The Impact of Domain Dynamics on Intelligent Robot Design

Douglas A. Reece and Steven Shafer

May 1990
CMU-CS-90-130

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This research was sponsored by the Defense Advanced Research Projects Agency (DOD) and monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, OH 45433-6543 under Contract F33615-87-C-1499, ARPA Order No. 4976, Amendment 20 and Contract DACA76-85-C-0003.

Keywords: System architectures, real time systems, mobile robots, architectures for intelligence

Abstract

Autonomous mobile robots should be able to perform useful tasks in complex and dynamic environments. In an attempt to achieve this goal, artificial intelligence and robotics research has given rise to a wide variety of intelligent systems. Only recently have these systems begun to address the issues involved in solving problems and planning action with limited resources in complex and dynamic domains. To this point, the literature has not provided an analysis of how these factors affect system architecture. This paper discusses how resource limitations and domain characteristics affect the basic structure of system architecture, and how this structure impacts the system's ability to solve cognitive problems. We set forth four general design issues for systems: modularization, parallelism, execution control, and communication. In each case we explain how the demands of problem solving ability and real time response conflict, thereby making it difficult to integrate planning and control in the same system. With this understanding of the implications of design choices, it is possible to evaluate the strengths and weaknesses of real and potential architectures. The range of design choices is illustrated with examples of real planners, controllers, and other intelligent systems.

Table of Contents

1. Introduction

One of the aims of robotics research has been to develop intelligent, mobile agents capable of performing tasks in the complicated environments in which humans live. Attempts to design such agents have given rise to a wide variety of systems. Artificial Intelligence has created planners, and Robotics has developed vehicle control systems. Both kinds of systems tend to concentrate on a narrow subset of intelligence: planners address puzzle solving, learning, and other cognitive problems; vehicle controllers concentrate on low level sensor processing, navigation, and control tasks. Traditional AI planners are not adequate for controlling an agent in complex and dynamic domains because they do not address issues of interaction with the real world, limited resources, and system implementation. On the other hand, robot controllers operate vehicles in the real world, generally under the supervision of a simple planner, but sacrifice cognitive abilities to do so.

There are many reasons why particular systems are competent in only limited areas. The competence of a system for a particular task is affected by its architecture — how the system represents and uses knowledge, what inference mechanism or algorithms are used, how knowledge is grouped into modules, etc. In addition to these aspects of the architecture, there are several basic issues that more generally affect how well suited a system is for performing cognitive or real time control tasks. These issues concern general information structure — whether modules are used at all, and how data is communicated between them — and how the locus or loci of execution flows in the system. The choices for these *structural* aspects of architecture design are driven by the requirements of the task and environment. an understanding of the relationship between these requirements and choices is important for understanding why planners cannot control robots and robot controllers cannot solve puzzles. Unfortunately, the planning and robotics literature does not adequately explain why systems are designed the way they are — what assumptions are made about the problem domains, how the assumptions constrain system design choices and how these choices affect problem solving capabilities. In this paper we discuss the relationship between domain constraints and system architecture and how architectural features can limit rationality. The remainder of this section describes traditional planners, dynamic domains, and planners for dynamic domains in more detail. We then explain the importance of resource constraint considerations, and finally introduce the specific system design issues that are analyzed in the paper.

1.1. Traditional Planners

Early robot programs embraced the idea that an intelligent robot seeks goals; the robot executes a sequence of actions that changes the state of the world (and robot) so that some specified goal state is achieved. The job of these programs was to plan — to search for a suitable sequence of actions. STRIPS [16] is a favorite example of an early planner.

The major weakness of STRIPS-like planners for mobile robots is that they assume a very benign problem domain. In particular, they make the following assumptions:

- *The world is static.* Nothing ever moves or changes except as a result of robot actions.
- *There are no time constraints.* The robot may take as long as necessary to develop a plan. Also, there are no limitations on how long the plan could take to execute.
- *The robot has complete knowledge of everything in the environment.* There is essentially no need to

use any sensors.

A mobile robot is rarely able to make such assumptions. Physical processes and other agents change the environment; the changing environment and the robot's own motion impose time constraints; and real sensors limit the robot's knowledge.

There are other limitations in STRIPS having to do with problem solving ability. For example, it has problems dealing with interacting subgoals and consumable resources; it cannot take advantage of abstraction; and it cannot learn. These limitations are significant for any intelligent problem solver. Later planners, such as STRIPS with MACROPS [17], BUILD [15], ABSTRIPS [50], NOAH [51], SIPE [58], Soar [33], PRODIGY [39], Theo [40], etc. have improved considerably on STRIPS' problem solving limitations. Nevertheless, these planners still make the assumptions itemized above. In this paper, however, we are concerned with systems and domains in which these assumptions do not hold.

1.2. The Problem of Complex, Dynamic Domains

Complex, dynamic domains place additional requirements on planner capabilities. These requirements have been expressed in many ways — the need to interleave planning and execution, the need to make decisions quickly or in a bounded time, the need to sense the world frequently, etc. The key problem for the planner in all of these cases is simply that its resources and time are both limited. If a robot had an infinite amount of time, it could thoroughly examine its environment and then exhaustively search for the best action to take next. If it had infinite sensing and computational resources, the robot could do these exhaustive computations in an instant. However, when a robot has limited resources relative to the dynamics of its domain, it may have to abandon its search for the ideal solution and settle instead for a satisfactory one. Thus the dynamics of the domain combined with limits on resources force a planner to act differently than if it were in a more ideal problem environment.

In addition to planning problems, limited resources are the cause of some perceptual and uncertainty problems. The inability of a robot to know about everything around it is partly due to its inability to direct its sensors and perceptual processors at everything at the same instant. Of course, range limitations and other physical factors would still limit even infinitely fast sensors.

This statement of the problem may seem obvious, but the relation of resources to world dynamics is an important one to consider before deciding that a certain planning system "must have feature X" to meet a functional requirement. For example, suppose a planner is to control an industrial process that has time constants on the order of several hours. It is possible that the planner could search for the best action and always find it within a few minutes; even exponential algorithms have bounded times if the problem size is limited. This planner could use a simple execution scheme in which the system continually plans, updates its data base from polled sensors and then plans again. Would this be a useful approach, though, if the time demands on the system were much higher, or if the problem size grew? Perhaps this same planner *could* be used, using faster hardware that will be available in ten years. In this example, the performance requirement does not necessarily require execution control features beyond those in a "traditional" planning system.

Our point here is not to claim advantages for simple execution control schemes, but to show that not all

problem domains necessarily constrain planners to the same degree. Architectural constraints are often introduced into systems based on tacit assumptions about the domain and resources. It is not always apparent that these assumptions hold true for the range of applications intended for the systems. On the other hand, it is possible to identify aspects of complex and dynamic domains that do justify special system features. These domains require resource limited systems to have more sophisticated control architectures because the domains have these characteristics:

- they impose response time constraints;
- the computations required to solve the problem grow as problem size increases;
- the problem complexity is variable over time and unpredictable;
- and situations in the environment change outside of the agent's control and at times independent of the agent's actions.

Such environments are important for intelligent agents because they are common in the real world that humans inhabit.

In the literature, problem domains with severe time constraints are referred to as "real-time" domains. A real-time planning or control system is one that is "fast" with respect to the dynamics of the domain. In control applications, real-time usually means that a processor executes fixed-time pieces of program code at regular, clocked intervals. This idea has been refined and generalized to apply to planning systems that do not repeatedly execute the same code. For example, O'Reilly and Cromarty [46] define a real-time system as one that has the ability to "guarantee a response after a fixed time has elapsed, where fixed time is provided as part of the problem statement."

"Real-time" requirements can be further generalized by combining concepts of response time and gradual performance degradation. In a dynamic domain, many of an agent's goals will be related to time. For example, a goal of getting to a particular destination will probably have a *time* limit associated with it; a robot might have a goal of tracking a car whose position is changing with *time;* a robot might need the ability to react within a certain *time* to an obstacle that moves in front of it. The performance of the robot in executing its task is thus often tied to the time it takes to think and act. Failure to act within a certain time interval may not necessarily result in total mission failure, but just reduce the quality of the robot's performance. For instance, consider a low level servo controller on the robot. As the time between successive command outputs increases, the bandwidth and stability of the controller degrade. Another example is viewing the scene ahead of the robot to find the road. As the motion planner waits longer and longer for new information about the road, it must slow the robot more and more to prevent it from "falling off" the edge of the known road. This slowing is a loss of performance if speed is a goal. A third example is braking or swerving to avoid an object that suddenly moves into the path of the robot. If the robot is to maintain a higher speed (again, a relation to time) and still be able to swerve or stop in time, then it must be able to react within an shorter time. System response within a fixed time must only be guaranteed absolutely if failure to do so would result in an unacceptable performance degradation or mission failure.

