CMP464/788

Multitasking, Concurrency, Events

Objectives

Learn to use finite state machines for event-based programming

Define a finite state machine for going from here to there

Finite state machines

This is not so much a technique to be learned as a way of thinking about things – if that's really a distinction. You decide that for yourself!

A finite state machine is a method for simplifying descriptions of what some complicated programs do. A finite state machine has a state, which describes what it is doing right now, and a collection of actions that can be performed on it. For example, everyone is familiar with a telephone. Let's consider just the telephone functions of a simple cell phone (ignoring voice mail and call waiting for the moment).

The states are:
Idle: the cell phone isn't doing anything
Dialing: the cell phone user is entering a number to call
Calling: the cell phone user is listening to the ringtone in his cell phone, while waiting for the other party to answer
Talking: the cell phone user is talking to someone on another phone
Ringing: the cell phone is ringing (or perhaps vibrating) because someone has called

The actions are:
Dial: A user enters a number in the cell phone
Send: A user hits the send button, to call another telephone
Answer: A user hits the send button because the phone is ringing
Hangup: A user hits the end button to end the call

Each action takes the cell phone from one state to another (or, if you use it in the wrong state, it may just do nothing). Figure 1 is an example of a drawing of a finite state machine.

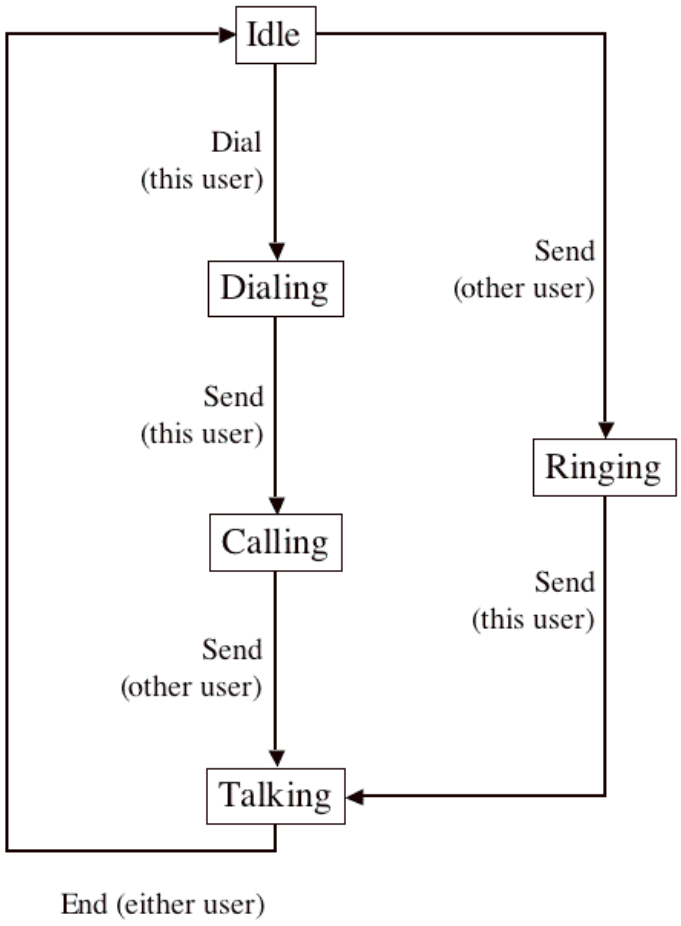

Figure 1. A Finite State Machine representing a telephone.

Using Finite State Machines to Simplify Programming Robots

Assume your robot for going from here to there is also using its light sensors to keep it going straight (by turning away whenever it hits the wrong color line). Suppose you program your robot to alternate counting lines and continue going straight. What problem could happen?

Right, it could miss the fact that it has crossed a line because it was trying to straighten out from hitting another line. This kind of thing can happen when there is just one task, alternating between different functions – nothing is looking out for one event while handling the other!

A similar problem arises if you do any waiting. At least one other task must be running for any event to be detected while the robot is using the wait command.

On the other hand, if multiple tasks are trying to control the motors, the robot can get into some rather undesirable situations[1]. For example, the Bugbot (see chapter 7) performs turns when an antenna is hit by turning off the motor opposite the hit antenna until the touch sensor is released. But if both antennae are hit, the Bugbot just stops, and the touch sensors behind the antennae are never released.

But we can develop a style of programming that avoids both of these problems by using finite state machines.

Using state machines

The central idea is to use several tasks to keep track of what's happening to the robot (that is, to update what its current state is). We'll call these "sensing tasks," because they monitor sensors and record their current value in global variables. Meanwhile, the main routine continually examines the current state (the values of the global variables) and decides what to do given the state.

Doing it this way means that only one task controls the activity of the robot (the main task), so the kind of conflict we saw with the Bugbot can't occur.

