithms exist for scheduling preemptible tasks on uniprocessor systems. These can handle preemptible periodic jobs. In general, though, we must know the worst case performance of every task.

What is needed are similar scheduling algorithms for the multi-processor and non-preemptible cases. Situations that are schedulable for a uniprocessor may not be when multiple processors are used. Only a little is known in these cases. The case with two processors and identical non-preemptible execution times can be solved with complexity $n \times \log n$. But, with only three processors, the problem becomes NP-hard. For as few as 50 tasks, the computations are already beyond the point of reasonable computation time.

3.2 New Views on Algorithm Development

The emphasis on the discussion up to this point has been on algorithms for scheduling. However, it is important to recognize that these are not the only categories of algorithms that must be dealt with in embedded AI systems. Most AI systems require searches of various kinds, and heuristic search algorithms are themselves a major area of research. Moreover, many optimization algorithms require iterative search; it is typically difficult to bound or predict the time required to complete such a search. Thus, the process of developing algorithms is, in general, important to the embedded AI area.

The real-time perspective described above strongly suggests new viewpoints on algorithm development. Algorithms should not only progress towards finding a solution, but must accept a time constraint within which they must provide some kind of an answer. One can imagine also requiring that the algorithm provide some indicator of confidence in the answer provided. Another view is to divide the algorithm into two or more parts. The first part would be considered a mandatory, and will always be computed. The others will be optional and will be computed only if there is time. The mandatory part must provide some level of usable answer. Ideally, if one

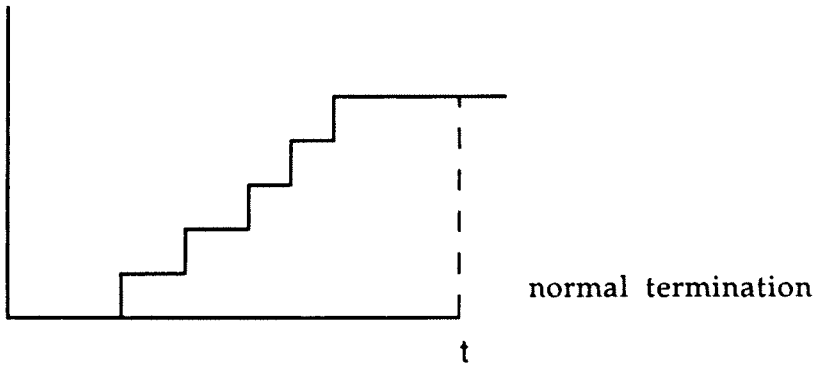

Figure 2: Desired monotonic performance of segmented algorithm.

plots the quality of the answer produced by the algorithm as a function of time it is monotonically improving, as shown in Fig. 2. Such an algorithm would provide more flexibility in scheduling because it allows a trade-off between the quality of the result and the time required to produce the result.

Some examples of algorithm types that are being investigated from this perspective are: iterative algorithms, statistical techniques, successive doubling for Fast Fourier Transforms (FFTs), image generation from holograms, phased arrays, and partial query processing.

Still another possibility is to increase the priority of a computation as a function of missed deadlines. This, obviously, can only be used when the deadlines are soft, e.g., as in Fig. 1.b. That is, it is application dependent.

It would seem then, that one might be able to do better by considering the broader aspects of the application when developing the algorithms used. Nevertheless, it is still valuable to investigate application independent methods. They are extremely demanding intellectually, but will have much wider application if success can be achieved. The general idea of separation will be to have application designers determine the value functions shown in Fig. 1 and let an algorithm scheduler take over from there.

Since many of the searches needed for AI applications are essentially database searches, some of the key problem areas for embedded AI applications will be in this area. Three of the major problems are:

- Scheduling database transaction processing to meet deadlines.
- Achieving temporal consistency.
- How to define, use and process imprecise queries.

4 Languages and Implementations

Prof. Richard LeBlanc, Presenter

Historically, there has been little overlap between the languages used to develop real-time applications and those used to develop AI applications[1]. There are many reasons for this. However, the reasons of interest to us here are those related to the features and implementations of the languages.

For purposes of embedded real-time AI systems, then, one can classify the computer languages into two categories, those that are typically used for embedded systems and those that are typically used for the development of artificial intelligence applications. Languages in the former category include: assembly language, Fortran, CMS-2, Jovial, C and Ada. Languages in the latter category are: Lisp, Prolog and various functional languages. We emphasize though, that this classification is by *use* of the languages named, not the features they contain. Some of the languages contain features that would allow them to be used for either application. The analysis that follows, however, is more related to specific kinds of features than to specific languages.

4.1 Language Characteristics

Although there are some major differences between individual languages within each group that may favor one over another, there are also significant characteristics that distinguish the categories.

Embedded System Languages

Languages used for embedded applications have features that allow predictability of program operation. In other words, operations must have predictable execution times. Memory accesses must take known times. Program control must use only structures whose execution times can be reliably predicted.

This leads first of all to a dependence upon the concept of typed variables. It does not, however, mean that strong typing is required, just that every variable has a statically (compile time) determinable type, whether explicitly declared or implicitly determined. Then a compiler can determine the specific type of instruction to be used in operations on the variables.

Static name resolution is also required so that the compiler can determine bindings between names and memory locations. As there is a fixed relation between names and memory, the operations in the language are viewed as modifying the state of memory.

Although several of the languages placed in the embedded category have features enabling recursion and dynamic data structures, these features are seldom, if ever, used for embedded applications. Embedded applications use only the fixed data structure features of the languages. Control structures are also restricted to the looping and alternation structures provided. The latter implies that iterative control rather than recursive is used. Moreover, direct control of machine resources such as timers and memory is typically possible.

[1] We distinguish here between development of an AI application and the rewriting of that application in a different language for production use. When one considers the languages in which AI applications are rewritten for commercial use, there are overlaps. However, this detail is not of particular significance to the discussion here.

AI Languages

While the languages typically used for embedded-systems have extensive architecture-oriented features, the languages typically used for AI applications emphasize problem-oriented features that do not as directly translate to machine level functions. Most evident among the problem oriented features is the use of dynamic data structures. Where traditional real-time applications would use static data structures almost exclusively (even if the language used supports dynamic structures), AI applications (and their supporting languages) use dynamic structures almost exclusively.