The planners discussed in section 1.1 create complete plans and then execute them blindly in static problem domains. Although this control method is adequate for static worlds, system designers have realized that much more interaction is required when the domain is dynamic and resources are limited.

For example, Munson [43] described how planning and execution could be interleaved for agents in uncertain environments. In the last four or five years, many intelligent agent designs have emphasized capabilities for dealing with dynamic environments, time constraints, and the need to use perception to get information [25], [4], [18], [20], [29], [34]. This research, as well as general research in "real-time" AI [32], [46] and time-dependent reasoning [13], [23], [26], [31] , has established several *functional* requirements for planners in dynamic domains. These requirements include the ability to choose actions quickly, guarantee a response time, focus attention, sense and act continually, respond to asynchronous events, and adjust the quality of reasoning to match time available. On the other hand, mobile robot controllers have generally started with a commitment to sense, plan movement, and monitor execution with perception [3], [6], [12], [14], [21], [24], [27], [35], [37], [41], [45], [47], [52], [54], [57]. As their tasks and environments became more complicated, robot system designers have sought to better decompose the problems, improve their perceptual capabilities, and better integrate various sensing and computation components. While all of these systems have made progress towards achieving intelligence in real-world environments, all lack some important ability — either reacting to dynamic events, accommodating limited resources (especially perception), or applying knowledge to solve problems. In other words, none yet combine powerful cognitive ability with realistic control capabilities.

In summary, it is resource limitations of the agent relative to the complexity and dynamics of the problem domain that make such domains difficult for autonomous agents. The ability of the agent to achieve its goals will degrade steadily as time constraints are tightened. Various systems have been proposed to solve problems in these domains; the architectures of these systems usually favor either problem solving or real time behavior control, but not both. This paper explores the rationale behind different architectural features and elucidates the problems inherent in integrating planning and control.

In this paper we use the term *integrated robot architecture* (IRA) to refer to a intelligent robot software system. This term is used because "planner" is often associated with a program that has little or no interaction with the environment, while "controller" is often used in the context of a purely numerical, reactive system. An IRA is an integrated planner and controller with limited resources that operates in a complex, dynamic domain.

1.3. The Importance of Addressing Resource Constraints

Resource constraints at first may seem unimportant relative to the basic task problems. For example, it seems reasonable to determine how an image of a scene can be interpreted and what robot motion should be made before worrying about how to implement the computations quickly. However, there are several reasons to integrate resource considerations into IRA's at an early stage in the design. First, time constraints actually form part of the intelligent agent problem. A plan to achieve a goal is not useful if the agent does not have time to execute it, or even complete the plan in the first place. If a planner is not guaranteed enough time to complete its work, it must use a different technique to find some answer in the time it has available. Thus the original problem has been changed by time and resource constraints. The IRA may approach this problem by using declarative knowledge about computation times and deadlines, or it may use parallel hardware and priority assignments, or some other design. Similarly, parallel processing, modularization, and data communication issues may seem secondary to understanding how an agent can accomplish its goals; however, as we will show, these structural

characteristics can limit the upper bound of the agent's rationality. Therefore, task solution methods should not be addressed in isolation from the system structure.

Several researchers have asserted that the special problems of performing tasks in the real world are even more important than solving abstract problems. Their emphasis is on handling the interaction with the complex and dynamic environment first, and working on abstract tasks second. Brooks comments that the hard part of building an intelligent agent is not in the abstract reasoning, but in making the abstractions from the real world in the first place [7]. Thus, once objects in a room have been classified by support an mobility properties, it is easy for the abstract monkey-planner to reason that a chair should be placed beneath the dangling bananas. Brooks and Moravec [42] point out that while abstract puzzle-solving systems have attained human-level expertise, complete, autonomous systems can barely function with the capabilities of insects. We may not be able, as humans inspecting our own behavior, to break down abstract problems into pieces that lead to realistic systems of low level perception and control functions. On the other hand, once systems that can perceive and perform simple tasks in the real world have been built, they may constrain or suggest structures and functions for the cognitive parts of the system. Chapman and Agre [10], along with Brooks and Moravec, suggest that the cognitive processes may actually use the perception and control hardware for abstract reasoning — the "mind's eye." While it is not clear that intelligent agents should be completely designed from the bottom up, it is certainly true that the implementation of perception and real time control will have a large impact on the overall capabilities of the agent.

Finally, more and more intelligent agent research is being conducted with real vehicle platforms, and in real outdoor environments. The performance — and even basic "survival" — of these agents is directly affected by the ability of their control systems to operate with time and resource constraints. Therefore, IRA designers must consider these constraints in their systems.

1.4. Design Issues

The requirements of complex, dynamic domains have expanded the dimensionality of IRA design space. General architectural issues that are important to consider in these task domains include:

- **Modularization.** Systems can be divided into information-hiding modules to reduce design complexity and improve performance. How subdivided does a system need to be? How well separated should the modules be?
- **Integration of parallelism.** Computation for robot control can be done on parallel processors. At what level should activities be parallel? How is parallel activity managed?
- **Execution control.** Within each processor, execution will flow between various modules as subproblems are solved. How is this flow controlled? Can it depend only on internal state changes, or does external state also have to be considered?
- **Internal communication.** Parallel processes and other modules must share some information. How is the flow of data controlled?

These are general issues that are independent of any particular system or knowledge representation, and can be used to characterize, evaluate and compare various design approaches. This paper characterizes an ideal cognitive system in terms of these issues and explains why a system with a different architecture may be less rational than the ideal. We also describe how domain complexity and

dynamics require these different design choices; this analysis helps one to evaluate how well the requirements of a particular domain match the assumptions and corresponding design choices used in an IRA. The space of design choices is outlined using examples from existing systems.

2. Modularization

Almost all large systems are composed of smaller parts. In this paper we will refer to the parts of an IRA as "modules." Modules contain related declarative or procedural knowledge. Modules are separated from one another, so that it is easier to get at information within the module than it is to get at information in another module. In fact, modules typically hide some information so that it is not accessible at all from outside. Furthermore, communication between modules is more limited than communication within them. This information hiding is done deliberately, so that modules don't *have* to consider what others are doing.

One of the interesting questions about IRA design is how to decompose the control problem into small pieces. It is common to have separate system pieces for performing *functions* such as perception, planning, and command generation. It is also common to see a second axis of decomposition that runs from *details* to *abstractions*. Another proposed problem division is along *task* lines, so that each system piece performs all functions necessary to implement obstacle avoidance, wandering, wall following, etc. A modularized IRA could implement any of these subdivision schemes; thus the *degree of modularization* is a more general issue than the *content* of the modules. In this section we address the more general issue.

2.1. Knowledge Access

Many traditional planners are relatively monolithic; for example, they have one fact database and one rule or operator set. Simple problems can be solved with simple programs, so it is not surprising that some of the first planners were designed this way. However, having a central knowledge base is advantageous for any intelligent agent because it allows reasoning processes to have unrestricted access to all of the agent's knowledge. Unrestricted access is an advantage for the following reason: an agent's ability to make rational decisions is limited by its processing capabilities and the amount of knowledge it has. If an agent is forced, because of some limitation of its information processing ability, to use only a subset of the available knowledge, then it can only be considered rational within that more limited context. This is the principle of bounded rationality, as described by Simon for human problem solving [55]. An ideally intelligent agent would not have its problem solving capability limited by an inability to use knowledge, but would always apply all relevant knowledge to its problems [44]. When information is hidden in modules, it is always possible that this division will hinder the ability of a reasoning process to access the knowledge needed for the current problem. Thus the ideal arrangement for problem solving would be for the system not to be decomposed into modules at all.

As an example, consider the task of driving down a road. We might imagine a module that performs this task independently of other problems the robot is trying to solve. An IRA that found itself on a road could presumably activate the road-following module and trust that the computations would not impact other activities. We might define the road-following module to take a maximum speed as an input, and report back failure when it could no longer find a road. For almost any definition of the module

boundaries, however, we can find situations that require them to be blurred as more information is passed across the interface to the rest of the system. The road-following module could perhaps perform better if it had more information from outside — e.g., where on the road to drive, what to do if lane lines are detected, what to do if lines disappear, how important it is to avoid potholes, how important it is to avoid sudden steering or speed changes, how crucial it is to keep all the wheels (or feet) on the road, etc. Likewise, other parts of the IRA could use more information from inside the module, such as how confident the perception is in locating the road, how accurately the road is being followed, what features are being used to track the road, how accurately the servos are able to maintain speed and steering, etc. A rich exchange of knowledge between different parts of the IRA thus discourages the use of strong modularization.

