AIAA-94-1242-CP

INTEGRATION FOR NAVIGATION ON THE UMASS MOBILE PERCEPTION LAB

Bruce Draper**, Claude Fennema†, Benny Rochwerger+
Edward Riseman*, Allen Hanson*

Abstract

The Mobile Perception Laboratory (MPL) is an autonomous vehicle designed for testing visually guided behaviors, such as road following, passive obstacle detection, landmark-based navigation and model acquisition [Hanson, Riseman, & Weems 93]. The research is being conducted under the sponsorship of the ARPA Unmanned Ground Vehicle (UGV) Program in the Computer Vision Laboratory at the University of Massachusetts.

The focus of this paper is on integrating multiple behaviors into a single, coherent system. It presents the ISR3, a new tool for interprocess communication, the storage and retrieval of transient, image-based data, and the long-term management of 3D data. It also presents the script monitor, a process control mechanism for invoking, monitoring and destroying concurrent sets of visual behaviors in response to dynamic events and the systems stated goals.

1. The Mobile Perception Lab (MPL)

The experimental laboratory vehicle for this effort is the UMass Mobile Perception Lab (MPL), a heavily modified Army HMMWV ambulance (Figure 1) that is equipped with actuators and encoders for the throttle, steering and brake. The interface to the on-board computer system is through a 68030-based controller board. Electrical power is provided by an on-board 10kW diesel generator feeding uninteruptable power supplies and conditioners. The MPL closely matches CMU's NavLab II, with modifications and component installation performed by RedZone, Inc., a Pittsburgh-based firm specializing in custom robotics.

Figure 1. The Mobile Perception Laboratory

The vehicle's sensor package includes a Staget, which is a stabilized platform capable of rotating a full 360^{o}. The Staget is mounted at the center of the cab roof and contains a CCD color camera and a FLIR sensor in a weatherproof enclosure. Two forward-looking stereo cameras and a forward looking color CCD camera are mounted in a rectangular enclosure at the front edge of the cab's roof. The primary computing engine for vision processing, goal-oriented reasoning and path planning is a Silicon Graphics 340GX four-node multiprocessor. The multiprocessor is interfaced to the sensor suite through a Datacube MaxVideo20 processor, which provides frame rate image processing for certain types of operators. Space and power has been provided for the possible addition in the future of a 16K Image Understanding Architecture (IUA) [Weems 1993], a massively parallel heterogeneous processor.

*Prof., Computer Science, Univ. of Massachusetts
**Sr. Postdoctoral Researcher, Computer Science, Univ. of Massachusetts
†Asst. Prof., Computer Science, Mt. Holyoke College, So. Hadley, MA
+Graduate student, Computer Science, Univ. of Massachusetts

The physical lay-out of equipment on the vehicle is depicted in Figure 2. The first programmer station is located in the HMMWV's passenger seat, with a 17" color X-terminal fixed to the metal platform between the passenger's and driver's seats. The second programmer station is located behind and slightly above the driver, and includes a car seat, mounting brackets for both an SGI color terminal and a small SONY monitor for viewing

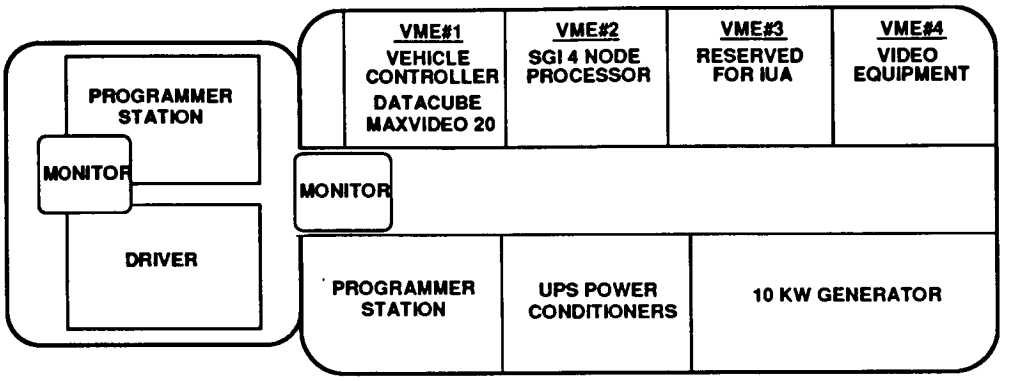


Figure 2. Interior layout of the MPL.

raw TV signals. Behind this programmer station is the diesel generator and power conditioning systems. The passenger's side rear hold four enclosed, air conditioned 19" computer frames for the on-board computer systems. The first frame holds the vehicle controller, image digitizers and frame stores, the MaxVideo20, and the Staget controller and interface. The second computing frame contains the Silicon Graphics multiprocessor, disk drives, power supply and (removable) tape drive. The third frame is reserved for the Image Understanding Architecture (IUA). The fourth frame contains video recorders for collecting experimental data.

