Automation in Construction 1 (1993) 371-401
Elsevier

A concurrent control architecture for Autonomous Mobile Robots using Asynchronous Production Systems *

S.S. Iyengar, Jeffrey Graham, V.G. Hegde, Phill Graham

Robotics Research Laboratory, Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803, USA

F.G. Pin

CESAR Laboratory, Oak Ridge National Labs, Bethel Valley Road, Oak Ridge, TN 37831, USA

The quest for efficient real-time response by autonomous robots in hazardous environments necessitates not only fast computational schemes in expert systems but also requires insights on high performance data structures and concurrent algorithms that lead to elegant problem-solving methods in the AI discipline. Towards this objective, this paper presents an Asynchronous Production System (APS) of architecture capable of monitoring and processing real-time information.

Keywords: Asynchronous production system; parallel computation; multiprocessing; multitasking; real-time expert system; distributed expert systems; autonomous robots.

Introduction

Intelligent robots are artificially created operational systems which can make decisions and take action autonomously. These capabilities derive from their use of expert systems to reason using and from their ability to learn from information acquired through sensory input.

When an intelligent robot operates in a hostile environment, its expert system must have the capability to recognize environmental threats. It must also be able to provide a real-time response to these threats. Traditional knowledge based expert systems do not possess this real-time response capability due to the sequential nature of their control architecture. Expert systems for Autonomous Mobile Robots thus require the integration of real-time response and control capabilities with traditional knowledge based techniques. To address this requirement, we are currently exploring the Asynchronous Production System (APS), a concurrent, rule-based inference engine capable of monitoring and processing real-time information [1-3].

* Discussion is open until August 1993 (please submit your discussion paper to the Editor on Construction Technologies, T.M. Knasel).

This paper describes the implementation of a concurrent control architecture (an Asynchronous Production System) for the HERMIES Robot. Part A describes the concurrent control architecture for the Autonomous Mobile Robot (AMR) using APS. Part B describes specific issues pertaining to the implementation of APS for the HERMIES robot.

Part A: Concurrent control architecture for the Autonomous Mobile Robot

Robots are increasingly employed in diverse applications that involve monotonous or tedious tasks, and in hazardous environments such as nuclear reactors, under-sea exploration, and battlefields. The environment in which an Autonomous Mobile Robot operates often changes rapidly. The robot encounters environmental threats which are hazardous to itself and its environment, such as the outbreak of a fire, the failure of one of the robot's internal systems, etc. These threats may occur at any time and are therefore asynchronous (environmental threats may be called external events, as these are external to the robot). A dynamic and complex envi-

0926-5805/93/$06.00 © 1993 - Elsevier Science Publishers B.V. All rights reserved

ronment like this imposes the following requirements [4] on autonomous robotic systems:
1. Rapid sensing: the capability to sense external events rapidly.
2. Real-time response: the ability to respond within a limited time period to external events occurring in its domain.
3. Interruptability: the capability to interrupt normal operation when an external event occurs and then to resume the interrupted task after responding to the external event.
4. Fault-tolerance: the capability to rely on the other functional units to continue operations in the event of an internal failure.

1. Specifications for the Autonomous Mobile Robot control system

The heart of an Autonomous Mobile Robot is its control system. The control system features which 'are critical to an AMR's successful operation are multitasking, multisensing, robustness, and interruptability [5].

Multitasking: Robots operating in dynamic and hostile domains must often pursue multiple goals due to asynchronous changes in environmental conditions. These multiple goals may conflict with each other. On such occasions, the control strategy must be able to weigh their relative importance and attend to the most important one.

Multisensing: Sensors are the most important part of Autonomous Mobile Robot systems, as they communicate external events to the robot. The robot control system must be able to process data from multiple sensors concurrently and must be able to rapidly assimilate it into its data structure for quick decision making and subsequent response. This implies that the control system must be able to begin a fresh control cycle concurrently with existing ones.

Robustness: When some of the sensors fail, the robot must be able to continue functioning with the remaining ones. This calls for an efficient control strategy which provides intelligent decisions even under conditions of incomplete or uncertain data.

Interruptability: Higher priority environmental threats must be able to interrupt normal operations of the robot. The robot must also be able to resume its original task after responding to the threat. Therefore, the robot's control system must be able to halt an existing control process and later resume that process after completing the new control cycle initiated by the higher priority task.

The following techniques have been used by several researchers to build a parallelized control system for Autonomous Mobile Robots:
1. Decomposing the control system with respect to parallel task-achieving behavior of the robots, rather than by the traditional way of decomposing along functional units [6].
2. Using concurrent programming techniques [7].
3. Using a distributed control system consisting of master/slave processes [8].

In this paper we focus our efforts on the concurrent and distributed programming techniques for building an expert system to achieve modularity of control and response.

2. A real-time expert system for Autonomous Mobile Robot control

Earlier, we discussed the characteristics of the complex environments in which the mobile robots must operate. We also discussed the need for AMR control systems to make intelligent decisions in order to respond to environmental threats. Our task is to build a real-time expert system to make intelligent inferences from the environmental data. It must employ an efficient control strategy and must meet the specifications listed in the previous section.

2.1. Limitations of traditional expert systems

Some of the various limitations of traditional expert systems which must be overcome to achieve effective, real-time control in AMR applications are their inadequate infrastructure, slow speed, and non-interruptability.

Inadequate infrastructure: Traditional Expert Systems like OPS5 do not have the infrastructure for collecting external data and representing them in their data structure. Thus they are insensitive to external events occurring in their domain.

Slow speed: The execution speed of these systems cannot guarantee a real-time response to the external events.

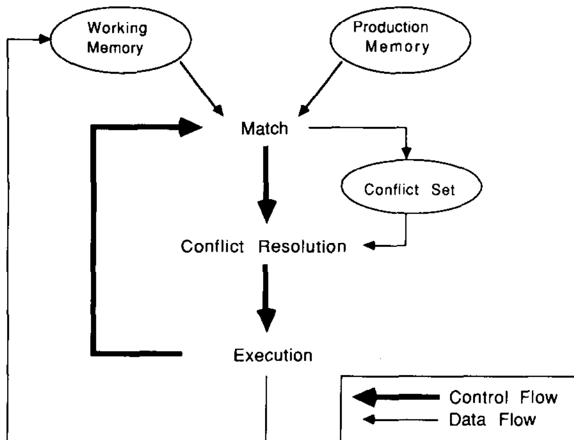


Fig. 1. The recognize-act of a conventional production system.

No interruptability: The control structure or production cycle of these systems as indicated in Fig. 1. is essentially a sequential and synchronized process [8] and hence the production cycle cannot be interrupted. Further, during an external emergency event a new cycle cannot be concurrently initiated to respond to the situation. Due to the sequential nature of the control structure, the control processes are implemented in a uniprocessor environment. This deteriorates the real-time performance as an emergency event has to wait to be processed by the only available processor.

The limited control structure of traditional production systems is thus a major hurdle which prevents their adoption for use in real-time control applications. Considering the above shortcomings, it was imperative for us to build an Asynchronous Production System to overcome them and achieve real-time performance.

3. Asynchronous Production System

An APS is a rule-based expert system which is capable of monitoring and responding to real-time events in its domain. It integrates the ease of knowledge representation of traditional production systems with real-time response and control capabilities. The real-time capabilities of an APS thus make it an ideal choice for controlling Autonomous Mobile Robots operating in hazardous, dynamic environments.

3.1. Data structure of APS

An APS has a working memory and a production memory similar to traditional production systems. In addition to these, we have incorporated a new data structure called the External Input. This is a global dataset which is used by control processes for interprocess communication. External Input has elements to represent facts and assertions about asynchronous external events. External Input elements are syntactically similar to working memory elements. The asynchronous stimuli are obtained through sensors and are integrated into the External Input data structure.

3.2. Partitioning the data structure for concurrency

The data structure of an APS is partitioned to form four, well-defined Global Data Sets in order to achieve concurrency of control. The four global datasets are Working Memory (WM), External Input (EI), Conflict Set (CS) and Select Rule (SR). Various production cycle processes communicate through these global datasets. Each global dataset serves as an input dataset for some process and as output dataset for some other pro-