Control structures are another area of significant difference. While real-time applications avoid the use of recursion (even though several of the languages in this category support it), recursion is the mainstay of AI applications. Many algorithms have a much simpler form when expressed recursively than when control iterations are written without recursion. Again, the issue from the real-time perspective is the predictability of the operations; the recursive structures are much harder to predict and are thus avoided in real-time applications.

A third significant area of difference is the view of machine level resources. The languages in the AI category abstract machine level resources away from the programmer and provide little or no direct control.

The concept of state involved is at a higher level than that of most of the languages used for embedded applications. It is associated with variables that may be bound to different memory locations at different points in a program.

Lisp also allows untyped variables, which implies dynamic type resolution. In this same spirit, Lisp has dynamic name resolution. Again, from a real-time perspective, these make predictability difficult.

Prolog is logic based. Variables are bound by unification. The resolution process, however, typically requires a great deal of backtracking, and is another operation whose execution time is difficult to predict. Actually, many algorithms involving backtracking are also easily expressed in Lisp as well.

4.2 Implementation Issues

From the perspective of embedded-real time systems, the central language implementation issue is the predictability of the execution time. An important subissue is the extent to which predictability of a given feature is inherent in the feature or implementation dependent. The difference in the characteristics of embedded languages and AI languages described above lead to several important implementation differences. These will again be considered by language category.

Embedded System Languages

One of the first important implementation characteristics of this category of languages is that they are compilable. Being compilable does not guarantee that one can predict the execution time, but being compilable increases the chances of doing so. Moreover, compilability usually

means that it is possible to develop efficient storage and/or operation mechanisms. For example, it is possible to select all operations at compile time, and object addressing can be sufficiently well determined to allow all object addressing instructions to be issued at compile time. Data structures can generally be decomposed and efficient storage and access mechanisms set up. Often the run-time system can be made small and efficient as well.

Some languages in this category provide access to low level machine features. Ada, for example, provides representation clauses that allow a programmer to specify actual storage layout for records and to reference specific memory addresses. Ada further provides low level I/O mechanisms. However, one must be careful that the particular implementation chosen in fact supports these mechanisms.

It is important to note some things that are **not** used by real-time applications, even if the language might support them. Dynamic storage management heads this list. Dynamic storage management leads to two problems. First, most implementations of storage allocation are hierarchical in nature. They start with a modest sized block of storage and allocate from it reasonably rapidly until the block is exhausted. Then, a higher level, but generally slower, storage allocation module is invoked to obtain another block of storage. It is common to find at least three such levels of allocation. The point of entry into the allocation hierarchy can also vary with the size of the block of storage being obtained. Obviously, it can thus be difficult to predict how long it will take to perform a storage allocation operation; it can depend upon past history.

Garbage collection is another major problem that is a consequence of using dynamic storage allocation. Many systems invoke a garbage collector implicitly, and often at a high priority. That can totally destroy predictability of any part of the program.

The avoidance of recursion is not so much a translator implementation issue as it is a difficulty in bounding the recursion depth and thus establishing an execution time bound.

AI Languages

The two most important implementation issues in this category are the heavy dependence upon dynamic storage schemes and the fact that programs are not always fully compilable. The impact of the dependence upon dynamic storage was discussed in the previous section. We concentrate here on the other issues.

Lisp has the ability to dynamically define functions (through the EVAL function, for example). Since these cannot possibility be known at compile time, they must be interpreted at run-time. This makes predictability very difficult, if not impossible. Furthermore, the fact that variables need not be statically typed requires run-time type determination and dispatching to code to perform the operations. This both increases the size of the run-time system and makes it impossible to statically determine the time required to perform the operations on the variables. Similar effects accrue from dynamic name resolution.

Prolog's implementation is based upon unification of program goals and logic rules. This leads to heavy use of recursion and backtracking. Often this is both slow and unpredictable. Prolog also has dynamic binding of simple names. In conjunction with unification, this leads to the need for dynamic storage management identified above. Prolog is also only poorly matched to the architectures of standard processors.

4.3 Summary and Issues on Languages and Implementations

In summary, the panel felt that the problems and issues were not with specific languages, but with individual features that might be used. The principal issue from the real-time perspective is the predictability of application code execution time. Use of language features that involve dynamic storage allocation, garbage collection, dynamic operation interpretation or dynamic typing lead to implementations that make predictability difficult or impossible.

The major research issues raised are:

- The overall architecture of embedded AI systems.
- Design methodology for embedded AI systems.
- Real-time, background garbage collection routines that can be cyclically scheduled, or made non-interfering with application code (i.e., they run only during slack times).

5 Performance Estimation and Measurements

Dr. Harlan Sexton, Presenter

The performance of a programming language may be defined as the performance of programs written in that language - this means that both the potential as well as the actual performance of programs written in this language must be considered. In particular, it is necessary but not sufficient to be concerned with the low-level implementation of the basic components of the language (such as function-call discipline, storage allocation and deallocation, etc.). One must also provide for the instrumentation and analysis of programs in order to have a "high-performance" programming language implementation. To put this another way, a high-performance language requires that the language have well-engineered "atoms" and well-developed support tools for the users of the language.

5.1 Limiting Factors on Potential Language Performance

First we examine some of the considerations which must be faced in implementing a real-time AI language with good "potential performance"; that is, with properly designed language atoms. These considerations are, clearly, an amalgamation of those facing the implementors of conventional real-time languages and of conventional AI languages. Forming such an amalgam will require the collaboration of experts from several areas.

In the case of "conventional" programming, the constraints on a piece of code are typically soft – the programmer is usually concerned with maximizing some statistical property of the code such as average response time. While such considerations are usually important to the embedded systems programmer, this programmer often has to deal with hard constraints, as well. As was discussed in the section on languages, it is because of the need to handle hard constraints that the embedded systems programmer must understand the costs associated with language constructs.

Programming languages used in AI are typically more abstract than are conventional languages in that the atomic operations of the language are often far removed from the machine-instructions of conventional hardware. Further, the implementation process of an "AI language" usually involves a good deal more than writing a compiler – typically these languages provide high-level runtime support for programs, too. Some of the Common Lisp atomic operations which affect potential language performance most are function calling, arithmetic and data-structure operations, and type-checking and type-dispatching (the latter operation is especially important in object-oriented programming systems). Runtime features which are usually present and which affect potential system performance are CATCH, THROW, UNWIND-PROTECT, the dynamic binding of values to special variables, and the dynamic allocation, management and deallocation of memory (garbage collection).