2. Integration of Vision Modules for Real-Time Control: An Overview

MPL is an experimental laboratory for testing and integrating different approaches to problems in autonomous navigation, including, but not limited to, landmark-based navigation, obstacle detection and avoidance, model acquisition and extension , road following, and path planning. It is therefore important that MPL have a software environment where multiple visual modules, addressing different subtasks, can be easily integrated, and where researchers can quickly experiment with different combinations and parameterizations of those modules. At the same time, MPL's software environment must be efficient enough to meet the demands of real-time navigation research.

The need to balance between flexibility and efficiency has led us to design a software environment around the ISR3, an in-memory database that allows users to define structures for storing visual data, such as images, lines and surfaces [Draper 93a,b]. ISR3 serves as a process communication interface, so that, for example, lines produced by one module can be used by another, even if the second module is run later or on a different processor than the first. ISR3 also provides modules with efficient spatial access routines for visual data, and protects data from being simultaneously modified by two or more concurrent processes. A graphical programming interface allows programmers to easily sequence modules and modify their parameters.

Our work on planning has concentrated on plan execution rather than plan generation; thus, we assume that plans are developed by the user (by hand) or by an appropriate planning subsystem. With this goal in mind, a control system, called the Script Monitor, has been implemented to support real-time execution of plans [Rochwerger et. al. 94]. The task of autonomous navigation, as with many other complex tasks, can be decomposed into specific subtasks like road following, obstacle avoidance and landmark recognition. In order to achieve a fully autonomous capability, the solutions to all these subproblems must be integrated into a coherent system. This paper focuses on the preliminary work done on the problem of integration for the UMass Mobile Perception Laboratory (MPL). Since each of the

subproblems is still a field of active research in which approaches and solutions may change rapidly over time, the integrated system should be flexible enough to allow testing different subsystems and different control strategies.

To achieve the desired flexibility we have implemented the control system as a "programmable" finite state machine (FSM), in which the states represent the different modes of operation of the system (behaviors) and the state transitions correspond to the system's reactions to events (either external or internal). The composition of the states and the transitions is not fixed, i.e., the FSM can be tailored to the particular task the system is trying to achieve.

3. Integration via the ISR Database.

MPL's database is built around ISR3, an in-memory database for computer vision. Historically, ISR (an acronym for *intermediate symbolic representation*) has been the name of a series of symbolic databases for vision developed at the University of Massachusetts [Brolio et. al. 89]. One version, ISR1.5, is now commercially available as part of KBVision™ [Williams 90], while the most recent version, ISR3, is used on-board the MPL. The ISR databases reflect a belief that computer vision requires more than image-like arrays of numerical data; computer vision depends on symbolic representations of abstract image events such as regions, lines, and surfaces, and on mechanisms for efficiently accessing data objects by a model-based system under various types of constraints (such as spatial proximity). Although each version of ISR has been a refinement of its predecessor, they all assume that visual procedures operate on symbolic records, called tokens, or on groups of tokens, and that visual procedures manipulate tokens both for internal computations and for exchanging data with other procedures.

ISR3 is an in-memory database for C structures. It reads a C header file of structure definitions when it is first initialized, and records the size of the structures (called *tokens*) and the data types and offsets of their fields. ISR3 then establishes a database for these structures, establishing queues for allocating and freeing tokens, semaphores for preventing simultaneous access, and functions for storing and retrieving tokens. Once one process has initialized ISR3, other processes can attach to it (since all tokens are kept in shared memory), making ISR3 a communication mechanism as well as a data store.

3.1 ALVINN

In order to emphasize ISR3's role as an integration mechanism, its communication, synchronization, and data storage capabilities will be described in the context of specific MPL tasks. One of MPL's most basic tasks is to drive down roads, using the ALVINN neural-net road-following system [Pomerleau 90, 92]. Developed at Carnegie-Mellon University, ALVINN is a neural network with a single hidden layer that produces steering vectors from reduced (32x30) and color-compensated intensity images[1]. When run alone it implements a simple road following behavior by grabbing images from a forward-looking camera and sending commands to the (low-level) vehicle controller to correct for drift and turns in the road in order to keep the vehicle on course.