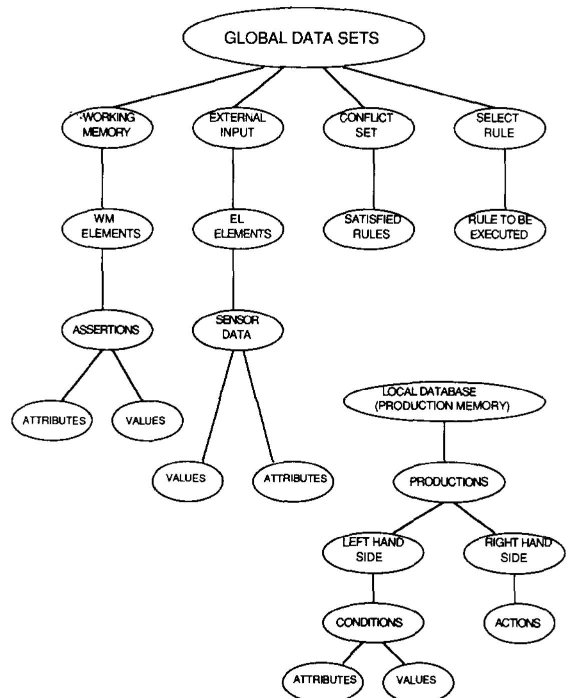


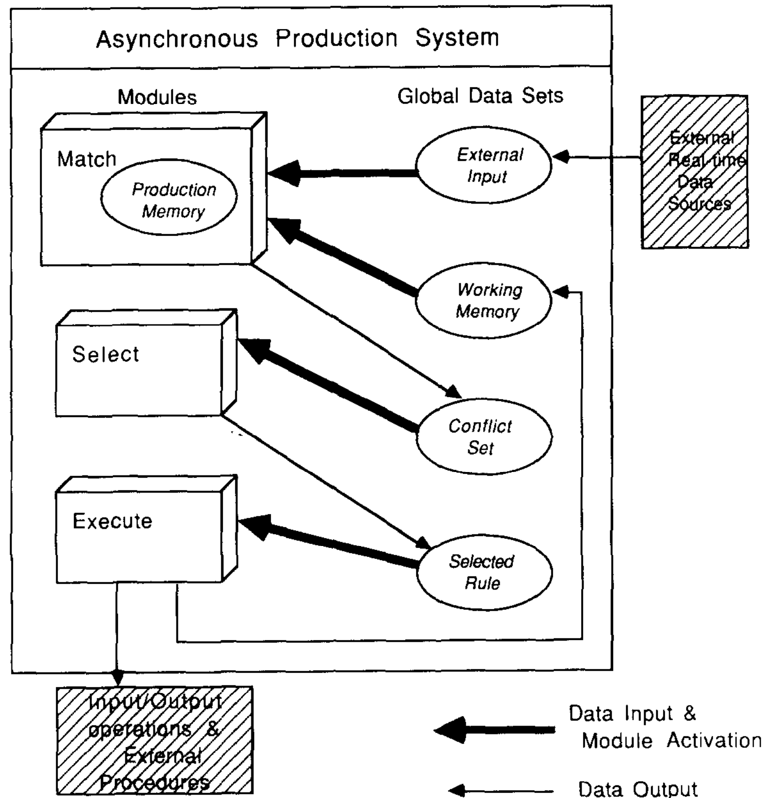
Fig. 2. APS database.

cess. The data structure (Fig. 2) of an APS plays a vital role in achieving real-time performance.

3.3. Control architecture of APS

The APS control architecture (inference engine) consists of three distinct processes: MATCH, SELECT, and EXECUTE. Each of these processes has one or more input datasets and an output dataset (see Fig. 3). Each process is invoked when there is a change in one of its input datasets. The module halts after writing appropriate results to the output dataset. We can observe that the processes are essentially asynchronous in nature due to the fact that changes in different global datasets occur asynchronously. The completion of a process indirectly results in the invocation of another process since the output dataset of one process is the input dataset of another process. Each process deserves some further detail.

The MATCH process matches External Input Elements with productions in addition to its traditional role of matching Working Memory Elements with productions. The input datasets for the MATCH process are Working Memory and



APS Execution Mechanism		
Modules	Input Data Sets	Output Data Sets
Match	Working Memory External Input	Conflict Set
Select	Conflict Set	Select Rule
Execute	Select Rule	Working Memory

Fig. 3. The execution mechanism for the APS.

External Input. The output dataset is Conflict Set. The MATCH process is activated by changes in the Working Memory or the External Input. After the MATCH process is completed, it writes the set of satisfied productions to the Conflict Set and waits for another change in its input datasets.

The SELECT process reads its input from Conflict Set dataset and writes its output to the Select Rule dataset. This process is activated by changes in the Conflict Set. It performs conflict resolution algorithms to select one of the rules for execution.

The EXECUTE process reads its input from the Select Rule dataset and writes output to the Working Memory dataset. It executes the right-hand side (RHS) of the selected rule, which may consist of some specific instructions to manipulate the Working Memory, to command HERMIES to perform some task, or to invoke a user-defined function. The ability to call user-defined functions from within a rule makes it possible to use the APS for a variety of applications which require domain-specific algorithms. The EXECUTE process is activated by a change in the. Select Rule dataset. It halts after executing all the RHS actions of the selected rule. This process differs substantially from the act phase of the traditional production systems.

For more details about the Data structure and Control structure of an APS, the reader is referred to previous publications on this subject [1-3].

4. Task partitioning of the APS production cycle for concurrency

We observed in the previous section that the the production cycle has been partitioned into three modular, independent processes, each of which functions asynchronously with respect to the other. This is the key factor in achieving concurrency. Each process is activated by a change in the corresponding input dataset and hence more than one process may be invoked concurrently if there is a change in more than one dataset at any time. To discuss this in detail, we will take a close look at the control process.

A change in Working Memory or External Input triggers the control mechanism. A change in any one of these datasets invokes the MATCH process. The MATCH process individually matches the data from both of these datasets with productions to find a set of productions which are completely satisfied. It then writes this set of productions to the dataset "Conflict Set" and halts. Since the MATCH process is the initial and most crucial process, it must be rapidly invoked by each external event. Hence, we must be able to invoke this process concurrently with the other active processes to respond to external events immediately. This is clearly achieved by the APS, as the external emergency event changes the External Input data structure directly and this in turn immediately activates the MATCH process.

Furthermore, the activation of the MATCH process concurrently with other active processes triggers a chain of processes. The MATCH process halts by writing the set of satisfied productions into the "Conflict Set". Since the Conflict Set is the input dataset to the "Select process", this process is activated at the completion of MATCH process. Observe that this is also activated concurrently with the other processes. SELECT process, in turn, activates the EXECUTE process by writing the selected rule into the dataset "Select rule". Thus there is complete modularity and concurrency between the various stages of the control process.

5. Levels of separation of the control mechanism

The effects of parallelizing each task of the APS production cycle can be viewed from three distinct points of view: Process Level, Processor Level and Interrupt Level. Each of these levels contributes to the overall concurrency of the control task. The different levels of separation are interrelated. Synchronization of all the elements of a level is the key factor in the efficient operation of a concurrent control mechanism. In the following section we discuss the salient features of each of these levels and the synchronization scheme involved in various stages.

5.1. Process level separation

Process level separation denotes the separation of the control task into multiple processes, each of which is independent and performs a specific function in the control structure. The

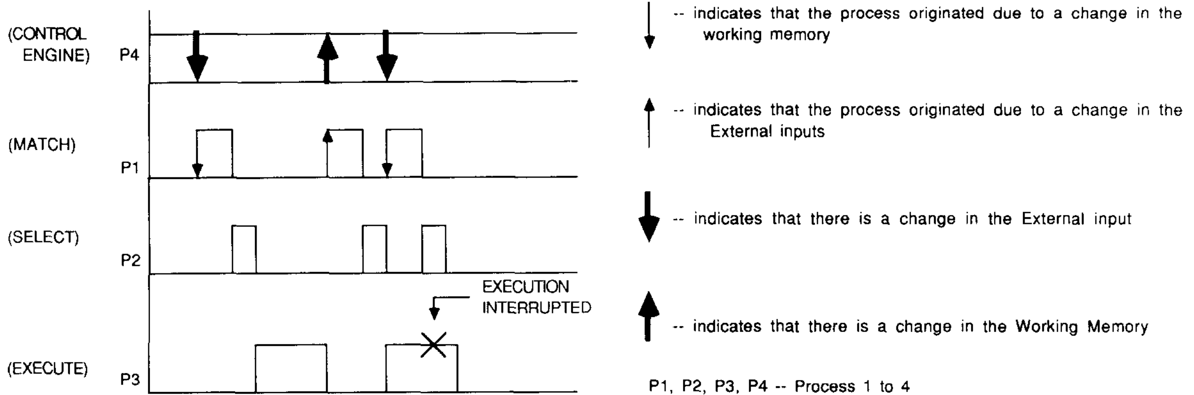


Fig. 4. Concurrent execution of production cycle processes.

APS control architecture consists of four processes. Three processes perform the function of the MATCH, SELECT and EXECUTE of the APS production cycle. The fourth process, the inference engine of APS, monitors the real-time environment and provides process synchronization between the various processes. It also resolves global dataset access conflicts between processes. The different processes execute concurrently, as they are generally implemented in separate processors. Fig. 4 indicates the concurrency in the execution of various processes. The figure shows that while the set of processes {MATCH, EXECUTE} or {SELECT, EXECUTE} may be activated simultaneously, the set {SELECT, MATCH} cannot. This is because the SELECT process takes a small part of the total production system cycle time and is activated only when the Conflict Set is modified. This observation is important with respect to the "Retraction of a fact," discussed later.

5.1.1. Process sychronization

The synchronization of the various processes involves sensing the completion of one process and activating the next process in the production cycle. The various processes of the production cycle are event-driven. An event declares a change in the status of the system. There are two types of events. An External Event and an Internal Event. An External Event is one which indicates a significant occurrence in the External world or the domain, such as a fire alarm or radiation leak, which causes a change in the External data. An Internal Event is one which indicates the completion of one of the processes which, in turn, implies that there is change in one of the global datasets. An Internal Event originates at a process and is interpreted by the inference engine, giving an activation signal to activate the process which awaits the occurrence of that particular Internal Event. The External Event is sensed only by the fourth process, the inference engine. The other events originate at the respective process and are sent to the inference engine process which then sends an activation signal to the next corresponding process in the production cycle. The process synchronization scheme is indicated in Fig. 5.

5.2. Processor level separation

The essential criterion for achieving concurrency and task partitioning of the production cycle is that the different concurrent processes must be implemented on different processors in a

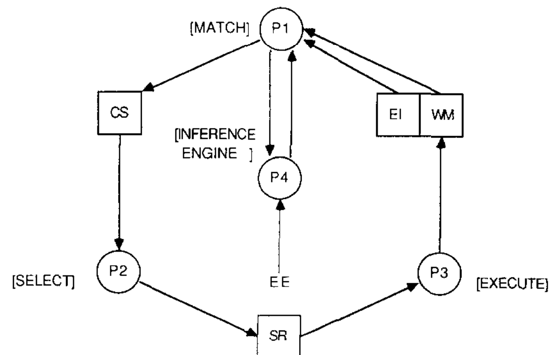


Fig. 5. Event based process synchronization scheme.

distributed processing environment. Ideally each process is implemented on a separate processor. However, we may combine the SELECT and EXECUTE processes and implement them on a single processor. This is possible because SELECT takes only a small part of the total production system cycle time. At this level, the different processors communicate through shared memory and thus they influence the execution of each other by altering the global datasets. Therefore synchronization between the different processors is essential for shared memory access. Apart from the three processors for three production cycle processes (viz MATCH, SELECT, EXECUTE), another processor is necessary for implementing the APS inference engine. The inference engine controls the activation of the three processes of the APS, arbitrates the shared memory access requests from different processors, and asynchronously monitors the external real-time inputs. Hence the multiprocessor architecture is an essential part of the concurrent control architecture.

