SMC Programmer's Manual

April 2, 2016
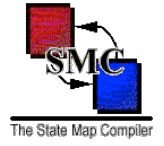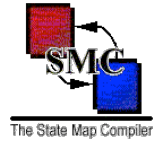
Table of Contents

Preface

This manual describes how to use the state machine compiler. Hopefully, you will find this tool as useful as I have. State machines are a natural way to describe object behavior but there has not been an easy way to mate state machines with objects. This compiler allows you to add a state machine to just about any class you desire (more about that later.)

I encourage you to first look at the annotated SMC code in section 1. This code demonstrates SMC's powerful simplicity. The state machine code is simple, meaning you can easily learn it. Its simplicity also allows you to readily build powerful, robust finite state machines.

For those of you who hang around comp.lang.c++ or comp.object, you may have read some of Bob Martin's articles. You may have even read about his state machine compiler and gotten a copy of it. You may then notice a striking similarity between my compiler and his. There is a reason for that: this compiler was derived from Bob Martin's original state machine compiler.

While an employee of Clear Communications Corporation (renamed Clear and now defunct), he developed state machine classes and later a compiler to automatically generate those classes. About this time, I came to work for the same company and was intrigued by what Bob had done. Some six months later, Bob struck out on his own to form Object Mentor Associates and I was left to maintain the state machine compiler. I have added many features to the original system (arguments, default transitions, push and pop transitions, transition guards, a more YACC-like language structure, etc.)

But no matter how much I have added, the state machine compiler will always be Bob Martin's invention. I'd like to think that I made a good thing better.

Section 1: .sm File Layout

Your Task Class

SMC generates finite state machines for objects - not processes or applications but for an individual object. If you have objects that receive asynchronous callbacks and how objects respond to those callbacks are based on the object state, then SMC provides a powerful solution.

(Note: this example is based on simple Java and is easily translated to other languages. Also, this example assumes that you know object-oriented programming and finite state machines (FSMs).)

In this example you are developing a Task object as part of your SuperCron! product:
(Note: The ellipsis (...) code will be filled in as we go along.)

```java
package com.acme.supercron;

public final class Task
    implements TaskEventListener, TimerEventListener
{
    public Task() {
        // Object initialization.
        ...
    }

    // TaskEventListener Interface Implemenation.

    // Time for the incomplete task to continue its work for the
    // specified time slice.
    public void start(long timeSlice) { ... }

    // Called when a running, incomplete task should suspend
    // running even though its time slice is not expired.
    // Note: the task's running is also suspended when the time
    // slice expires.
    public void suspend() { ... }

    // Called when an incomplete task is blocked. Blocked tasks
    // are able to continue running when unblocked.
```

```
    public void block() { ... }

    // Called when a blocked task is unblocked and allowed
    // to continue running.
    public void unblock() { ... }

    // Called when an incomplete task is permanently stopped.
    // Stopped tasks are then deleted.
    public void stop() { ... }

    // Called when the task is deleted. Tasks are deleted when
    // either 1) the task has completed running and is now
    // stopped or 2) when the system is shutting down and all
    // are to terminate immediately.
    public void delete() { ... }

    // TimerEventListener Interface Implementation.

    // Called when the time slice timer expires. If running,
    // the task is suspended.
    public void handleTimeout(TimerEvent event) { ... }


    // Remainder of class definition.
    ...
}
```
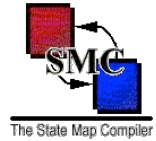
How the Task class should respond to the start, suspend, etc. method calls depends on what the Task is currently doing - that is, it depends on the Task's state.

The Task Finite State Machine

The Task Finite State Machine (FSM) diagram is:
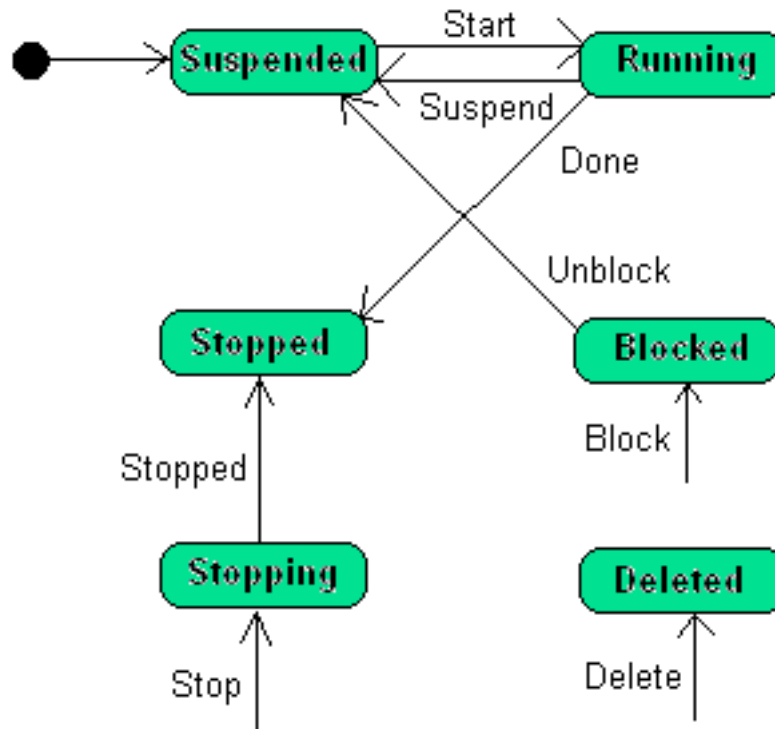
The Task's states are:

- Running: the task is actively doing work. The task is allowed to run for a specified time limit.
- Suspended: the task is waiting to run again since it is not yet completed.
- Stopped: the task has either completed running or externally stopped.
- Blocked: the uncompleted task is externally prevented from running again. It will stay in this state until either stopped or unblocked.
- Stopping: the task is cleaning up allocated resources before entering the stop state.
- Deleted: the task is completely stopped and all associated resources returned. The task may now be safely deleted.
  This is the FSM end state.

Some notes on this FSM:

- The Task object starts in the Suspended state.