2.2. Advantages of Modules

The problem of designing an IRA is very complex. The description of a robot's mission is often full of diverse activities, functions, and implicit goals. Furthermore, the robot itself is composed of many different kinds of physical resources, including various types of sensors, processors, and effectors. In order to make the IRA design manageable, it must include modules that are strongly independent and that have well defined interfaces. This principle has been well established in software and systems engineering.

Modularization can also provide a performance improvement to help meet time constraints. There are several reasons for this. The first is that strongly independent modules can be implemented in parallel with little efficiency loss in communicating with other modules (see the discussion of parallelism in section 3). In addition, if module interfaces are well defined and some information is restricted to flow between two particular modules, these modules may communicate data directly on a private connection rather than indirectly via a global resource. When the system is modularized appropriately, all information relevant to the problem at hand is localized; the IRA does not always have to search a large, global database. This problem could be especially severe in large systems that match facts from a single working memory against rules or operators from a global set.

An IRA can often gain these advantages in organization and performance without sacrificing too much cognitive capability from limited knowledge access. This is due to the fact that there is structure in the world and coherence in problem contexts over time and space. This structure and coherence mean that there are not as many occasions for using remotely related data as there are for accessing local, directly related data. For example, if a robot is driving down a road at some point in time, it is expected that in the near future the robot will still be on the road, and that the characteristics of the road will change only a limited amount. Therefore, road-following control algorithms will be needed, but knowledge about repairing the vehicle will not be required. Agre and Chapman [1] [10] have also observed that for this kind of concrete activity, *general* reasoning is not often required. Thus, rather than using knowledge like "`for all cars $C, if the robot is moving toward $C and overtaking $C, then slow down`," the IRA can use knowledge that is *indexed to the sensors:* "`if the car in the center of the visual field is getting closer, slow down.`" This latter approach avoids having to search an internal world model for instantiations of the variable $C. Instead, the "vision" module can always send the same object to the "collision prevention" module. The robot behavior is the same in both

cases, but data access is much more limited in the latter case.

One further argument for modularization comes from the claim that intelligent behavior arises from the interaction of many independent little agents. This idea is in direct contrast to the concept of intelligence as a centralized process of searching for problem solutions and applying knowledge to all decisions. Minsky [38] was one of the first AI researchers to present the idea, calling the collection of behavioral agents the "society of mind." Agre and Chapman [1] use a set of independent behavioral rules working together to control their Pengi system. Brooks uses a flock of simple, "solipsist" [6] modules to control a real robot. Several other robot researchers [3] [14] [30] [36] have also stated that they assume that intelligent robot behavior must be achieved by the aggregation of "several independent processing units that work in parallel" [30]. In most cases these modules have a few well defined communication paths with other modules and use *indexed* facts as described above. This philosophy has shown some promise for generating useful robot behavior. However, such a bottom-up approach to designing purposeful behavior would seem to be much more complex than a top-down approach. Whereas traditional problem decomposition attempts to break tasks down into simpler components that can ideally be designed individually, the aggregation approach potentially requires the management of many interacting components at the same time. The difficulty of this design process implies that effective learning algorithms may have to be employed to develop cognitive capabilities semi-autonomously.

2.3. Modularization in Real Systems

Monolithic Systems: The early planners and expert systems (e.g., STRIPS [16] and MYCIN [53]) as well as several more recent systems (e.g. Soar [33] and Prodigy [39]), have no modules; all facts are kept in one working memory, and all rules and operators are in one group. These systems avoid modularization in order to have the maximum flexibility to access knowledge.

Some of the more recent systems use special techniques to improve their performance even with central bottlenecks. For example, Soar (and other systems based on OPS5) use a RETE [19]network to quickly find productions that are affected by new facts. However, RETE cannot ultimately guarantee the same performance that a restricted module can. An action that can be performed by a fixed routine — a low level control loop, for instance — can be isolated in a module, and receive only the same limited inputs, regardless of problem size. The routine can thus operate at a guaranteed speed even if the problem becomes more complex. On the other hand, RETE has unpredictable response times when new facts are added to the database [23]; thus a routine that is tied to the central database can have worse response times when the problem complexity grows. Without modules, all routines must share the central database.

Procedural Modules: Several IRA's incorporate structure to group procedural knowledge, while leaving facts in a central database. This grouping of procedural knowledge allows the task activities to be decomposed into independent parts, even if the data is not. The IRA can reap some of the benefits of complexity management, parallelism, and knowledge localization discussed above. The shared working memory can reduce the both the advantages and disadvantages of modularization, especially when the modules use it extensively to post partial results, hypotheses, etc. PRS [20] and BB* [25] are examples of such systems.

Partly Local Data: Many vehicle-inspired IRA's have more modularization. In these systems, procedural knowledge is kept in structured modules, and facts are kept partly in a central data base and partly in private memories with each module. The central databases in these systems are often called "blackboards." The IRA is built up from independent modules, and the central database is added to provide a special means for sharing and combining information at relatively low rates [3], [21], [24], [35], [47], [52]. The database is often a map of the physical world.

Local Data: The most highly modularized systems do not use any shared global memory, but instead allow modules to send data directly to other modules. In some cases, the connections are made via a shared communication resource such as a bus or a central switching module [14], [37], [45], [54]. Other systems use independent, direct connections between modules; for example, facts can flow directly between the Theo frames [40] as a result of a simple memory reference (however, Theo's modules are very weak because there is really no limitation on the connections between them). Brooks' Subsumption architecture [6] uses "simple" finite state automata as modules, and defines wires between them for communication. This latter system clearly avoids slow memory searches and allows a parallel implementation without communication bottlenecks, but causes the biggest limitation on knowledge access.

2.4. Summary

Modularization is a hindrance to pure cognitive ability because it limits access to knowledge. For abstract problems, a broad flow of knowledge is required, so modularization must be weak. Strong modularization allows complex problems to be decomposed into manageable pieces, makes parallelism more efficient, and allows faster access to relevant data. At the level of rapid interaction with the world, strong modularization is needed for performance — even at the expense of perfectly intelligent behavior. We will discuss the impact of performance requirements on architecture further in the next section.

3. Parallelism

The concurrent solution of problem parts is a key to improving the overall performance of a system. Thus parallelism is a major design consideration for systems in dynamic domains.

Parallel system implementations require independent modules to make efficient use of the parallel resources. Independence is important because modules that depend too much on information stored in the rest of the system (or vice versa) spend much of their time retrieving (or shipping) the information. For example, to ensure that all modules had access to all of the most recent knowledge, each module would have to update its local memory every time any other module inferred a new fact. The previous section argued that the ideal cognitive system would not have modules; thus it could not effectively use multiple processors either.

System designers sometimes do not distinguish between modularization and parallel implementation. Independent modules are assumed to execute simultaneously on parallel hardware. However, modularization does not imply parallelism because the simultaneous execution of modules can be simulated on a sequential machine. While modularization is required mostly by the *complexity* of the problem, simultaneous execution affects *performance.* This section discusses the use of parallelism in a

system that is already modular.

3.1. Sequential Reasoning

Early planners performed sequential searches through the space of solutions to find plans. Many more recent problem solvers have developed sophisticated methods for controlling search, applying knowledge, and learning solutions, but they still can work on only one problem or subproblem at a time. In view of the time demands presented by dynamic domains, it would seem that multiple processors would have to give better performance. However, parallelism is not always efficient in an IRA, and is not always obviously worth incorporating into an architecture.

Central bottlenecks can reduce the efficiency of parallel problem solving. Consider that the ultimate result of an agent's deliberations must be one action (for any one effector at one instant of time). A mobile robot cannot, for example, drive in more than one direction at a time. Even at more abstract levels of the problem, if we look at problem solving as moving from state to state toward a goal in a problem space, the system can only apply one operator at a time and go to one other state. Thus the results of parallel evaluation have to be funneled through one combining function to produce a final decision. Furthermore, if an IRA does not have enough processors to handle all subproblems simultaneously, a decision must be made as to which subproblems should be allocated a processor. In a system that strives for the best use of knowledge available, this is a central decision that must be made before the processors can begin work.

Besides the difficulty of bottlenecks, it is generally more difficult to design a system that not only allows parallel implementation, but integrates it and even coordinates it. In return, using N processors in a system will result in a speed up of at most N; this is insignificant if the computational load can vary by orders of magnitude during a task. The speedup is less than N if, for example, there is a lot of communication overhead or if all processors cannot be kept busy at some stage in the problem. Even if all processors can be kept busy, the N processors can be simulated by a single processor, and the same performance might be achieved in the near future by a single processor that is N times as fast as today's. Thus the limited benefits of parallelism may not always be worth the cost in knowledge access and system integration ease.