To give a concrete example of how these implementation details interact, consider the case of a multitasking utility such as is often present in a modern Common Lisp system. In such a system it is usual for the various tasks to share the global "name space" of the underlying Lisp system, but for the dynamic bindings of special variables within the various tasks to be independent of one another. This means that when the context of one process replaces another the special bindings must be replaced, too. There are two primary binding strategies used in Lisp implementations, referred to as deep and shallow binding. Without going into great detail, the differences are generally that for deep binding systems, the process of establishing a binding involves adding a "binding cell" to a stack, and for shallow binding the process involves adding a similar "binding cell" (representing the variable's PREVIOUS value) to a stack and then changing the value in a global cell associated with the variable being bound (in both cases these stacks are usually the control stack of the program). Clearly, the costs of looking up the value of a variable are smaller for the shallow binding strategy than for the deep binding one – in fact, special-value lookup is of fixed cost for shallow-binding systems while for deep-binding systems the cost may be arbitrarily large. On the other hand, task switches in shallow-binding systems may be arbitrarily expensive, while for deep-binding systems they can be of fixed cost.

The choice of binding strategy used in a given implementation and the performance consequences of the choice are precisely the sort of details which are often needed by embedded systems programmers, and it is unfortunately one about which they are rarely told. Further, language implementors often make the design tradeoffs implicit in such choices without being completely aware of the significance of their choices to users of their implementation. It seems especially important that benchmarks be developed that test for such design choices in languages used for embedded systems applications. Such a set of benchmarks, analogous to but more extensive than those in the reference at the end of this section, would serve both to help language users make more informed choices and to help implementors construct more useful systems.

An area where design decisions are especially critical and the tradeoffs especially complicated is that of memory management. The more primitive the memory allocation/deallocation system is, the more difficult and error prone are programs which make complex use of storage – in effect, such programs on these systems must implement a garbage collector for themselves each time. On the other hand, the garbage collector on a standard Common Lisp system is a complex amalgam of software, runtime support, and low-level programming conventions (including built-in support in the system's compiler). Such a garbage collector is intended to behave in some "acceptable" manner in all circumstances and to provide high levels of performance in cases perceived as "most important". Unfortunately, such a garbage collector is unlikely to have behavior which is provably

acceptable in any realistic embedded system applications that require a garbage collector in the first place. It actually is unlikely that a general-purpose real-time garbage collector is possible (as any general-purpose garbage collector would probably look like current ones), but it does seem that different sets of "real-time" requirements can be met with different sorts of garbage collectors. This is clearly an area that needs study, and equally clearly this study needs to involve experts in real-time programming, AI programming, and Lisp implementation.

5.2 Support Tools for Program Development

Next we consider the problem of providing a supporting environment for the development of embedded AI systems. This problem is considerably more difficult and the way to proceed much less clear, but experience of conventional AI developers and the understanding of some of the special problems of real-time programmers provide some suggestions as to promising first steps.

Larry Masinter (of Xerox PARC) is often quoted as saying, "Premature optimization is the source of all bugs." While perhaps a bit extreme, it is very important to remember that the process of programming is a human one, and that no matter how fast a program might be potentially, it is infinitely slow until it actually runs.

One may regard the programming process as successive refinement of the programmer's understanding of his or her problem. From this point of view, performance monitoring tools are one of the natural final steps in this process. Once the "static correctness" of the program has been established (however formally or informally this is done), it is a natural next step to seek an understanding of the dynamic behavior of the "proposed solution" (i.e. the program), and usually to refine this solution in light of this enhanced understanding.

On a related point, many Lisp programmers view the runtime type support common in most modern Lisps in pretty much the same way. In most parts of most "finished" Lisp programs the types of all variables do not change during execution. (This is not so true in many object-oriented systems, where the type information of the arguments is used DYNAMICALLY to determine how to invoke generic functions, but even in these systems such "generic function code" is in the minority.) That is, the association of specific type information to variables is often viewed as belonging to the later stages of the programming process. Support for runtime-types, which is typically removed by the compiler from the "product code", is just a way of verifying and affirming the programmers solution of the problem during the process of development and debugging.

The common theme here is that having the development system provide support tools for the programmer will almost always improve the quality of the resulting code in every respect. Another important point is that this development support NEED NOT exact a price from the ultimate program – performance monitoring tools can be left out of the final program and runtime type-checking can be "compiled away".

The challenges faced by AI programmers were (and are) such that high levels of development support were deemed essential. To put it more plainly, it is a view all but universally held in the AI community that good development support *of all kinds* is essential for good software. While difficult to substantiate, it is frequently claimed that the powerful support systems provided programmers on special-purpose Lisp machines result in improved quality of code and productivity increases of factors of 5 to 10. This is not likely to convince the most skeptical AI debunkers,

but it is verifiable that many of the "modern" CASE and programmer tools in use in the general-purpose computing world have antecedents (often in more elaborate and sophisticated forms) in the AI programming world. While the absolute performance of these AI tools were often not as good as their current general-purpose counterparts, these costs were willingly born. Further, as we have indicated here, these costs are not *intrinsic* to the final code being developed, just to the machine and system on which the development was done. This trade-off of machine for programmer costs is one that is seen as a bargain in the AI community.

It is very distressing to many AI developers to discover that while the challenges faced by the real-time embedded systems programmer are in many ways even more difficult than those they are familiar with, the support tools available to the real-time programmer are usually very much inferior. This is especially distressing when the complexities of real-time AI systems are contemplated. Certainly the embedded AI systems programmer needs the sorts of development tools found in the best conventional AI programming environments, but also desperately needs tools to help guarantee that hard constraints imposed by the task are being met.

As an obvious first example, it should be feasible to provide compiler support to give the programmer parametrized (somewhat abstracted in terms of the arguments) upper bounds on the cost of the execution of many functions, especially if the programmer can supply some declarative information to help. Such a tool could be quite valuable in helping the programmer identify areas where hard constraints might be violated, such as in finding places where patterns of memory allocation and deallocation are problematic for the particular garbage collector strategy chosen for this program. This is definitely an area where a great deal of research is needed, and one which should yield a high rate of return.

5.3 Summary

Providing high-performance languages for embedded AI systems is a problem having two parts. It is first of all necessary to design and implement a basic language that has the capabilities and features needed by the AI programmer and that, at the same time, can be implemented so that the resource consumption of these capabilities and features is predictable. Second, it is essential that the development systems for such a language provide as much support as possible to aid these programmers in dealing with problems of unprecedented complexity.

