Executing Reactive Behavior for Autonomous Navigation

Benny Rochwerger, Claude L. Fennema,
Bruce Draper, Allen R. Hanson and Edward M. Riseman

CMPSCI TR94-05

February 1994

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

Executing Reactive Behavior for Autonomous Navigation *

Benny Rochwerger Claude L. Fennema† Bruce Draper
Allen R. Hanson Edward M. Riseman

Computer Vision Laboratory
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003

Abstract

The aim of the Unmanned Ground Vehicle (UGV) project at the University of Massachusetts is to develop a system capable of navigating both on-road and cross country, avoiding obstacles, and determining its position using landmarks. Complex problems, such as driving, can be solved more easily by decomposing them into smaller sub-problems, solving each sub-problem, and then integrating the solutions. In the case of an autonomous vehicle, the integrated system should be able to "react" in real time to a changing environment and to "reason" about ways to achieve its goals. This paper describes the approach taken on the UMass Mobile Perception Laboratory (MPL) to integrate independent processes (each solving a particular aspect of the navigation problem) into a fully capable autonomous vehicle.

Keywords: Vision-guided robotics, real-time systems and applications, control, integration, finite state machines.

---

*This research was supported by DARPA via TACOM under contract number DAAE07-91-C-R035 and by the National Science Foundation under grant number CDA-8922572.

†Computer Science Program, Mount Holyoke College - South Hadley, 01075 MA

1 Introduction

One of the goals of the autonomous vehicle effort at the University of Massachusetts is the development of a landmark-based navigation system capable of robust navigation both on-road and cross-country. While driving, the vehicle must exhibit classic driving behaviors, such as staying on the road or prescribed path, avoiding obstacles, etc., while using landmarks for determining vehicle "pose" - that is, the position and orientation of the vehicle - with respect to a 3D world model. The emphasis of this effort is model-based navigation.

The experimental laboratory vehicle (Figure 1) for this effort is the Mobile Perception Lab (MPL), a heavily modified Army HMMWV ambulance that is equipped with actuators and encoders for the throttle, steering and brake. The interface to the on-board computer is through a 68030-based controller. On board diesel generators provide power.

The vehicle's vision sensor package includes a Staget, which is a rotating stabilized platform mounted on the roof of the cab with a CCD color camera and a FLIR sensor. 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 image processor, which provides frame rate image processing for certain type of operators. Space and power is also provided for the Image Understanding Architecture (IUA) [23], a massively parallel heterogeneous processor.

The 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 to provide autonomous driving functions. It is also important to provide an environment in which 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. In addition, control of the system must be both planned as well as driven by external events.

To illustrate the problem, consider the task of driving to your friend's new house. You were given instructions such as "at the second light on Main street turn right, then stop at the third house on your left". Obviously, this is not enough information to get you there and you will most likely consult

Figure 1: The Mobile Perception Laboratory, built on a modified HMMWV chassis, has computer controlled steering, throttle, and brakes, a complete research laboratory on the back.

a map to find out how to get from where you are to Main street. Once you have a general idea of where are you heading, you will get into your car and start driving towards your friend's house. On the way you may have to take actions that were not in your original plan, like stopping suddenly to let a child cross the street. In some cases you may have to completely rethink your plan, when one of the streets you wanted to take is closed for repairs.

For an autonomous vehicle to mimic human driving, it is not sufficient for it to have the ability to drive down the road, or recognize traffic lights. It must also know how to combine all of its capabilities (count traffic lights while driving), as well as how to react to unexpected events (stop, there is a child crossing the street); how to keep going when completely new situations are encountered (a road is closed); and how to complete the missing information (how to get to Main street).

The task of autonomous navigation, as with many other complex tasks, can be decomposed into more specific subproblems like road following, obstacle avoidance and landmark recognition. In order to achieve a fully autonomous vehicle, 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 sub-systems.

To achieve the desired flexibility we have implemented the control system as a "programable" finite state machine (FSM), where the states are the different modes of operation of the system (behaviors), and the transitions are 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.

The paper is structured as follow: the first section presents a brief review of behavior based systems, and the terminology used throughout this paper is introduced. Then, a formal definition of a *script* (section 3) and the *script monitor* (section 3.1), are presented. Experiments on the MPL are described in section 4. Section 5 discusses our thoughts on how to incorporate a planner into the FSM model, and section 6 presents some of the implementation open issues. Finally, section 7 summarizes the work.