3.2. Parallel Reasoning

The primary benefit of parallelism is performance. Given the difficulties and limitations of parallelism described in the above paragraphs, is it worth while to consider parallelism in IRA design? The answer clearly depends on just how limited the resources of the agent are, but for typical domains of interest and the lifetimes of research programs, parallelism is definitely required. General purpose computers are far too slow to support an agent with the capabilities we have described; Moravec [42], for example, estimates that a 1 MFLOP computer is about 10^5 times too slow to emulate human vision capabilities, and another order of magnitude slower than the brain as a whole. While it is impractical at this point to put a million general purpose computers in an agent, significant performance improvements can be achieved with more limited parallelism. This gain is achievable for two reasons: first, some computations can be implemented on multiprocessors with special architectures that match the flow of data in the computation. Such processors perform these particular computations much faster than processors that

support more general information flow and in fact can often use many processing elements very efficiently. Second, system modularization can be organized by time criticality so that tasks that have short time constraints are separate from less critical tasks. If the time critical tasks can be implemented with bounded time algorithms, then dedicated hardware for each of these tasks allows the system to guarantee that the hard time constraints can always be met. Section 2 discussed how such use of independent modules does not always result in a great loss in cognitive ability because of structure in the problem.

One of the best examples of the effective use of parallelism is low level perception. Extraction of simple features may be done the same way independent of the scene or the agent's goals. The running time of the algorithms often depends only on the image size, and not on the pixel values. Input generally comes from the same sensor and goes to the same object recognition processes. The number of pixels in the input and the dynamics of even a simple task may require an enormous amount of computation per unit time. Finally, the algorithms may map easily onto certain parallel computers with special topologies. All of these factors suggest that a great performance improvement could result from the separation of low level perception into a module, and subsequent implementation on a separate processor that runs independently of the other problem solving activity. Experience with driving robots on paths has shown that some vision and control procedures really can be implemented independent of a planner and with limited communication between modules [22], and that they can be implemented on a special purpose computer to improve performance [2] [11].

In addition to performance gains, there are secondary reasons for parallelism. We have previously discussed the idea that intelligence emerges from the interaction of simple, independent agents. Research in distributed problem solving indicates that distributed systems have practical advantages in the areas of cost, reliability, fault isolation, adaptability, etc. [5]. These features arise from domain requirements other than complexity and dynamics, so we will not discuss them further.

3.3. Parallelism in Real Systems

Sequential Systems: In the most basic planning systems, parallelism is ignored. Not only are there no modules in the planning system, but planning and execution are sequential. Contemporary intelligent agent systems such as BB* [25], the reactive RAP planner [18], PRS [20], Soar [33], PRODIGY [39], and Theo [40] have continued the sequential nature of the reasoning in traditional planners and expert systems. Other systems explicitly address time constraints on planning and acting, but are nevertheless use sequential execution [13]. All of these systems assume that the proper execution ordering of various modules will allow performance goals to be met.

Independent Parallel I/O: Many planners mostly ignore parallelism, except that they assume the presence of computation *outside* of the planner. These external "black boxes" take care of important functions such as perception and control. These planners concentrate on cognitive problems and use abstractions for perception and control; from the planner's point of view, facts from the environment just appear in the data base, and commands are blindly executed. Robosoar [34] and PRS [20] are examples of existing systems that use such abstractions. The overall system design in these planners is reasonable as far as it goes (parallelization of perception, control, etc.), but it does not address the implementation of or

communication between the modules outside the planner.

Parametrized Parallel I/O: BB* [25] uses a similar arrangement of sequential problems solving coupled with parallel perception and control. However, perception and control are explicitly included in the design of the overall system. The problem solving part of the system can modify the behavior of the perceptual and control functions somewhat by commanding them to filter unwanted data or sample less frequently. The interface to the parallel processors is managed via data buffers in the working memory.

Controllable Parallel Behaviors: Payton's Reflexive Control architecture [47] exhibits a bit more integration of parallelism. Here, modules running bounded-time algorithms can be implemented as parallel processes connected with dedicated data paths. The central reasoning system controls the *behavior* of the robot by activating different sets of these modules with variable parameters and dynamically establishing the data paths between them. This scheme takes full advantage of specialization to get a fast response for time-critical computations, while retaining the ability to use knowledge to change the response. Unfortunately this system hasn't been implemented with parallel hardware, and it is not clear how efficiently it would work with the dynamic module activation and connection.

Distributed Processing: Most robot systems, as mentioned earlier, are broken into pieces that are run on separate general or special-purpose computers. The parallelism in these systems is fairly large-grained, at the level of related groups of tasks. Communication of information between processors often requires the use of a global communication resource such as a bus or a routing processor; for this reason, information transfer is generally restricted. For example, messages may contain only a limited set of commands or status reports. This limited communication has not been too inhibiting so far because the robots perform fairly simple planning tasks and the modules do not need to share a great deal of information. Examples of these systems include Dolphin [14], HILARE [21], the GSR [24], the FMC system [37], NAVLAB [52] [22], and TCA [54].

Fine Grain Parallelism: Finally, some systems are highly parallel at a low level; that is, there are many simple modules. These systems were discussed in section 2.2. In addition to the characteristics discussed there, it is clear that they have the best potential for high performance due to their highly parallel design.

3.4. Summary

While modularization and parallelism are related design issues, parallelism more directly affects system performance than modularization does. The ideal problem solving system does not require parallelism, but instead continually uses knowledge to select one action to take next. Parallelism can actually hinder reasoning in as much as it requires strong modularization to work effectively. However, the computation demands of agents in dynamic environments require the performance improvements parallelism can provide. Even limited parallelism can yield significant performance gains because special multiprocessors can be used effectively on subproblems that match the topology of the computer. Low level perception is an example of such a task. A capable IRA will require massive parallelism for demanding, isolated tasks like perception; small parallel modules for reactions that must be quick at the possible expense of cognitive ability; and large-grain parallelism for tasks that are computationally

demanding but which share data at a lower rate. The next section describes how multiple tasks can share the same processor and still meet time constraints.

4. Execution Control

In the ideal case an intelligent system would be able to find a complete solution to its current problem before the world changed significantly. In other words, for the duration of the solution computation the problem would essentially be static. However, resource limitations make it impossible to search the problem space completely before the domain dynamics change the problem. Thus an IRA must consider resource limitations and time constraints as part of the problem, and must be prepared to change its problem solving strategy from moment to moment as the changing world dictates. Even traditional planners in static domains recognize the impracticality of performing exhaustive search to solve problems. Planners in non-trivial domains limit their search by using heuristic knowledge to choose which node to expand next and which operator to apply next at the node. IRA's, as we have shown, are modular and parallel; thus they also use knowledge to decide how to allocate system resources to modules. In static domains, these choices are based solely on internal state changes. In domains in which the world changes during problem solving, these decisions are based on both internal and external state changes. Below we discuss the requirements complex, dynamic domains place on execution control, and examine some design features available to meet the requirements.

4.1. The Need for Dynamic Execution Control

4.1.1. Limits of parallelism

Although it is possible to speed up the operation of an IRA with parallel processors, there are limits to the usefulness of parallelism. Consider first the problem solving abilities of the IRA. We argued previously that an ideal problem solver facilitates the sharing of knowledge globally among weakly divided modules, and that only the requirements of the domain and available technology force stronger modularization and parallelism. We believe that in order to solve difficult cognitive problems, an IRA will have to retain some part that can support this rich interaction between (weak) modules. This part of the IRA would be difficult to implement efficiently with multiple processors; thus these modules would be implemented on a single processor. Not all researchers agree with this view; Brooks, for instance, has suggested that loosely connected modules communicating at a low bandwidth can reason about objects, make plans, etc. [6]. However, as we will show below, such a highly parallel architecture can still suffer from problems of inefficient resource utilization.

Parallelism is also limited in its applicability to independent computations. If *all possible* tasks and subproblems were allocated their own processor, then performance would be high and there would be no need to juggle the execution of several problems on a processor. However, there could be combinatorially many of these subproblems, and only a limited set would be active for any one task. The processors associated with the remainder of the subproblems would be inactive, and thus most of the system would be idle all of the time. An IRA with limited resources might be unable to afford this luxury. Even if a way could be found to allocate resources only to active tasks, some resources might still be too expensive to dedicate to one task. For example, several tasks might be required to share a special

sensor or sensor data processor. Thus while many IRA modules can be implemented in parallel, there will also be modules that have to share resources.