On the MPL ALVINN is almost never run in isolation, however. At the very least it is run with an obstacle detection program to ensure the safety of the vehicle (otherwise ALVINN is more than happy to run into obstacles). ALVINN is integrated with the obstacle detection program by interposing an arbiter between ALVINN and the vehicle controller. The arbiter takes the steering vectors produced by ALVINN and combines them with the results of obstacle detection to make sure the vehicle avoids obstructions.

ALVINN is integrated with the rest of MPL's software by declaring its steering vectors to be ISR3 tokens stored in shared memory. ALVINN can then be tested in isolation from other systems by having the controller read ALVINN's steering tokens directly, and combined with other systems by having the arbiter retrieve ALVINN's tokens and produce its own (modified) steering tokens for the controller. A similar principle can be applied to image acquisition, with a data server producing image tokens and storing them in shared memory for ALVINN (or any other process) to retrieve.

3.2 Landmark Recognition and Positioning

Although ALVINN demonstrates how ISR3 can help integrate software modules, including modules developed at other sites, it does not exercise all of the ISR3 capabilities. MPL's

[1]Properly speaking, an in-memory database is called a data store.

landmark recognition system is a combination of four software modules that together match 3D landmark (or object) models to images and determine the position and orientation of the camera relative to the landmark. The first module uses color and texture information to limit the search for the landmark to a specific region of interest (ROI), the second extracts straight (2D) line segments from the ROI, the third projects the 3D landmark model according to the estimated viewing point and determines which (3D) lines should be visible, and the fourth matches (2D) image lines to visible (3D) model lines and determines the relative position of the vehicle to the landmarks in the world coordinate system.

As with ALVINN and the arbiter, ISR3 is the communication mechanism that allows the four landmark recognition modules to exchange data, this time by declaring ROIs, (2D) image lines, (3D) model lines and (3D) faces to be ISR3 tokens. The landmark recognition modules are more typical of vision applications than ALVINN, however, in that they produce large numbers of tokens which are typically accessed by name, feature value or spatial location. Spatial access is particularly important for the matching modules, which must repeatedly access image lines near the projections of model lines (according to the current pose estimate).

Most landmark recognition tokens, particularly ROIs and image lines, exist for only a short period of time (in this case the time required to process one image) and should then be deallocated. Consequently, file I/O is not optimized; although data is sometimes saved for later analysis in the lab, most data can be kept in memory and then erased without ever being saved to disk (ignoring the effects of paged virtual memories). Token allocation and deallocation, on the other hand, are critical, which is why ISR3 maintains its own token queues.

The landmark recognition processes are also typical of many vision application programs in that they both produce and consume sets of tokens rather than single tokens. The matching module, for example, finds (potentially many-to-many) correspondences between sets of image lines and sets of model lines. Therefore most storage and retrieval commands in ISR3 are in the form of set operations, such as a request to access "all long, straight lines in the upper corner of an image". Synchronization is provided, not at the level of individual tokens, but of sets of tokens, so that processes may iterate over sets of tokens without having to lock and unlock each token individually. Special facilities for optimizing spatial retrieval over individual or arbitrary sets of tokens are also provided, as are macros for iterating over the tokens in a set, and functions for taking the union, intersection and differences of sets[2].

Although most tokens are temporary, ISR3 also provides permanent storage for those few sets of tokens (critical features, updated maps, etc.) that correspond to significant results and should persist over time. Model acquisition processes, such as one described by Sawhney [Sawhney 93] produce 3D models of the type used for object recognition (i.e. 3D points, lines and faces). These models, once learned, should be permanently stored, and the landmark recognition processes should always have access to the most recent version of any model. With regard to these tokens, therefore, ISR3 serves as a permanent data base system that provides storage for, and structured access to, long-term data.

3.3. ISR3 Protection and Memory Management Mechanisms

Although ISR3 acts as a general database system, it has been optimized for real-time vision research by reducing overhead wherever possible while still supporting those functions most often used in computer vision. For example, one task of a database in a multiprocess environment is to stop processes from accidentally overwriting or destroying each other's data. In a typical computer vision application, hundreds or thousands of tokens may be in memory at one time. If multiple processes have uncontrolled access to these tokens and modify them, unpredictable interactions will cause elusive and non-repeatable bugs. On the other hand, it is not uncommon for a process such as the model matcher to access hundreds of tokens at a time. If it had to lock and unlock a token each time it reads a feature value, protection would become unacceptably expensive, especially given the relatively slow speed of semaphores under UNIX.