2 What is a behavior?

> **behavior** 1: the manner of conducting oneself; 2a: anything that an organism does involving action and response to stimulation; 2b: the response of an individual, group, or species to its environment; 3: the way in which something (as a

machine) behaves. *Webster's 7th. dictionary.*

In the robotics literature, *behavior* has generally been used to describe processes that connect perception to action, i.e., ā behavior senses the environment and does something 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 [3, 4], the task of robot control is decomposed into levels of competence; each level, in combination with lower levels, defines a behavior. Payton, Rosenblatt and Keirsey [17, 18] in their Distributed Architecture for Mobile Navigation system (DAMN), 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 [20], as well as Rivlin [22], events are considered the alphabet, Σ, of a formal language; a behavior is 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. To make the distinction clear, we have chosen to think of a behavior as a *mode of operation* [18], in which several *perception-action processes* [6] 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 [1] . The system *reacts* 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. We employ a finite state machine (FSM) - with states representing behaviors and transitions representing reactions to events - to specify the system. For example, the following set of statements describe a system that will drive on the road while obeying traffic lights, until a given distance is travelled:

[1] In its current incarnation, the blackboard is built on top of the ISR3 - a symbolic real-time database for vision [2, 7]. Although the ISR3 was mainly designed as an in-memory database, it provides a very efficient memory management mechanism (on top of UNIX) and a set of primitives necessary for shared memory based communication.

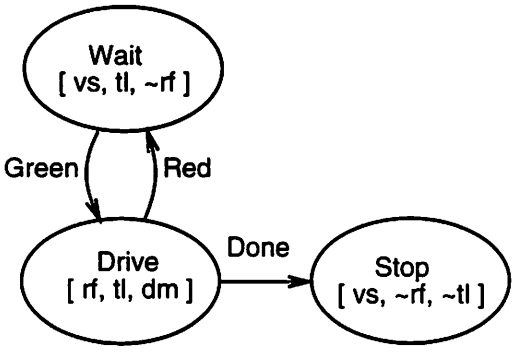
While driving down the road,
if the traffic light turns red, wait.
if goal is reached, stop.

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

These statements correspond to the FSM in Figure 2a. Note that a state represents a behavior or mode of operation, i.e, a set of concurrent perception-action processes.

Based on the notion of behaviors represented as states of a finite state machines, 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 [2] . The transition table specifies what to do for each of the valid events (events not specified in the state description are not valid). A simple BDL example is shown in Figure 2b; for a more complete example see section 4.

[2]The choice of two sets implies that processes that were running when the behavior started will continue to run unless explicitly killed.



