Vehicle control architectures

R.G. Longoria
Spring 2015
Recall: Driver-Vehicle-Environment Interactions

Classical view of passenger vehicle systems and principal types of operating modes — *annotated with systems/modes*

Adapted from Wong (2001)
Objectives

- Think about how we need to have the ‘low-level’ type systems we’ve been focusing on interact with other sensors, human(s), and other types of controlling elements.
- Review the type of software/system architecture that seems to be most common for vehicle applications.
- Plan how to experiment with this approach in the lab, especially using state machine architecture in LabVIEW.
Useful references

On UT library site

Autonomous Ground Vehicles
Ozguner, Umit; Acarman, Tankut; Redmill, Keith

In the near future, we will witness vehicles with the ability to provide drivers with several advanced safety and performance assistance features. Autonomous technology in ground vehicles will afford us capabilities like intersection collision warning, lane change warning, backup parking, parallel parking aids, and bus precision parking. Providing you with a practical understanding of this technology area, this innovative resource focuses on basic autonomous control and feedback for stopping and steering ground vehicles. Covering ...

Title: Autonomous Ground Vehicles
Author(s): Ozguner, Umit; Acarman, Tankut; Redmill, Keith
Publisher: Artech House
Series: Dowoy: 620.2042
Publication Date: 01 Aug, 2011
Pages: 288
Category: Engineering

Print ISBN: 97816008071920
eISBN: 97816008071937

Also: Bekey, Valvano, Jones, etc.
From Ozguner, et al

CHAPTER 3
System Architecture and Hybrid System Modeling

3.1 System Architecture
   3.1.1 Architectures Within Autonomous Vehicles
   3.1.2 Task Hierarchies for Autonomous Vehicles

3.2 Hybrid System Formulation
   3.2.1 Discrete Event Systems, Finite State Machines, and Hybrid Systems
   3.2.2 Another Look at ACC
   3.2.3 Application to Obstacle Avoidance
   3.2.4 Another Example: Two Buses in a Single Lane

3.3 State Machines for Different Challenge Events
   3.3.1 Macrostates: Highway, City, and Off-Road Driving
   3.3.2 The Demo '97 State Machine
   3.3.3 Grand Challenge 2 State Machine
   3.3.4 The Urban Challenge State Machine
   References
Task hierarchies

From Ozguner, et al, Chapter 3

Invert this…

Figure 3.1  The generic structural/functional architecture implemented on OSU ACT.
Example

From Ozguner, et al, Chapter 3

Figure 3.2 The architecture for the OSU Team car used in Demo ’97.
3.1.2.1 High-Level Control

High-level control can be thought of as behavior generation, and in our autonomous vehicles is expressed as a hierarchy of finite-state machines. At the top level, overall families of behavior appear as independent state machines, which we designate as metastates.

The DARPA Grand Challenges of 2004 and 2005 were both off-road races. As such, the only behavior and thus the only metastate required would be path following with obstacle avoidance from point A to point B. However, since there was no path or lane that could be discerned from a roadway, the only method of navigation is to rely on GPS- and INS-based vehicle localization and a series of predefined waypoints. Obstacle avoidance techniques were needed, although in the less structured off-road scenario greater freedom of movement and deviations from the defined path were allowed. The Grand Challenge race rules ensured that there were no moving obstacles and different vehicles would not encounter each other in motion. General off-road driving would of course not have this constraint.

Fully autonomous urban driving introduces a significant number of new metastates—situations where different behavior and different classes of decisions need to be made. Figure 3.4 shows the highest-level metastate machine that defined the urban driving behavior of OSU-ACT. The DARPA Urban Challenge, although quite complex, did have fairly low speed limits, careful drivers, and no traffic lights. Visual lane markings were unreliable and thus true to life. The terrain was fairly flat, although some areas were unpaved, generating an unusual amount of dust and creating problems for some sensors.
Figure 3.4 General hybrid system showing the interfaces.
3.1.2.4 Sensor Interpretation and Situation Analysis

The term *situation* is defined to be knowledge concerning the vehicle and/or the prevailing scenario and surroundings. **From a practical viewpoint, situations are the switching conditions among metastates and all the substates inside the high-level control state machines.** Thus, the aim of situation analysis is to provide the high-level controller with all the switching conditions in a timely manner. The situation analysis software analyzes the current vehicle state, the current and upcoming required behavior for the route plan, the map database, and the sensor data to identify specific situations and conditions that are relevant to the vehicle’s immediate and planned behavior.

For the off-road Grand Challenge scenario, the situation is always obstacle and collision avoidance. The software is required to identify, when analyzing the occupancy grid map blocks, the desired path and to adjust the planned path as needed.

For an urban scenario, we are interested in all the targets in our path and the targets in surrounding lanes or on roads intersecting our lane. We are not interested in targets that do not affect the current situation and planned behavior. While an autonomous vehicle is navigating through the city, many different situations may arise. The situations may vary if the vehicle is on a one-lane road, a two-lane road, an intersection, and so on.

