U.S. DEPARTMENT OF
ENERGY

PNNL-22595

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

The State Machine Library

DES-0014

Revision 2

Charlie Hubbard

May 2014

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by **Battelle** *Since 1965*

The State Machine Library

Charlie Hubbard

DES-0014
Revision 2
May 2014

DISCLAIMER

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under
Contract DE-AC05-76RL01830

Printed in the United States of America

Available to DOE and DOE contractors from the Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831-0062
ph: (865) 576-8401
fax: (865) 576-5728
email: reports@adonis.osti.gov

Available to the public from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161
ph: (800) 553-6847
fax: (703) 605-6900
email: orders@ntis.fedworld.gov
online ordering: http://www.ntis.gov/ordering.htm

Contents

1 Introduction

This document describes the *State Machine Library*, a collection of code designed to make it easy to directly implement processes modeled as state machines in C++ code. The State Machine Library is defined and implemented by the two files *statemachine.h* and *statemachine.cpp*. These two files define and implement the `StateMachine` class, which provides a framework for implementing state machines. The primary documentation for the `StateMachine` class is its associated Doxygen-generated HTML files. The Doxygen documentation should be consulted for a detailed description of the class' methods and attributes. The document you are reading now is supplemental, and is intended to provide deeper background for the state machine implementation. In case of any conflict between these two documents, the Doxygen documentation should be considered correct.

1.1 Why State Machines?

To implement automatic control of a system, the various phases of the system's operation are broken into a series of steps that must be performed in a specific sequence to achieve the desired result. In the high-level design, it is useful to view these various steps as *states* in a *state machine*.

State machines define a process as a collection of well-defined, discrete states together with a set of well-defined transitions that indicate how the process is allowed to move from one state to the next.

The state machine model for defining a process is very useful. It is easy for the scientists that design the control process to describe it in terms of a state machine; it is easy for operators of the system to understand the state machine, and, with a little help from the State Machine Library, it is easy to convert the state machine description directly into C++ control software.

1.2 Our State Machine Model

State machines can be organized in different ways. The model we use puts all the emphasis on the states rather than the transitions between them. Each state defined in the state machine has associated with it four things:

1. An optional list of actions that must be performed when entering the state.
2. An optional list of actions that must be performed when exiting the state.
3. A set of conditions that indicate when one state must transition to another state (this is not optional).
4. Occasionally, a well-defined piece of work must be performed while in the state. We call this *internal processing* to distinguish it from work that is performed while entering

or exiting the state. Frequently, the amount of internal processing required can be reduced or eliminated by adding additional states. In general, eliminating internal processing is encouraged. However, for practical reasons, that is not always possible.

2 The State Machine Library

2.1 The StateMachine Class

The heart of the State Machine Library is the `StateMachine` class. This is a base class. It provides the basic functionality to implement a state machine, but it does not define any states or transitions itself. Users of the class are expected to derive a new class from `StateMachine` to implement their specific state machine. A simplified class definition showing only the public and protected members is shown below. This is intended only as a reference for discussion. For the most up-to-date, detailed documentation for the class, see its associated Doxygen-generated HTML documents and the source code files.

```
#define ADD_STATE(name, entryFunc, exitFunc, selectorFunc) \
    AddState(name, static_cast<EntryFunc>(entryFunc),      \
    static_cast<ExitFunc>(exitFunc),                       \
    static_cast<SelectorFunc>(selectorFunc))

struct StateID {
   string name;
   int    number;

   StateID();
   string Serialize() const;
   void   Deserialize(const string &s);
};
typedef vector<StateID> StateIDVector;

class StateMachine {
protected:
   typedef void (StateMachine::*EntryFunc)();
   typedef void (StateMachine::*ExitFunc)();
   typedef string (StateMachine::*SelectorFunc)();

   void AddState(string name, EntryFunc entry, ExitFunc exit, SelectorFunc selector);
   virtual void StateChanged();

public:
   StateMachine(string startState, bool enableJumps = false);
   virtual       ~StateMachine();
   bool           Step();
   void           JumpToState(string name);
   StateID        GetCurrentStateID() const;
   StateID        GetPreviousStateID() const;
   int            GetNumStates();
   StateIDVector  GetAllStateIDs();
};
```

Within `StateMachine` derived classes, individual states are identified by unique text names that are assigned by the designer as appropriate. In addition to an identifying name, each

state is associated with a mandatory *selector* function, and optional *entry* and *exit* functions. These functions are defined as member functions in the derived class, and they must follow the "typedef'ed" prototypes shown above. That is to say, entry and exit functions return `void` and take no parameters. Selector functions return type `std::string` and also take no parameters.