The related problems of language performance for embedded AI programming are more a matter of engineering than of theory. A great deal is known about implementing languages such as Common Lisp on a wide range of hardware, but this knowledge has not been applied in an intensive way to meet the very difficult requirements of real-time embedded systems. It seems likely, however, that a systematic study by experts from real-time systems, AI programmers, and AI language implementors could catalog the important design tradeoffs and critical features needed by the "hybrid" embedded AI systems programmer. It also seems that this is essential if really reliable, high-quality "real-time" AI language implementations are to be developed.

The task of developing support tools for providing hard estimates of software performance are more research oriented, but there seem to be no theoretical obstacles to developing such tools. Significant progress could certainly be made simply by proceeding in the more or less obvious manner, and the lessons learned there would almost certainly be worth the cost.

Unfortunately, while high-levels of development support have been accepted among AI pro-

Problem complexity	Problem size solvable with N computers
N	N^2
N^2	$N^{1.5}$
N^k	$N^{1+1/k}$
2^N	$N + \log_2 N$

grammers for some time, this view is not universal among all programmers (nor among their managers). The fact is that an AI language cannot do anything new or different from any other programming language – it simply enables fallible, confused, and otherwise all too human programmers to succeed at tasks on which they might have failed with a language and environments that made their work harder. The fact is that languages and environments CAN be implemented to support programmers and still produce software which is fast and efficient, if anyone is willing to pay for these languages and environments to be implemented and for feasible target architectures on which to run the final programs.

REFERENCE
Performance and Evaluation of Lisp Systems, by Richard P. Gabriel. MIT Press, 1985.

6 Parallel AI

Prof. Benjamin Wah, Presenter

An important technique for speeding up the execution of programs is parallel processing. In some cases this technique may be applied to embedded AI systems in order to meet real-time deadlines. However, parallel processing is not a guaranteed way of meeting deadlines: ordering dependencies between various parts of a program usually limits the number of processors that can be used in parallel. Moreover, if a problem cannot be done in a reasonable time on a sequential machine, for example when the problem does not have a polynomial time solution, then it is unlikely that the problem can be solved in a reasonable time using parallel processing. To illustrate this point in another way consider the table below. The lefthand column shows a range of complexities for sequential algorithms in terms of N, the input data size. The righthand column shows the problem size that could be solved in the same amount of time if N processors were used in parallel. An (optimistic) assumption is that the problem is perfectly parallelizable, i.e., all of the N processors can be profitably used all the time during the execution of the parallel version of the algorithm. Examining the table, we see for a problem whose complexity is linear in the input data set size (first line of the table) that N processors will allow us to solve a problem that is N times as large in the same amount of time. Unfortunately, for inherently more complex problems the advantages of using N processors to meet a deadline declines dramatically. The final line in the table shows that for a problem whose solution time grows exponentially with its input size the use of N processors will only accommodate a problem size increase of $\log_2 N$,

or a factor of $(1 + \frac{\log_2 N}{N})$. In other words, if an algorithm has exponential complexity, parallel processing is unlikely to help meet a real-time deadline for any but small problems. Clearly, then, parallel processing is not a substitute for good algorithm design for embedded AI applications, or any other types of application for that matter. Indeed, examining the above table would suggest that parallel processing can only be used with any significant effect to meet a deadline if the underlying algorithm is of linear or quadratic complexity unless the problem is small.

A common theme of AI algorithms is the idea of search. This can take many forms. Two examples are the search for a sequence of moves that leads to a goal and the search for the logically permissible steps in a deductive process. Many others examples could be cited which have a wide range of applications from robot navigation to battle management decision making. Search procedures, in their simplest form, are usually exponential in their input data size; for example, the possible paths that a robot may explore is exponential in the number of junctions in the map that it must navigate. In existing algorithms this potential for exponential growth of the problem complexity has been dealt with in a number of ways. Indeed the introduction of "intelligence" into the search has been responsible in part for the term artificial intelligence. This intelligence can take many forms, examples of which are heuristic rules and knowledge about the application domain. These in turn can be built into the system by the programmer, as is the case with expert systems, or learned through a training period and/or during the operation of the algorithm. The resulting algorithms are ones in which the overall complexity is no longer exponential and, in many cases, is reduced to the point at which parallel processing becomes a practical means for meeting deadlines for realistic problem sizes. However, this reduction in complexity is typically traded for the optimality or quality of the solution.

In practice, intelligent search algorithms typically make use of parallel processing to meet deadlines by using several processors to initiate searches on non-overlapping regions of the search space. Parts of the search space are usually created as needed during the running of the algorithm to avoid creating an unmanageably large data structure. However, this dynamic aspect of many AI algorithms makes them more difficult to parallelize than, say, a matrix multiply, whose steps can be (statically) scheduled on N processors before the algorithm is run. It also leads to a number of issues that are still being explored by researchers and whose solutions will significantly impact the effectiveness with which parallel processing can be used to meet real-time deadlines. Some of the major issues are as follows: 1) dynamic load balancing and scheduling work among the processors; and 2) determining the order in which the search space is explored. Both of these issues have an important effect on the execution time of the algorithm (see Section 8.3). In fact, researchers have reported "superlinear speedup" (N processors complete the problem more than N times faster than 1 processor) for some AI algorithms involving heuristic searching. This would appear to contradict our earlier discussion on the limitations of parallel processing as a means to meet real-time deadlines. Closer inspection of examples of superlinear performance improvements shows that superlinearity comes from comparing the parallel algorithm with a poorly designed serial one, often one in which the search heuristic is incorrectly applied. This and related issues in the parallel processing of intelligent search algorithms has recently been explored in depth in the reference at the end of this section.