Also, it won't be necessary to use the wait command. Instead of having the robot enter a wait until it clears an obstacle, for example, we have it continually monitoring the sensors and also continually checking its current state (in the main task) so that the main task knows as soon as anything changes.

An example

We will construct a robot that tries to find another robot with a light sensor inside an area, including obstacles. Our robot will have bumpers, a light sensor, and two motors to run its treads or wheels. We will have tasks to monitor events on the light sensors and bumpers.

Tasks:

> Moving: run motors to move toward light and avoid obstacles
> Checking light sensor: set saw_light to 0 (none seen) or 1 (seen); this will require two tasks, one for the "light detected" event and one for the "no light detected" event.
> Checking bumpers: set hit_bumper to 0 (none hit), 1 (left hit), 2 (right hit), or 3 (both hit); this will require four tasks, one to detect when a bumper is pressed and the other when it is released, for each of the two bumpers.

Finite State Machine

State variables:

---

[1] It turns out that there is an acquire statement that effectively provides a semaphore for controlling access to resources. We will investigate the use of this statement in a future lecture.

saw_light: 0 or 1 – this state variable will be set by the sensing task for detecting a light and the sensing task for detecting no light (or for detecting dark)
hit_bumper: 0, 1, 2, or 3 – this state variable will be set by the sensing tasks for detecting bumpers pressed and released
direction: stopped, forward, reverse, spin left, spin right – this state variable will be set by the main task as it controls the motors to do the corresponding actions.

Actions:

Stop
Go forward
Go back
Spin left
Spin right

The following table shows all possible states (that is, all possible values of saw_light, hit_bumper, and direction) and what to do in each state. Note that the only thing that the robot can influence directly is the new direction, so that's all it changes; then the sensing tasks must update the other parts of the state as they change (e.g., when a bumper is hit).

Note that when the robot has to avoid something, it deals with that first (ignoring the light). In general, it spins right when the left bumper has been hit, left when the right bumper has been hit, and reverses when both bumpers are hit. However, if it's already spinning right when the left bumper is hit, something is strange so it goes into reverse to get away; similarly, if it's already spinning left when the right bumper is hit, it goes into reverse.

If it doesn't have to avoid anything, it checks if there is a light visible and goes toward it. But if it doesn't see any light, it spins. The choice of spin direction is either the current direction, if it is already spinning, or random (just "spin") if it is not.

saw_light	**hit_bumper**	**direction**	**new direction**
0	0	stopped	spin
0	0	forward	spin
0	0	spin right	spin right
0	0	spin left	spin left
0	0	reverse	spin
0	1	stopped	spin right
0	1	forward	spin right
0	1	spin right	reverse
0	1	spin left	spin right
0	1	reverse	spin right
0	2	stopped	spin left
0	2	forward	spin left
0	2	spin right	spin left
0	2	spin left	reverse
0	2	reverse	spin left
0	3	stopped	reverse

0	3	forward	reverse
0	3	spin right	reverse
0	3	spin left	reverse
0	3	reverse	reverse
1	0	stopped	forward
1	0	forward	forward
1	0	spin right	forward
1	0	spin left	forward
1	0	reverse	forward
1	1	stopped	spin right
1	1	forward	spin right
1	1	spin right	reverse
1	1	spin left	spin right
1	1	reverse	spin right
1	2	stopped	spin left
1	2	forward	spin left
1	2	spin right	spin left
1	2	spin left	reverse
1	2	reverse	spin left
1	3	stopped	reverse
1	3	forward	reverse
1	3	spin right	reverse
1	3	spin left	reverse
1	3	reverse	reverse