5.2.1. Multiprocessor architecture

An APS requires a multi-processor architecture where four processors work in MIMD mode to host the different processes of the production cycle and inference engine of APS. The processors exchange information through a global memory, access to which is controlled by the processor which hosts the APS inference engine (Fig. 6). We may implement the control structure using three processors. One CPU for MATCH process, another one for SELECT/EXECUTE process, and a third processor for Real time Monitoring In this configuration, the third processor also hosts the APS inference engine. The global memory access requests are routed through this processor. The MATCH processor will write the set of selected rules (viz, Rule buffer) on to the Shared Memory. This rule buffer will be read by the SELECT/EXECUTE processor which in turn writes the selected rule on to the shared memory. The SELECT/EXECUTE processor also executes the RHS of the selected rule and modifies the global memory.

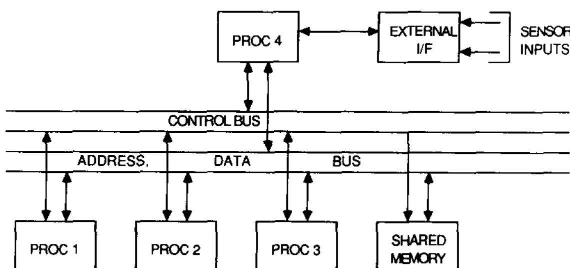


Fig. 6. Multiprocessor architecture (PROC 1 hosts MATCH process; PROC 2 hosts SELECT process; PROC 3 hosts EXECUTE process; and PROC 4 hosts inference engine).

5.3. State interrupt level separation

In order to activate a new MATCH process when an External Event (Interrupt) occurs, the process management techniques used by the operating system may be employed. This technique calls for spawning the current MATCH/SELECT process and thereby creating a child process. Then a new foreground MATCH process is created for the higher priority interrupt which occurred most recently. When the new MATCH process corresponding to the higher priority input is completed, the spawned process may be brought to the foreground and executed, if it is still incomplete. The different processes must be prioritized for proper scheduling at the process management level.

6. Distributed processing for improved concurrency

An Autonomous Mobile Robot needs to perform several distinct tasks: navigation, analysis of terrain topology, obstacle evasion, etc., each requiring a certain type of knowledge base. A distributed expert system capable of integrating several expert subsystems, each having a large independent, knowledge base for handling different prioritized tasks, will result in a higher level of concurrency. Such a system consisting of a central coordinating shell can invoke a number of expert subsystems concurrently, each addressing a particular prioritized task. This results in better real-time performance, as each one of them processes only the relevant knowledge-base, thereby avoiding garbage collection. This also avoids the combinatorial explosion.

7. Salient features of the concurrent control architecture

7.1. Interruptibility and state recovery of processes

As different processes are concurrently activated due to changes in the data structure, more than one goal may contend for execution. This can occur during an emergency input because the MATCH process (which is the initial phase of the production cycle) executes concurrently and in turn gives rise to the execution of SELECT, EXECUTE processes. Thus, a SELECT process may complete its execution and the selected rule may be awaiting execution while the execution of the previously selected rule is still in progress. If the

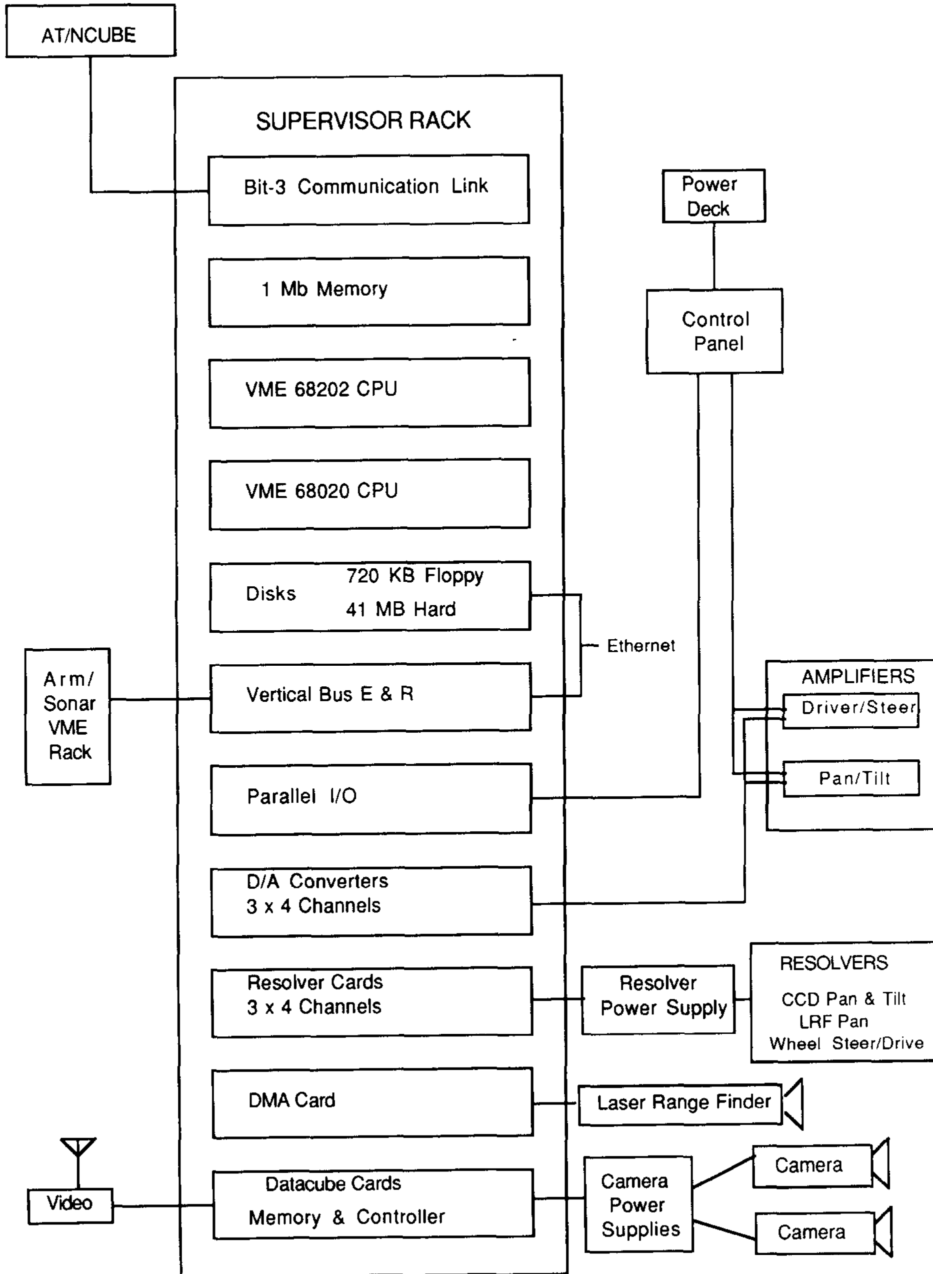


Fig. 7. The HERMIES III robot.

emergency must be attended to immediately, then the execution of the existing rule should be interrupted, as the emergency event has the higher priority. After processing this exception the execution process has to recover its previous state and resume the execution of the previous rule. So the system must have the capability to be interrupted and also the capability to recover its old state.

7.2. Prioritized execution

As discussed in the previous section, if there is an execution contention the system must be capable of choosing one goal to execute from among the goals which are in contention. This means that each goal must have a priority assigned to it before it reaches the execution stage. Also, there must be perfect co-ordination among the various processes in processing the prioritized goals.

7.3. Dynamic reconfiguration

Concurrency results in additional computing load on the system due to invocation of multiple processes simultaneously. This is because the system has only limited resources. Concurrency results in meaningful improvements in real-time performance only if the system dynamically reconfigures itself to the changing environment and properly allocates resources according to the priority of the events.

The above features indicate that the control structure of an APS satisfy all the real-time requirements of an Autonomous Mobile Robot. From the foregoing discussion on the concurrency of control, it is evident that concurrency can be added to some of the most important criteria for the real-time performance (listed in the introduction) of the Production systems. Hence it can be seen that APS satisfies all the major criteria for Real-time performance.

Part B: APS implementation issues

1. Software development environment

APS and the Autonomous Robot Simulation System (ARSS) were developed in an environment rich in state of the art hardware and software. Most noticeable was ORNL's HERMIES III robot—a high-technology autonomous mobile platform complete with a laser range finder, sonars and cameras. Development of APS centered on the Motorola 68020 processors found in HERMIES, while ARSS was developed on a Silicon Graphics IRIS workstation. An ethernet LAN provided process communication and high-speed data transfer between the two systems, allowing the development team to test APS on the ARSS simulator. Details of the development hardware are provided in the following section.

Complementing the high-tech hardware, the software development environment included the Motorola OS-9 real-time operating system and the powerful simulation software IGRIP. Details of these packages are provided in Section 1.2.

1.1. Hardware

1.1.1. HERMIES III

HERMIES III is a battery-powered mobile robot designed to support the development of autonomous capabilities in performing complex navigation and manipulation under time constraints while dealing with imprecise sensory information [9].

In order to achieve the ultimate level of autonomy in a hostile environment, HERMIES is made fully aware of its environment by the acquisition and fusion of data from a sensor suite that includes an Odetics laser range finder, four ccd cameras and thirty-two sonar transceivers.

Sensory processing and mechanical control are handled by a 16 node Ncube hypercube machine in conjunction with five Motorola 68020 processors mounted in VME bus racks. In the event more computing power is necessary, HERMIES can receive instructions and data via radio link to other computer systems. For a detailed description of HERMIES III, the reader is referred to [9] and Fig. 7.