The string returned by a selector function is the name of the state that should be run on the next iteration of the state machine. It is via this mechanism that the designer implements state transitions. Note that transitions are never declared explicitly (that is to say, there is nothing like a `Transition` class, for example); instead transitions are established by which state names a particular state's selector function is allowed to return. Terminal states are defined by providing selectors that, at least on occasion, return the empty string (""). If any state returns an empty string as the next state to be transitioned to, that is the signal that a terminal condition has been met, and the state machine *terminates*. Here this just means the call to `Step()` that resulted in the terminal condition will return `true` (see the description of the `Step()` method below). It is up to the application to decide how to react.

Each state added to the state machine is automatically assigned a unique integer identifier. These are assigned sequentially starting with a value of zero. These numeric state identifiers are assigned and tracked by the base `StateMachine` class, but they are not actually *used* by the base class for anything. They exist because many real-world applications benefit from a numeric state identifier in addition to the unique text names that are assigned to each state. Most typically they're used for state-of-health logging[1].

New states are added to the state machine by calling the `ADD_STATE()` macro. As you can see from its definition, the macro simply associates a state name with the entry, exit and selector functions the state uses. It is often the case that a state does not use an entry or exit function. In that case, the user can simply pass `NULL` in their place. Typically, all states are added to the state machine in the derived class' constructor.

The `StateMachine` class provides several methods for operating the state machine, the most important of which is the `Step()` method. Each call to `Step()` performs one state machine iteration. Specifically, it calls the selector function of the current state and examines its return value. Recall that the return value is the name of the state the state machine will be in on the next iteration. If the returned name is the same as the current state, then no other actions are performed, and the `Step()` method exits. However, if the returned name is different than the current state (indicating a state transition from the current state to the state returned by current state's selector function), then the following sequence of events occurs:

1. The *current state* state now becomes the *previous state*. The state returned by the selector function becomes the new current state.

2. The exit function of the previous state is run. This function should perform any actions

[1]The ability to log the current state ID as a state-of-health parameter using the PNNL state-of-health server was the primary and original motivation for assigning numeric state IDs. See design documents ***DES-0006, The State-of-Health Server*** and ***DES-0072, The Number Server*** for more details on this specific application of the numeric state IDs.

the designer wants to occur *every time* the state is exited.

3. The `StateChanged()` method is called. `StateChanged()` is a virtual method that can be overridden to provide any actions a designer may want to occur when control is transferred from one state to another. This is different than a state's exit function, because an exit function is specific to the state it is associated with. The `StateChanged()` method runs the same code for *any* state change. In practice, this is typically used as a convenient place to record the state change event to a log file.
4. The entry function of the current state is run. This function should perform any actions the designer wants to occur *every time* the state is entered.

Other methods available to the state machine designer include `JumpToState()`, `GetCurrentState()`, `GetPreviousState()`, `GetNumStates()`, and `GetAllStateIDs()`. Most of these are self-explanatory, but the `JumpToState()` deserves some more explanation.

Typically any process defined by a state machine is only valid if the state machine is allowed to proceed from its start state to a terminal state following only defined transitions between the states. Allowing a user to force arbitrary transitions from any state to any state is usually a recipe for disaster. However, during development and testing, this can also be an extremely useful ability so long as the state machine application is designed to handle arbitrary transitions or at least the operator understands the consequences. For that reason, the base class provides the `JumpToState()` method. Calling `JumpToState()` will immediately force a transition from the current state to the target state (including calling the appropriate entry and exit functions). However, because this action is risky, by default the `JumpToState()` method is disabled. It can be enabled at the user's discretion by passing a flag to the `StateMachine` class constructor.