A number of parallel computers have been proposed that are aimed at AI applications. Figure 3 lists some of the most notable. Their status, which ranges from "commercially available" to "paper design", is also shown, and it can be seen that a surprising number exist or are in an advanced state of development. Figure 3 is organized by language paradigm (functional, logical,

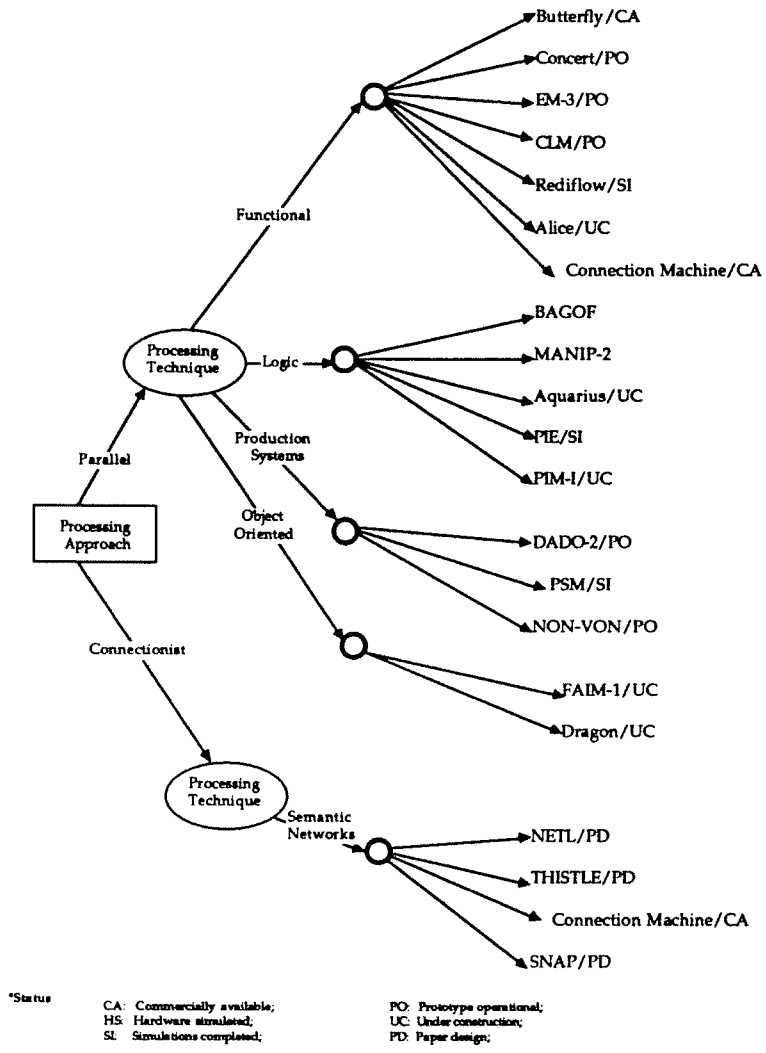

Figure 3: Proposed Architectures for Parallel AI.

etc.) because many AI applications areas are bound to these paradigms. Although, in principle, each paradigm is capable of expressing any Turing computable function, it is not clear how well a particular machine will perform outside of its paradigm.

REFERENCE

Parallel Processing of Best-First Branch and Bound Algorithms on Distributed Memory Multiprocessors, Tarek Saad Abdel-Rahman, Ph.D. Thesis, University of Michigan, 1989.

7 Practices and Requirements for AI Applications

Mr. Bradley Allen, Presenter

AI applications are software systems that perform or support a task involving problem solving at some level of processing. Such applications can take a variety of forms: e.g., as stand-alone expert systems, or as large software systems with embedded knowledge-based components. AI applications can be characterized in one of two ways: in terms of what tasks they perform and in terms of how they are implemented. A discussion of real-time issues in AI applications must focus on what tasks performed by AI applications may have real-time aspects, and how such systems can be implemented so as to satisfy real-time constraints.

The problem-solving tasks that present-day AI applications accomplish can be divided into two categories: classification tasks and synthesis tasks. Classification tasks work from a description of a object and arrive at a classification that may be simply presented to a user or may determine actions to be taken. Such tasks include diagnosis, pattern recognition, and situation assessment. Synthesis tasks work from a description of desired goals and generate a plan for satisfying the goals. Example synthesis tasks include planning, scheduling, design and configuration. Real-time constraints in AI applications are driven by the time constraints imposed by the specific task to be performed.

Current practice is to implement AI applications in one of two ways: directly in a procedural language (e.g., Lisp or C), or in a very high level language that has an associated interpreter implemented in a procedural language i.e., using an AI application "shell". The latter approach is by far the most frequently used, mainly for reasons of software productivity. For this reason, any discussion of real-time issues in AI applications must address the issue of AI application shells.

A variety of implementation-level characterizations for AI applications have evolved over the past two decades of work. These include rule-based architectures, model-based architectures, case-based architectures and connectionist architectures, as well as architectures that combine several of these. Each of these architectures has a number of characteristic algorithms that are used in their implementation. For rule-based architectures, implementation depends on algorithms such as resolution, unification, and matching. Model-based architectures depend on demand-driven inheritance and message passing. Case-based architectures use associative retrieval and object instantiation. Connectionist architectures primarily depend on value passing algorithms. The evolution of AI application shells has progressed from initial university prototypes on specialized hardware to commercially supported products available on widely-used machines, ranging from PCs to mainframes. The consequence of this progression is that arguments about the relative merit of implementation languages for AI applications (e.g., Lisp vs. Ada) or the need for specialized hardware support are beside the point. The real-time properties of AI applications depend mainly on the algorithms used in their implementation, and much less on the choice of implementation language or supporting hardware.

An important starting point for any effort to explore real-time issues for AI is to attempt to taxonomize AI applications from both the task and implementation perspectives. Once suitable taxonomies have been arrived at, the impact of real-time concerns on the choice of task and implementation can be determined. By focusing on the specific algorithms involved in each implementation, we can use concrete results about their complexity, boundedness, resource requirements, and decomposability into mandatory and optional sections to arrive at an analysis

of the real-time performance of a given application. Because AI applications are increasingly embedded in much larger systems, an analysis of real-time performance of AI applications also involves a careful analysis of the real-time properties of the overall system of which they are a part.

Additional areas to be considered include machine learning, verification and validation, and explanation. Each of these areas are currently in transition from research to use in applications, and each has a variety of consequences for real-time AI applications. There is also a growing body of experience in the area of "soft" real-time AI application architectures (i.e., architectures designed to be "fast enough" in a non-quantitative sense for a particular application area) that those investigating hard real-time problems should consider.

In conclusion, real-time AI applications require a thorough analysis of both the real-time constraints encountered in typical problem-solving tasks and the algorithms used to implement the primitive operations in a problem-solving architecture. Real-time software researchers should formulate constraints relevant to a variety of problem-solving tasks, thus generating requirements for AI tool and application developers to satisfy. Developers should then satisfy these requirements by basing their architectures on a careful analysis of the computational characteristics of their underlying algorithms.

8 Presentation and Discussion of Recommendations

As discussed in the introduction, the initial set of working groups were combined into three groups for the working group sessions: 1) Requirements, Constraints and Algorithms, 2) Languages and Performance, and 3) Parallel AI. This section reports on the recommendations from each of the revised working groups.