1.1.2. Vislab VME system

Vislab ("Vision Laboratory") is a standalone computer system consisting of three Motorola 68020 processors mounted in a VME bus. Each processor has 2 megabytes of resident random access memory. An additional 2 megabytes is shared between them. Vislab is a node in an

ethernet LAN shared by the Silicon Graphics workstation. Vislab was the main develop-and-test platform for the APS system.

1.1.3. Silicon Graphics workstation

An IRIS 4D/60T workstation from Silicon Graphics was used to develop the ARSS simulation system.

1.2. Software

1.2.1. OS-9

The OS-9 operating system on HERMIES and Vislab is a multiuser, multitasking operating system featuring a real-time kernel. Because of the time-critical nature of APS, OS-9 seemed an appropriate choice for the project; however, because it is not multiprocessing, OS-9 lacked some of the support tools needed for the development.

Two problems with OS-9 proved constantly irritating during development. A severe lack of error handling by the operating system often lead to VME bus errors, cryptic error messages, and suspended processes due to the apparent corruption of the kernel. The second problem was the most detrimental to the project. Because OS-9 is not multiprocessing, there are no system-level commands for handling processor synchronization, i.e. no atomic hardware instructions (Test-and-Set, Swap, semaphores) for dealing with the critical sections of APS. A software solution to APS's Readers/Writers problem had to be implemented. See Sections 2.3.3 and 2.4 for details.

1.2.2. Unix

Unix is known as an excellent portable development platform. The inherent relationship between Unix and the C language was a definite plus, offering an extensive library of system calls for device control and communications. The Silicon Graphics workstation uses the Unix Operating system.

1.2.3. IGRIP

IGRIP is the acronym for Interactive Graphic Robot Instruction Program. This software package provides a convenient framework for building the various components of a complete simulation system, and offers an effective scheme for defining relationships between the components such as degrees of freedom, positional dependencies, and motion velocities. The IGRIP software package is layered on top of Unix on the Silicon Graphics workstation.

2. Development of APS

The development of APS followed a structured software engineering approach, framed within the successive versions lifecycle model. This section presents the details of APS construction-from requirements specification and design decisions through implementation and testing. It concludes with an evaluation of the development effort.

2.1. Requirements specification

2.1.1. Real-time, continuous monitoring for environmental threats

APS shall immediately recognize any of a set of predefined environmental threats and react to them. This requirement presupposes the ability to acquire and process data from the environment in time to effect a favorable change.

2.1.2. Graceful suspension and resumption of non-threat rules

At the time when an environmental threat is recognized, the currently executing non-threat rule shall be suspended. After the threat is abated, the suspended rule shall resume execution. This requirement implies that a facility must exist to capture and store the current state of the expert system (and robot), execute the threat rule, restore the saved state, compensate for state changes due to execution of the threat-rule, and resume execution of the suspended rule.

2.1.3. Most dangerous threats attended first

If during the execution of a threat rule a more dangerous threat is recognized, the low-threat rule shall be suspended and resumed after execution of the high-threat rule. This requirement is a direct corollary of requirement 2.1.2. above, adding the ability of APS to handle more than one threat at a time in some predetermined prioritized fashion.

2.1.4. Execution of existing rule bases

APS shall execute CLIPS rule bases with predictable results. This requirement is necessary

due to the large number of CLIPS rule bases currently in use at Oak Ridge National Labs. The rule implies that if an environmental threat is not detected, APS will behave in exactly the same manner as CLIPS.

2.1.5. Portable, platform-independent code

APS shall be coded using only ANSI-standard C language functions, constructs and features. This requirement guarantees that APS will be portable across hardware platforms.

2.2. Lifecycle model: successive versions

2.2.1. Description

Product development by the method of successive versions is an extension of prototyping in which an initial product skeleton is refined into increasing levels of capability. In this approach, each successive version of the product is a functioning system capable of performing useful work ([10], p. 52).

2.2.2. Choice justification

The successive versions lifecycle has several advantages over the conventional waterfall model of software development. It permits the exploration of technical issues concerning implementation. It also allows evaluation before proceeding to the next stage of development. Since all technical issues could not possibly be known or accounted for before hand concerning APS, this development model matched the requirements of the project perfectly.

2.3. Design alternatives and decisions

2.3.1. Genesis—modification of CLIPS and starting from scratch

The premier design decision for the project was whether to modify the existing CLIPS production system or to develop an entirely new production system from the ground up.

CLIPS was developed by the Artificial Intelligence Section of the Mission Planning and Analysis Division of NASA. It represents state of the art production systems- utilizing efficient RETE pattern matching networks for high-speed selection and execution of complex rule bases and providing easy extensibility and modification of it's C code.

Modifying CLIPS has the advantage of satisfying requirement D (execution of existing rule bases) with little effort. In addition, the development time of APS by modifying CLIPS would be much shorter than constructing an entirely new system from scratch.

The major drawback of using CLIPS as a starting point of APS lies in its data structures. CLIPS is very efficient, and that efficiency is due to several factors:

1. Strong common coupling: modules are tightly bound together by the global data structures ([10], p. 148). By separating the modules from the global data, we increase the complexity of interprocess communication.
2. Weak communicational cohesion: modules refer often to the same set of input and/or output data ([10], p. 149). This condition made it difficult to decompose the CLIPS system into the separate, asynchronous modules MATCH, SELECT, and EXECUTE.
3. Large "scope of effect": a significant change in the run-time behavior of CLIPS occurs when the outcome of a decision within a module (such as in an "IF-THEN-ELSE" construct) is changed. Tracing the flow of execution through CLIPS was difficult.

Another (slight) disadvantage of modifying CLIPS is its extra code for supporting features that are not useful to APS or that may slow its execution speed. This code and its supporting data structures must be removed.

Starting from scratch offers the advantage of designing a system that exactly satisfies the requirements specification; a small, fast inference engine with distinct, separate modules for each phase of execution. However, starting from scratch involves extensive time for planning and design—time not available to the development team. For these reasons, the APS project team focused on the metamorphosis from CLIPS to APS.

2.3.2. Load balancing—asynchronous processes and processors

The decision to modify CLIPS lead to the next major design alternative—determining the number of processors and processes running on those processors in order to achieve the required level of performance.

Adhering to the philosophy of Task Level Partitioning, there would be four processes in APS—MATCH, SELECT, EXECUTE, and a real-time monitor process RTM for detecting environmental threats. To run at the fastest possible speed each process should be dedicated to a separate processor; however, faced with only three processors available for the APS system (the others are dedicated to performing other necessary robotic tasks), the four processes had to be combined in such a way so that the CPU load was distributed evenly among them. The logic behind mapping processes onto processors is of some interest.

By far, the most complex and computationally expensive task in an expert system is the search for matching rules. Therefore it is empirically obvious that MATCH should have an entire processor dedicated to it. That leaves either SELECT or RTM to be shared with EXECUTE.

SELECT is simplistic and not computationally intensive. It is also dataflow dependent with EXECUTE, writing data to the Select Rule dataset. RTM is I/O bound (constantly interrogating physical devices for indications of threats) as is EXECUTE (robotic motion commands), so these should not be combined on a processor. That forces SELECT and EXECUTE to share a processor, with RTM having an entire processor to itself. This is also a practical decision because it gives APS the best chance of fulfilling requirement A (real-time continuous monitoring for environmental threats); the RTM runs at the fastest possible speed on a processor with no other tasks to perform.

Summarizing, MATCH will be alone on processor #1. SELECT and EXECUTE will share processor #2, and RTM will be alone on processor #3. Figure 8 graphically illustrates the process-processor map.

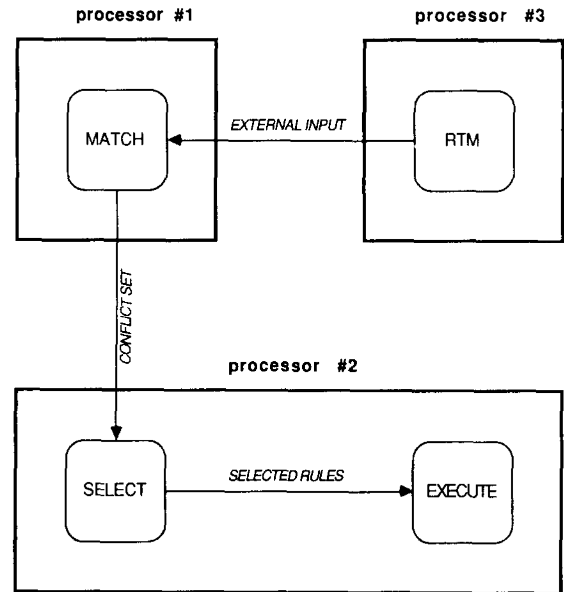


Fig. 8. APS processor/process map.

2.3.3. Interprocess communications—pipes, shared memory and signals

After determining the process-processor map of APS, attention focused on how best to implement interprocess communications (ipc). The choices available were pipes, shared memory, and signals.

Pipes offer several useful features. Implementation is straightforward—processes read and write to pipes just like to a file. The serialization of data through a pipe ensures that data will not be stepped on or lost by the process at the receiving end. A disadvantage of pipes is that they are relatively slow—a fact that cannot be ignored when constructing a real-time system. And, although serialization may be viewed as an advantage in some sense, it is its biggest disadvantage; incoming data is not randomly available, i.e. important data is not revealed until all data before it has been processed. This situation is not acceptable, especially for communication with RTM which must deal with prioritized threats and notify SELECT/EXECUTE to handle it. For these reasons pipes were removed from consideration for ipc.

Signals offer efficient interprocess communications, but they are characterized by their interrupt-like nature. Interrupting a rule in mid-execution would leave APS in an uncertain state, preventing its resumption after the interrupt is handled. This would prevent satisfying the requirement of Section 2.1.2, so signals were also excluded from consideration.