2.2 State Machine Example

To better understand how states with the above attributes can be used to implement control processes, we'll look at an (admittedly contrived) example. Assume we need a state machine to control a device that can sort a mixture of black and white marbles into two buckets, one containing only black marbles and the other containing only white marbles. One possible state machine that can handle that control task is shown below.

2.2.1 Marble Sorting State Machine Definition

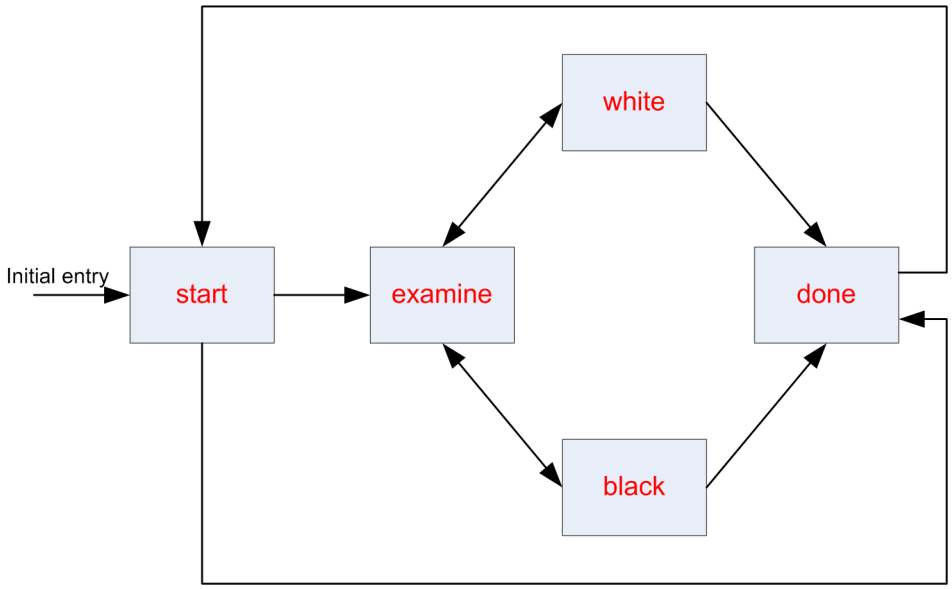

Figure 1: An example state machine for sorting marbles

Look at the action lists and the transition logic for each state to understand how it works.

Table 1: State machine control actions for the marble sorting state machine

State	Entry Actions	Exit Actions	Internal Processing	Transition Logic
start	Turn on the "system is running" light	None	None	If there are more marbles to be processed, jump to **examine**, otherwise jump to **done**.
examine	Get a new marble	None	None	If the marble is white, jump to **white**, else jump to **black**.
white	Put the marble in the white bucket	None	None	If there are more marbles to be processed, jump to **examine**, otherwise jump to **done**.
black	Put the marble in the black bucket	None	None	If there are more marbles to be processed, jump to **examine**, otherwise jump to **done**.
done	Turn off the "system is running" light.	None	None	If the operator presses the start button, jump to **start**.

Notice that in the above description, we never specify any exit actions for any of the states. Whether or not to use entry actions, exit actions or both is a design decision. The marble sorting process could be represented with a state machine that used only exit actions (although it may not have the same number of states, transitions or transition logic). For this example, we just chose to do everything on the entry actions. Other, equally valid, solutions are possible. A state machine could be developed for this process that used entry actions for some states and exit actions for other states or provided states that had both entry and exit actions. The important point is that work only happens when entering or leaving a state. How that work is divided between these two options is left up to the designer.

Also notice that none of the states defined incorporate any internal processing logic. In general, that is good design. When possible, internal processing should not be used. However, incorporating internal logic can substantially reduce the total number of states in a state machine. Whether or not that is a good idea is a decision that needs to be made by the designer.

2.2.2 Marble Sorting State Machine Implementation

In this section, we'll look at a quick implementation of the marble-sorting state machine using C++-like pseudocode just to see how the `StateMachine` class could be employed to create this state machine.