4.1.2. Variable processing delays

When several tasks are sharing one processor, it may be difficult for each of them to meet its time constraints when the processor is executing the other tasks. Chapman [9] showed that even simple planning is exponentially complex; thus in a complex domain where problem sizes are not predictable there could be a great variation in the amount of time needed to finish different planning problems. If some other module were using the same processor as the planner, the other module could be prevented from executing for a long time if a planning problem were particularly difficult. If that module's task had time constraints associated with it, system performance could degrade severely. Other computation tasks, such as the identification of objects in an image, are also time consuming and can vary greatly in cost. In this example, system failures could result if an unexpectedly complex image caused some modules to be starved for execution. To solve this problem, the IRA must be able to *stop modules* before they have run to completion and *start up* others in their place. Furthermore, if potentially long-running modules are to be used in such a system, it would be advantageous if they were implemented with algorithms that can provide some useful information even before they complete (e.g. the "anytime" algorithms espoused by Dean and Boddy [13]). An iterative deepening search is an example of such an algorithm [56]. For potentially long-running modules, it is also useful if they can save context and continue processing when they regain access to the processor, thus computing a solution in an incremental fashion over a period of time [28].

4.1.3. Unexpected state changes

A resource-limited IRA cannot assume that it has enough time to completely solve its problems, and it must take external time constraints into account when choosing what to do next. If the world is changing in "predictable" ways — e.g., errors grow predictably, random variables follow known probability distributions, state changes can be predicted in a probabilistic sense — then knowledge can often be applied ahead of time to determine what problems should be solved first and at what point in the future planning should give way to acting. For example, given the "anytime" algorithms mentioned above and the deadlines for solving various parts of the problem, Dean and Boddy [13] describe an algorithm for choosing the best sequence of modules to run. Horvitz [26] and Russell *et al* [49] discuss the use of decision theory to choose between computations. Horvitz shows that the *intrinsic* utility of a computation (or action) could be combined with the cost of performing it at various times to yield a *net* utility that is a function of time. These utility functions are combined with predictions of future world states to determine the best computation to perform. These techniques can help a robot solve problems such as deciding how long it can afford to go before closing a servo loop, or whether it must use sensors to find more road ahead.

The decision-theoretic approaches, while not yet perfected, offer a way to select actions in a relatively predictable environment. Many dynamic domains, however, are not very predictable. The problem with unpredictable domains is that a planning system could be in the middle of a computation or action when an unexpected event changes the situation. Execution control techniques that work in static domains are generally inadequate because these external events are not synchronized with module operations.

Decision-theoretic analyses are not entirely useful because the system cannot predict how much time is available before the event occurs, or what event it might be. The IRA must be able to detect the external state change and react to it within a limited time (determined by performance goals), even if this means taking control from a module that was previously allowed to run. For example, a decision to use cameras for a less-critical task such as searching for navigational landmarks to the side of the robot might have to be rescinded if rangefinders discover road signs or other interesting objects in front of the robot. It is apparent that there are two independent capabilities that IRA's in dynamic domains must have — selecting activities based on the current and predicted state, and reacting to unexpected events. This latter ability requires the IRA to handle *interrupts* as well as starting and stopping modules.

4.2. Control Mechanisms

4.2.1. Interrupts

A planner in a predictable environment can sample the state of the world at carefully chosen times and perform blind computations between samples. In unpredictable domains, it is impossible to predict when to sample the world for new information. Interrupts allow a system to make plans as if the world were predictable, and still react to important events even if they occur during the blind computations.

The traditional interrupts in conventional computer systems are signaled by simple binary flags. An identifying number and a priority value may be associated with the flag. The hardware of the system supports a fast context save and switch to a new module. A robot planner, however, may require more sophisticated interrupt capabilities. The detection of an interrupt may be more involved than reading a binary condition. Detection may require sensor allocation, data processing, and situation interpretation. For example, detecting a Stop sign requires having sensors pointed in the appropriate direction and doing the processing to identify the sign — if the robot isn't already looking for the sign, it will never notice it. Detection could also involve matching a logical expression (such as the left hand side of a production rule) to a database of current facts. Deciding whether and how to react to the detected condition may be more complicated than comparing priorities; it ideally involves the application of knowledge to select the most useful computation or action. Similarly, the completion of the interrupting process constitutes another internal state change, which could generally cause selection of any other module for execution. The best next action is not necessarily the one that was in progress when the interrupt occurred. Thus the stack-based flow of control provided by traditional interrupt schemes may not be appropriate for a robot planner.

A system that seeks to provide flexible interrupts like these needs a special control module that has the power to instantiate, start, suspend and kill other modules. This is the same power that a traditional multiprocessing operating system (OS) has over user processes; however, OS's do not normally select processes for execution based on their current utility for robot control, nor do they allow user processes to program the selection algorithm. Thus the IRA may provide its own control module that can accept inputs from the world, compute interrupt triggering conditions, and select modules for execution. This module may well be the kernel of the IRA. Like an OS, this kernel must sit between the hardware and the rest of the modules so that it can control their execution. The kernel may in fact interpret the modules, which are written in a special language. However, since the IRA is itself generally an application

program that runs on top of an OS, it is still possible to use the simple interrupts of the underlying OS to interrupt the control module's deliberations, should they take too long. It may even be possible for the control module to reason about its own time constraints and use of resources.

4.2.2. Polling

We have described in a previous section how a resource-limited IRA must be able to interrupt its problem-solving modules to react to unexpected state changes. Without strong resource limitations, the IRA can be designed so that all perception and detection of interrupt conditions (as discussed above) is done continuously and in parallel with other problem solving. Thus conditions in the outside world come into the system from the bottom up and interrupt problem solving tasks. However, if resources are more limited, execution control must be periodically taken away from problem solving tasks in order to process inputs and execute command outputs. This periodic processing scheme is known as polling. Polling may be implemented in a variety of ways; for example, by processing inputs after each step of an interpreted module (as mentioned above), or by using an alarm clock to trigger input processing (discussed further below).

If resources are very limited relative to the dynamics of the domain, the interrupt-based control of execution may be dominated by the polling process. For example, many real autonomous vehicles spend a high percentage of their time processing perceptual data (e.g, [52]); in these systems it is much more important to reason about perception and control operations than about problem-solving activities. IRA's for such systems are sometimes designed around a polling or scheduling mechanism to guarantee adequate performance in a dynamic environment.

4.2.3. Clocks

Virtually all real time systems use clocks. A clock is essentially a counter that ties the time-based external world to the instruction-based world inside a planner. Clocks can be used to reason implicitly or explicitly about time.

Traditional real time control systems employ interval clocks. A system designer creates a program that solves a particular problem under all conditions that he/she expects it to encounter. The designer then determines the worst case execution length of the program under these conditions. The program is run on hardware that is dedicated enough to guarantee a low enough execution time to meet performance requirements in the worst case, and an "alarm" clock is used to trigger execution at regular intervals. Servo controllers and polling input handlers are examples of such systems. Programs can also be run continually without an explicit clock, depending instead on the processor instruction execution rate for reference.

Clocks can be much more explicit in planners. We discussed above the possibility of reasoning about execution times, time varying utilities,and deadlines; this requires some knowledge of the current time to know how much time is available, and when to start and stop different modules. A clock in this case is generally an accumulating counter, rather than a timer or pulse train. An IRA planner could do some of this reasoning just based on an implicit clock, such as a count of the number of instructions executed or some other measure that was tied to time when the system was designed. However, this reasoning would fail if the planner were implemented on different hardware with a different processor clock speed.

Clocks can be used by modules as timers to trigger the module after a period of time has elapsed. Such alarm clocks are useful for implementing a delay, executing an action at a time, and avoiding deadlocks and execution starvation. These alarms essentially allow very simple OS interrupts to replace pattern matching computations; they are also a simple, distributed way of using clocks to reason about time and deadlines. For example, suppose a Stop-response module is to constrain the robot's speed based on traffic controls. When a Stop sign is detected in the distance, the module is to maintain the robot's speed at the speed limit until it gets close to the sign, and then begin deceleration. An IRA could handle this a number of ways. First, a system that guaranteed response time by frequent sampling might run the detection algorithm repeatedly to track the sign, and then take the appropriate action when it was close. This is probably wasteful of resources. A second possibility would be to establish interrupt conditions based on the distance to the sign; the Stop module would only be run again when these conditions triggered it. Of course, the distance would have to be measured continuously anyway, thereby tying up resources. Third, an IRA that reasoned explicitly about time and resources would recognize that there was an increasing utility of checking the sign again as time went on, and a very high cost of *not* checking it after a point. This utility value would have to be continually compared to the utility of other tasks as the robot's situation changed. A fourth way is to estimate the time to the deceleration point, and set an alarm for that time. The sign can be checked earlier if desired, but the alarm guarantees that the system will not "forget" the Stop-response module because of other (inappropriately weighted!) concerns. In fact, without such a guarantee, the Stop-response module might have no choice but to assume that it might never run again, and output a deceleration command immediately.