A compromise used in ISR3, therefore, was to associate semaphores with sets of tokens. When a

[2]The complement of a set is not well defined, since there is an infinite universe of possible tokens.

set of related tokens is created, for example the set of lines extracted from a ROI, a semaphore is allocated to protect those tokens. Any ISR3 function that accesses that set of tokens will first check the semaphore; ISR3 access functions that create subsets of a defined set of tokens assign the same semaphore to the subset that is used for the parent set. If users access tokens surreptitiously through C pointers[3], they are expected to lock the semaphore before accessing the first token and to unlock it after the last token access. As a result, as long as users do not circumvent its safeguards, ISR3 is able to provide process synchronization with very little overhead.

Similarly, ISR3 provides a level of memory management on top of the UNIX operating system. For non-real-time, file-based systems, memory management is not a critical issue; for continuous, real-time systems, however, it is crucial. ISR3 applications operate in real-time loops, allocating new tokens on each iteration. Memory allocation must be rapid, and must avoid fragmenting memory (as repeated calls to *malloc* would). Memory must be recycled, with space allocated to old tokens being reassigned to new ones once the old data is no longer needed. ISR3 satisfies these requirements by providing token buffers (at a level hidden from the user). Calls to create a token actually allocate one from the buffer for that token type, and freed tokens are returned to the appropriate buffer. For users who store tokens in hierarchies, ISR3 also provides functions for tracing through a hierarchy and freeing all the tokens in it, so that, for example, one can free all the memory associated with an image once the image is no longer current.

4. Executing Reactive Behaviours: Scripts and the Script Monitor

The term behavior has generally been used in the literature to describe processes that connect perception to action, i.e., a behavior senses the environment and takes an appropriate action based on what was perceived. A combination of behaviors is also called a behavior; thus, a complex behavior can be achieved by combining simpler behaviors. In Brooks' subsumption architecture [Brooks 86], the task of robot control is decomposed into levels of competence; each level, in combination with lower levels, defines a behavior. In their Distributed Architecture for Mobile Navigation system (DAMN), Payton, Rosenblatt and Keirsey [Payton 86; Payton et. al. 87] refer to behaviors as very low level decision-making processes which are guided by high level plans and combined through arbitration. In their DEDS work, Ramadge and Wonham [Ramadge 89], as well as Rivlin [Rivlin], events are considered the alphabet, Σ, of a formal language. A behavior is then a sequence of events, or a string over Σ^*. Note that in this terminology every prefix of a string is also a behavior, i.e., the sequential combination of behaviors is a behavior.

These definitions, although consistent, can be confusing - the same term is used for individual processes and for the composition of these processes. We have chosen to think of a behavior as a mode of operation [Payton et. al. 90], in which several perception-action processes [Draper 94] are executed concurrently. Each process converts sensory data into some kind of action (either physical or cognitive), and at any time may generate an event - a signal to let the system know that "something" significant has occurred. All inter-process communication is achieved through a global blackboard -a section of shared memory accessible to all processes. The blackboard is built on top of the ISR3 which, as described earlier, provides a very efficient memory management mechanism (on top of UNIX) and a set of primitives necessary for shared memory based communication.

4.1. A Finite State Machine Representation for Behaviors

An autonomous system must react to events by changing its behavior; hence the sequence of behaviors actually executed depends upon the sequence of events. Since the latter is unpredictable, so is the former. To program such a system, one must specify which processes constitute a behavior and for each possible event describe the system's reaction. In order to specify such a system, a finite state machine (FSM) formalism has been chosen, in which the states represent behaviors and the transitions reactions to events.

As a simple illustrative example, the following set of statements describe a system that will drive on

[3]Since ISR3 stores arbitrary C Structures, there is no way to prevent users from accessing tokens without using ISR3 functions, once the user has obtained one pointer into the database.