In addition to the technical recommendations, the panel as a whole felt that the issues raised are extremely important and that there should be a continuing forum for discussion of this problem area. There was a call for another workshop within a year. Three groups were identified as technical communities that should be brought together: 1) the Real-time Computer community, 2) the Artificial Intelligence Community, and 3) the Intelligent Control community.

8.1 Requirements, Constraints and Algorithms

The first conclusion reached by this working group was that we do not even have a good definition of the problem yet. We are solving problems in a very ad hoc way, often without knowing the problem domain for which the proposed solution must work. There is much work to be done, both in defining the problem more completely and in developing new approaches and techniques for solving it. Among the research issues that this working group believes are important to address are:

1. Develop a requirements document giving the functional components of embedded AI systems:

This is the beginning point. Once this is in place, one can begin to address the problem in an organized manner. Without an adequate requirements description, one cannot be sure that one is addressing the right problems or that the techniques being developed will be useful. It was suggested that such a requirements statement be developed for a number of different applications.

2. Develop a systems/software architecture:

   Again it is suggested that several possible architectures be developed and experimented with.

   (a) Allow application time constraints to drive lower level designs.

      i. Develop and use temporal reasoning.

      ii. Develop and use knowledge-based scheduling.

      In view of the wide variation of time constraints that are known to occur in examples examined to date, this is a particularly important issue. If the time constraints can be incorporated into the architecture in a formal manner, then there is hope of automating the process of developing methods to satisfy them.

   (b) Develop techniques to characterize and control non-determinism.

      Non-determinism seems to fly in the face of all of the requirements of predictability expressed by the real-time computing community. Yet nondeterminism is a mainline characteristic of a number of modern computer languages for AI. A technique to allow constrained use of non-determinism could be very useful.

   (c) Develop an approach to verify the techniques used.

      Verification & validation (V & V) is currently a very important part of building real-time systems. The same will be true of embedded AI systems. It will be important to understand how, if at all, V & V will be different for these systems and what techniques must be used.

   (d) Study numeric vs. symbolic processing.

      Symbolic processing is widely used in AI systems, but rarely used in real-time computing. Nevertheless, it could lead to simpler problem expression or simpler expression of solutions. However, what are the predictability characteristics of symbolic processing?

   (e) Develop an interface between application programs and the scheduler: – Who does the scheduling? And at what levels of granularity?

      On the one hand, scheduling is not yet handled well enough that it can be automatically incorporated into systems. On the other, it is a burden and complexity one would not like the user to have to bear because scheduling requires information on *all* tasks in the system, not just the one the user is writing.

   (f) How does one reason about limited resources?

      In particular, limited time and memory resources are particularly important.

   (g) How does one quantify the quality of results? What is the tradeoff between quality and complexity?

3. Develop new characterizations of time constraints and new mechanisms for recovering from missed deadlines.

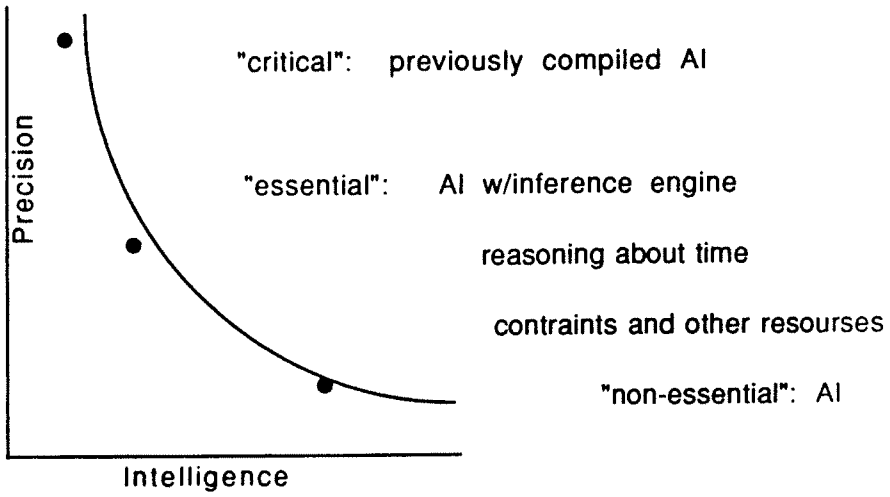

Figure 4: Hypothesized relation between precision and intelligence.

Examples given earlier in this document demonstrate that the current definition of deadline scheduling is insufficient to cover all important cases, and hence the solutions basis upon that definition are also insufficient to cover all important cases. Moreover, the consequences of missing deadlines need to be better understood. Some relaxation in the hardness of deadlines could significantly alter scheduling methods and results.

4. Develop a systems theory for knowledge-based control systems.

   Just as there exists an extensive theoretical basis for servo control systems, a theory is needed for the rule-based systems that seem assured of playing a major role in embedded AI systems. When is a set of rules consistent? Complete? Stable? How is consistency maintained in the underlying distributed database that seems destined to become part of many systems?

It was also suggested that there may be a relation between the degree of intelligence and the precision of a system. This is illustrated in Fig. 4 which shows a hypothesized relation between precision and the different levels of AI involved in the system.

The working group suggested three directions of research they feel are promising:

1. Monotonic (or as they are sometimes called, Anytime) algorithms.

   This allows algorithms to be divided in such a way that each increment of computation is assured of improving the result. Such algorithms make scheduling and resource management to meet time constraints easier by allowing trade-offs between result quality and time and resource requirements.

2. Resource bounded AI algorithms.

   This seems essential for embedded systems.

3. Synthesis of intelligent hierarchical schedulers.

   This, again, seems essential. The level of intelligence that one might be able to achieve at any level may be proportional to the severity of the time constraints that must be met. On the other hand, the amount of preplanned "reflex" type action at a given level may be inversely proportional to the magnitude of the time constraints.

8.2 Languages and Performance

The Languages and Performance working group believes that there are a number of significant language and hardware oriented issues that must be addressed if progress is to be made on embedding AI applications into real-time systems. Considerable experimentation is required. To obtain different viewpoints and comparative approaches, the research should involve several research groups. Again, the proposed research issues reflect the belief that we do not yet fully understand the problem. The research issues recommended for study by this group are:

1. Build a repository for problems and solutions.

   It was suggested that the Software Engineering Institute (SEI) would be an appropriate place for the repository. SEI has good network connections to the rest of the research community, and is conducting work in related areas.