(a)

```
PROCS={
  rf     "driveOnRoad",
  tl     "checkTrafficLight",
  vs     "vehicleStop",
  dm     "distanceMonitor"
}

STATES = { drive, wait, stop }

EVENTS = { red, green, done}

MSGS = {distance}

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 2: Script of a simple driving system: (a) Finite state machine (FSM) representation. Listed below the state name are the perception-action processes that should be run and killed (marked with a ~) in that state. This example is composed of four perception-action processes: follow the road (*rf*), check for traffic lights (*tl*), monitor the distance traveled (*dm*), and stop the vehicle (*vs*). (b) BDL representation. 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.

3 Scripts - Augmented finite state machines

Returning to the simple driving system, consider the following task:

1. Drive for 100 meters (while obeying traffic lights).
2. Turn left.
3. Drive for 150 meters (while obeying traffic lights).

In the FSM model presented in the previous section, this task will be represented as a six state FSM (Figure 3a). As the complexity of the vehicle's task grows, this approach will lead to long and repetitive representations. To alleviate this problem, the FSM is augmented with a *fetch-goal* state, and the behavior description is parameterized. A particular behavior (like drive for 100 meters or drive for 150 meters) is instantiated from this generic description through the use of blackboard messages at run time. The system starts in the *fetch-goal* state where it reads goals (in terms of behaviors) from a precompiled *plan*. When a goal is retrieved, the *script monitor* (section 3.1) writes relevant blackboard messages into the blackboard before creating the perception-action processes. Once such a process is running, it looks for its parameters in the blackboard (Figure 4). With this approach the same task is represented in four states. (Figure 3b).

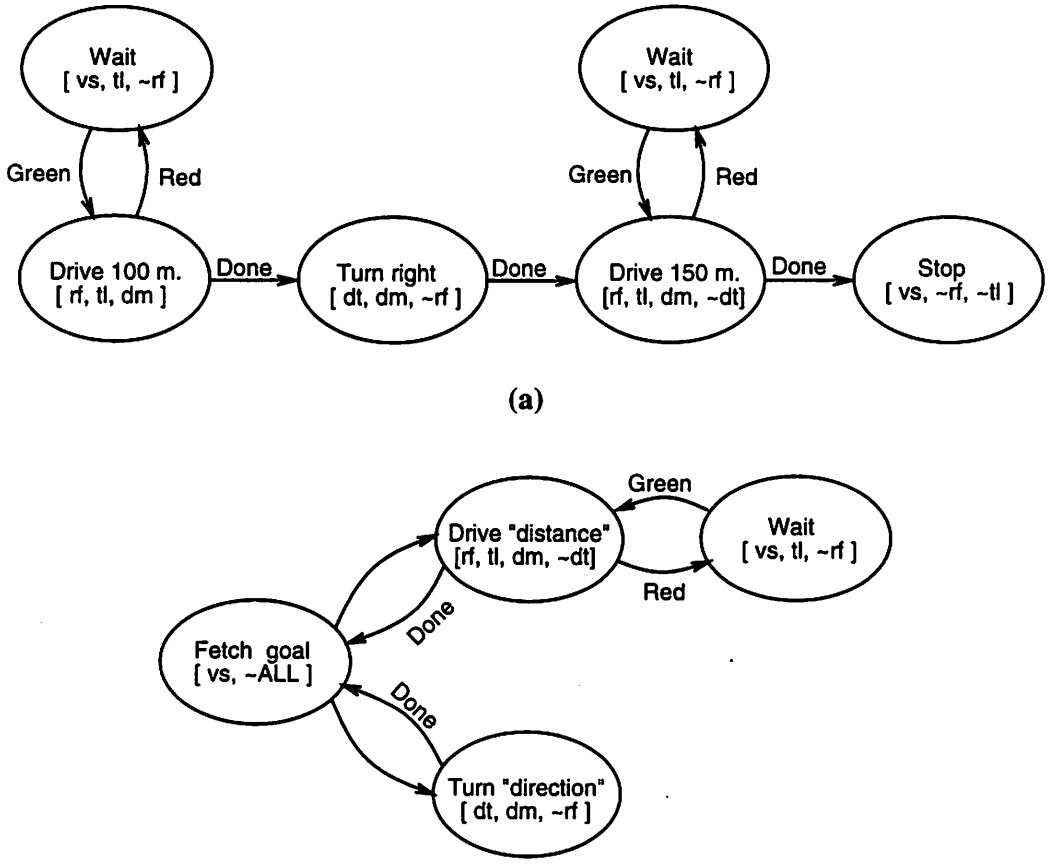


Figure 3: Script as an augmented finite state machine: (a) Specifying a task in terms of a sequence of behaviors (*dt* is a dead reckoning turning process). (b) Parameterizing the states and adding the *fetch-goal* state can greatly simplify the FSM. Blackboard messages (in quotes) are used to refine the behavior at run time. The *fetch-goal* state differs from all 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 (*ALL* is a shortcut to "all running processes").

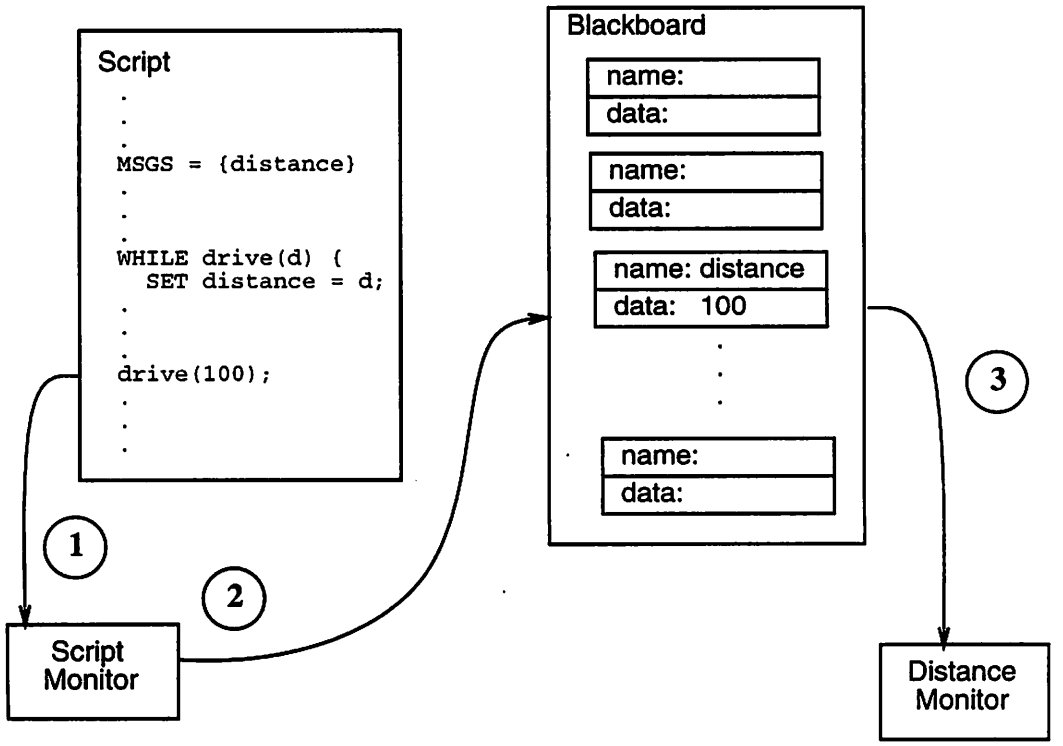


Figure 4: Passing parameters thru the blackboard: 1 - The *script monitor* interprets the script. 2 - When the *drive* state is entered, the *script monitor* writes a blackboard message for the *distance monitor*. 3 - The *distance monitor* reads the *distance* message from the blackboard.