The advantage of using shared memory is that data is available to all processes without the work involved in reading and writing though a pipe. Setting up shared memory is not difficult, and is much more efficient (faster) than the pipe architecture. Shared memory would seem a more appropriate ipc implementation. The disadvantage

Table 1
APS environmental threats

Threat category	Priority
Fire alarm	100
Security alarm	50
Power failure emminent	1000
Radiation leak	150
Camera failure	1000
End-effector failure	1000
Collision emminent	1000
Sonar failure	1000

of shared memory is that, as mentioned in Section 1.2, OS-9 lacks any multiprocessing capabilities and as such lacks the atomic instructions necessary to ensure data integrity when multiple processes attempt to read/write a common memory address. To handle the read/write synchronization between processes and global datasets a Bakery algorithm should be implemented ([11], p. 335).

2.3.4. Environmental alarms: threat detection and prioritization

The final design alternative is the choice of environmental conditions that must be monitored in order to identify the existence of a threat, and the priority in which simultaneous threats will be handled.

For testing purposes, eight environmental factors are continuously monitored: radiation detector, fire alarm, security alarm, collision detector, power supply, sonar failure, ccd camera failure and end-effector failure, These "threats" are prioritized as shown in Table 1, with the higher priority threats demanding the greatest attention.

2.4. Implementation

With the critical design decisions made, APS implementation began. As discussed in Section 2.2, the Successive Versions lifecycle model was used. All together, four distinct versions of APS were created.

2.4.1. APS Version 1—one processor, one process

In the initial version of APS, the MATCH, SELECT, and EXECUTE processes were modularized into separate, callable routines. No attempt was made to decouple the data structures or to alter the control flow of CLIPS. Version One provided the initial exposure to the CLIPS code and to the data and control flow mechanisms within.

Experimentation was conducted during this version to establish the basis for the shared memory and to test the algorithm for handling the read/write contention for the global datasets.

2.4.2. APS Version 2—one processor, three processes

In APS Version 2, MATCH, SELECT, and EXECUTE were decoupled into separately compiled and executable modules. Shared memory, tested during Version 1, was implemented with success. At this point, CLIPS and APS executed any rulebase with exactly the same results.

2.4.3. APS Version 3—two processors, three processes

APS Version 3 saw the addition of a 68020 processor to the system and migration of SELECT and EXECUTE to it. It was at this point that the problem of bus errors began to appear; APS was in resource contention for the address bus! The problems were intermittent, and no solution was implemented. Also noticeable at this point was the first variations in run-time behaviors between CLIPS and APS. The change in execution proved to be caused by the change of order in which facts are asserted by the EXECUTE process. By prioritizing each rule in the rulebase, comparable run-time behavior between the two expert systems was reestablished.

The problem of rule suspension and resumption was also studied in Version 3. The difficulty lies not in suspending the currently executing rule, but in resuming execution at some later time. The state of the expert system must be preserved in some structure for resolution when the rule is reactivated. Another problem arises at the time of reactivation because the possibility exists that the robot itself is no longer in the same state as before (it has probably moved), so some plan must be in place to resolve the differences in state so that the rule may resume execution. Though this problem is solvable, time constraints and the focus on real-time execution prevented the implementation of this requirement.

2.4.4. APS Version 4—three processors, four processes

The final version of APS was marked by the addition of the real-time monitor process RTM

and its processor. Environmental alarms and triggers were simulated using keyboard input to the APS system. After an alarm goes off, RTM accesses the shared memory module and initializes a flag indicating the occurrence of the proper threat. MATCH reacts to the flag, finding all rules that will handle the threat. Next, it signals SELECT/EXECUTE to suspend the current rule and execute the threat rule. This method of handling the external input proved the most reliable. Response time of HERMIES to the threats was excellent. No problems were detected in this final phase of development.

3. Testing

Throughout the development lifecycle of APS, exhaustive testing was conducted to ensure proper execution of existing CLIPS rulebases. To test the APS requirements of Section 2.1, a new rulebase had to be constructed.

3.1. Test data

APS was designed to run on the HERMIES III autonomous mobile robot at ORNL. Unfortunately, at the time of APS development the robot was unavailable. Testing proceeded using the ARSS that was developed as part of this project. The test "data" used for APS consisted of the rulebase found in APPENDIX A. Note the prioritization of the rules (known as "salience" in CLIPS) used to maintain a stable behavior of the rulebase. Also note the calls to external subroutines. These routines are provided for reference in APPENDIX B.

Of interest in the rulebase is the mechanism employed to determine if, after a hardware failure, the current task can be executed to successful completion. For example, a series of rules is executed with the goal of closing a radiation valve. One of the rules requires the use of the end-effector. While executing the navigation rule to travel to the valve location, the RTM detects a servo failure in the end-effector—it is damaged and will not function correctly. The navigation rule is suspended so that the hardware failure can be attended by its special rule. One of the rhs actions of the failure rule is to retract any rule that requires the end-effector; the rule to close the radiation valve is retracted. An alternative plan of action may now be executed to achieve the same goal—perhaps the robot will flip an electrical breaker. This feature of APS will prove very useful in the real world of robot hardware failures, contingency plans, redundancy and survivability.

3.2. Test results

Requirement 1.2.1.
Real-time, continuous monitoring of environmental threats was achieved. The response time to the external events was almost instantaneous.

Requirement 1.2.2.
Graceful suspension and resumption of non-threat rules was not pursued due to time constraints and focus upon the real-time responsiveness of the system. Initial design was completed however, and this requirement should be met in the next version of APS.

Requirement 1.2.3.
Most dangerous threat attended first was achieved. APS successfully handled the simultaneous occurrence of multiple threats—interrupting the current threat rule after notification by RTM of a higher-priority event, executing the new rule, and invalidating consequential rules after a hardware failure would have prevented its successful completion.

Requirement 1.2.4.
Execution of existing rulebases was achieved. Test rulebases included tic-tac-toe, an automotive diagnostics system and the rulebase found in Appendix A.

Requirement 1.2.5.
Portable, platform independent code was partially achieved. Syntactic implementation of shared memory was specific to the OS-9 operating system and is not part of the ANSI-standard C definition. These system-dependent modules were isolated to a separate object file.

4. Autonomous robot simulation system

Testing the APS on the Autonomous Mobile Robot in a real-world situation may endanger the equipment and the people involved. In order to

avoid this problem it is necessary to use simulation techniques to characterize the problem and find the best model which can be used for testing the robotic system. For this purpose a front-end simulation software package called the Autonomous Robot Simulation System (ARSS) was developed. ARSS is layered on top of the generic simulation product IGRIP.

The ARSS system allows the testing of robotic systems in theoretical situations and environments. Further, this system allows us to test the actual real-time data in a simulated environment. An elegant feature of the ARSS is that it allows the user to click a mouse on menu buttons such as zoom, translate and rotate while the simulation is in progress to obtain varying visual perspectives.

4.1. APS interface to IGRIP using command sets

Two sets of commands were designed and implemented into ARSS to support the HERMIES robot commands given by the APS. SET A commands require movement of the robot or one of its peripherals, whereas SET B commands require no movement. These sets are:

SET A

FMOVE(x,i)	Move forward x feet and i inches
BMOVE(x,i)	Move backward x feet and i inches
FRMOVE(x,i)	Move forward x feet and i inches, fire the forward looking sonar
RTURN(i)	Turn right i degrees
LTURN(i)	Turn left i degrees
MJOINTx(i)	Move joint x of the arm by i degrees
LCAMx(i)	Turn pan table to the left by i degrees
RCAMx(i)	Turn pan table to the right by i degrees
UTILTx(i)	Turn tilt table x up by i degrees
DTILTx(i)	Turn tilt table x down by i degrees
ZTILTx	Return tilt table x to 0 degrees
LHEAD(i)	Turn the LRF pan table to the left i degrees
RHEAD(i)	Turn the LRF pan table to the right i degrees
ZHEAD	Return the LRF pan table to 0 degrees
MATCH	Rotates the match graph-ic by ten degrees
SELECT	Rotates the select graph-ic by ten degrees

SET B

GET_JOINT(a)	Get joint degrees and store in array a
SNAPx_TO_CUBE(a)	Acquire CCD camera x data and store in array
SNAPL_TO_CUBE(b)	Acquire LRF data and store in array b
SONARx(y)	Fire single sonar x and store range in y
WIDE_SCAN(b)	Fire all sonars and store the data in array b

A parser program was constructed to translate HERMIES commands into these ARSS command sets.

4.2. Features of IGRIP applied to the simulation

4.2.1. Parts

In IGRIP each piece of equipment having a degree of freedom must be created and stored separately as parts, which are composed of objects such as blocks, polygons and cylinders. For instance, the robotic arm has seven degrees of freedom and so it has seven different parts with each part made up of any combination of blocks, cylinders, polygon etc. Similarly the laser range finder, the cameras etc, are created and stored separately.

4.2.2. Devices

A Device is constructed by attaching a series of parts to each other and defining their relationship to each other. The HERMIES III used in simulation is a single device consisting of the seven parts of the robot arm, cameras, tilt tables, laser range finder, wheels and the body of the robot. Each part is attached to its position and later its attributes such as the link type (either

rotational or translational) and link speed etc are defined.

4.2.3. Work-cells

Work-cells comprise of devices, positioned in arbitrary locations. The work-cell created consists of three devices, the HERMIES II, match graphic, and execute graphic. HERMIES II moves across a grid that has lines spaced seven feet apart when it receives commands from the APS. On the other hand, the match and select graphics rotate by ten degrees in order to show the amount of work being performed by each processor. This is achieved a Graphics Simulation Language Program which reacts to the incoming commands by joining the links of the appropriate parts of a particular device.

4.2.4. GSL—A simulation language

IGRIP is a powerful tool for graphic simulation. It supports an efficient simulation language called GSL. IGRIP and GSL provide a convenient framework for building the various components of the simulation and offers an effective scheme for defining the relationship between them.