- The transitions match the TaskEventListener interface's methods.
- The Stopped state is reached when either the Running task completes or is externally stopped.
- The Stop, Block and Delete transitions do not start in any specified state. More on that in coding up the FSM.

Now the problem is: how to take this FSM and put it into your code? The first step in that is encoding the FSM in the SMC language.

Creating an SMC .sm file

The .sm listing below is a skeleton with no states or transitions defined. It contains only the following features:

- A verbatim code section which is copied verbatim into the generated source code file. In this case the verbatim code is the boilerplate copyright comments. This section is delimited by the `%{ %}` pair.
- The `%class` keyword which specifies the application class to which this FSM is associated: Task.
- The `%package` keyword which specifies to which class package this FSM belongs. This is the same package as the Task class.
- The `%fsmclass` keyword specifies the generated finite state machine class name. If `%fsmclass` is not specified, then the finite state machine class name would default to `TaskContext`. This keyword is not required.
- The `%fsmfile` keyword specifies the generated finite state machine class file name. The appropriate suffix for the given programming language is appended to this file name to form the complete file name. This suffix may be changed from its default using the `-suffix` and `-hsuffix` command line parameters. See this table for the default file name and suffix settings for each supported target language.
- The `%access` keyword is used to specify the generated class' accessibility level (this works only when generating Java and C# code). In this case the FSM can only be accessed within the `com.acme.supercron` package.
- The `%start` keyword specifies the FSM's start state. For the Task FSM it is the Suspended state.
- The `%map` keyword is the FSM's name.

Name the source code file `TaskFSM.sm` because both the `%fsmclass` and `%fsmfile` directives specify the finite state machine class name as TaskFSM. The `.sm` suffix is required.

(Note: the `%fsmclass` directive was added to SMC version 6.0.1 and `%fsmfile` was added to version 6.6.0.)

```
%{
//
// Copyright (c) 2005 Acme, Inc.
// All rights reserved.
//
// Acme - a name you can trust!
//
// Author: Wil E. Coyote (Hungericus Vulgarus)
//
%}

// This FSM works for the Task class only and only the Task
```
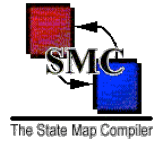
```
// class may instantiate it.

%class Task
%package com.acme.supercron
%fsmclass TaskFSM
%fsmfile TaskFSM
%access package

// A %map name cannot be the same as the FSM class name.
%start TaskMap::Suspended
%map TaskMap
%%
    ...
%%
```

Defining FSM States

The state definitions are placed inside the `%map TaskFSM %% ... %%` delimiter:

```
%{
//
// Copyright (c) 2005 Acme, Inc.
// All rights reserved.
//
// Acme - a name you can trust!
//
// Author: Wil E. Coyote (Hungericus Vulgarus)
//
%}

// This FSM works for the Task class only and only the Task
// class may instantiate it.
%class Task
%package com.acme.supercron
%fsmclass TaskFSM
%fsmfile TaskFSM
%access package

// A %map name cannot be the same as the FSM class name.
%start TaskMap::Suspended
%map TaskMap
%%
Suspended { ... }
Running { ... }

// Wait here to be either unblocked, stopped or deleted.
Blocked { ... }
Stopping { ... }
Stopped { ... }
Deleted { ... }
...
%%
```

Like all C-syntax languages, the opening, closing braces are not needed if there is only one expression inside the braces. But like all C-syntax languages, it is *good* practice to always use them. If you don't follow this rule now, you will after you spend two days tracking down a bug due entirely to you not following this rule.

Notice the ellipsis before the closing `%%`? There is one more state to define even though the diagram's six states are declared. There is an implicit state. Remember how the Stop, Block and Delete transitions have no start state?

Defining FSM Transitions

A transition definition consists of four parts:

1. The transition name.
2. An optional transition guard (not used in Task FSM).
3. The transition end state.
4. The transition actions.

Only the "standard" transitions are defined for now. The Stop, Block and Delete transitions will be covered here.

```
%{
//
// Copyright (c) 2005 Acme, Inc.
// All rights reserved.
//
// Acme - a name you can trust!
//
// Author: Wil E. Coyote (Hungericus Vulgarus)
//
%}

// This FSM works for the Task class only and only the Task
// class may instantiate it.
%class Task
%package com.acme.supercron
%fsmclass TaskFSM
%fsmfile TaskFSM
%access package

// A %map name cannot be the same as the FSM class name.
%start TaskMap::Suspended
%map TaskMap
%%
Suspended
{
    // Time to do more work.
    // The timeslice duration is passed in as a transition
    // argument.
    Start(timeslice: long) // Transition
        Running // End state
        {
            ... // Actions go here
        }
}
```
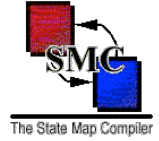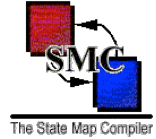
```
Running
{
    // Wait for another time slice.
    Suspend
        Suspended
        { ... }

    // Task has completed.
    Done
        Stopped
        { ... }
}

// Wait here to be either unblocked, stopped or deleted.
Blocked
{
    // The task may continue working now.
    Unblock
        Suspended
        { ... }
}

Stopping
{
    // The task is now stopped.
    Stopped
        Stopped
        { ... }
}

Stopped { ... }
Deleted { ... }
...
%%
```

Defining FSM Transition Actions

Transition actions are the first coupling between the FSM and the application class `Task`. Actions are `Task` methods. These method must have the following attributes:

- Be accessible to the FSM. This means at least public methods or, if in the same package, then package methods.
- Have a `void` return type. If the method does return a value, the FSM ignores it.

