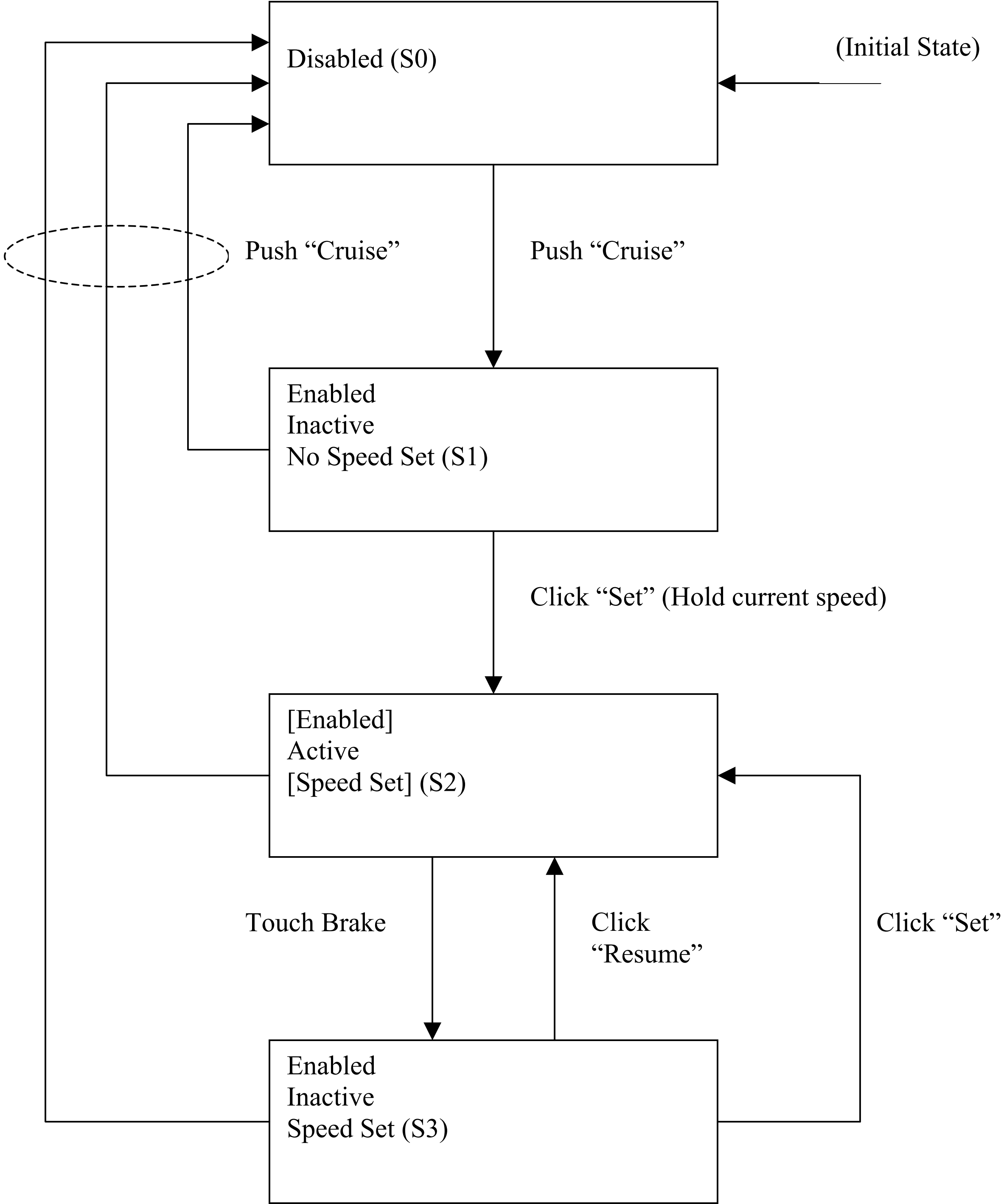
Cruise Control as a Finite State Automaton with Data
(Revised 12 April 2004)

Our new car has a Cruise Control. It works like this:



The preceding diagram depicts a finite state machine with four states. The depiction of four states actually (and beneficially) oversimplifies the machine, as we shall see presently.

The input alphabet for this machine consists of four “letters”:

Push “Cruise”
Click “Set”
Click “Resume”
Touch Brake

Not all inputs are shown for all states. The convention is that if an input is not shown for a state, receipt of that input does not cause a change of state. For example, if you touch the brake while cruise control is disabled, cruise control remains disabled. This convention reduces clutter. Also, no accepting states are shown, since accepting states are not relevant to this example.

In realistic programming, the state of this machine would be defined by the values of three Boolean variables:

```
boolean bEnabled, bActive, bSpeedSet;
```

Now there are eight possible combinations of values for these three Boolean variables, but only four states. Here is the correspondence. Certain combinations (well, one combination) are not allowed. It is the responsibility of the software to ensure that the disallowed combination never occurs. In order to determine the current state, the program must inspect the values of all three variables.

Enabled	Active	Speed Set	State
T	T	T	S2
T	T	F	disallowed
T	F	T	S3
T	F	F	S1
F	T	T	S0
F	T	F	S0
F	F	T	S0
F	F	F	S0

Now we come to data. An essential component of the machine state is the value of the speed setting. There is an integer variable, `iSpeedSetting`, that plays a key role. Let’s assume that the electronics in the car can measure speed with a resolution of 1 mph. Let’s also assume that speed settings less than 40 mph are not allowed by the cruise control processor. (This is quite realistic; it is certainly true of my car.) Let’s also assume that speed settings exceeding 100 mph are also not allowed. (I don’t know whether this is true of my car, and I don’t intend to find out.) Then we have an `int` variable that is allowed to assume values in the range 40 to 100. There are 61 possible values.

Lets declare

```
int iSpeedSetting;
```

There are 61 allowed values of `iSpeedSetting`. So with the four states shown in the figure, and with the 61 allowed values of `iSpeedSetting`, we actually have a finite state automaton with 4*61 = 244 states.

Now organizing a computer program as a finite state automaton with 244 states is impractical except for very complex programs. But it is entirely practical to organize a program as a finite state automaton with four states, supplemented by an `int` data item (or even a lot more data)!

Therein lies the power of the finite-state machine concept for organizing a program. At a high level, define a manageable number of states that the program can be in between inputs, and define the necessary data structures to supplement the state selection. Then the main control loop of the program is completely defined by the transition diagram of the finite-state machine, and the rules for modifying the data (and producing output) in each case.