Clocks can also be important for responding to input data. Sensor data is usually passed directly to some processing module in the system, where it can be interpreted and used to trigger other modules. Since the data does not go through the underlying OS, there is no basis for using OS interrupts. However, an OS clock can interrupt processing at regular intervals to transfer control to the IRA's interrupt controller. If this *polling* is done frequently enough, the system can simulate module interrupts.

4.3. Execution Control in Real Systems

Local Execution Control: Intelligent systems described in the literature use a variety of execution control schemes. The most basic controller utilizes local decisions to pass control between modules. Execution flow can vary in response to external events, but only if a module requests information. For example, single process systems written in a conventional programming language have subroutines as modules, and control passes directly from one to another via call and return mechanisms. RSPL [48] is an operating system and programming language for "schema" modules that define robot control functions. This language has flexible mechanisms for dynamically creating, suspending, and killing modules, and for specifying execution frequencies and deadlines. Although there is a central scheduler that actually runs the modules, it cannot itself activate or suspend modules or change their specified repetition rate. These decisions are still made locally by modules for themselves or the modules they have instantiated. There is no central mechanism to interrupt module execution and start an appropriate service routine.

Theo Agent [4] is a frame based system in which control is passed between modules (taking frames as weak modules) when slot values are requested or returned. There is not currently a global means to interrupt such a chain of references if something important changes in the world. Thus the system relies

on always being able to finish its computations before its response time is up. It would take much too long to do this continually in a dynamic domain, so "truth maintenance" and "running arguments" are used to shortcut all the parts of the problem that have not changed since the last iteration. When the situation does not change much, little computation is needed and the re-evaluation cycle time is very short; new inputs can be accepted rapidly, and the system can react quickly. There is no guarantee, however, that a certain situation change will not invalidate the current problem solution and cause a long delay while the new solution is recomputed. This behavior is arguably a reasonable response to events of varying importance in the world. In some domains, though, the possibility of having a planner spend a relatively long time in an uninterruptable state is unacceptable.

Highly parallel systems often have an independent locus of execution in each module, so moving execution between modules is not an issue. These systems also depend on having adequate resources to be able to handle all important tasks within the time constraints of the task. Brooks' Subsumption architecture [6] is an example of such a system.

Other parallel systems must run several modules on each processor, so execution within each processor must be considered. In some cases there is no central coordinating processor, so local decisions are used to control execution. Processors can coordinate and synchronize with other processors by using a local module with OS features to issue and accept control messages. This allows processors to block execution and wait for data, request data and interrupt their modules when it arrives, etc. The modules in the NAVLAB [52] and TCA [54] have this capability.

Global Execution Control: Many robot control systems are composed of groups of modules running in parallel on different processors. In many cases, one module takes the role of a system manager and issues commands and requests to the functional modules, which then run independently. The GSR [24], for example, has a planning module that can send messages to the rest of the system to request that modules be started or stopped or perform some particular function. The central control module is not assumed to be fast enough to react quickly to external events, so the parallel functional modules communicate low-level information directly to one another to provide real time behavior. Dolphin [14], Reflexive control [47], and HILARE [45] also use this strategy.

Global Interrupts: Another category of systems comprises those that implement a new language and an interpreter for the language. Soar [33] and PRS [20] are examples of such systems. The interpreters in this case execute one language element (e.g., one elaboration cycle or one procedural step) and then pause to accept new data from the world. After incorporating the data, a new module may be selected and started. The interpreter effectively sits between the underlying OS and the application program, and implements interrupts through a frequent polling mechanism. Clocks are not used, so the polling is not guaranteed to occur at a frequency referenced to the external world. Application programs can not reason about the resources used by the interpreter itself, but the interpreter executes each step in a bounded time.

BB* [25] relies on OS timeout interrupts (a clock) to regain control if a module executes too long. This is because modules in BB* are not restricted to an interpreted language, so the "kernel" of the system cannot step through them. When the kernel gains control this way, it can check for internal or external

state changes and execute a new module if appropriate. The control module can even be interrupted itself and modify its own deliberations because of time constraints.

4.4. Summary

While ideal problem solvers use control knowledge to guide their search and find solutions more quickly, they do not change their focus of attention in response to external events. On the other hand, dynamic domains require systems to interrupt their activities. This requirement hinders the problem solving process because resources have to be used to detect and respond to external events; furthermore, the best inference procedures may have to be replaced by procedures that can produce a reasonable result in a limited time. This problem cannot be solved simply by using a different processor for each task because some tasks do not map well onto parallel hardware, and because resources are limited. One of the most important tools of dynamic execution control is an operating system-like interrupt mechanism that can gain control of the processor from an executing module, process internal and external data, and select a (possibly) new module for execution. Real time clocks are also indispensable for connecting the deliberations of the IRA with the passage of time in the real world. The next section will deal with a related issue, that of controlling the flow of data.

5. Communication

As we have discussed in the previous sections, the ideal problem solving system would have no strong modules and would be implemented on a single processor. The weak modules can communicate through a global, shared database. All reasoning processes thus have complete access to all system knowledge. For modularized IRA's in dynamic domains, communication is more limited, more direct, and more parallel. Communication methods help to determine how effective the modules are on one hand, and how well knowledge can be shared on the other. This section discusses various communication arrangements and their impacts on modularization and execution control.

5.1. Access Issues

The various methods of communicating between modules are distinguished by what access they allow modules to other module's data and what global information the modules need to refer to the data. Access can range from modules being able to reference facts associated with any module, to modules being able to use only a specific quantity from one other module. The reference information can be the name of another module, a value associated with the data, or the location of the data. When access is limited and communication requires specific name and location information, the modules tend to be connected closely with certain other modules; communication is potentially very fast because finding data requires no pattern matching, and because no centralized database bottlenecks data flow. In contrast, finding data by matching patterns against and implicitly specified source database allows modules to remain totally anonymous and free to be modified without big impacts to the system.

Consider first a system with several modules all sharing a common working memory. Facts inferred by the modules are posted to the implicit recipient, the working memory. No destination is explicitly given. To obtain data, modules match pattern expressions against the memory and find successful instantiations. The source of the data is implicitly just the common memory. The only global information

that must available explicitly to the modules is the list of keywords naming the facts. For example, the modules would need to know whether Stop signs are referred to by `(object (type sign) (marking "stop"))` or `(object Stop_sign)`. This associative knowledge retrieval is ideal for general reasoning.

Unfortunately, associative retrieval can be time consuming and domain dynamics may require higher speed and latency guarantees. A less flexible, but more efficient way to access data is to refer to the data by address rather than by value. This is possible if facts are indexed to concrete system resources, as described in section 2. For example, if a key concept in the system were "the closest stop sign in front of me," then a particular location in the working memory could be dedicated to that concept and its address used directly by the modules. References to this particular fact would not require any matching on the memory, and modules would not need to know any global names for the fact (although they do need to know the address).

In strongly modularized systems, modules may include specifications of a limited set of data that can be shared only with certain other modules. These limitations on access can inhibit the problem solving ability of the system unless the problem task has relatively independent components, as discussed earlier. Data references in such a direct-connection scheme require the specification of the name of the module that is sending or receiving the data as well as the name or address of the data. For instance, a source module can write data directly to a receiving module's input port. In this case the source module must know the name or address of the other module and the port; however, it only needs to know this information for the modules to which it sends data. The receiving module does not need to know anything about the source module or the connection, and can use a local name for the data it receives. The opposite connection is also possible; that is, the source module merely sets the value of a local variable, and the receiving module uses the global name of the module and port to read from the source. In both cases there are modularization benefits because of the limited global knowledge needed, but disadvantages because modules and module names cannot be changed without affecting other modules.

Another variation is to use "wires" or "pipes" between modules. Wires can be physical connections between modules, or entries in a global routing table. This scheme allows both source and sink modules to use only local names for data and to be unaware of the module on the other end of the wire. However, the wires must be established by the designer or a central module with global access to all module and port names. Modularization with wires is good from the modules' point of view, since all global information is hidden. Wires also allow the possibility of truly parallel connections, thereby providing higher performance by removing the bottleneck of shared communications channels. Unfortunately, the wires themselves prevent modules from being modified or replaced easily without disturbing the rest of the system.