2. Language issues requiring study are:

   (a) Time abstractions.

   This is particularly important for distributed systems in which each node may have its own sense of time. The issues are not only developing some kind of updating policy (a number already exist), but determining the semantics of time that take into account the fact that clocks on different nodes will not have precisely the same values.

   (b) Sources of unpredictability.

   These need to be better understood. What are they? How does one identify them? Can they be bounded? Can better implementations remove the unpredictability?

   Performance measurement will likely be an important aspect of the needed work in this area. The development of suitable measurement techniques is thus also important.

   (c) Visibility of implementation choices.

   At present, compiler vendors make many implementation choices that significantly affect the predictability and performance of a compiler. Important examples are: the storage allocation scheme, the garbage collection mechanism, the task scheduling mechanism, and inter-task synchronization and communication mechanisms. Many examples of (unnecessarily) inefficient or unpredictable implementations have been found. Yet these mechanisms are almost always hidden from the user, and one seldom can find someone within the vendor that knows or will provide the information. Yet for real-time embedded applications, knowledge of this kind of information is essential.

   Again, measurement techniques are likely to be important.

   (d) Support for expressing distributed programs.

There is no commonly accepted way of expressing the distribution of a program, though there have been a number of proposals. Under what circumstances should it be done explicitly by the programmer? For many embedded systems, the distribution is fairly obvious, but the mechanisms for expressing the distribution do not exist. However, when one gets into the use of massively parallel systems, manual expression of distribution is probably not practical. To what extent can it be automated? How application dependent is it?

3. Develop *application generators* for real-time systems.

   The complexity of embedded real-time AI systems will make it very difficult for practitioners to build good systems using current language tools. Higher level methodologies are essential if the ability to build such systems is to become widespread.

4. Study the significance of object-oriented design/implementation.

   Object oriented design has gained great favor in recent times. There are many indications that it does, indeed, significantly improve programmer productivity and product reliability in non real-time situations. However, what are the implications of its use for real-time applications?

5. Build supportive development systems.

   One of the major reasons that many researchers find rapid prototyping in Lisp to be very effective is the environment support that is typically provided with the systems. Syntax directed editors, pretty printers, incremental execution, etc. have been shown to be very useful. Not only are these kinds of tools needed for embedded AI systems, but new classes of tools addressing the timing and predictability issues will be needed.

6. There are relevant hardware issues as well:

   (a) Provide adequate support in terms of mechanisms like timers.

   (b) Ensure that there are no obstructions, e.g., priority mechanisms, that can prevent proper operation of real time scheduling mechanisms.

   (c) Ensure that modern technology is available.

   This issue is also an extremely important one. Most real-time computer implementations today are with relatively old hardware that is substantially below current day technology. The ability to use current hardware would make a substantial difference in what could be accomplished.

8.3 Parallel AI

During the course of the workshop discussion in this area, the following issues were raised:

1. What is the definition of an embedded AI system?

2. What classes of problems are suitable for parallel AI?

3. What is the model of the architecture for such systems?

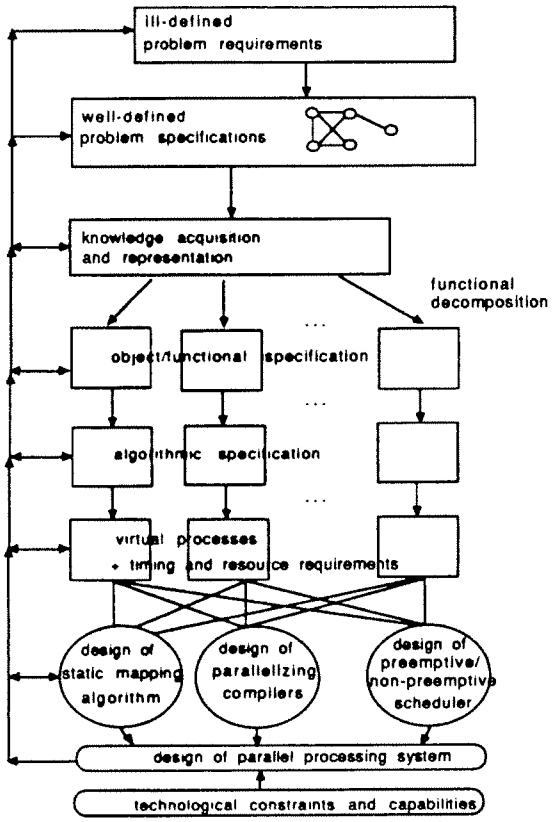

Figure 5: Design Methodology.

4. Who schedules parallel tasks?
5. Deadline and computation decomposition – how can the compiler detect them for efficient use of parallel computation?
6. Load balancing, i.e., how can resources in a distributed system be scheduled and managed?
7. What is the minimum amount of information that must be specified to allow the design of a system?
8. Design methodologies for mapping real-time applications to parallel systems.

This led the Parallel AI group to begin by developing several block diagrams that they believe will help organize efforts toward designing and building an appropriate system. Figure 5 shows their view of a design methodology leading to the design of a parallel processing system for implementing an embedded AI system. Figure 6 shows the place of the design methodology in the broader scope of the embedded AI system being built.

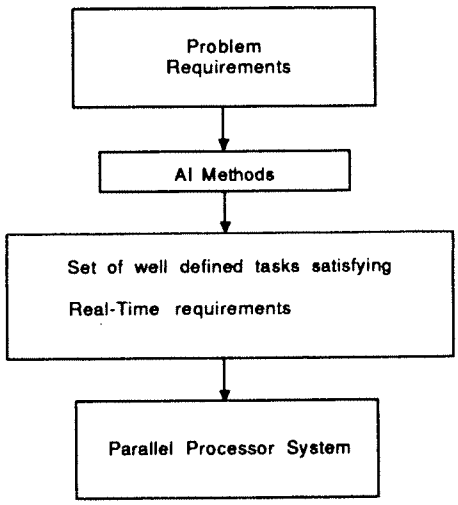

Figure 6: Interpretation of a real-time embedded AI system.