To begin, we need to derive a new class from the `StateMachine` class. Among other things, we need to add methods to this class to act as the entry, exit and selector functions for all the states our state machine will use. One possible derived class might look like the following.

```
class MarbleMachine : public StateMachine {
public:
   MarbleMachine();

protected:
   void   StartEntry();
   string StartSelector();

   void   ExamineEntry();
   string ExamineSelector();

   void   WhiteEntry();
   string WhiteSelector();

   void   BlackEntry();
   string BlackSelector();

   void   DoneEntry();
   string DoneSelector();

   virtual void StateChanged(string from, string to);

   Marble marble;
};
```

As you can see, the derived class is pretty simple. It contains a simplified constructor, and

entry and selector functions for the marble-sorting state machine's five states. We've also overridden the `StateChanged()` method, just to provide an example of how that might be used.

We've defined one class variable, called `marble` (an instance of the fictitious `Marble` class), that we'll use to store information on the current marble under examination. In this case, it is necessary to use a class variable for `marble`, because the data in `marble` needs to remain valid across calls to `ExamineEntry()` and `ExamineSelector()`. When designing state machines, you'll find that it is frequently the case that certain bits of information need to be retained across multiple function calls.

The next step is to implement the constructor. It is here where we'll actually add new states to the state machine.

```
MarbleMachine::MarbleMachine() : StateMachine("done")
{
   //    state name              entry function          exit func          selector function
   //==================   ==========================   =========   ================================
   ADD_STATE("start",     &MarbleMachine::StartEntry,       NULL,    &MarbleMachine::StartSelector);
   ADD_STATE("examine",   &MarbleMachine::ExamineEntry,     NULL,    &MarbleMachine::ExamineSelector);
   ADD_STATE("white",     &MarbleMachine::WhiteEntry,       NULL,    &MarbleMachine::WhiteSelector);
   ADD_STATE("black",     &MarbleMachine::BlackEntry,       NULL,    &MarbleMachine::BlackSelector);
   ADD_STATE("done",      &MarbleMachine::DoneEntry,        NULL,    &MarbleMachine::DoneSelector);
}
```

The constructor is simple. We call the base constructor with one parameter, `"done,"` which means that the state machine will begin in the `done` state. The second parameter to the base constructor we omitted, which means the base class `JumpToState()` method will be disabled. In the body of the constructor we do nothing but use the `ADD_STATE()` macro to create five new states. Each state is assigned a name and associated with whatever entry, exit and selector functions it uses. In this example, no states use exit functions, so `NULL` is passed instead. Note that it is also here in these calls to `ADD_STATE()` where the state integer IDs are automatically assigned. These are assigned in the order that states are added, so, in this case, "start" will be given an ID of zero, "examine" will be given an ID of one, and so on.

Next we need to implement the entry, exit and selector functions defined for each of the states. For brevity, we won't show the implementations for all of them. They are all similar. Refer back to Table 1 on page 5 to compare the actions and the transition logic listed there with the following state functions.

```
void MarbleMachine::DoneEntry()
{
   SetRunningLight(false);  // turn the "system is running" light off
}

string MarbleMachine::DoneSelector()
{
   if (StartButtonPressed()) {
     return("start");
   }
```

```
   return("done");
}

...

void MarbleMachine::ExamineEntry()
{
   marble = GetNewMarble();
}

string MarbleMachine::ExamineSelector()
{
   if (marble.IsWhite()) {
      return("white");
   }
   return("black");
}

...
```

Hopefully, it is clear from examining the two selector functions above how transitions are implemented in our state machine model. Allowed transitions out of a particular state are completely determined by the possible return values of the state's associated selector function.

Finally, we'll show one possible (admittedly trivial) use of the base class' `StateChanged()` method. Here we just redefine it to print out the state transition information to the terminal.

```
void MarbleMachine::StateChanged()
{
   StateID  current;
   StateID  previous;

   current  = GetCurrentStateID();
   previous = GetPreviousStateID();

   cout << "State " << previous.name << " has transitioned to the " << current.name << " state." << endl;
}
```

With our derived state machine class now complete, the final question is, "How is the state machine actually run?" The state machine is run by making calls to the class' `Step()` method until `Step()` returns `true`, indicating it has reached a terminal state[2]. Each call to `Step()` runs the current state's selector function, and if a transition to a new state is initiated, it runs the appropriate entry and exit functions. There is no provision in the base class to automatically step the state machine. It is up to the user to decide how this is done. In practice, we typically set up a timer to step the state machine at regular intervals.