SMC places no syntax limitations on transition arguments except they are enclosed in parens "()" and are comma-separated. Go here for more information on transition actions.

```
%{
//
// Copyright (c) 2005 Acme, Inc.
// All rights reserved.
//
// Acme - a name you can trust!
//
// Author: Wil E. Coyote (Hungericus Vulgarus)
//
%}

// This FSM works for the Task class only and only the Task
// class may instantiate it.
%class Task
%package com.acme.supercron
%fsmclass TaskFSM
%fsmfile TaskFSM
%access package

// A %map name cannot be the same as the FSM class name.
%start TaskMap::Suspended
%map TaskMap
%%
Suspended
{
    // Time to do more work.
    // The timeslice duration is passed in as a transition
    // argument.
    Start(timeslice: long) // Transition
        Running // End state
        {
            continueTask();
            startSliceTimer(timeslice);
        }
}
```
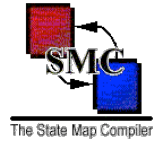
```
Running
{
    // Wait for another time slice.
    Suspend
        Suspended
        {
            stopSliceTimer();
            suspendTask();
        }

    // Task has completed.
    Done
        Stopped
        {
            stopSliceTimer();
            releaseResources();
        }
}

// Wait here to be either unblocked, stopped or deleted.
Blocked
{
    // The task may continue working now.
    Unblock
        Suspended
        {}
}

Stopping
{
    // The task is now stopped.
    Stopped
        Stopped
        {
            releaseResources();
        }
}

Stopped { ... }
Deleted { ... }
...
%%
```

The transition actions methods in `Task` are:

```java
package com.acme.supercron;

public final class Task
    implements TaskEventListener, TimerEventListener {

    public Task() {...}

    // TaskEventListener Interface Implementation.
    <snip>

    // TimerEventListener Interface Implementation.
    <snip>

    // State Machine Actions.

    // Activate the underlying task and get it running again.
    /* package */ void continueTask() {
        ...
        return;
    }

    // Inactivate the underlying task.
    /* package */ void suspendTask() {
        ...
        return;
    }

    // Start the timeslice timer for the given milliseconds.
    /* package */ void startSliceTimer(long timeslice) {
        ...
        return;
    }

    // Stop the timeslice timer.
    /* package */ void stopSliceTimer() {
        ...
        return;
    }

    // Return system resources from whence they came.
    /* package */ void releaseResources() {
        ...
        return;
    }
...
}
```

Defining FSM Default Transitions

Now the mystery transitions `Stop`, `Block` and `Delete` are defined. The reason why these transitions have no start state is because they are taken no matter the current state. Well, not exactly.

- **Stop**: If a task is still alive (in the `Suspended`, `Running` or `Blocked` state), then it must immediately transition to the `Stopping` state. If a task is not alive (in the other three states), this transition is ignored because the task is no longer alive.
- **Block**: If a task is either `Suspended` or `Running`, then it transitions to the `Blocked` state. Otherwise this request is ignored.
- **Delete**: If a task is in any state other than `Deleted`, then it must transition to the `Deleted` state.

SMC provides two ways to define default transitions: the `Default` state and the `Default` transtion. Manual section 2 describes how `Default` state and transition work. Go there to learn more about them. `Task.sm` is updated with the default `Stop`, `Block` and `Delete` transition definitions:

```
%{
//
// Copyright (c) 2005 Acme, Inc.
// All rights reserved.
//
// Acme - a name you can trust!
//
// Author: Wil E. Coyote (Hungericus Vulgarus)
//
%}

// This FSM works for the Task class only and only the Task
// class may instantiate it. %class Task
%package com.acme.supercron
%fsmclass TaskFSM
%fsmfile TaskFSM
%access package

// A %map name cannot be the same as the FSM class name.
%start TaskMap::Suspended %map TaskMap
%%
Suspended {
    // Time to do more work.
    // The timeslice duration is passed in as a transition
    // argument.
    Start(timeslice: long)
        Running {
            continueTask();
            startSliceTimer(timeslice);
```

```
        }

    Block
        Blocked {
            blockTask();
        }
}

Running {
    // Wait for another time slice.
    Suspend
        Suspended {
            stopSliceTimer();
            suspendTask();
        }

    Block
        Blocked {
            stopSliceTimer();
            blockTask();
        }

    // Task has completed.
    Done
        Stopped {
            stopSliceTimer();
            releaseResources();
        }
}

// Wait here to be either unblocked, stopped or deleted.
Blocked {
    // The task may continue working now.
    // No actions needed.
    Unblock
        Suspended {}
}

Stopping {
    // The task is now stopped.
    Stopped
        Stopped {
            releaseResources();
        }

    // We are stopping.
    Stop
        nil {}
}
```
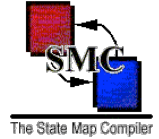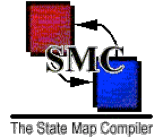
```
Stopped {
    // We are stopped.
    Stop
        nil {}

    // Ignore all transitions until deleted.
    Default
        nil {}
}

Deleted {
    // Define all known transitions as loopbacks.
    Start(timeslice: long)
        nil {}

    Suspend()
        nil {}

    Block()
        nil {}

    Unblock()
        nil {}

    Done()
        nil {}

    Stop()
        nil {}

    Stopped()
        nil {}

    Delete()
        nil {}
}

Default {
    // Three states follow this transition, three states ignore.
    // So define the active definition.
    Stop
        Stopping {
            stopTask();
    }

    // Block is ignored by four of six states.
    // Force the other two states to define this.
    // Note the "nil" end state. This is a loopback transition
    Block
        nil {}
```

```
    // All but the Delete state follow this transition. Define it here.
    Delete
        Deleted {}

    // Ignore a transition by default.
    Default
        nil {}
}
%%
```

The `blockTask()` and `stopTask()` methods are added to the `Task` class:

```
package com.acme.supercron;

public final class Task
    implements TaskEventListener, TimerEventListener {

    public Task() { ... }

<snip>

    // State Machine Actions.

<snip>

    // Block the underlying task from running.
    /* package */ void blockTask() {
        ...
        return;
    }

    // Permanently stop the underlying task.
    /* package */ void stopTask() {
        ...
        return;
    }

<snip>
...
}
```

There is one more improvement to the FSM that needs to be made before we finish. Notice that the `Running` state's transitions must stop the slice timer. If a new transition is added to `Running`, the developer must remember to include the `stopSliceTimer()` action. This is a potential problem because a different developer maintaining this FSM may not know about this. But there is a solution to this.

Defining State Entry/Exit Actions

The slice timer should be stopped when not in the `Running` state. The way to enforce this is to add an `Exit` block to `Running` and move the `stopSliceTimer()` action there.