Formally, a *script* S is 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).

3.1 The script monitor

The *script monitor* in our system is in charge of "high level" control [3] : reading, interpreting, and executing BDL scripts. Essentially, the script monitor is a *plan executer* [11] very similar to PRS [13, 16]. The monitor does not perform any action on the vehicle by itself but controls the set of running processes at any time.

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

1. $b_g \leftarrow$ *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)

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

(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)

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

$$\forall b \in Q \;\; \exists s_b \in E^+ \;\; s.t. \;\; \hat{\delta}(b, s_b) = fetch - goal$$

where $\hat{\delta}$ is the transition function applied to a sequence of events [12].

4 An example

The following perception-action processes have been successfully tested on the MPL:

- Vehicle pose determination based on landmark model matching [1, 5, 15].
- Neural-network road following (ALVINN) [19].

- Servo-based steering [10, 9].
- Obstacle detection via stereo.
- Reflexive obstacle avoidance.
- A distance monitor.
- Turning via dead reckoning.

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 tested:

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, while avoiding obstacles, for z meters.

The full BDL script for this coordinated action, with $x = 100$, $y = 150$ and $z = 50$, is shown in Figure 5; the FSM representation is shown in Figure 6. Note that the same result can be achieved with a different script: if in the driving states the *success* condition takes the system back to the *fetch-goal* state and a *compute-pose* statement is added after each *drive-onroad/drive-offroad* statement in the plan, the result will be similar. As in other programming languages, in BDL some things can be expressed in more than one way.

5 Planning

The script mechanism attempts to decompose the navigation problem into three sub-problems: *goal planning, reaction planning and behavior planning.*

Goal planning - Given a particular finite state machine, and a task (e.g. drive to Main street), what is the sequence of behaviors that the system must execute to achieve this task. In a BDL script this sequence is specified in the goal list, G.

Reaction planning - Given a set of behaviors, a set of possible events, and an abstract task (drive), what should be the reactions to detected events, i.e., how should the behaviors be connected into a finite state machine. This is given in the transition table, δ.