GSL is a procedural language used to construct programs for individual devices in a simulation system to govern their actions. GSL syntax is similar to that of PASCAL with specific enhancements for device motion, display control, viewpoint choreography, etc. Apart from processing the input, GSL is used in the ARSS for opening windows to graphically display the simulation and its status messages and for spawning the communications program and command parser.

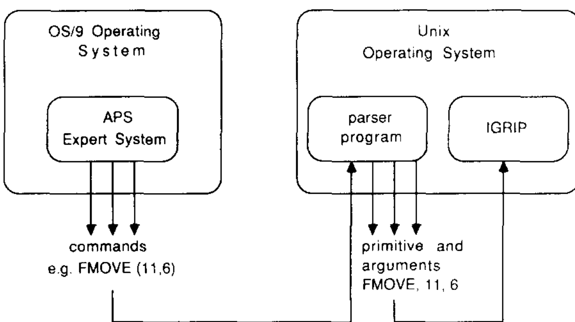


Fig. 9. Integration of graphics software to APS.

4.2.5. Communication between APS and ARSS

In order to simulate the robotic action during the execution of the selected rule APS must communicate with ARSS. Both Vislab and the Iris workstation were connected to ethernet. The design team built up communications primitives on using the socket facilities found on both systems. Figure 9 shows the integration of the ARSS/IGRIP graphics software with the APS expert system.

5. Conclusion

We have discussed the need for using a real-time expert system with a concurrent control architecture for the control of an autonomous mobile robot operating in a complex hostile environment. We have described the various factors contributing to the concurrency of control of Asynchronous Production System—a powerful expert system base built with real-time capabilities. We have also observed how concurrency of control enhances the real-time response of the mobile robots. We have described in detail how the concept of concurrency of control can be effectively implemented in the real-world situation for controlling an Autonomous Mobile Robot, particularly for controlling the HERMIES III robot which is used for experiments in a hostile domain.

The following key ideas were established during the discussion:

1. The real-time response by an Autonomous Mobile Robot to environmental threats depends on the parallelization of the control system used on the robot.
2. The expert system used for time-critical, intelligent control of an Autonomous Mobile Robot must possess an inference engine with a concurrent control architecture in order to provide real-time response to environmental threats.
3. APS—a real-time expert system—possesses a concurrent control architecture and is therefore ideally suited for the control of Autonomous Mobile Robots.
4. The concurrency of control of APS is due to the following factors:
   (a) partitioning of the control task into four independent processes, with one process

for the control and coordination between different processes;

(b) partitioning of the database into four distinct global datasets which are uniquely associated with the various processes of the control mechanism. Inclusion of a special data structure for external events;

(c) using a multi-processor architecture which ensures the concurrent execution of different processes and the continuous monitoring of the real-time events in the robot environment.

5. The implementation of the concept of APS on HERMIES III and the thorough testing of the overall robot system calls for effective modelling of real-world problems such as domain modelling, path planning, etc. This requires the development of a graphic simulation software package.
6. The Autonomous Robot Simulation System provides a convenient and portable platform for characterizing the problem and for exhaustive testing of the entire robot system under theoretical and real-world conditions, using both theoretical and actual data.

Acknowledgments

This research was sponsored by the Office of Basic Energy Sciences, US Department of Energy, under Contract Number DE-AC05-87OR21400 with Martin Marietta Energy Systems Inc. through the Oak Ridge National Laboratory.

References

[1] A. Sabharwal, S.S. Iyengar, F.G. Pin and C.R. Weisbin, "Asynchronous production systems", *J. Knowledge Based Systems 2* (2) (June 1989).

[2] A. Sabharwal, S.S. Iyengar, F.G. Pin and C.R. Weisbin, "Asynchronous production system for real time expert systems", *Proc. AVIGNON,88: 8th Int. Workshop on Expert Systems and their Applications*, Avignon, France, 1988.

[3] A. Sabharwal, A. Amawy, S.S. Iyengar, G. de Saussure, F.G. Pin and C.R. Weisbin, "Asynchronous production system: implementation issues", *Proc. Communication and Control Conf.*, October 21–23, 1988.

[4] T.J. Laffey, P.A. Cox, J.L. Schmidt, S.M. Kao and J.Y. Read, "Real-time knowledge-based systems", *The AI Magazine* (Spring 1988) 27–36.

[5] R.A. Brooks, "A robust layered control system for a mobile robot", *IEEE J. Robotics and Automation Ra-2* (1) (March 1986).

[6] R.A. Brooks, "A hardware retargetable distributed layered architecture for mobile robot control", *IEEE Trans. Systems, Man and Cybernetics* (March 1987) 106–110.

[7] Y. Kanayama, "Concurrent programming of intelligent robots", *Proc. IJCAI* (1983) 834–838.

[8] A. Elfes and S.N. Talukdar, "A distributed control system for the CMU rover", *Proc. IJCAI* (1983) 830–833.

[9] C.R. Weisbin et al., "Hermies III: a step toward autonomous mobility, manipulation and perception" *Robotica, 8* (1990) 7–12.

[10] Richard Fairley, *Software Engineering Concepts* (McGraw-Hill, New York, 1985) pp. 52–53.

[11] J. Peterson and A. Silberschatz, *Operating System Concepts* (Addison-Wesley, Reading, MA, 1985) pp. 334–344.