Code

```
//Sensors
#define EYE SENSOR_2
#define LBUMP SENSOR_1
#define RBUMP SENSOR_3

//Motors
#define LEFT OUT_A
#define RIGHT OUT_B

//Events
#define LEFT_HIT 0
#define RIGHT_HIT 2
#define LEFT_UNHIT 1
#define RIGHT_UNHIT 3
#define LIGHT 4
#define NOT_LIGHT 5

// state variables
int saw_light = 0;    // 0 for no light or 1 for light
```

```
int hit_bumper=0;   // 0 for no bumpers hit, 1 for left, 2 for right, 3 for both
int direction=1;        // bits 4-5 are left motor, 6-7 are right motor
                        // 0 off, 1 forward, 2 reverse
                        // 6 is spin right
                        // 9 is spinning left
                        // 5 is forward
                        // 10 is reverse

sub spinLeft() { Fwd(RIGHT); Rev(LEFT); direction = 9; On(LEFT+RIGHT); }

sub function spinRight() { Fwd(LEFT); Rev(RIGHT); direction = 6; On(LEFT+RIGHT); }

task main() {
        int i;

        // set sensors
        SetSensor(EYE, SENSOR_LIGHT);
        SetSensor(LBUMP, SENSOR_TOUCH);
        SetSensor(RBUMP, SENSOR_TOUCH);

        // set events
        SetEvent(LEFT_HIT,SENSOR_1,EVENT_TYPE_PRESSED);
        SetEvent(LEFT_UNHIT,SENSOR_1,EVENT_TYPE_RELEASED);
        SetEvent(RIGHT_HIT, SENSOR_3, EVENT_TYPE_PRESSED);
        SetEvent(RIGHT_UNHIT, SENSOR_3, EVENT_TYPE_RELEASED);
        SetEvent(LIGHT, SENSOR_2, EVENT_TYPE_HIGH);
        SetEvent(NOT_LIGHT, SENSOR_2, EVENT_TYPE_LOW);

        start looking;
        start feeling_left;
        start feeling_right;

        // start rotating
        spinRight();

        while(true)
        {
                if (hit_bumper == 3) {                          // blocked – back up
                        OnRev(LEFT+RIGHT);
                } else if (hit_bumper == 1 && direction == 6) { // oops – back up
                        OnRev(LEFT+RIGHT);
                } else if (hit_bumper == 2 && direction == 9) { // oops – back up
                        OnRev(LEFT+RIGHT);
                } else if (hit_bumper == 1) {           // left bumper – right spin
                        spinRight();
                } else if (hit_bumper === 2) {          // right bumper – left spin
                        spinLeft();
                }
                else if (saw_light == 1 && hit_bumper == 0) {    // see light & not blocked
                        OnFwd(LEFT+RIGHT);
                } else if (saw_light == 0 && hit_bumper == 0) {    // no information - spin
```

```
                        if (direction != 6 && direction != 9) {         // not already spinning
                                i = Random(1,2);
                                if (i == 1) { spinRight(); }
                                else { spinLeft(); }
                        }
                }
        }
}

task lookForLight()
{
        monitor(EVENT_MASK(LIGHT)) { while(true); }
        catch  { saw_light = 1; }
}

task lookForDark()
{
        monitor(EVENT_MASK(NOLIGHT)) { while(true); }
        catch  { saw_light = 0; }
}


task hitLeft()
{
        monitor(EVENT_MASK(LEFT_HIT)) { while(true); }
        catch  { hit_bumper += 1; }
}

task hitRight()
{
        monitor(EVENT_MASK(RIGHT_HIT)) { while(true); }
        catch  { hit_bumper += 2; }
}

task releaseLeft()
{
        monitor(EVENT_MASK(LEFT_UNHIT)) { while(true); }
        catch  { hit_bumper -= 1; }
}

task releaseRight()
{
        monitor(EVENT_MASK(RIGHT_UNHIT)) { while(true); }
        catch  { hit_bumper -= 2; }
}
```

Advantages

1. Should respond as soon as anything happens, i.e., as soon as sees light or bumper is hit.
2. Can respond to a combination of events happening close together.

3. If decision-making is wrong, it should be clear where to change it?

Subroutines versus inline functions

Subroutines do not have arguments
Cannot call another subroutine.
Inline functions are copied into the code in place.

Function arguments

int: call by value (all Java arguments are "by value") – the value is copied into a temp, and that's what is used inside the function

const int: must be a constant (no variables allowed)

int &: call by reference. The value is not copied – this saves on memory, and now the function can change the value. This is like (but not really the same as) when you pass a reference to an object and call a method on the object that changes an instance variable of the object. It's the same object, but you can change something about it.

const int &: call by reference, but it must be a constant – more efficient than const int; anything can be passed this way, but it cannot be changed in the function. Anomaly when x == x for x declared const int &

Expressions

See programmer's manual section 2.4: conditions are also expressions, and evaluate to integers. 0 is false, non-zero is true

Appendix

```
switch(hit_bumper) {
case 3:                 // both bumpers hit
        OnRev(LEFT+RIGHT);
        break;
case 2:                 // right bumper hit
        if (direction == 9) OnRev(LEFT+RIGHT);
        else spinLeft();
        break;
case 1:                 // left bumper hit
        if (direction == 6) OnRev(LEFT+RIGHT);
        else spinRight();
        break;
case 0:                 // no bumpers hit
        switch(saw_light) {
        case 0:         // no light seen
                if (direction != 6 && direction != 9) {
                        int i = Random(1,2);
                        if (i == 1) spinLeft();
                        else spinRight();
```

```
                }
                break;
        case 1:         // light seen
                OnFwd(LEFT+RIGHT);
                break;
        default:        // oops
                PlaySound(TONE_DOWN);
                break;
        }
        break;
default:                // oops
        PlaySound(TONE_DOWN);
        Break;
}
```