Since the state's `Exit` actions are being defined, it would appear natural to put the `startSliceTimer()` action into a `Entry` block. But there two reasons against it:

1. There is only one transition into the `Running` state. Moving `startSliceTimer()` from `Suspended`'s `Start` transition to `Running`'s entry actions gains nothing.
2. `startSliceTimer()` takes the `Start` transition's `timeslice` argument. If `startSliceTimer()` is an entry action, then it cannot access that transition argument. The only way around it is to store the slice time in the Task class and then retrieve it immediately in the entry action (`startSliceTimer(ctxt.getSliceTime())`). Now moving the action to the entry block is worse than doing nothing.

(Go here to learn more about state entry and exit actions.)

```
%{
//
// Copyright (c) 2005 Acme, Inc.
// All rights reserved.
//
// Acme - a name you can trust!
//
// Author: Wil E. Coyote (Hungericus Vulgarus)
//
%}

// This FSM works for the Task class only and only the Task
// class may instantiate it.
%class Task
%package com.acme.supercron
%fsmclass TaskFSM
%fsmfile TaskFSM
%access package

// A %map name cannot be the same as the FSM class name.
%start TaskMap::Suspended
%map TaskMap
%%

<snip>
```

```
Running
    Exit { stopSliceTimer(); }
{
    // Wait for another time slice.
    Suspend
        Suspended {
            // stopSliceTimer(); moved.
            suspendTask();
        }

        Block
            Blocked {
                // stopSliceTimer(); moved.
                blockTask();
            }

    // Task has completed.
    Done
        Stopped {
            // stopSliceTimer(); moved.
            releaseResources();
        }
}

<snip>
}
%%
```

There is one final task: connecting the task FSM to the `Task` application class.
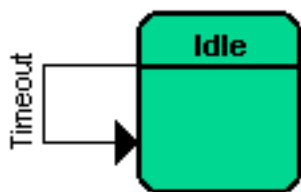

```
// State
Idle
{
// Trans    Next State Actions
    Timeout Idle        {}
}
```

Connecting Task and Task FSM

Connecting FSMs to their application class is as simple as:

- Add the data member `TaskFSM _fsm` to `Task` class.
- Instantiate `TaskFSM` in `Task`'s constructor.
- If the start state has entry actions that must be executed when the FSM is created, then call
  `_fsm.enterStartState()` *outside* of `Task`'s constructor.
- When you need to issue a transition, call `_fsm`'s appropriate transition method. For example:

  `_fsm.Start(timeSlice);`

```java
package com.acme.supercron;

public final class Task
    implements TaskEventListener, TimerEventListener {

    public Task() {
        ...

        // Instantiate the FSM here but perform the initial
        // state's entry actions outside of the constructor
        // to prevent referencing this object before its
        // initialization is complete.
        _fsm = new TaskFSM(this);
    }

    // Execute the start state's entry actions by calling this
    // method. This method should be called only once and prior to
    // issuing any transitions. Therefore this method should be
    // called before registering this Task instance as a task and
    // timer event listener.
    public void startFSM() {
        _fsm.enterStartState();
        TaskManager.addListener(this);
    }

    // TaskEventListener Interface Implementation.

    // Time for the incomplete task to continue its work for the
    // specified time slice.
    public void start(long timeSlice) {
        _fsm.Start(timeSlice);
    }

    // Called when a running, incomplete task should suspend
    // running even though its time slice is not expired.
    // Note: the task's running is also suspended when the time
```

```
    // slice expires.
    public void suspend() {
        _fsm.Suspend();
    }

    // Called when an incomplete task is blocked. Blocked tasks
    // are able to continue running when unblocked.
    public void block() {
        _fsm.Block();
    }

    // Called when a blocked task is unblocked and allowed
    // to continue running.
    public void unblock() {
        _fsm.Unblock();
    }

    // Called when an incomplete task is permanently stopped.
    // Stopped tasks are then deleted.
    public void stop() {
        _fsm.Stop();
    }

    // Called when the task is deleted. Tasks are deleted when
    // either 1) the task has completed running and is now
    // stopped or 2) when the system is shutting down and all
    // are to terminate immediately.
    public void delete() {
        _fsm.Delete();
    }

    // TimerEventListener Interface Implementation.

    // Called with the time slice timer has expired. If running,
    // the task is suspended.
    public void handleTimeout(TimerEvent event) {
        _fsm.Suspend();
    }

 <snip>

    // The associated finite state machine.
    private final TaskFSM _fsm;
}
```

Voíla! `Task`'s behavior is now defined by a finite state machine.

Section 2: From Model to SMC

This section shows a quasi-UML state machine snippet and the equivalent SMC code. I use the word "quasi" because SMC is not directly derived from UML or Harel state machines. That means that there are capabilities in UML that are not in SMC and vice versa. See the SMC FAQ for why this is.

Instantiating a Finite State Machine

Care must be taken when instantiating an SMC-generated finite state machine. The application class passes a its reference to the FSM context and this reference is used when FSM actions are performed. It is safe to instantiation an FSM within a constructor but unsafe to enter the start state while in a constructor because the start state entry actions will call your application object before its constructor has completed.

```
private final AppClassContext _fsm;

public AppClass() {
    // Initialize you application class. Instantiate your finite state
    // machine.
    // Note: it is safe to pass this to the the FSM constructor because
    // the FSM constructor only stores this in a data member.
    _fsm = new AppClassContext(this);
}

// Enter the FSM start state after instantiating this application object.
public void startWorking() {
    _fsm.enterStartState();
    return;
}
```

Prior to SMC v. 6, the FSM constructor incorrectly used the this pointer which meant instantiating the FSM within the application constructor could lead to incorrect behavior. SMC v. 6 corrects this problem and it is now safe to instantiate the FSM within your application constructor.

The enterStartState method should be called only once after instantiating the finite state machine and prior to issuing any transition. This method is unprotected and does not prevent its being called multiple times. If enterStartState is called after the first transition, then incorrect behavior may occur.