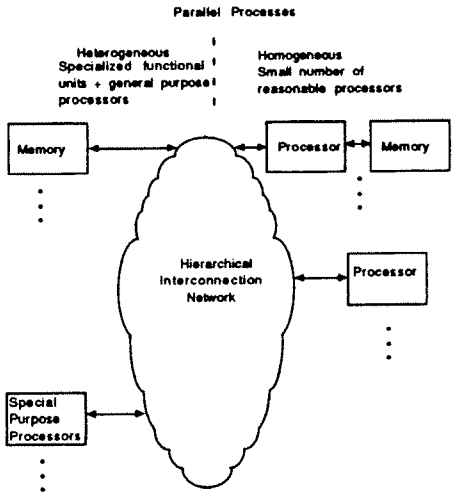

Figure 7: Parallel Processor system for embedded AI systems.

It can also be expected that embedded AI systems will not be built entirely with homogeneous processors. Figure 7 presents of view of the interconnections between different kinds of processors that might be present in the system.

The working group then coalesced the issues into the following set of major issues, many of which were also identified by the other working groups:

1. Develop an architectural model of real-time embedded systems.

   Once such a model is established, it will provide a conceptual framework within which

people can describe and talk about embedded AI systems in a meaningful way. It will also provide a basis for the development of a theory of such systems.

Several system architectures should be developed and compared.

2. Definition of an embedded real-time AI system.

   As expressed earlier, the working group does not feel that we yet have a full definition of the problem. The relationships between the characteristics of typical AI systems and the hardness of real-time systems constraints poses a problem. Once we do develop an adequate problem definition, it will be important to develop a *design methodology* for building these systems.

3. Compiler-detected vs. user-defined parallelism:

   (a) What is the minimum amount of information that must be specified by the user in order to allow compiler detected parallelism?

   (b) One must consider both deadline and computation decomposition.

4. Develop suitable application domains.

   While a small number of examples of systems under development today exist, there is no general description of domains of problems for which embedded real-time AI systems are suitable. It is likely that there will be different problem domains that require (allow) the use of different techniques in their solution. The severity of time constraints, for example, might be one discriminator. An explication of several suitable problem domains would help one develop an understanding of the architectures and solution methods that could be employed to obtain solutions.

A List of Workshop Participants

EMBEDDED A.I. LANGUAGES WORKSHOP
NOVEMBER 16-18, 1988

James Abello
Texas A&M University
Computer Science Department
College Station, TX 77843

Bradley P. Allen
Product Research Manager
Inference Corporation
5300 W. Century Blvd.
Los Angeles, CA 90045

Dean Lynn A. Conway
University of Michigan
College of Engineering
Ann Arbor, MI 48109

Prof. Susan Davidson
University of Pennsylvania
Computer Information Science
Philadelphia, PA 19104

Prof. Edmund Durfee
University of Michigan
EECS Dept.
Ann Arbor, MI 48109

Prof. Tzilla Elrad
Illinois Inst. of Tech.
Dept. of Computer Science
Chicago, IL 60616

Lt. Col. Norb Eyrich
U.S. Army A.I. Center
Room ID 659, CS DS AI
Pentagon
Washington, D.C. 20310-0200

Edward Ferguson
Symbolic Computing Lab
Computer Science Center
Texas Instruments

Dr. Steven A. Gordon
Computer Science Dept.
Illinois Inst. of Tech.
Chicago, IL 60616

Dr. David Hislop
U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

Prof. Keki Irani
University of Michigan
EECS Dept.
Ann Arbor, MI 48109

Lt. Col. John James
Headquarters HQ TRA DOC
Att: ATRM - D (LTC James)
Fort Monroe, VA 23651

Prof. Robert Kessler
University of Utah
Dept. of Computer Science
Salt Lake City, UT 84112

Dr. John Knight
Software prod. Consortium
1880 Campus Commons, North
Reston, VA 22901

Dr. Toshiaki Kurokawa
IBM Japan, Ltd.
5-19, Sanbancho, Chiyoda-ku
Tokyo 102, JAPAN

Prof. Jane W.S. Liu
University of Illinois
1304 W. Springfield Ave.
Urbana, IL 61801

Prof. John Laird
University of Michigan
EECS Dept.
Ann Arbor, MI 48109

Prof. Wm. W. Lively
Texas A&M University
Dept. of Computer Science
College Station, TX 77843

Prof. Richard LeBlanc
Georgia Inst. of Tech.
School of Info. & Comp. Sci.
Atlanta, GA 30332

Dr. Douglas Locke
IBM Software Concepts
Fed. Systems Div., Bodle Hill Rd.
Owego, NY 13827

Prof. Insup Lee
Univ. of Pennsylvania
Computer Science Info.
Philadelphia, PA 19104

Dr. Jed Marti
The RAND Corp.
1700 Main St.
Santa Monica, CA 90406

Prof. Gary Lindstrom
University of Utah
Comp. Sci. - 3190 MEB
Salt Lake City, UT 84112

Prof. Al Mok
University of Texas
Dept. of Computer Science
Austin, TX 78712

Dr. Joseph Linn
Inst. for Defense Analysis
1801 North Beauregard St.
Alexandria, VA 22311

Prof. Trevor Mudge
University of Michigan
EECS Dept.
Ann Arbor, MI 48109

Dr. Reed Little
Software Eng. Inst.
Carnegie Mellon Univ.
Pittsburgh, PA 15213

Sundar Narasimhan
M.I.T.
545 Technology Sq., Room 826
Cambridge, MA 02139

Prof. John Painter
Texas A&M Univ.
Dept. of Elect. Eng.
College Station, TX 77843

Dr. Brian Unger
Jade Simulation International
1833 Crowchild Tr., N.W.
Calgery, Alberta, T2M 4S7 Canada

Prof. Tony Reeves
Cornell Univ.
Dept. of Comp. Sci.
Ithaca, NY 14853

Prof. Richard Volz
Texas A&M Univ.
Dept. of Comp. Sci.
College Station, TX 77843

Henry Rueter
Vector Research
Ann Arbor, MI

Benjamin Wah
Program Director
National Science Foundation

Dr. Lawrence Siedman
Ford Aerospace & Comm.
3939 Fabian Way, M/S G13
Palo Alto, CA 94303

Dr. Abraham Waksman
AFOSR/NMBolling AFB
Washington, D.C. 20332-6448

Dr. Harlan Sexton
Lucid, Inc.
707 Laurel St.
Menlo Park, CA 94025

Prof. Terry Weymouth
University of Michigan
EECS Dept.
Ann Arbor, MI 48109

Prof. John Stankovic
Univ. of Massachusetts
Dept. of Computer Science
Amherst, MA 01003

Phillip Topping
Manager
Lockheed Missiles and Space
2710 San Hill Road
Menlo Park, CA 94025