```
PROCS={
  pe   "poseEstimation",
  rf   "roadFollow",
  od   "obstacleDetect",
  oa   "obstacleAvoid",
  se   "servo",
  dm   "distanceMonitor",
  dt   "deadReckoningTurn",
  vs   "vehicleStop"
}

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

EVENTS = {success, obstacles, clear}

MSGS = {distance, direction}

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);
}
```

Figure 5: BDL script to achieve the following goal: drive on the road for 100 meters, then for 150 meters, take a left turn and drive off road for 50 meters.

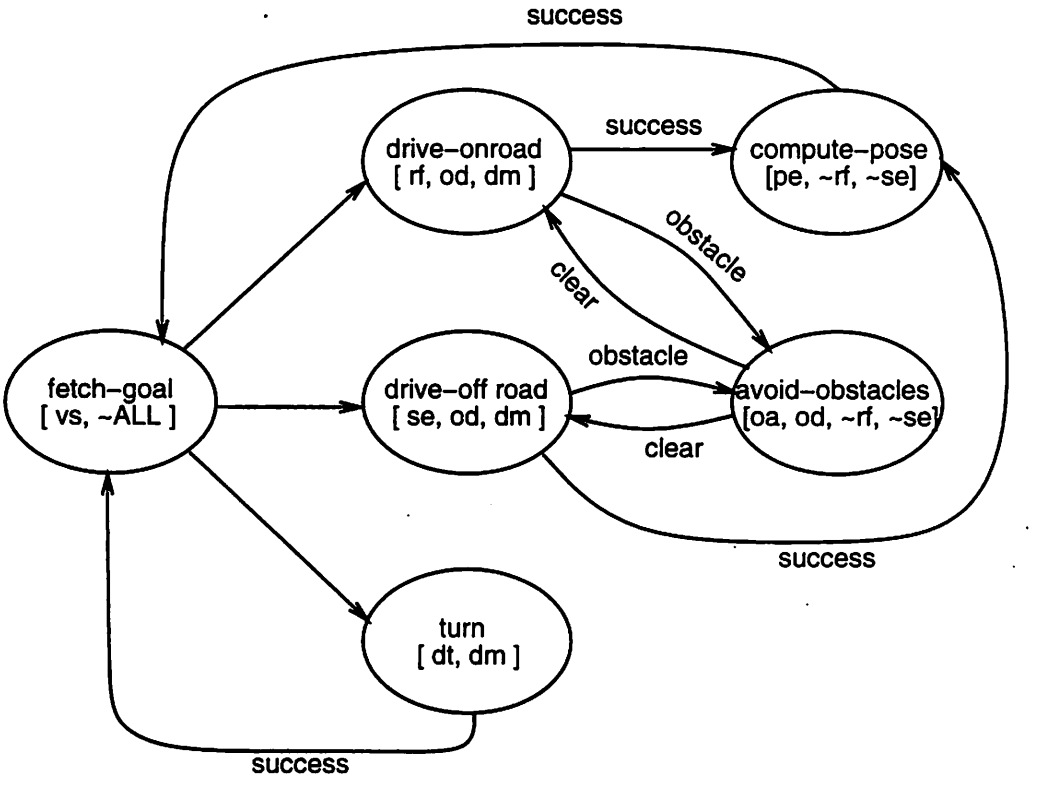


Figure 6: An augmented FSM for on/off road navigation : drive while avoiding obstacles for a specified distance, then check position. It can also take turns. The perception-action processes involved include pose estimation (pe), road following (rf), obstacle detection (od), obstacle avoidance (oa), servoing (se), distance monitor (dm) and dead reckoning turning (dt).

Behavior planning - How to construct behaviors from a set of perception-action processes, i.e., how should processes be combined to form intelligent behaviors. This is achieved in the script thru the run (ρ), and kill (κ) tables.

Conceptually, a solution to each of these sub-problems (a script) could be generated automatically. The first problem can be viewed as a traditional planning problem, where each of the behaviors can be considered to be primitive actions. At this level, a hierarchical planner, such as the one implemented in the RML system [9], can be easily integrated with the script monitor. The monitor fetches a subgoal from the stack G; if the specified action is not primitive, the planner is invoked to refine the subgoal. Note that planning is another behavior and hence planner invocation is done through the state transition mechanism. If a plan failure occurs the planner may also be invoked by the behaviors, to dynamically create a new plan, or at least part of a plan. Adding a planner in this way to the example of the previous section will produce the FSM in Figure 7. In this particular example, the *planning* state may be entered when the *compute-pose* behavior cannot estimate the current position (i.e. the system is *lost*). Note that at this level only the subgoals stack, G, is modified dynamically.

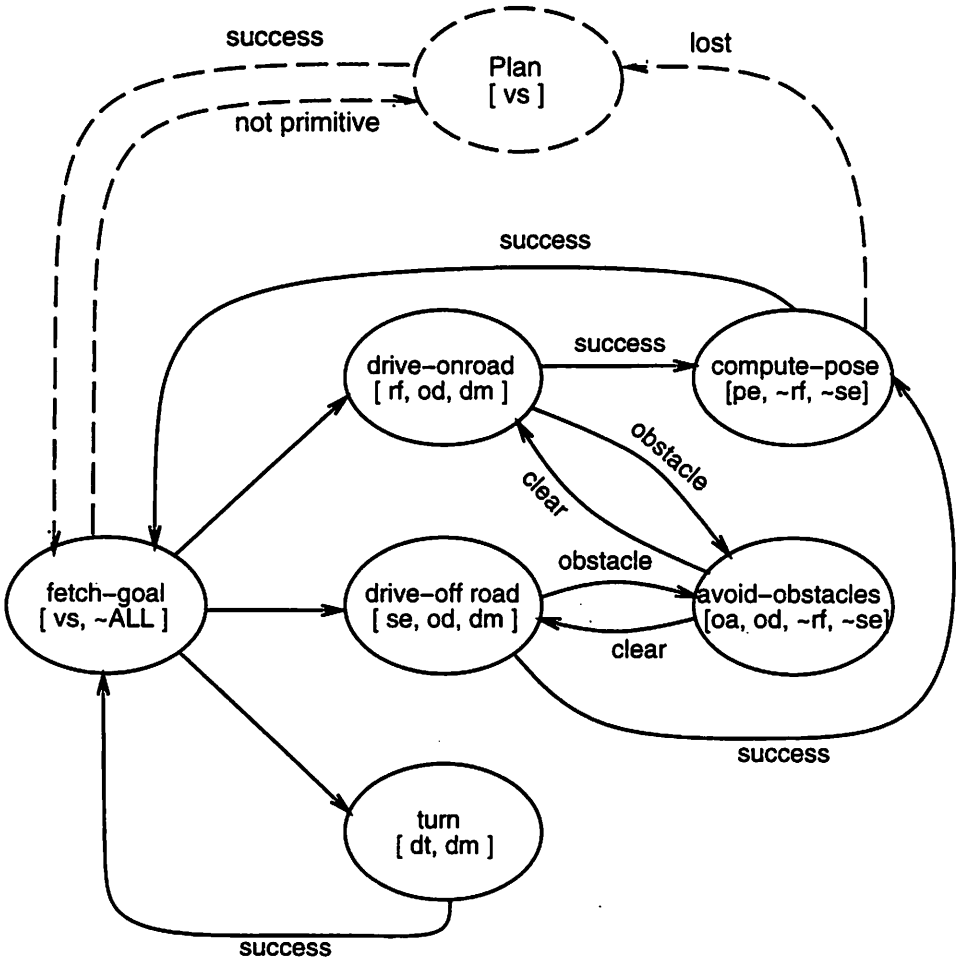


Figure 7: Integrating the monitor and the planner. If the monitor finds a non-primitive action, the planner is called to decompose it into primitives. If the *compute-pose* behavior cannot determine the position of the vehicle, the planner is called to generate a plan to get to a known point.

The second problem raises the issue of dynamic modification of reactions (FSM transitions, δ). There are several reasons we may want to have this capability:

- Partial planning - Planning reactions to all possible events can take a long time, when in most cases only a small subset of events typically occur in any particular state. The system should be able to pre-plan only for those events that are likely to occur - or even those which are unlikely but require an immediate reaction - thereby saving considerable amounts of pre-planning time. This can be useful in situations that rarely occur, and if they do occur, then the need to stop the vehicle to deliberate is not a bad tradeoff. An example could be discovering a mechanical failure in the vehicle; if this happens the best thing to do is to stop and think.

- Modifying reactions - With only partial information about the environment, and possibly inaccurate heuristics guiding the planning, some reactions may be either wrong or not optimal. As more information becomes available the system may improve its reactive capabilities. For example consider driving on a rainy day; under normal (dry) conditions the best reaction to an obstacle which suddenly appears in front of the vehicle is to brake sharply. On a wet road, this may be a very bad de-

cision, as soon as the planner realizes that the road is wet, the reaction to a sudden obstacle should be reconsidered.

The composition of behaviors (sub-problem 3) raises many questions. Each state in the FSM (a behavior in our terminology) can be viewed as an instance of Brooks' layered control [3]; as such, arbitration of commands to the actuators, and allocation of sensors (and other resources), become problems (see section 6). The solution of these problems is fundamental for a planner that composes behaviors.

6 Open Issues

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. These are difficult problems and an efficient solution to them is essential.

6.1 Resource Sharing

Concurrent processes accessing sensors and sending commands to actuators will inevitably cause problems if resource allocation is not handled correctly. For example, consider the case where the vehicle is to travel down the road, looking for a landmark while avoiding obstacles. This behavior can be achieved by three perception-action processes, *road-follow*, *obstacle-avoidance* and *find-landmark*, running concurrently. The problem is that the resources needed by these processes are limited and conflicts will surely arise (see Figure 8). Even in this simple example each resource poses some difficult problems:

- *The vehicle controller* - Commands from different processes have to be combined (arbitration). The way this combination is achieved is crucial if the vehicle is to execute rational behaviors.

- *The image processor* - Interleaving processes in this unit may considerably slow down the system. On the other hand it is clear that while moving some processes (such as avoid obstacles!) cannot be suspended for long periods of time while the image processor is being used by some other process.

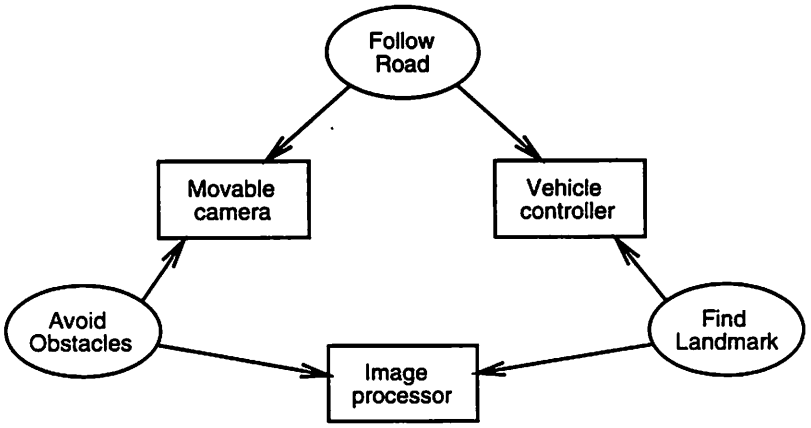


Figure 8: Resource allocation graph for a driving state. Perception-action processes are represented as ovals, and resources as rectangles. A directed edge from a process P to a resource R indicates that P uses R. Assuming there is only one instance of each resource, the potential for deadlocks is large.