Simple Transition


```
// State
Idle
{
//  Trans      Next State      Actions
    Run        Running         {}
}
```
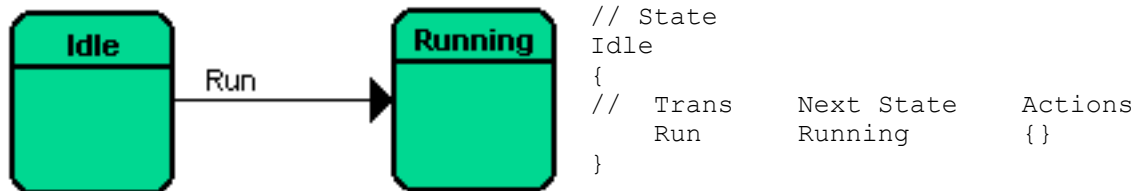
A state and transition names must have the form "[A-Za-z_][A-Za-z0-9_]*".

Jump Transition


```
// State
Idle
{
//  Trans      Next State      Actions
    Run        jump(Running)   {}
}
```
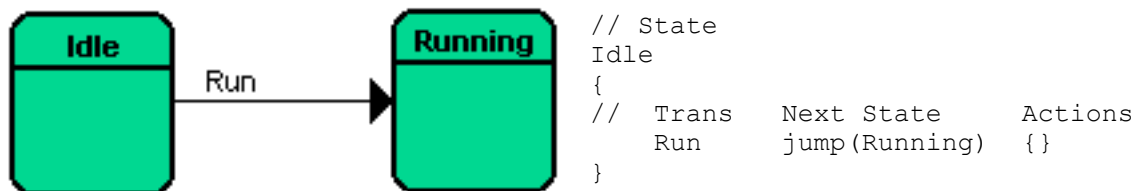
The Jump transition is equivalent to the Simple Transition and is provided since this transition is used in an Augmented Transition Network.

In a future SMC release, the Jump transition will become the only way for make an end state outside the current map. The syntax will be: `jump(AnotherMap::NextState)`

External Loopback Transition

An external loopback does leave the current state and comes back to it. This means that the state's exit and entry actions are executed. This is in contrast to an internal loopback transition.


```
// State
Idle
{
//  Trans  Next State  Actions
    Run    Idle        {}
}
```
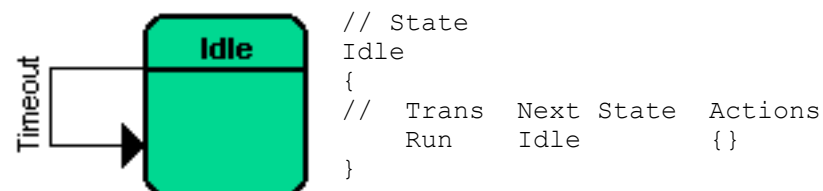
Internal Loopback Transition
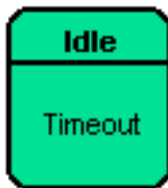
```
                                // State
                                Idle
                                {
                                //  Trans    Next State Actions
                                    Timeout nil        {}
                                }
```

Using "nil" as the next state causes the transition to remain in the current state and not leave it. This means that the state's exit and entry actions are not executed. This is in contrast to an external loopback transition.

Transition with Actions
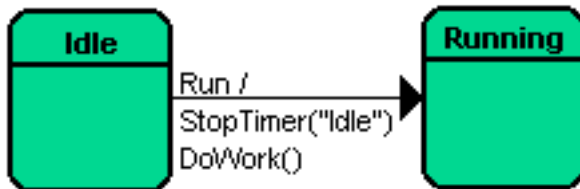
```
                                // State
                                Idle {
                                    // Trans
                                    Run
                                        // Next State
                                        Running {
                                            // Actions
                                            StopTimer("Idle");
                                            DoWork();
                                        }
                                }
```

1. A transition's actions must be enclosed in a `{ }` pair.
2. The action's form is `[A-Za-z][A-Za-z0-9_-]*(<argument list>)`. The argument list must be either empty or consist of comma-separated literals. Examples of literals are: integers (positive or negative, decimal, octal or hexadecimal), floating point, strings enclosed in double quotes, constants and transition arguments.
3. Actions must be member functions in the %class class and accessible by the state machine. Usually that means public member functions in C++ or package in Java.

Action arguments include:

- An integer number (e.g. 1234).

- A float number (e.g. 12.34)
- A string (e.g. "abcde")
- A [transition argument](#)
- A constant, #define, or global variable.
- An Independent subroutine or method call (e.g. event.getType()). **Note:** this subroutine/method call may also include arguments.

If you want to call a method in the context class, then use the `ctxt` variable. For example, if the context class contains a method `getName()` and you want to call it inside an action's argument list, then write `ctxt.getName()`.

Go here for sample code using the `ctxt` variable.

Note: Only use `ctxt` inside argument lists and transition guards.

Transition Guards
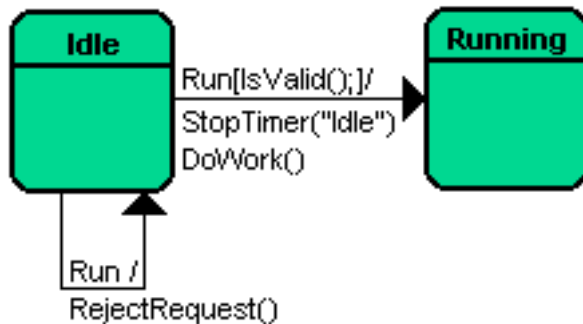

```
// State
Idle {
    // Trans
    Run
      // Guard condition
      [ctxt.isProcessorAvailable() == true &&
       ctxt.getConnection().isOpen() == true]

        // Next State
        Running {
            // Actions
             StopTimer("Idle");
             DoWork();
        }

        Run nil {RejectRequest();}
}
```

The guard must contain a condition that is valid target language source code - that is, it would be a valid "if" statement. Your guard may contain &&s, ||s, comparison operators (==, <, etc.) and nested expressions. SMC copies your guard condition verbatim into the generated output.

Note: If your are calling a context class method, then you must prefix the method with `ctxt` - SMC will not append `ctxt` for you.