*‘transitions’*
3.1.2.5 Low-Level Control

Command Interface
In a two-level control hierarchy as shown in Figure 3.4, the low-level control receives operational instructions from the high-level control module. These instructions take the form of:

1. A path to be followed, defined by a set of approximately evenly spaced control points;
2. A desired speed;
3. Commands to indicate starting and stopping;
4. Special commands indicating motions, which can be fully implemented at the lower level.

Figure 3.4 General hybrid system showing the interfaces.
**Longitudinal Control**

The interface and control of vehicle actuation is achieved by having a drive-by-wire car. Our experience has been that a simple control algorithm, for example a set of PID controllers, is adequate to generate a virtual torque command to achieve the commanded speed, and a state machine is used to select between the use of throttle, active braking, or engine idle braking. Speed commands are modified to constrain the acceleration and jerk of the vehicle to preset comfortable limits. There may also be emergency deceleration modes that are less comfortable.

Urban driving, in contrast to highway or off-road driving, requires the vehicle to execute a precise stop at predefined locations, for example the stop line of an intersection. To accomplish this, the low-level control determines the distance from the vehicle’s current position to a line drawn through the specified stopping point and perpendicular to the vehicle’s path of travel, taking into consideration the distance from the front bumper of the vehicle to its centroid. The speed of the vehicle is controlled to follow a specified, possibly nonlinear, deceleration trajectory.
**Lateral Control**

The path that the vehicle is to follow is specified as a set of control points. The lateral controller identifies both the current location of the vehicle and the look-ahead point (a prespecified distance ahead of the vehicle along its lateral axis) and extracts a subset of control points closest to each location. Constant radius circles are fitted to the points in each subset and these circles are used to compute the vehicle offset distances from the path and to estimate desired yaw rates. Each subset of points also defines a desired yaw angle for the vehicle. The offset distances, yaw angle error measurements, and desired yaw rates can be used to generating a feedback signal for the steering controller. There are a number of algorithms that can be used in this control loop, and a simple PID controller with fixed gains is not enough to cover all possible driving and path-shape scenarios. The variations here are speed dependent and turn-radius dependent.

Examples could be short movements along constant radius arcs, precision stops, and so forth. The low-level control will execute a given command set until the command is completed and the vehicle is in a stationary state, the vehicle has driven off the end of the path provided, at which point the vehicle will be stopped, or it receives a new command set.
Contrast these discussions with your reading of Jones (to be provided) on programming behaviors.

Also, relate to general concept of robot software architectures as discussed by Bekey Chapter 5

Many examples of how these architecture concepts have been used are reported in the literature on autonomous vehicles for both on and off road use.
3.2 Hybrid System Formulation

3.2.1 Discrete Event Systems, Finite State Machines, and Hybrid Systems

The high-level aspects of an intelligent vehicle can be modeled as a discrete event system. In this section we develop a modeling approach for modeling the discrete event system (DES). We represent the DES with a finite state machine (FSM), and then couple the FSM with a continuous time, dynamic system to create a hybrid system. One does not always need to go through the full formal development introduced here. Indeed, in many cases it is quite possible to directly develop an FSM.

We claim that hybrid system models are particularly useful in representing, simulating, and analyzing autonomous vehicles as they perform in complex environments, under prespecified scenarios and in possibly unplanned situations [4].

Let $X$ denote the set of DES states and $E$ denote the set of events. We define an enable function

$$g : X \rightarrow P(E) - \{\emptyset\}$$

which specifies which events are enabled at time $t$, and $P(E)$ denotes the power set of $E$. The DES state transition function is given by a set of operators

$$f_E : X \rightarrow X$$

where $E$ is a subset of $g$. The transition function specifies the next state when the event(s) in $E$ occur.

Alternatively, the state transitions can be shown on a graph, where the nodes represent the states and the directed arcs are labeled with the individual events $e$ in $E$, and are pointing from $x$ to $f_E(x)$.

Now let us consider the case where the events $e$ can be either generated externally, or can be generated by a separate continuous time, dynamic system. The interface from the continuous time system to the DES is described by a function $\Phi$. Similarly, the DES also affects the continuous time system through a function $\Psi$ (see Figure 3.4). Further details and examples in a nonvehicle context, can be found in [5].

In the following sections, we shall first look at ACC to formulate it as a DES and model it in terms of a finite state machine. We shall then consider first an obstacle avoidance operation and then a special bus service situation as hybrid systems.
3.2.2 Another Look at ACC

Consider first a cruise control system. The states of this system are given as \{Cruise, Manual, Speed up\}. It is assumed that the car slows to a speed below the desired cruise speed when it is in manual. When a return to cruise speed is desired, it can be accomplished either manually, or by setting cruise ON again. The events then are \{Set ON, Set OFF\}. It is assumed that the desired cruise speed is known.