- *The movable camera* - It is likely that different processes will try to point the camera in different directions or require its use at the same time. Generally, the time to move a mechanical device such as this is much longer than computation time. Again we encounter the problem of probable long waits.

6.2 Inter-process Communication

In the current implementation, all communication between processes is done thru 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, TCX [8], etc.).

Ideally, the blackboard should support several mechanisms for communication. Processes running on the same machine can still communicate through shared memory; processes on different machines use some other mechanism. Of course, whenever timing constraints allow it, the actual mechanism should be transparent to the perception-action processes, i.e., access routines should look the same regardless of where is the data or how it is copied from one process to another. In hard real-time situations this will not be possible.

To try to alleviate possible bottlenecks, the communication needs of each process should be considered when it is assigned a machine. The scheduler can try to run those processes with large communication needs (those sharing big chunks of data) in the same machine. All of this becomes part of the planning process.

7 Conclusions

An autonomous vehicle should be able to react in real-time to a changing environment, but it should also be able to reason about its goals. In our system, these seemingly contradictory capabilities are achieved through the use of a"programable" finite state machine. In this model, reactions to events (either internal or external) are represented as state transitions. The system can react rapidly by following a transition table. Goal-directed reasoning is supported by the "programability" of the FSM - the system does not enforce what goes into a state or a particular transition table, instead it executes "scripts" which eventually will be written by an automated planner.

8 Acknowledgments

Many people were involved in the design and implementation of the system described in this paper. Special thanks go to Gökhan Kutlu for his work on the ISR3, Srinivas Ravela for his work on the reflexive obstacle avoidance process, Sumit Badal for his work on the stereo based obstacle detection process, Jonathan Lim, Shashi Buluswar and Katja Daumeller for their work in the landmark based pose estimation process, Carnegie-Mellon University and Dean Pomerleau for making ALVINN available, Alan Boulanger for bringing up the ALVINN code, to Bob Heller for his general help, and to Gila Kamhi

for her careful review of this paper.

References

[1] R. Beveridge. Local search algorithms for geometric object recognition: Optimal correspondence and pose. Ph.D. Thesis CMPSCI TR93-71, University of Massachusetts at Amherst, 1993.