If the guard condition evaluates to true, then the transition is taken. If the guard condition evaluates to false, then one of the following occurs (ordered by precedence):

1. If the state has another guarded transition with the same name and arguments, that transition's guard is checked.

2. Failing that, If the state has another unguarded transition with the same name and argument list, that transition is taken.

3. If none of the above, then the default transition logic is followed.

A state may have multiple transitions with the same name and argument list as long as they all have unique guards. When a state does have multiple transitions with the same name, care must be taken when ordering them. The state machine compiler will check the transitions in the same top-to-bottom order that you use except for the unguarded version. That will always be taken only if all the guarded versions fail. Guard ordering is only important if the guards are not mutually exclusive, i.e., it is possible for multiple guards to evaluate to true for the same event.

Allowable argument types for a transition guard are the same as for a transition action.

Transition Arguments
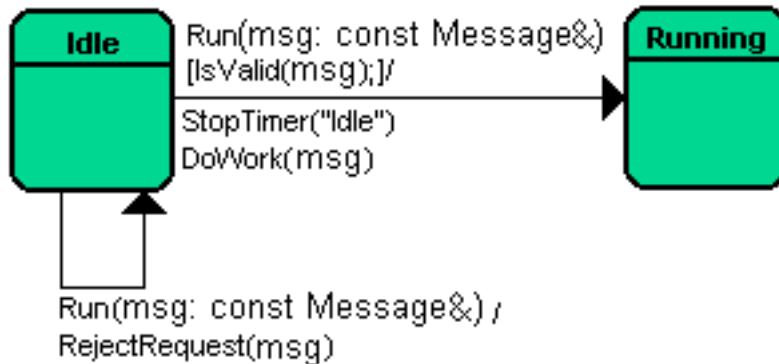

```
// State
Idle {
    // Transition
    Run(msg: const Message&)

     // Guard condition
     [ctxt.isProcessorAvailable() == true &&
      msg.isValid() == true]

        // Next State
        Running {
            // Actions
            StopTimer("Idle");
            DoWork(msg);
        }

        Run(msg: const Message&)
            // Next State Actions
            nil              {RejectRequest(msg);}
}
```

Note: When using transition guards and transition arguments, multiple instances of the same transition must have the same argument list. Just as with C++ and Java methods, the transitions `Run(msg: const Message&)` and `Run()` are not the same transition. Failure to use the identical argument list when defining the same transition with multiple guards will result in incorrect code being generated.

Tcl "arguments":

While Tcl is a type-less language, Tcl does distinguish between call-by-value and call-by-reference. By default SMC will generate call-by-value Tcl code if the transition argument has no specified type. But you may use the artificial types "value" or "reference".

If your Tcl-targeted FSM has a transition:

then the generated Tcl is:

```
DoWork(task: value)
    Working {
        workOn(task);
    }
```

```
public method DoWork {task} {
    workOn $this $task;
}
```

If your Tcl-targeted FSM has a transition:

then the generated Tcl is:

```
DoWork(task: reference)
    Working {
        workOn(task);
}
```

```
public method DoWork {task} {
    workOn $this task;
}
```

The method `workOn` must `upvar` the task parameter:

```
public method workOn {taskName} {
    upvar $taskName task;
    ...
}
```

Lua/Python/Ruby "arguments":

While Lua/Python/Ruby is a dynamically typed language and does not use types for function parameter definitions, you could provide a optional data type for transition arguments. This "data type" is ignored when generating the target Lua/Python/Ruby code. I suggest using meaningful type names.

```
DoWork(task: TaskObj, runtime: Ticks)
    Working {
        ...
    }
```

Groovy/PHP "arguments":

While Groovy/PHP gives the choice between static and dynamic typing, you could provide a optional data type for transition arguments. In this case, the type is used when generating the target Groovy/PHP code.

The PHP variable syntax is like Perl (named with '$').

Perl "arguments":
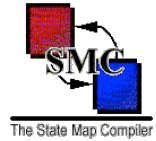
While Perl is a dynamically typed language and does not use types for function parameter definitions, you could provide a optional data type for transition arguments. This "data type" is ignored when generating the target Perl code. I suggest using meaningful type names.

Only Perl scalar values (i.e., named with '$') are allowed.

```
DoWork($task: TaskObj, $runtime: Ticks)
    Working {
        ...
    }
```

Entry and Exit Actions
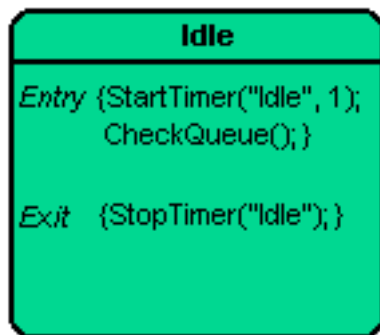
```
// State
Idle
    Entry {StartTimer("Idle", 1); CheckQueue();}
    Exit {StopTimer("Idle");}
{
    // Transitions
}
```

When a transition leaves a state, it executes the state's exit actions before any of the transition actions. When a transition enters a state, it executes the state's entry actions. A transition executes actions in this order:

1. "From" state's exit actions.
2. Set the current state to null. This denotes that a transition is in progress.
3. The transition actions in the same order as defined in the .sm file.
4. Set the current state to the "to" state.
5. "To" state's entry actions.

As of version 6.0.0, SMC generates a `enterStartState` method which executes the start state's entry acitons. It is now up to the application to call the start method after instantiating the finite state machine context. If it is not appropriate to execute the entry actions upon start up, then do not call `enterStartState`. You are not required to call this method to set the finite state machine start state. That is done when the FSM is instantiated. This method is used only to execute the start state's entry actions.

If you do call this method, be sure to do it outside of the context class' constructor. This is because entry actions call context class methods. If you call `enterStateState` from within you context class' constructor, the context instance will be referenced before it has completed initializing which is a bad thing to do.

`enterStartState` does not protect against being called multiple times. It should be called at most once and prior to issuing any transitions. Failure to follow this requirement may result in inappropriate finite state machine behavior.

Whether a state's `Entry` and `Exit` actions are executed depends on the type of transition taken. The following table shows which transitions execute the "from" state's `Exit` actions and which transitions execute the "to" state's `Entry` actions.