A proposed state machine representing this DES is given in Figure 3.5. Students are urged to discuss if this machine is correct, and if, in fact, the above discussion was truly representative of what a cruise control system is expected to accomplish.

![Diagram](image)

**Figure 3.5** Standard cruise control logic (state machine model).
An advanced cruise control system is shown in Figure 3.6. It is of course assumed that our car has the means of detecting a car ahead of us and measuring its distance and relative speed. It will then slow down and match speeds, and will speed up if the car ahead speeds up. This speeding up process will continue until the car ahead gets to be faster than our desired cruise speed. At that time, we return to regular cruise operation.

Figure 3.6 The state machine for an ACC system.
Later, we’ll look at the lateral control problem.

First, let’s now look at basic FSM representation
Other types of abstractions we like to use?
Types of FSMs

We will present two implementations of finite state machines. The **Moore FSM** has an output that depends on state, and the next state depends on input and current state. On the other hand, the **Mealy FSM** has an output that depends on both the input and the state, and the next state depends on input and current state. We will use a Moore implementation if there is an association between a state and an output. There can be multiple states with the same output, but the output defines in part what it means to be in that state. For example, in a traffic light controller, the state of green light on the North road (red light on the East road) is caused by outputting a specific pattern to the traffic light. Conversely, we will use a Mealy implementation if the output causes the state to change. In this situation, we do not need a specific output to be in that state; rather, the outputs are required to cause the state transition. For example, to make a robot stand up, we perform a series of outputs causing the state to change from sitting to standing. Although we can rewrite any Mealy machine as a Moore machine and vice versa, it is better to implement the format that is more natural for the particular problem. In this way the state graph will be easier to understand.
The goal of this section is to design a finite-state machine robot controller, as illustrated in Figure 2.26. Because the outputs cause the robot to change states, we will use a Mealy implementation. The outputs of a Mealy FSM depend on both the input and the current state. This robot has mood sensors, that are interfaced to Port A. The robot has four possible mutually exclusive conditions:

00 OK, the robot is feeling fine
01 Tired, the robot energy levels are low
10 Curious, the robot senses activity around it
11 Anxious, the robot senses danger

Figure 2.26
Robot interface.
There are four actions this robot can perform, which are triggered by pulsing (make high, then make low) one of the four signals interfaced to Port B.

PB3 SitDown, assuming the robot is standing, it will perform a sequence of moves to sit down
PB2 StandUp, assuming the robot is sitting, it will perform a sequence of moves to stand up
PB1 LieDown, assuming the robot is sitting, it will perform a sequence of moves to lie down
PB0 SitUp, assuming the robot is sleeping, it will perform a sequence of moves to sit up

For this design we can list heuristics describing how the robot is to operate:

- If the robot is OK, it will stay in the state it is currently in.
- If the robot’s energy levels are low, it will go to sleep.
- If the robot senses activity around it, it will awaken from sleep.
- If the robot senses danger, it will stand up.
These rules are converted into a finite-state machine graph, as shown in Figure 2.27. Each arrow specifies both an input and an output. For example, the “Tired/SitDown” arrow from Standing to Sitting states means if we are in the Standing state and the input is Tired, then we will output the SitDown command and go to the Sitting state. Mealy machines can have time delays, but this example just didn’t have time delays.

![Finite-State Machine Diagram]

The first step in designing the software is to decide on the sequence of operations.

1. Initialize directions registers
2. Specify initial state
3. Perform FSM controller
   a) Input from sensors
   b) Output to the robot, which depends on the state and the input
   c) Change states, which depends on the state and the input

From here, we’d jump to our software environment, e.g., LabVIEW
Refer also to the handout from Jones (see course log), for a nice discussion on how robot behaviors can be represented by finite state machines.

**FSM Example: Escape**

From Jones, Chapter 3

Let’s now consider a finite state machine example closer to a mobile robotic application—the classic Escape behavior. This is the behavior a robot runs after it has collided with an obstacle. Escape tries to extricate the robot from the near vicinity of the collision.

The robot knows that a collision has occurred because contact with an obstacle compresses the bumper. Many physical mobile robots incorporate a bumper instrumented with two switches, one at either side of the bumper. When the bumper compresses, because of collision with an obstacle, one or both of the “bump switches” trips, alerting the electronics to the collision. Escape is the software routine called on to respond and rescue the robot.

---

**Figure 3.6**

One FSM implementation of the Escape behavior is shown in the diagram. In the No action state, the behavior does not compute a command for the robot. If the bumper detects a collision on the left, the No action state outputs $d$ with a value of right and moves to the Backup state. If a collision occurs on the right side of...
Summary

• Reviewed prevailing view on how vehicle control architectures can be view from high to low level.
• We discussed why and how FSMs can be used not only to model but also to implement basic behaviors in vehicle control architectures.
• Human behavior abstraction? Yes, you can use ‘mental models’ in the form of FSMs to represent your understanding of how a human would interact or interpret the operation of a machine.
• See handouts provided on the clog.