[2] J. Brolio, B. Draper, R. Beveridge, and A. Hanson. The ISR: an intermediate-level database for computer vision. *Computer*, 22(12):22–30, 1989.

[3] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1), March 1986.

[4] R. A. Brooks. Intelligence without representation. In *Proceedings of the Workshop on the Foundations of Artificial Intelligence*, Cambridge, MA, 1987. MIT Press.

[5] B. Draper, S. Buluswar, A. Hanson, and E. Riseman. Information acquisition and fusion in the mobile perception laboratory. In *Proc. of Sensor Fusion VI*, pages 175–187, Boston, MA, Sept. 1993.

[6] B Draper, A. Hanson, and E. Riseman. Integrating visual procedures for mobile perception. In H. Christensen, editor, *Experimental Environments*. World Scientific Press, to appear. also in CVGIP:IU, March 1994.

[7] B. Draper, G. Kutlu, and J. Wong. The ISR3 user's manual. University of Massachusetts - Amherst.

[8] C. Fedor. TCX task communications (version 7.7). Robotics Institute - Carnegie Mellon University, Jan 1993.

[9] C. L. Fennema. Interweaving reason, action and perception. Ph.D. Thesis COINS TR91-56, University of Massachusetts, 1991.

[10] C. L. Fennema and A. R. Hanson. Experiments in autonomous navigation. In *Proceedings of the Tenth International Conference on Pattern Recognition*, pages 24–31, 1990.

[11] M. P. Georgeff. Planning. In *Readings in Planning*, chapter Introduction, pages 5–25. Morgan Kauffmann, 1990.

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

[13] F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert*, 1992.

[14] K. Kluge. Multisensor system integration for autonomous navigation tasks. In I. Masaki, editor, *Intellegent Vehicles.* IEEE Press, to appear in 1994.

[15] R. Kumar. Model dependent inference of 3D information from a sequence of 2D images. Ph.D. Thesis COINS TR92-04, University of Massachusetts at Amherst, 1992.

[16] J. Lee, M. J. Huber, E. H. Durfee, and P.G. Kenny. UM-PRS: An implementation of the procedural reasoning system. Artificial Intelligence Laboratory - University of Michigan.

[17] D. W. Payton. An architecture for reflexive autonomous vehicle control. In *Proceedings IEEE Robotics Automation Conference*, pages 1838–1845, 1986.

[18] D. W. Payton, K. Rosenblatt, and D. M. Keirsey. Plan guided reaction. *IEEE Transactions on Systems, Man and Cybernetics*, pages 1370–1382, 1990.

[19] D. A. Pomerleau. Neural network based autonomous navigation. In Charles Thorpe, editor, *Vision and Navigation: The CMU Navlab.* Kluwer Academic Publishers, 1990.

[20] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, January 1989.

[21] E. Rich and K. Knight. *Artificial Intelligence.* McGraw-Hill Inc., second edition, 1991.

[22] E. Rivlin. DEDS formalism for systems with vision. University of Maryland.

[23] C. Weems et al. Status and current research in the image understanding architecture program. In *Proc. Image Understandig Workshop*, pages 1133–1140. DARPA, April 1993.