Combinations of the above techniques are also possible. For example, modules may write to a limited database that is shared by only one or a few other modules. The names needed for the shared data must only be known to the sharing modules, and does not have to be global to the whole system. Another example is the use of an intermediate module to communicate data between modules. In this case modules only need to know the global address of the intermediary, which may be a fixed system manager. If a module is added or changed, only the connections between it and the manager need to

change.

All of the above communications schemes represent *logical* module interconnections. In general they may be *implemented* in a variety of ways, such as shared physical memory, networks, etc. However, only the limited, direct connection methods allow high performance parallel implementations. Thus the dynamics of the domain may require that direct communication be used.

5.2. Communication and Execution Control

The communication techniques just described are independent of the various execution control schemes discussed in the previous section. However, certain communication techniques fit particularly well with certain flows of control. For example, if data are always written to a shared database instead of directly to specific modules, then the database must be polled to find new data applicable to each module. Parallel modules may be interrupted by new information, but only if a system kernel or database manager is polling the database to match new facts with interrupt trigger conditions. Within a single processor, only one module can run at a time so the system must select one triggered module to execute. The actual flow of control is determined by what the modules put in the database; the data might be commands, subgoals, priority knowledge, information requests, computed values, etc.

Modules may also write directly or via a wire to a receiving module's input port. This technique suggests a system in which the data arrives asynchronously at the receiving module, causing the module to start up or interrupt its execution. This is especially true if the modules are implemented on parallel processors, and the wires are physical connections. Data and control can be thought to flow through such a system from source (sensors) to sink (memory or effectors) without a need for direct requests to flow in the other direction. Just the opposite is true if modules read their data from the output of another module. In this case it is convenient to think of data being provided — including sensor operation — only in response to direct requests. These examples illustrate that data communication issues are closely related to execution control issues.

5.3. Communication in Real Systems

Shared Database: Traditional problem solving systems, especially those without modules, use a shared database to hold all facts; access to domain knowledge is complete. These databases require matching operations to find appropriate data for problem solving operations. Examples include the logic-based world state description in PRODIGY [39] or the working memory in Soar [33]. Blackboard systems typically have modules that hide some information, but results are shared with all other modules on the blackboard database [24] [52] . The BB* blackboard [25] allows some data access by address reference. In particular, input data is accessed from an input buffer which lies in a particular location in the blackboard. Similarly, knowledge sources output data to the outside world by sending it to a particular output buffer.

Direct Connections: Mobile robot systems typically have several processors to perform sensor processing and navigation tasks in parallel. In order to achieve higher processing speeds, data is often sent directly between modules rather than going first to a generally accessible database. The actual implementations often utilize a memory that is shared by processes, a shared multi-processor bus or

network, or a central message router. The communications that go between modules on different processors usually require that the source module package the data or request with the name of the receiving module and send it as a message. Systems that use some form of direct connections include HERMIES-IIB [8], Dolphin [14], HILARE [45], NAVLAB (in addition to the general blackboard communications mentioned above) [52], FMC [37], and TCA [54].

Parallel Wires: Brook's Subsumption architecture [6] is a good example of truly parallel modules connected with wires. These modules do not need to know any global information about where their data comes from or goes to. Data in this system mostly flows in one direction from sensors to effectors, especially at low levels. New data arriving on module inputs can cause the module to respond immediately, if it was waiting.

Theo Agent [4] is just the opposite of Brooks' system is some ways. The frame modules in Theo compute their slot values using information obtained directly from other (globally named) frame slots. Slot values are only computed in response to a request from another module that needs that value. Thus requests flow from effector command modules to sensor modules, and data flows back in response. There is also a truth maintenance mechanism, however, that allows changes in data values to ripple forward from the source (invalidating other slot values, rather than recomputing them).

5.4. Summary

An ideal problem solver would have one central knowledge database, and therefore avoid transferring data anywhere. Systems in dynamic domains will have independent modules, though, and must specify how they communicate. Modules may reference data by performing an associative search on an implicitly shared working memory, thus finding instantiations for general knowledge, keeping module identities anonymous and decoupling module names from facts. Alternatively, modules can communicate directly or indirectly through named channels; this approach avoids the search, allows parallel communication and requires the use of more concrete objects. Execution flow in a system often naturally follows the movement of data. IRA communication should match the other design choices of the architecture; thus highly parallel computation will make good use of parallel communication, while more abstract tasks sharing a general processor can communicate via an implicit central working memory.

6. Conclusions

We have presented a set of design issues for implementing intelligent robots in complex, dynamic domains. Although they are not often discussed in the literature, there are clearly identifiable links between what systems are supposed to do and how the systems are structured. Basic structural architecture — how a system is decomposed and how data and control flow between the components — not only determines how well a system can meet time constraints, but also sets limits on the problem solving capabilities of the system. Systems that seek to be as rational as possible require designs that always allow the application of all relevant knowledge to the task. This requirement conflicts with the features needed to provide the fastest response using limited resources. This is a fundamental reason why problem-solving systems cannot control mobile robots in dynamic environments, and why

controllers cannot solve problems outside of a narrow context. For example, functional improvements obtained by adding more knowledge or new knowledge representations would not be sufficient to give a rational problem solver real-time capabilities; it would still require parallelization for speed and interruptability for reactivity. Likewise, a strongly modularized controller will never gain the ability to apply more of its knowledge to a problem by adding more processors.

An intelligent robot clearly needs features of both problem solvers and controllers. Unfortunately, these two extreme system types are too different to simply be grafted together. Several research efforts mentioned in this paper have explored how the necessary system features can be *integrated* into a system — for example, reasoning intelligently about sensor sampling rates [25], sensor conflicts [24], or parallel control processes [47]. Future work in robotics must incorporate both reasoning and reactive control to create systems that are intelligent in dynamic domains.

References

[1] Agre, Philip E. and Chapman, David.
Pengi: An Implementation of a Theory of Activity.
In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 268-272. Morgan Kaufman Publishers, Los Altos, 1987.

[2] Annaratone, M. *et al.*
The Warp Computer: Architecture, Implementation, and Performance.
IEEE Transactions on Computers C-36(12):1523 - 1538, December, 1987.

[3] Arkin, Ronald C.
Motor Schema Based Navigation for a Mobile Robot: An Approach to Programming by Behavior.
In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 264-271. IEEE, 1987.

[4] Blythe, Jim and Mitchell, Tom M.
On Becoming Reactive.
In *Proceedings of the 6th International Workshop on Machine Learning*. 1989.

[5] A. Bond and L. Gasser (editors).
Readings in Distributed Artificial Intelligence.
Morgan Kaufmann, San Mateo, 1988.

[6] Brooks, Rodney A.
A Robust Layered Control System for a Mobile Robot.
IEEE Journal of Robotics and Automation RA-2(1), March, 1986.

[7] Brooks, Rodney A.
Achieving Artificial Intelligence Through Building Robots.
AI Memo 899, MIT, May, 1986.

[8] Burks, B. L., de Saussure, G., Weisbin, C. R., Jones, J. P., and Hamel, W. R.
Autonomous Navigation, Exploration, and Recognition Using the HERMIES-IIB Robot.
IEEE Expert 2(4):18 - 27, 1987.

[9] Chapman, David.
Planning for Conjunctive Goals.
AI Technical Report 802, MIT, November, 1985.

[10] Chapman, David and Agre, Philip E.
Abstract Reasoning as Emergent from Concrete Activity.
In M. P. Georgeff and A. L. Lansky (editors), *Reasoning About Actions and Plans*, pages 411 - 424. Morgan Kaufman Publishers, Los Altos, 1987.

[11] Clune, Ed; Crisman, Jill; Klinker, Gudrun; and Webb, Jon.
Implementation and Performance of a Complex Vision System on a Systolic Array Machine.
Future Generations Computer Systems 4:15 - 29, 1988.

[12] Crowley, James.
Navigation for an Intelligent Mobile Robot.
IEEE Journal of Robotics and Automation RA-1(1), March, 1985.

[13] Dean, Thomas and Boddy, Mark.
An Analysis of Time-Dependent Planning.
In *Proceedings of the 7th National Conference on Artificial Intelligence*, pages 49 - 54. Morgan Kaufmann, 1988.

[14] Elfes, Alberto.
A Distributed Control Architecture for an Autonomous Mobile Robot.
The International Journal for Artificial Intelligence in Engineering 1(2):99-108, October, 1986.

[15] Fahlman, Scott E.
A Planning System for Robot Construction Tasks.
Artificial Intelligence 5:1 - 49, 1974.