[2]In practice, the *terminal state* feature has not proven to be particularly useful. In all real-world applications we've implemented using this library, we define no true terminal states at all (that is, no selector functions ever return the empty string). Instead, we may have one or more *idle* states, whose selector functions always return to themselves. In this way, the state machine stays alive and the application can begin new process sequences by using `JumpToState()` to jump to the appropriate start state of the desired sequence.

3 State Machine Design Considerations

This section discusses some of the issues involved when translating a real-world control process into a suitable state machine, and provides some guidelines for creating workable state machine designs.

The purpose of the state machine is to represent the system's process flow as a series of well-defined, discreet states and to provide a logical path through those states via well-defined transitions. So the first problem is how to split the system processing steps into discreet states. Let's start by taking a look at the types of processing steps that might need to be performed. The following is a snippet of a hypothetical processing sequence.

```
...
- Add liquid nitrogen to the main trap dewar.
- Wait for trap temperature to drop to a certain value
- Open a flow path from the sample source, to the main trap and out the exhaust
- Set the sample flow rate to a certain value
...
```

As a first attempt, we could continue to list all the steps for the full processing sequence and then decide to make each step a separate state. That may be a workable solution, but there are hundreds of steps involved, and taking this approach quickly generates a very large, difficult-to-manage state machine. Also, state-machine purists might point out that even breaking down the processing steps to this level of detail is not enough, because we've actually hidden a huge number of discreet sub-steps within each of our states. For instance, the step that begins "Open a flow path..." actually breaks down more like this...

```
- Open valve V101A
- Open valve V101B
- Open valve V101C
- Open valve V101D
- Open valve V201A
- Open valve V201B
- Open valve V201C
- Open valve V201D
- Open valve V791A
- Open valve V791B
- Open valve V791C
- Open valve V791D
...
```

When represented at this extreme level of detail, the entire control process consists of thousands of separate events. Representing each of these events as a separate state in our state machine, although technically possible, is not particularly useful. It results in an enormous number of states, and saying something like, "The system is in the 'Open Valve V791C' state" really gives no indication of what phase of the overall process the system is currently in.

So we reject a state machine that describes the system operation at this level of detail. We want something simpler. The following proposed state machine goes to the other extreme.

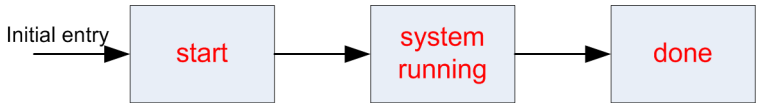

Figure 2: An example of a state machine solution with far too little detail.

Again, this isn't particularly useful. Saying, "The system is running" doesn't really tell the operator very much about what the system is actually doing at any particular moment.

Assume the purpose of our hypothetical control system is to take a sample of material, extract various products of interest from it, analyze those products, and then archive the products. This description breaks the process into a few separate phases, and we could build a state machine that captures those phases as follows.

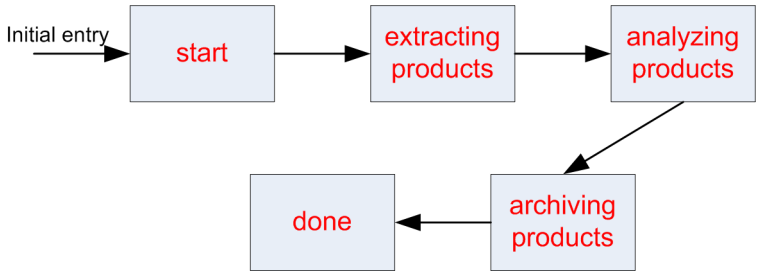

Figure 3: An example of an improved state machine solution with somewhat more detail.

This is somewhat better. By looking at the current state, the operator can now get some idea of what the system is currently doing. But the process view is still pretty coarse-grained at this point. A little more information would be useful. As can be seen, it is possible to create any number of viable state machines to describe the control process. How do we pick the *right* one? There are no hard-and-fast rules for this. Deciding how to split the processing steps into states is a decision that has to be made by the designer. However, we can develop certain guidelines to assist. Specifically...