Appendix A. Asynchronous Production System rulebase

```
(defrule file-based "the terrain reader"
=>
 (fprintout t crlf "*** ASYNCHRONOUS PRODUCTION SYSTEM DEMO          ***"crlf)
 (read_map)
 (fprintout t crlf "the robot starts at 0.0 facing North", crlf)
 (bind ?xs 0)
 (bind ?ys 0)
 (bind ?dir 1)
 (fprintout t "what are the robot", goal co-ords?" crlf)
 (fprintout t "x = ")
 (bind ?xd (read))
 (fprintout t "y = ")
 (bind ?yd (read))
 (assert (robpos ?xs ?ys))
 (assert (robdir ?dir))
 (assert (goal ?xd ?yd))
 (assert (dest ?xd ?yd)


;;; flags for the following facts are ordered "power vision effector"
;;; value TRUE=1, value FALSE=0
;;; for the initial task of navigating to goal, only power is required
 (assert (task-requires 1 0 0)
 (assert (task-available 1 1 1)


 (assert (rtm-location-fire 2 0))
 (assert (rtm-location-security 4 2))
 (assert (rtm-location-power 2 4))
 (assert (rtm-location-radiation 0 2))


 (fprintout t crlf "*** hermies begins navigation to goal          ***" crlf)
 (plan ?xs ?ys ?xd ?yd ?dir)
 (assert (navigate))
 (assert (plan-exec 0))
)


;;;***************************************************
;;;; navigation using the current plan                    *
;;;***************************************************


(defrule navigate "using the plan"
 (declare (salience 10000))
 ?nav <- (navigate)
 ?plan <- (plan-exec ?num)
 (total-plan ?total&: (<= ?num ?total))
 ?currplan <- (plan ?num ?prim ?para)
=>
 (retract ?plan)
 (bind ?var 1)
 (assert (plan-exec =(+ ?num ?var)))
 (retract ?nav)
 (retract ?currplan)
 (assert (nav ?prim ?para)
)
```

```
;;;*************************************************
;;; robot primatives on hermies                                          *
;;;*************************************************

(defrule formove "move forward"
 ?nav <- (nav 0 ?para)
 ?pos <- (robpos ?x ?y)
 (robdir ?dir)e (dest ?xd ?yd)
=>
 (fprintout t "fmove " ?para " 0", crlf)
 (ahparam ?x ?y ?xd ?yd ?dir)
 (fmove ?para 0)
 (retract ?pos)
 (if (= ?dir 0) then (assert (robpos (+ ?x (/ ?para 4)) ?y)))
 (if (= ?dir 1) then (assert (robpos ?x =(+ ?y (/ ?para 4)))))
 (if (= ?dir 2) then (assert (robpos (- ?x (/ ?para 4)) ?y)))
 (if (= ?dir 3) then (assert (robpos ?x =(- ?y (/ ?para 4)))))
 (retract ?nav)
 (assert (navigate))
)

(defrule rightturn "turn right"
 ?nav <- (nav 2 ?para)
 ?rob <- (robdir ?dir)
=>
 (fprintout t "rturn " ?para crlf)
 (rturn ?para)
 (retract ?rob)
 (bind ?diff (- ?dir (/ ?para 90)))
 (if (< ?diff 0) then (bind ?diff (+ 4 ?diff)))
 (assert (robdir ?diff))
 (retract ?nav)
 (assert (navigate))
)

(defrule leftturn "turn left"
 ?nav <- (nav 1 ?para)
 ?rob <- (robdir ?dir)
)

 (fprintout t "lturn " ?para crlf)
 (lturn ?para)
 (retract ?rob)
 (bind ?diff (4 ?dir (/ ?para 90)))
 (if (> ?diff 3) than (bind g?diff (- 4 ?diff)))
 (assert (robdir ?diff))
 (retract ?nav)
 (assert (navigate))
)

;;;*************************************************
;;; high-level rules for alarms                                          *
;;;*************************************************

(defrule alarm-fire " "
 (declare (salience 100))  ?rtm_rule <- (rtm_alarm_fire)
 task-available ?ap ?av ?ae)
 ?taskr <- (task-requires ? ? ?)
=>
 (if (= ?ap ?ay ?ae 1) then
  (retract ?taskr)
```

```
  (assert (clean work-mem))
  (assert (task-requires 1 1 1))
  (assert (process fire)
 else
  (fprintout t crlf "ignoring fire... hardware insufficient", crlf))
 (retract ?rtm_rule)
)

(defrule alarm-security " "
 (declare (salience 50))
 ?rtm_rule <- (rtm_alarm_security)
 (task-available ?ap ?av ?ae)
 ?taskr <- (task-requires ? ? ?)
=>
 (if (= ?ap ?ay 1) then
  (retract ?taskr)
  (assert (clean work-mem))
  (assert (task-requires 1 1 0))
  (assert (process security))
 else
  (fprintout t crlf "ignoring security... hardware insufficient" crlf))
 (retract ?rtm_rule)
)

(defrule alarm-power " "
 (declare (salience 10000))
 ?rtm_rule <- (rtm_alarm_power)
 ?taska <- (task-available ?ap ?ay ?ae)
=>
 (retract ?rtm_rule ?taska)
 (assert (clean work-mem))
 (assert (task-available 0 7av ?ae)
 (assert (process power))
)

(defrule alarm-radiation " "
 (declare (salience 150))
 ?rtm_rule <- (rtm_alarm_radiation)
 (task-available ?ap ?ay ?ae)
 ?taskr <- (task-requires ? ? ?)
=>
 (if (= ?ap ?ay ?ae 1) then
  (retract ?taskr)
  (assert (clean work-mem))
  (assert (process radiation))
 else
  (fprintout t crlf "ignoring radiation... hardware insufficient", crlf))
 (retract ?rtm_rule)

;;;*************************************************
;;; high-level rules for handling hardware failures         *
;;;*************************************************

(defrule failure-vision "handle failure of sonar end ccd camera"
 (declare (salience 1000))
 ?rtm rule <- (rtm_failure_vision)
 ?taska <- (task-available*ap 1 ?ae)
 (task-requires ? ?trv ?)
=>
 (if (= ?trv 1) then
  (fprintout t crlf "vision failed and was required" crlf)
```

```
 else
  (fprintout t crlf "vision failed... continuing" crlf))

 (retract ?rtm_rule ?taska)
 (assert (task-available ?ap 0 ?ae))
)

(defrule failure-effector "handle failure of end-effector"  (declare (salience 1000))
 ?rtm_rule <- (rtm_failure_effector)
 ?taska <- (task-available ?ap ?ay 1)
 (task-requires ? ? ?tre)
=>
 (if (= ?tre 1) then
  (fprintout t crlf "effector failed and was required" crlf)
 else
  (fprintout t crlf "effector failed... continuing" crlf))

 (retract ?rtm_rule ?taska)
 (assert (task-available ?ap ?ay 0)
)

;;;**************************************************
;;; mid-level rules for processing the interrupts                   *
;;;**************************************************

(defrule replan-fire "move to location of fire"
 (declare (salience 100))
 ?clean <- (clean-over)
 (robpos ?xs ?ys)
 (rtm-location-fire ?xfire ?yfire)
 (robdir ?dir)
 ?rtm-rule <- (process fire)
 ?dest <- (dest ? ?)
=>
 (plan ?xs ?ys ?xfire ?yfire ?dir)
 (assert (navigate))
 (assert (plan-exec 0))
 (retract 7clean ?rtm-rule ?dest)
 (assert (dest ?xfire ?yfire))
 (assert (rtm-event-occurred))
 (fprintout t crlf "*** hermies extinguished the fire         ***" crlf)
)

(defrule replan-security "move to location of security intrusion"
 (declare (salience 50))
 ?clean <- (clean-over)
 (robpos ?xs ?ys)
 (rtm-location-security ?xsecurity ?ysecurity)
 (robdir ?dir)
 ? rtm-rule <- (process security)
 ?dest <- (dest ? ?)
=>
 (plan ?xs ?ys ?xsecurity ?ysecurity ?dir)
 (assert (navigate))
 (assert (plan-exec 0))
 (retract ?clean ?rtm-rule ?dest)
 (assert (dest ?xsecurity ?ysecurity)
 (assert (rtm-event-occurred))
 (fprintout t crlf "*** hermies shot the intruder in the foot          ***" crlf)
)
```

```
(defrule replan-power "move to location of battery charger"
 (declare (salience 10000))
 ?clean <- (clean-over)
 (robpos ?xs ?ys)
 (rtm-location-power ?xpower ?ypower)
 (robdir ?dir)
 7 rtm-rule <- (process power)
 ?dest <- (dest ? ?)
 ?taska <- (task-available ?ap ?av ?ae)
=>
 (plan ?xs ?ys ?xpower ?ypower ?dir)
 (assert (navigate)
 (assert (plan-exec 0))
 (retract ?clean ?rtm=rule ?dest ?taska)
 (assert (dest ?xpower ?ypower))
 (assert (rtm-event-occurred))
 (assert (task-available 1 ?av ?ae))
 (fprintout t crlf "*** hermies recharged his batteries          ***" crlf)
)

(defrule replan-radiation "move to location of radiation leak"
 (declare (salience 150))
 ?clean <- (clean-over)
 (robpos ?xs ?ys)
 (rtm-location-radiation ?xradiation ?yradiation)
 (robdir ?dir)
 ?rtm-rule <- (process radiation)
 ?dest <- (dest ? ?)
=>
 (plan ?xs ?ys ?xradiation ?yradiation ?dir)
 (assert (navigate))
 (assert (plan-exec 0))
 (retract ?clean ?rtm-rule ?dest)
 (assert (dest ?xradiation ?yradiation)
 (assert (rtm-event-occurred))
 (fprintout t crlf "*** hermies sealed the radiation leak          ***" crlf)
)

;;;*************************************************
 ;;; low level rules for real time processing          *
 ;;;*************************************************

(defrule clean "the working memory"
 (declare (salience 10000))
 (clean work-mem)
 ?plan <- (plan-exec ?num)
 (total-plan ?total&: (<= ?num ?total))
 ?currplan <- (plan ?num ?prim ?para)
=>
 (retract ?currplan)
 (retract ?plan)
 (assert (plan-exec =(+ ?num ?)))
)

(defrule finish-clean "no plans left"
 (declare (salience 10000))
 ?clean <- (clean work-mem)
 ?plan <- (plan-exec ?num)
 ?atotal <- (total-plan ?total&: (> ?num ?total))
=>
 (retract ?clean ?plan ?atotal)
```

```
 (fprintout t "took out "all the plans" crlf)
 (assert (nav-clean))
)

(defrule no-plans-exist "no plans existed"
 (declare (salience 10000))
 ?clean <- (clean work-mem)
 (not (plan-exec ?))
 (not (total-plan ?))
=>
 (retract ?clean)
 (fprintout t "took out all the plans" crlf)
 (assert (nav-clean))
)

(defrule nav-clean "if navs were generated"
 (declare (salience 10000))
 ?clean <- (nav-clean)  7nav <- (nav ?prim ?para)
=>
 (retract ?clean ?nav)
 (assert (navigate-clean))
)

(defrule no-navs "no navs were found"
 (declare (salience 10000))
 ?clean <- (nav-clean)
 (not (nav ?prim ?para))
=>
 (retract ?clean)
 (assert (navigate-clean))
)
(defrule navigate-clean "if navigate is present"
 (declare (salience 10000))
 ?clean <- (navigate-clean)
 ?nav <- (navigate)
=>
 (retract ?clean ?nav)
 (assert (clean-over)
)

(defrule no-navigate "no navigate present"
 (declare (salience 10000))
 ?clean <- (navigate-clean)
 (not (navigate))
=>
 (retract ?clean)
 (assert (clean-over))
)

(defrule nav-finish "no plans left"
 ?nav <- (navigate)
 ?plan <- (plan-exec ?num)
 ?atotal <- (total-plan ?total&: (> ?num ?total)
=>
 (retract ?nav ?plan ?atotal)
 (fprintout t "navigation completed" crlf)
 (assert (recover))
)

(defrule success "robot at destination"
 ? rec <- (recover)
```

```
 (not (rtm-event-occurred))
 (robpos ?xs ?ys)
 (goal ?xs ?ys)
=>
 (retract ?rec)
 (fprintout t crlf "*** success !!! hermies arrived at goal          ***" crlf)
)

(defrule rtm=event-recover "recovery"
 ? rec <- (recover)
 ? rtm-rule <- (rtm-event-occurred)
 (robpos ?xs ?ys)
 (goal ?xd ?yd&: (|| (!= ?xs ?xd) (!= ?ys ?yd)))
 (robdir ?dir)
 ?dest <- (dest ? ?)
=>
 (retract ?rec ?rtm-rule ?dest)
 (fprintout t crlf "*** recovering from real-time interrupt          ***" crlf)
 (plan ?xs ?ys ?xd ?yd ?dir)
 (assert (navigate))
 (assert (dest ?xd ?yd))
 (assert (plan-exec 0))
```