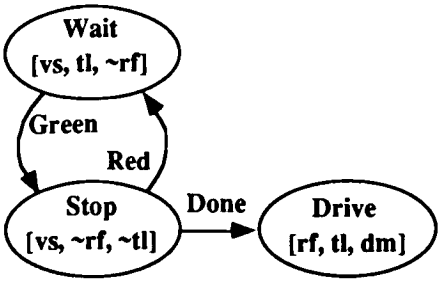
the road while obeying traffic lights, until a given distance is traveled:

While driving down the road
if the traffic light turns red, wait.
if goal is reached, stop.

While waiting at the traffic light,
if it turns green, start driving.

These statements correspond to the simple FSM in Figure 3a; note that a state represents a behavior or mode of operation, i.e, a set of concurrent perception-action processes. This example can be implemented with four perception-action processes: follow the road (*rf*), check for traffic lights (*tl*), monitor the distance traveled (*dm*), and stop the vehicle (*vs*). Listed below the state name are the perception-action processes that should be run and killed (marked with a ~) in that state.

Based on the notion of behaviors represented as states of a finite state machine, a Behavior Description Language (BDL) was designed and implemented. Behaviors are described as two sets of perception-action processes, and a transition table. The run set specifies the minimum set of processes that form the behavior; the kill set specifies those processes that should not be running for the correct execution of the behavior. The choice of two sets implies that processes that were running when the behavior started will continue to run unless explicitly killed. The transition table specifies what to do for each of the valid events (events not specified in the state description are not valid). The representation of our simple example expressed in the BDL is shown in Figure 3b. First, the perception-action processes available are listed. Then the set of states (or behaviors) and the set of events are declared. Finally, a description of each state is provided; parameters required for the perception-action processes associated with each state are fetched from the blackboard prior to their initiation. For a more complete example, see Section 4.4.



(a)

```
PROCS = {
    rf      "DriveOnRoad"
    tl      "CheckTrafficLight"
    vs      "VehicleStop"
    dm      "DistanceMonitor"}

STATES = {drive, wait, stop}

EVENTS = {red, green, done}

WHILE drive(d) {
    SET distance = d;
    RUN rf, tl, dm;
    EVENT red GOTO wait;
    EVENT done GOTO stop;}

WHILE wait () {
    KILL    rf;
    RUN     vs, tl;
    EVENT green GOTO drive;
```

(b)

Figure 3. Script of a simple driving system. (a) Finite state machine (FSM) representation. (b) Behaviour Description Language (BDL) representation.

4.2. Scripts - Augmented finite state machines

The simple FSM model discussed in the previous section has been augmented in two ways. First, a *fetch-goal* state was added to the set of states as a mechanism for compacting the representation. The system starts in the *fetch-goal* state where it reads goals (in terms of behaviours) from a precompiled plan. When a goal is retrieved, the script monitor writes relevant blackboard messages into the blackboard before creating the perception-action processes associated with the state. Once a perception-process is running, it looks for its parameters in the blackboard, which is implemented within the ISR structure described earlier. The *fetch-goal* state differs from all other states in two ways: (1) transitions out of the state are unlabelled since the next state is explicitly specified in the goal retrieved from the script; (2) Unless the plan is empty, the kill and run sets are ignored.

With these change, a script S is formally defined as the eight-tuple $(P,Q,E,M,\delta,\kappa,\rho,G)$, where:

- P is the set of available perception-action processes.
- Q is the set of states ($Q = B \cup$ *fetch-goal*, where B is the set of behaviors).
- E is the set of possible discrete events (transitions in the FSM).
- M is the set of valid blackboard messages.
- δ is the transition table, $\delta : B \times E \rightarrow Q$.
- κ is the kill table, $\kappa : B \times P \rightarrow \{0, 1\}$.
- ρ is the run table, $\rho : B \times P \rightarrow \{0, 1\}$.
- G is the plan expressed in terms of subgoals. Each subgoal is of the form $<b_g,M_g>$, for $b_g \in B$ and $M_g \subseteq M$.

Scripts can be generated by an automated planner, or by hand (using BDL).

4.3. The Script Monitor

The script monitor is in charge of "high level" control[4]: reading, interpreting, and executing BDL scripts. Essentially, the script monitor is a *plan execution system* [Georgeff 90] similar to PRS [Ingrand et. al. 92; Lee et. al. 93] in some aspects. The monitor does not perform any direct action on the vehicle controller by itself; rather, it controls the set of running processes which do take direct action.

The script monitor consists of two modules, an *interpreter* and an *execution system*. The interpreter takes a BDL script S and builds the transition (δ), kill (κ) and run (ρ) tables. After this, the subgoals in S's plan are stacked into the *execution stack* G. The execution system simulates a finite state machine as shown in Table 1.