[16] Fikes, Richard E. and Nilsson, Nils J.
STRIPS: A New Approach to the Applications of Theorem Proving to Problem Solving.
Artificial Intelligence 2:189 - 208, 1971.

[17] Fikes, R; Hart, P; and Nilsson, N.
Learning and Executing Generalized Robot Plans.
Artificial Intelligence 3(4):251-288, 1972.

[18] Firby, James R.
An Investigation into Reactive Planning in Complex Domains.
In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 202-206. Morgan Kaufmann, 1987.

[19] Forgy, C.L.
RETE: A Fast Algorithm for the Many pattern/may Object Pattern Match Problem.
Artificial Intelligence 19:189 - 208, 1980.

[20] Georgeff, Michael P. and Lansky, Amy L.
Reactive Reasoning and Planning.
In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 677-682. Morgan Kaufman Publishers, Los Altos, 1987.

[21] Giralt, Georges; Chatila, Raja; and Vaisset, Marc.
An Integrated Navigation and Motion Control System for Autonomous Multisensory Mobile Robots.
In Michael Brady and Richard Paul (editors), *First International Symposium on Robotics Research*, pages 191-214. MIT Press, 1984.

[22] Goto, Yoshimasa; Shafer, Steven; and Stentz Anthony.
The Driving Pipeline: A Driving Control Scheme for Mobile Robots.
Technical Report CMU-RI-TR-88-8, Carnegie Mellon University Robotics Institute, 1988.

[23] Haley, Paul V.
Real-time for Rete.
In *Proceedings of the 3rd Annual Workshop on Robotics and Expert Systems*, pages 277 - 282. Instrument Society of America, 1987.

[24] Harmon, Scott Y.
The Ground Surveillance Robot (GSR): An Autonomous Vehicle Designed to Transit Unknown Terrain.
IEEE Journal of Robotics and Automation RA-3(3):266-279, June, 1987.

[25] Hayes-Roth, Barbara.
Making Intelligent Systems Adaptive.
In K. Van Lehn (editor), *Architectures For Intelligence.* Erlbaum Associates, 1990.

[26] Horvitz, Eric J.
Reasoning about Beliefs and Actions under Computational Resource Constraints.
In L.N. Kanal, T.S. Levitt, and J.F. Lemmer (editors), *Uncertainty in Artificial Intelligence*, pages 301 - 324. Elsevier Science Publishers, 1989.

[27] Isik, Can and Meystel, Alexander.
Pilot Level of a Hierarchical Controller for an Unmanned Mobile Robot.
IEEE Journal of Robotics and Automation 4(3):241-255, June, 1988.

[28] Kaelbling, Leslie P.
An Architecture for Intelligent Reactive Systems.
In M. P. Georgeff and A. L. Lansky (editors), *Reasoning About Actions and Plans*, pages 395-410. Morgan Kaufman Publishers, Los Altos, 1987.

[29] Kaelbling, Leslie P.
Goals as Parallel Program Specifications.
In *Proceedings of the 7th National Conference on Artificial Intelligence*, pages 60 - 65. Morgan Kaufmann, 1988.

[30] Kanayama, Yutaka.
Concurrent Programming of Intelligent Robots.
In *Proceedings of the 8th International Joint Conference of Artificial Intelligence*, pages 834 - 838. IJCAI, 1983.

[31] Korf, Richard E.
Real-time Heuristic Search: First Results.
In *Proceedings of the 6th National Conference on Artificial Intelligence*, pages 133 - 138. Morgan Kaufmann, 1987.

[32] Laffey, Thomas J.; Cox, Preston A.; Schmidt, James L.; Kao, Simon M.; and Read, Jackson Y.
Real-Time Knowledge-Based Systems.
AI Magazine :27-45, Spring, 1988.

[33] Laird, John E; Newell, Allen; and Rosenbloom, Paul S.
Soar: an Architecture for General Intelligence.
Artificial Intelligence 33:1 - 64, 1987.

[34] Laird, John E.; Yager, Eric S.; Tuck, Christopher M.; and Hucka, Michael.
Learning in Tele-autonomous Systems using Soar.
In *Proceedings of the 1989 NASA Conference on Space Robotics*. NASA, 1989.

[35] Lowrie, James .
The Autonomous Land Vehicle (ALV) Preliminary Road-Following Demonstration.
In David P. Casasent (editor), *SPIE Vol. 579, Intelligent Robots and Computer Vision*, pages 336-350. SPIE, September, 1985.

[36] Maes, Pattie.
The Dynamics of Action Selection.
In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 991 - 997. IJCAI, 1989.

[37] McTamaney, Louis S.
Real Time Intelligent Control.
IEEE Expert :55-68, Winter, 1987.

[38] Minsky, Marvin.
The Society of Mind.
Simon and Schuster, 1986.

[39] Minton, Steven; Carbonell, Jaime; Knoblock, Craig; Kuokka, Daniel R.; Etzioni, Oren; and Gil, Yolanda.
Explanation-Based Learning: A Problem-Solving Perspective.
Technical Report CMU-CS-89-103, Carnegie Mellon University, Pittsburgh, PA, January, 1989.

[40] Mitchell, Tom M. et al.
Theo: A Framework For Self-Improving Systems.
Architectures For Intelligence.
Erlbaum, 1989.

[41] Moravec, Hans P.
The Stanford Cart and the CMU Rover.
Proceedings of the IEEE 71(7), July, 1983.

[42] Moravec, Hans P.
Locomotion, Vision and Intelligence.
In Michael Brady and Richard Paul (editors), *Robotics Research*, pages 215 - 224. MIT Press, Cambridge, Masschusetts, 1984.

[43] Munson, John H.
Robot Planning, Execution, and Monitoring in an Uncertain Environment.
In *Proceedings of the Second International Joint Conference on Artificial Intelligence*, pages 338-349. IJCAI, 1971.

[44] Newell, Allen.
The William James lectures.
1987.

[45] Noreils, Fabrice R. and Chatila, Raja G.
Control of Mobile Robot Actions.
In *Proceedings of the IEEE Conference on Robotics and Automation*, pages 701 - 707. IEEE, 1989.

[46] O'Reilly, Cindy A. and Cromarty, Andrew S.
"Fast" is not "Real-Time": Designing Effective Real-Time AI Systems.
In *SPIE Vol. 548, Applications of Artificial Intelligence II*, pages 249-257. SPIE, April, 1985.

[47] Payton, David W.
An Architecture for Reflexive Autonomous Vehicle Control.
IEEE Journal of Robotics and Automation , 1986.

[48] Pocock, Gerry.
A Distributed, Real-Time Programming Language for Robotics.
In *Proceedings of the IEEE Conference on Robotics and Automation*, pages 1010 - 1015. IEEE, 1989.

[49] Russell, Stuart and Wefald, Eric.
Decision-Theoretic Control of Reasoning: General Theory and an Application to Game-Playing.
Technical Report UCB/CSD 88/435, University of California, Computer Science Division, 1988.

[50] Sacerdoti, Earl D.
Planning in a Hierarchy of Abstraction Spaces.
In *Proceedings of the 3th International Joint Conference on Artificial Intelligence*, pages 412 - 422. IJCAI, 1973.

[51] Sacerdoti, Earl D.
The Nonlinear Nature of Plans.
In *Proceedings of the 4th International Joint Conference on Artificial Intelligence*, pages 206 - 214. IJCAI, 1975.

[52] Shafer, Steven A; Stenz, Anthony; and Thorpe, Charles E.
An Architecture for Sensor Fusion in a Mobile Robot.
Technical Report CMU-RI-TR-86-9, CMU RI, April, 1986.

[53] Shortliffe, E.H.
Computer-Based Medical Consultation: MYCIN.
American Elsevier, New York, 1976.

[54] Simmons, Reid and Mitchell, Tom.
A Task Control Architecture for Autonomous Robots.
In *Proceedings of SOAR 89.* 1989.

[55] Simon, Herbert A.
Models of Man.
John Wiley & Sons, 1957.

[56] Slate, D. J. and Atkin, L. R.
CHESS 4.5 - The Northwestern University Chess Program.
Chess Skill in Man and Machine.
Springer Verlag, New York, 1977, pages 82 - 118.

[57] Waxman, Allen; LeMoigne, Jacqueline J.; Davis, Larry S.; Srinivasan, Babu; Kushner, Todd R.; Liang, Eli and Siddalingaish, Tharakesh.
A Visual Navigation System for Autonomous Land Vehicles.
IEEE Journal of Robotics and Automation RA-3(2), April, 1987.

[58] Wilkins, David E.
Domain-independent Planning: Representation and Plan Generation.
Artificial Intelligence 22:269 - 301, 1984.