1. In our state machine model, ideally all actions occur when entering a new state or exiting the current state (or both). Other than checking to see if the transition conditions have been met, we would prefer that there be no internal processing performed inside a state. We should design our states with this in mind. However, internal processing (that is processing performed by the selector function that is not related to checking transition conditions) is not forbidden. Using some small amount of internal processing in some states can greatly reduce the total number of states required to implement a process.

2. The state machine should provide enough detail that the operator can get a good idea of what phase of operation the system is currently in just by glancing at the current state, but it should not be overly detailed. Too much detail is one reason the designer might decide to reduce the total number of states by using internal processing.

3. Transitions from one state to the next should be governed by a small number of simple conditions. Fewer conditions are better. One way this can be accomplished is to make each state perform only a few actions, all of which are related to a particular phase of operation (regeneration, collection, purification, analysis, archiving, etc.).

4. More states provide a finer-grained look at what part of the process the system is currently performing, but we've already seen that too much detail is not helpful. As a rule of thumb, try to limit the total number of states to 20 or 30. Of course the exact number will depend on the nature of the process.

Even with these goals in mind, how the control process is split into separate states is largely a judgment call, but it does give us some guidelines to help us reject poor solutions. We conclude with the following diagram, which is provided as an example of an appropriately detailed state-machine process for a simple sample-processing and analysis system.

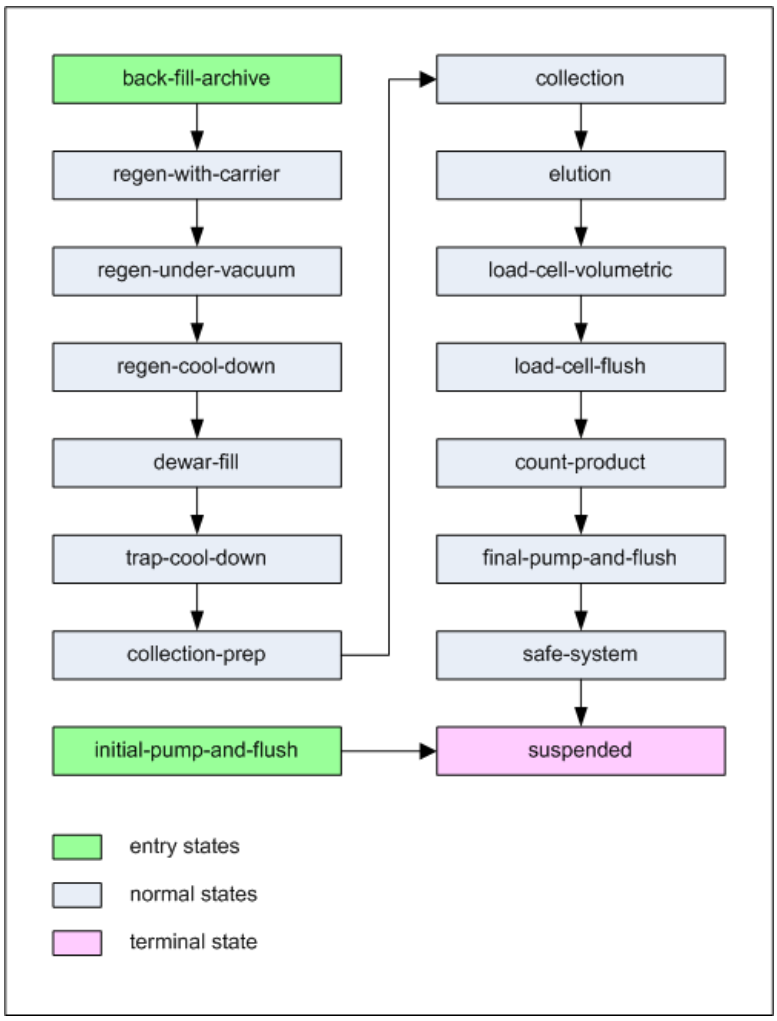

Figure 4: A process state machine with a reasonable amount of detail

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by **Battelle** *Since 1965*

U.S. DEPARTMENT OF
ENERGY

902 Battelle Boulevard
P.O. Box 999
Richland, WA 99352
1-888-375-PNNL (7665)
www.pnnl.gov