Clearly, for the system to complete the task in G, the following must hold:

$$\forall b \in Q \; \exists s_b \in E^+ \text{ s.t. } \delta'(b,s_b) = \textit{fetch-goal}$$

where δ' is the transition function applied to a sequence of events [Hopcroft and Ullman 79].

[4] In this context, ``high level'' control is used to differentiate the control of processes from the "low level" control of the vehicle actuators.

4.4. A More Complete Example and an Experiment

A set of perception-action processes have been implemented for the MPL. These include:

- Vehicle pose determination based on landmark model matching [Beveridge 93; Draper 93b; Kumar 92,93].
- Neural-network road following (ALVINN) [Pomerleau 90].
- Servo-based steering [Fennema and Hanson 90; Fennema 91].
- Obstacle detection via stereo [Badal et. al. 94].
- Reflexive obstacle avoidance [Ravela et. al. 94].
- A distance monitor.
- Turning via dead reckoning.
- Servo to compass heading.
- Harmonic function path planner [Connolly et. al. 92,93]

In the future, additional processes will be added, including landmark tracking, recognition and modeling of natural landmarks, automatic landmark extension, etc.

Using a subset of these processes, an experiment was designed in order to demonstrate the capabilities of the vehicle and the performance of the independent perception-action processes. The following script was successfully tested on the vehicle at the UMass test site:

(1) Drive on the road, while avoiding obstacles, for x meters.
(2) Estimate vehicle position using landmarks.
(3) Drive on the road, while avoiding obstacles, for y meters.
(4) Estimate vehicle position using landmarks.
(5) Turn left (at the experimental site this command is a transition to off-road navigation).
(6) Drive off road (by servoing on a compass heading), while avoiding obstacles, for z meters.

1. $b_g \rightarrow$ *fetch-goal* (Start at the fetching state)	
2. if (b_g = *fetch-goal*) then	
(a) if G is empty then $\forall p \in P$	
i. kill (p)	(Terminate ALL processes)
ii. if ($\rho(b_g, p) = 1$) then run(p)	(Run cleanup processes)
iii. stop	(S was successfully executed)
(b) $<b_g, M_g> \leftarrow$ pop(G)	(Fetch next goal)
(c) blackboard $\leftarrow M_g$	(Write blackboard messages)
3. $\forall p \in P$ if $\kappa(b_g, p) = 1$) then kill(p)	Terminate processes)
4. $\forall p \in P$ if ($\rho(b_g, p) = 1$) then run(p)	(Create processes)
5. Wait for an event $e \in E$	(Wait)
6. $b_g \leftarrow \delta(b_g, e)$	(React to event - follow transition table)
7. goto 2	(Repeat until all goals are achieved)

Table 1. Script Monitor Execution Subsystem

The perception-action processes involved in this example include pose estimation (*pe*), road following (*rf*), obstacle detection (*od*), obstacle avoidance (*oa*), servoing to a compass heading (*se*), distance monitor (*dm*) and dead reckoning turning (*dt*). The full BDL script for this coordinated action, with x = 100, y = 150 and z = 50, is:

```
PROCS = {
    pe      "PoseEstimate"
    rf      "RoadFollow"
    od      "ObstacleDetect"
    oa      "Obstacle Avoid"
    se      "Servo"
    dm      "DistanceMonitor"
    dt      "DeadReckoningTurn"
    vs      "VehicleStop"}

STATES={drive-onroad, drive-offroad, turn,
        compute-pose, avoid-obstacles}

EVENTS = {success, obstacles, clear}

WHILE drive-onroad (dist) {
    SET distance = dist;
    RUN rf, od, dm;
    EVENT success GOTO compute-pose;
    EVENT obstacle GOTO avoid-obstacles;}

WHILE drive-offroad (dist) {
    SET distance = dist;
    RUN se, od, dm;
    EVENT success GOTO compute-pose;
    EVENT obstacle GOTO avoid-obstacles;}

WHILE turn (dir, dist) {
    SET direction = dir;
    SET distance = dist;
    RUN dt, dm;
    EVENT success GOTO fetch;}

WHILE avoid-obstacles ( ) {
    KILL rf, se;
    RUN oa, od;
    EVENT clear GOTO BACK;}

WHILE compute-pose ( ) {
    KILL rf, se;
    RUN pe;
    EVENT success GOTO fetch;}

GOALS {
    drive-onroad (100);
    drive-onroad (150);
    turn (left, 10);
    drive-offroad (50);}
```

The augmented finite state machine for this script is shown in Figure 4. Note that the figure shows the addition of the *fetch-goal* state discussed earlier. The FSM also shows a potential link to the high level planning system (not yet implemented) which is activated (in this example) by failure of the pose estimation state to localize the vehicle. In this case, the vehicle is considered to be lost - it then stops and initiates planning to resolve its location. Since planning may result in modifications of the goal stack, the addition of the

planning state[5] may theoretically change the model to a push-down automata (PDA).