Transition Type	Execute "From" State's Exit Actions?	Execute "To" State's Entry Actions?
Simple Transition	Yes.	Yes.
External Loopback Transition	Yes.	Yes.
Internal Loopback Transition	No.	No.
Push Transition	No.	Yes.
Pop Transition	Yes.	No.

Table 1: Entry/Exit Execution.

WARNING! Entry and exit actions are *not* supported for the Default state which is not an actual state. See more in the Default Transitions section.

From this point on, SMC diverges from UML. SMC uses the idea of multiple machines and pushing and popping states as way of breaking complicated behavior up into simpler parts. UML achieves much the same by grouping states into superstates. They may be equivalent in ability but I find the idea of pushing to a new state easier to understand because it is similar to the subroutine call.

Push Transition


```
Running {
    Blocked
        push(WaitMap::Blocked) {
            GetResource();
        }
}
```
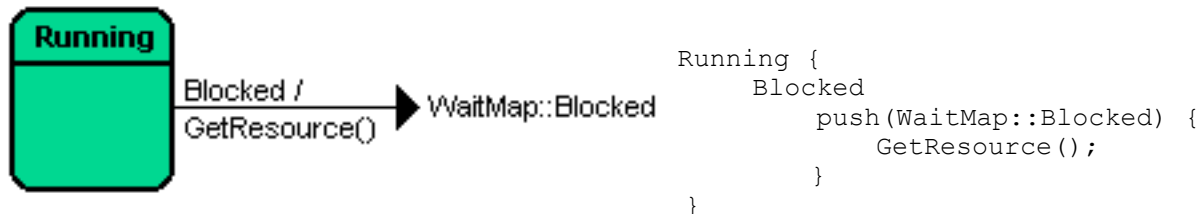
Note: The end state does not have to be in another map - it could be in the same `%map` construct.

Conversely, a plain transition's end state may be in another map. But chances are that you will set up maps so that you will push to another map's state and simple transitions will stay within the same map. You use multiple maps for the same reason you create multiple subroutines: to separate out functionality into easy-to-understand pieces.

With SMC v. 1.3.2, the push syntax was modified yet is backward compatible with the initial syntax. The new syntax is:

```
Running {
    Blocked
        BlockPop/push(WaitMap::Blocked) {
            GetResource();
        }
}
```

This causes the state machine to:

1. Transition to the `BlockPop` state.
2. Execute the `BlockPop` entry actions.
3. Push to the `WaitMap::Blocked` state.
4. Execute the `WaitMap::Blocked` entry actions.

When `WaitMap` issues a pop transition, control will return to `BlockPop` and the pop transition issued from there.

Use this new syntax when a state has two different transitions which push to the same state but need to handle the pop transition differently. For example:

```
Idle {
    NewTask NewTask/push(DoTask) {}
    RestartTask OldTask/push(DoTask) {}
}

NewTask {
    TaskDone Idle {}

    // Try running the task one more time.
    TaskFailed OldTask/push(DoTask) {}
}

OldTask {
    TaskDone Idle {}
    TaskFailed Idle {logFailure();}
}
```

Pop Transition


```
Waiting
{
    Granted pop(OK) {cleanUp();}
    Denied pop(FAILED) {cleanUp();}
}
```
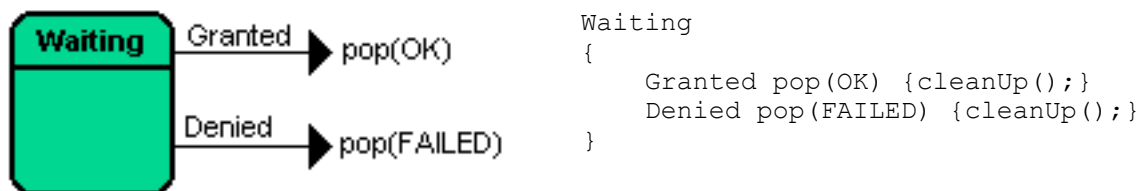
The pop transition differs from the simple and push transition in that:

- The end state is not specified. That is because the pop transition will return to whatever state issued the corresponding push.
- There pop transition has an *optional* argument: a transition name.

In the above example, if the resource request is granted, the state machine returns to the corresponding state that did the push and then takes that state's OK transition. If the request is denied, the same thing happens except the FAILED transition is taken. The code for the corresponding push transition is:

```
Running {
    Blocked push(WaitMap::Blocked) {GetResource();}

    // Handle the return "transitions" from WaitMap.
    OK     nil {}
    FAILED Idle {Abend(INSUFFICIENT_RESOURCES);}
}
```

As of SMC v. 1.2.0, additional arguments may be added after the pop transition's transition argument. These additional arguments are like any others passed to an action and will be passed into the named transition. Following the above example, given the pop transition pop(FAILED, errorCode, reason), then the FAILED should be coded as:

```
FAILED(errorCode: ErrorCode, reason: string)
    Idle {
        Abend(errorCode, reason);
}
```

Default Transitions

What happens if a state receives a transition that is not defined in that state? SMC has two separate mechanisms for handling that situation.

The first is the "Default" state. Every `%map` may have a special state named "Default" (the uppercase D is significant). Like all other states, the Default state contains transitions.

```
Default {
    // Valid run request but transition occurred in an invalid
    // state. Send a reject reply to valid messages.
    Run(msg: const Message&)
      [ctxt.isProcessorAvailable() == true && msg.isValid() == true]
        nil {
            RejectRequest(msg);
        }

    // Ignore invalid messages are ignored when received in
    // an invalid state.
    Run(msg: const Message&)
        nil
        {}

    Shutdown
        ShuttingDown {
            StartShutdown();
        }
    }
```

Default state transitions may have guards and arguments features as non-default transitions. This means the Default state may contain multiple guarded and one unguarded definition for the same transition.