Appendix B. Asynchronous Production System external functions

```
#include <stdio.h>
#include "clips.h"

#define HUGE 10000.0

int terrain[5][5];

struct unit {
 int x;
 int y;
 int dir;
 float g;
 int h;
 float f;
 int parx;
 int pary;
 struct unit *parent;
};

int xstart, ystart, xdest, ydest, robdir, robdist;
int solution [25][3], lastsol;
int primsol[2][2], index;

extern int     s_socket;
char           buf[48];
int                   rc, buf_len;

/*************************************************/
/* low level robot routines                      */
/*************************************************/

ahparam()
 xstart = rfloat(1);
 ystart = rfloat(2);
```

```
 xdest  = rfloat(3);
 ydest  = rfloat(4);
 robdir = rfloat(5);
 return (0);
 }

bmove()
 {
 float x,y;

 x = rfloat (1);
 y = rfloat(2);

 sprintf(buf,"BMOVE (%d,%d)\0", (int) x, (int) y);
 buf_len = strlen(buf)+1;
 rc = send(s_socket, &buf_len, sizeof(buf_len), 0);
 rc = send(s_socket, buf, buf_len, 0);

 sleep(5);
 return(0);
 }
la20fmove()
 {
 float x,y;

 x = rfloat(1);
 y = rfloat(2);

 sprintf(buf,"FMOVE (%d,%d)\0", (int) x, (int) y);
 buf_len = strlen(buf)+1;
 rc = send(s_socket, &buf_len, sizeof (buf_len), 0);
 rc = send(s_socket, buf, buf_len, 0);

 sleep(5);
 return (0);
 }

lturn()
 {
 float x;

 x = rfloat(1);

 sprintf(buf,"LTURN (%d)\0", (int) x);
 buf_len = strlen(buf)+1;
 rc = send(s_socket, &buf_len, sizeof(buf_len), 0);
 rc = send(s_socket, but, but_len, 0);

 sleep(5);
 return(0);
 }

rturn()
 {
 float x;

 x = rfloat(1);

 sprintf(buf,"RTURN (%d)\0", (int) x);
 buf_len strlen(buf)+1;
 rc = send(s_socket, &buf_len, sizeof(but_len), 0);
 rc = send(s_ socket, buf, buf_Ten, 0);
```

```
 sleep(5);
 return(0);
 }

aplan()
 float a, b, c, d, e;
 a = rfloat(1);
 b = rfloat(2);
 c = rfloat(3);
 d = rfloat(4);
 e = rfloat(5);
 plan (int) a, (int) b, (int) c, (int) d, (int) e);
 return(0);
 }

/***********************************************/
/* high level robot routines        */
/***********************************************/
read_map()
 {
 FILE *terdat, *fopen();
 int i, j;

 if ((terdat = fopen("terrain.dat", "r")) == NULL)
  {
  printf("can't open terrain.dat\n");
  return(0);
  }
 else
  {
  printf("\n\n\n");
  printf("The Terrain           N  \n");
  printf("                    W + E\n");
  printf("    X                 S  \n");
  printf("0 1 2 3 4               \n\n");

  for (j = 4; j>-1; j--)
   {
   for (i = 0; i<5; i++)
    {
    fscanf(terdat, "%d", &terrain[i][j]);
    printf("%d ", terrain[i][j]);
    }
   if (j == 2)
    printf(" %d Y\n",j);
   else
    printf(" %d\n",j);
   }
  }
 fclose (terdat);
 printf ("\n");
 return 1);
 }

plant xs, ys, xd, yd, dir)
int xs, ys, xd, yd, dir;
{
 int sol;

 if (terrain[xs][ys] == 1)
  {
  printf("bad start position\n"];
```

```
   return(0);
   }

  if (terrain[xd][yd] == 1)
   {
   printf("bad destination position\n");
   return(0);
   }

  xstart  = xs;
  ystart = ys;
  xdest = xd;
  ydest = yd;
  robdir = dir;
  printf("goal = %d, %d\n", xdest, ydest);
  printf("start = %d, %d, %d\n", xstart, ystart, robdir);
  sol = astar();
  if (sol == 0)
   printf("no solution av&ilable\n");
  else if (sol == 1)
   {
   modsol();
   assertsol();
   }
  return(1);
}

astar()
{
 struct unit *open[25];
 struct unit *close [25];
 struct unit *successor;
 int i = 0, j = 0, k, l, m, openi, found;
 struct unit *init, *bestnode, *succeed(), *try;

 init = (struct unit *) malloc (sizeof (*init));
 init->x = xstart;
 init->y = ystart;
 init->dir = robdir;
 init->g = 0.0;
 init->h = estimate(init);
 init->f = init->g + (float) init->h;
 init->parx = -1;
 init->pary = -1;
 init->parent = NULL;
 open[i] = (struct unit *) malloc (sizeof (*try));
 equalnode(open[i], init);
 i++;
 bestnode = (struct unit *) malloc (sizeof (*bestnode));
 successor = (struct unit *) malloc (sizeof (*successor));

 while (1) {
 equalnode (bestnode, open[0]);
 openi = 0;
 for (k = 1; k < i; k++) {
  if (open[k]->f <= bestnode->f) {
   equalnode (bestnode, open[k]);
   openi = k;
  }
 }
 if (bestnode->f == HUGE) return (0);
```

```
 open[openi]->f = HUGE;
 open[openi]->x = 5;
 open[openi]->y = 5;
 close[j] = (struct unit *) malloc (sizeof (*try));
 equalnode (close[j], bestnode);
 j++;
 if (bestnode->x = xdest && bestnode->y == ydest) {
  try = (struct unit *) malloc (sizeof (*try));
  equalnode (try, bestnode);
  lastsol = -1;
  while (!((try->x = - xstart) && (try->y == ystart))) {
   lastsol++;
   solution[lastsol][0] = try->x;
   solution[lastsol][1] = try->y;
   solution[lastsol][2] = try->dir;
   for (k = 0; k <j; k++) {
    if ((close[k]->x == try->parx) &&
     close[k]->y == try->pary))
     equalnode (try, close[k]);
   }
  }
  return (1);
 }
 else {
 for (k=0; k <4; k++) {
  if (succeed(bestnode, k) != NULL) {
   equalnode (successor, succeed(bestnode, k));
   found = 0;
   for (1=0; 1 <i: 1++) {
    if ((open[1]->x == successor->x) &&
     (open[1]->y successor->y)) {
     found = 1;
     if (successor->f < open[1]->f)
      equalnode(open[1], successor);
    }
   }
   if (found == 0) {
    for (1=0; 1<j; 1++) {
     if ((close[1]-> == successor->x) &&
     (close[1]>y == successor->y)) {
     found = 2;
     if (successor->f < close[1]->f)
      equalnode (close[1], successor);
    }
    }
   }
   if (found == 0) {
    open[i] = (struct unit *) malloc (sizeof (*try));
    equalnode(open[i], successor);
    i++;
   }
  }
 }
 }
 }
}

int estimate(point)
struct unit *point;
{
 int x, y;
```

```
 if ((x = xdest = point->x) <0)
  x = -x;
 if ((y = ydest = point->y) <0)
  y = -y;
 return (x + y);
}

struct unit *succeed(point, dir)
struct unit ⩾int;
int dir;
{
 struct unit *temp;
 int tempx, tempy;
 float rot;

 tempx = point->x;
 tempy = point->y;
 if ((dir == 0) &&
  ((tempx+1 > 4)) | | (terrain[tempx +1][tempy] == 1)))
  return(NULL);
 if ((dir == 1) &&
  (tempy+1 > 4) | | (terrain[tempx])tempy +1] == 1)))
  return(NULL);
 if ((dir == 2) &&
  ((tempx - 1 <0) | | (terrain[tempx -1][tempy] == 1)))
  return(NULL);
 if ((dir == 3) &&
  (tempy - 1 <0) | | (terrain[tempx][tempy -1] == 1)))
  return(NULL);
 temp = (struct unit *) malloc (sizeof (*temp));
 switch (dir) {
  case 0: temp->x = tempx +1;
          temp->y = tempy;
          break;
  case 1: temp->x = tempx;
          temp->y = tempy +1;
          break;
  case 2: temp->x = tempx -1;
          temp->y = tempy;
          break;
  case 3: temp->x = tempx;
          temp->y = tempy -1;
          break;
 }
 temp->dir = dir;
 if (point->dir - dir == 0) rot = 0.0;
 if ((point->dir - dir == 1) | | (point->dir - dir == -1)
  ⊇ ⊇ (point->dir - dir == 3) | | (point->dir - dir == -3))
  rot = 0.5;
 if ((point->dir - dir == 2) | | (point>dir - dir == -2))
  rot = 1.0;
 temp->g = point->g + 1.0 + rot;
 temp->h = estimate(temp);
 temp->f = temp->g + (float) temp->h;
 temp->parent = point;
 temp->parx = point->x;
 temp->pary = point->y;
 return (temp);
}

equalnode(blank, full)
struct unit *blank, *full;
{
```

```
 blank->x = full->x;
 blank->y = full->y;
 blank-> dir = full->dir;
 blank->g = full->g;
 blank->h = full->h;
 blank->f = full->f;
 blank->parent = full->parent;
 blank->parx = full->parx;
 blank->pary = full->pary;
}

printsol()
{
 int i;

 printf("the solution\n");
 for (i = lastsol; i >= 0; i--) {
  printf("%d, %d, %d\n", solution[i][0], solution[i][1],
   solution[i][2]);
 }
}

modsol()
{
 int curr, next, dist = 0, i, diff;

 curr = robdir;
 index = -1;
 for (i = lastsol; i >= 0; i--) {
  next = solution[i][2];
  diff = next - curr;
  curr = next;
 switch (diff) {
   case 0: {
           dist++;
           break;
           }
   case 1:
   case -3: {
            if (dist > 0) {
             index++;
             primsol[index][0] = 0;
             primsol[index][1] = dist;
             }
            index++;
            primsol[index][0] = 1;
            primsol[index][1] = 90;
            dist = 1;
            break;
            }
   case -1:
   case 3: {
           if (dist > 0) {
            index++;
            primsol[index][0] = 0;
            primsol[index][1] = dist;
            }
           index++;
           primsol[index][0] = 2;
           primsol[index][1] = 90;
           dist = 1;
           break;
           }
```

```
   case 2:
   case -2: {
            if (dist > 0) {
             index++;
             primsol[index][0] = 0;
             primsol[index][11 = dist;
             }
            index++;
            primsol[index][0] = 2;
            primsol[index][1] = 180;
            dist = 1;
            break;
            }
  }
 }
 it (dist > 0) {
  index++;
  primsol[index][0] = 0;
  primsol[index][1] = dist;
 }
 robdir = curr;
}

printprim()
{
 int i, prim, para;

 printf("the solution in terms of primitives\n");
 for (i = 0; i <= index; i++) {
  prim primsol[i][0];
  para = primsol[i][1];
  if (prim == 0) printf("fmoves %d 0\n", para*4);
  else if (prim == 1) printf("lturn%d\n", para);
  else if (prim == 2) printf("rturn%d\n", para);
 }
}

assertsol ()
 {
 int i, prim,.para;
 char fact[50];

 for (i = 0; i <= index; i++)
  {
  prim = primsol[i][0];
  para = primsol[i][1];
  if (prim == 0)
   sprintf(fact, "plan %d 0 %d", i, para*4);
  else if (prim == 1)
   sprintf(fact, "plan %d 1 %d", i, para);
  else if (prim == 2)
   sprintf(fact, "plan %d 2 %d", i, para);
  assert fact);
  }
 sprintf(fact, "total-plan %d", index);
 assert (fact);
 }
```