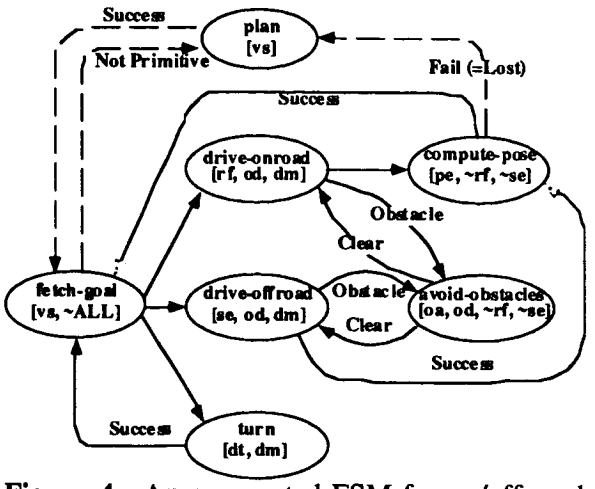


Figure 4. An augmented FSM for on/off road navigation.

5. Summary and Conclusions

One of the requirements for MPL's software environment was that it should support the integration of real-time visual procedures with as little overhead as possible. This meant that the focus of the system design had to be on the two critical areas of data storage/exchange and process control. Data storage and exchange is supported by the ISR3 real-time database/datastore system, which provides a central data repository and communication mechanism for all the perception-action processes running on the MPL. Behaviors are collections of concurrent perception-action processes whose interaction and execution are controlled through a script; MPL's script monitor is a low-overhead control system that switches from one behavior to the next in response to external conditions.

Although limited, the script monitor system in its current form has given us an idea of the complexity involved in building intelligent controllers, particularly in a real-time application domain where safety must be ensured. Encoding system reactions as a finite state machine seems a reasonable approach, but it is not clear how to optimally construct the individual states, i.e. which perception-action processes constitute each state, and how these processes interact with one another.

[5]Or any state that can write to the goal stack.

In the current implementation, all communication between processes is done through the blackboard. In this particular domain, where potentially large amounts of data are shared (images, maps, etc.), the shared memory paradigm seems the most efficient method of communication for processes running on the same machine. But if perception-action processes were to run in a distributed architecture, other means of communication will be necessary (UNIX sockets or a system such as TCX [Fedor 93]). In the current implementation, it was assumed that scarcity of resources was not an issue. However, independently executing concurrent processes which access sensors and send commands to actuators, or otherwise affect other scarce system resources, will inevitably cause problems if resource allocation is not handled correctly.

Resource scheduling and sharing, inter-process communication and real-time control are difficult problems, particularly in a real-time dynamic environment, and efficient solutions to them are essential if robust autonomous systems are to be constructed.

Acknowledgments

This work has been supported in part by Advanced Defense Research Projects Agency (via TACOM), under contract number DAAE07-91-C-RO35, and by the National Science Foundation under grant number CDA-8922572.

References

Badal, S. and B. Draper, "Stereo Obstacle Avoidance on the MPL", forthcoming Computer Science Technical Report, 1994.

Beveridge, J.R., "Local Search Algorithms for Geometric Object Recognition: Optimal Correspondence and Pose", Ph.D. Dissertation, University of Massachusetts at Amherst, Computer Science Department Technical Report TR93-71, 1993.

Brolio, J., B. Draper, R. Beveridge, and A. Hanson, "The ISR: An Intermediate System Representation for Computer Vision:, IEEE Computer, 22(12), 1989, pp. 22-30.

Brooks, R. A., "A Layered Robust Control Systems for a Mobile Robot," 2(1), 1986, pp. 14-25.

Connolly, C., and Grupen, R., "Harmonic Control", Proc. of the International Symposium on

Intelligent Control, Glasgow, Scotland, August, 1992, pp. 503-506.

Connolly, C., and Grupen, R., "On the Applications of Harmonic Functions to Robotics", Journal of Robotic Systems, Vol.10, No.7, October, 1993, pp. 931-946.

Draper, B., Hanson, A., and Riseman, E., "ISR3: A Token Database for Integration of Visual Modules", Proc. ARPA Image Understanding Workshop, Washington, D.C., April 1993a, pp. 1155-1161.

Draper, B., S. Buluswar, A. Hanson, E. Riseman, "Information Acquisition and Fusion in the Mobile Perception Laboratory," Proc. of Sensor Fusion VI, Boston, MA, Sept. 1993b, pp. 175-187.

Draper, B., A. Hanson, and E. Riseman, "Integrating Visual Procedures for Mobile Perception," in Experimental Environments (H. Christensen, Ed.), World Scientific Press, to appear; also to appear in CGVIP:IU, March 1994.

Fedor, C., "TCX Task Communications (Version 7.7)", Robotics Institute, Carnegie Mellon University, January 1993.

Fennema, C., "Interweaving Reason, Action, and Perception," Ph.D. Dissertation, University of Massachusetts at Amherst, Computer Science Department Technical Report TR91-56, 1991.

Fennema, C. and A. Hanson, "Experiments in Autonomous Navigation", Proc. 10th International Conference on Pattern Recognition, 1990, pp. 24-31.

Georgeff, M. P. Planning. In Readings In Planning, Morgan Kaufmann, 1990, pp. 5-25.

Hanson, A., and Riseman, E., and Weems, C., "Progress in Computer Vision at the University of Massachusetts", Proc. ARPA Image Understanding Workshop, Washington, D.C., April 1993, pp. 39-47.

Hopcroft, J. E. and J. U. Ullman. Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 1979.

Ingrand, F. F., M. P. Georgeff, and A. S. Rao, "An Architecture for Real-Time Reasoning and System Control, IEEE Expert, 1992, pp.

Kumar, R., and Hanson, A., "Robust Methods for Estimating Pose and a Sensitivity Analysis", CGVIP:IU to appear.

Kumar, R., "Model Dependent Inference of 3D Information from a Sequence of 2D Images", Ph.D. Dissertation, University of Massachusetts at Amherst, Computer Science Technical Report 92-04, February 1992.

Lee, J., M. J. Huber, E. H. Durfee, and P. G. Kenny, "UM-PRS: An Implementation of the Procedural Reasoning Systems," Artificial Intelligence Laboratory, Univ. of Michigan, 1993.

Payton, D.W., "An Architecture for Reflexive Autonomous Vehicle Control", Proc. IEEE Robotics and Automation Conference, 1986, pp. 1838-1845.

Payton, D. W., K. Rosenblatt, and D. M. Keirsey, "Plan Guided Reaction", IEEE Trans. on Systems, Man, and Cybernetics, 1990, pp. 1370-1382.

Pomerleau, D. A., "Neural Network-Based Autonomous Navigation," in Vision and Navigation: The CMU NavLab (C. Thorpe, ed.), Kluwer Academic Publishers, 1990.

Ramadge, P. J. and W. M. Wonham, "The Control of Discrete Event Systems", Proc. of the IEEE, 77(1), January 1989, pp. 81-98.

Ravela, S., and Draper, B., "Reflexive Obstacle Avoidance on the MPL", forthcoming Computer Science Technical Report.

Rochwerger, B., C. Fennema, B. Draper, A. Hanson, and E. Riseman, "Executing Reactive Behavior for Autonomous Navigation," forthcoming technical report, Computer Science Department, University of Massachusetts at Amherst, 1994.

Rivlin, E., "The DEDS Formalism for Systems with Vision," University of Maryland.

Sawhney, H., "Spatial and Temporal Grouping in the Interpretation of Image Motion", Ph.D. Dissertation, University of Massachusetts at Amherst, Computer Science Technical Report 92-05, February 1992.

Weems, C. Herbordt, M., Dutta, R., Daumueller, K., Weaver, G., and Dropsho, S., "Status and Current Research in the Image Understanding Architecture Program", Proc. ARPA Image Understanding Workshop, Washington, D.C., April 1993, pp. 1133-1140.

Williams, T., "Image Understanding Tools," 10th International Conference on Pattern Recognition, Atlantic City, NJ, June 1990, pp. 606-610.