The second mechanism is the "Default" transition. This is placed inside a state and is used to back up all transitions.

```
Connecting {
    // We are now connected to the far-end. Now we can logon.
    Connected
        Connected {
            logon();
    }

    // Any other transition at this point is an error.
    // Stop the connecting process and retry later.
    Default
        RetryConnection {
            stopConnecting();
        }
    }
```

Because any transition can fall through to the Default transition, Default transitions:

- May *not* have an argument list.
- A Default transition may take a guard.
- Putting a Default transition in the Default state means that all transitions will be handled - it is the transition definition of last resort.

Transition Precedence

Transition definitions have the following precedence:

1. Guarded transition.
2. Unguarded transition.
3. The Default state's guarded definition.
4. The Default state's unguarded definition.
5. The current state's guarded Default transition.
6. The current state's unguarded Default transition.
7. The Default state's guarded Default transition.
8. The Default state's unguarded Default transition.

> **Since SMC does not force you to specify a Default state or Default transition, it is possible that there is no transition defined. If SMC falls through this list, it will throw a "Transition Undefined" exception.**

Section 3: Adding a State Machine to your Class

The SMC-generated code is designed to be loosely coupled with your application software. The only changes that you need to make to your code is to:

1. Include the SMC class definitions into your application (stored in smc/lib by programming language name):
    *   **C:** Have lib/C/statemap.h in the include path.
    *   **C++:** Have lib/C++/statemap.h in the include path.
    *   **C#:** Have lib/DotNet/statemap.dll included in the build library list.
    *   **Groovy:** Have lib/Groovy/statemap.jar in the class path.
    *   **Java:** Have lib/Java/statemap.jar in the classpath.
    *   **JavaScript:** Have lib/JavaScript/statemap.js accessible to `<script>` tag.
    *   **Lua:** Have the lib/Lua/statemap.lua module on you Lua `package.path` (initialized by the environment variable `LUA_PATH`).
    *   **Objective-C:** Have lib/ObjC/statemap.h in the include path.
    *   **Perl:** Have StateMachine::Statemap module on you Per library path `@INC`.
    *   **PHP:** Have StateMachine/statemap.php in your PHP `include_path`.
    *   **Python:** Have the statemap module on your import source path `sys.path`.
    *   **Ruby:** Have the statemap module on your Ruby library path `$LOAD_PATH`.
    *   **Scala:** Have lib/Scala/statemap.jar in the classpath.
    *   **Tcl:** Have the lib/Tcl/statemap1.0 package on you path.
    *   **VB.net:** Have lib/DotNet/statemap.dll included in the build library list.

2. Include the state machine source file:
    *   **C:** #include "*AppClass*_sm.h"
    *   **C++:** #include "*AppClass*_sm.h"
    *   **C#:** put the *AppClass*_sm.cs file into the product source list.
    *   **Groovy:** If *AppClass*Context class is in the same package as *AppClass*, no importation is needed.

- **Java:** If *AppClass*Context class is in the same package as *AppClass*, no importation is needed.
- **JavaScript:** Have *AppClass*_sm.js file accessible to `<script>` tag.
- **Lua:** require '*AppClass*_sm'.
- **Objective-C:** #import "*AppClass*_sm.h"
- **Perl:** use *AppClass*_sm;
- **PHP:** require_once '*AppClass*_sm';
- **Python:** import *AppClass*_sm
- **Ruby:** require '*AppClass*_sm'
- **Scala:** If *AppClass*Context class is in the same package as *AppClass*, no importation is needed.
- **Tcl:** source ./*AppClass*_sm.tcl
- **VB.net:** put the *AppClass*_sm.vb file into the project source list.

3. Instantiate the state machine context object.
4. If you want to execute the start state's entry actions call the state machine context's `enterStartState` method. This is *not* needed to set the start state as that is done when the state machine context is instantiated. `enterStartState` only executes the start state's entry actions (if any exist).

That's all you need to do. Whenever you want to issue a transition, call the context's object transition method.

SMC does not change your code or require you to change your code's logic. SMC does not require that your class inherit or implement any SMC class. SMC state machines are easy to add with minimal impact to your existing code.

CAUTION: THIS BEARS REPEATING. Do **NOT** issue a transition from within an action - it will cause the state machine to throw an exception since actions are not allowed to issue a transitions.

For an explanation of why this is, see the SMC FAQ question: Why can't an action issue a transition?

If you do need to issue a transition from an action, see the section 7, "Queuing Up" explaining how to use a "transition queue".

Cautions

The following class and file names are generated by SMC. Be careful not to use them yourself.

- *<AppClass>*Context
- *<AppClass>*State
- *<MapName>*
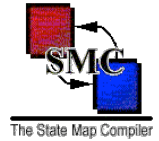
- *<MapName>*_Default
- *<MapName>_<StateName>*
- *<smc file name stem>*_sm.h
- *<smc file name stem>*_sm.*<ext>*

where:

- *<AppClass>* is the name of your application class using that FSM.
- *<MapName>* is a `%map  <MapName>` in *<smc file name stem>*.sm.
- *<StateName>* is a state in *<smc file name stem>*.sm.
- *<smc file name stem>* is that part of the .sm file's name before the ".".
- *<ext>* is the source file extension used for the target programming language.

C

C language is not Object Oriented. But this behavior could be emulated in order to follow the SMC State pattern.

Assumptions

- Your application class is named `NetworkIF`.
- Your class is stored in NetworkIF.h and NetworkIF.c.
- The state machine is in NetworkIF.sm.
- As explained in section 1, you have added the lines:
      ```
      %class NetworkIF
      %header NetworkIF.h
      %start MainMap::Start
      ```
  at the appropriate location in NetworkIF.sm. Note: your .sm file does not need to use the map name `MainMap` nor have a state named `Start`. This is for example purposes only.

- You have successfully compiled NetworkIF.sm.

Changes to NetworkIF.h

1. Add the following #include to NetworkIF.h:
      ```
      #include "NetworkIF_sm.h"
      ```
2. Add the data member to `NetworkIF`:

      ```
      struct NetworkIFContext _fsm;
      ```

      (I use `_fsm` but you can use another name.)

Changes to NetworkIF.c

1. In NetworkIF initializer `NetworkIF_init`, add the line:

      ```
      void NetworkIF_Init(struct NetworkIF *network) {
          NetworkIFContext_Init(&this->_fsm, network);
      }
      ```

      Again, the data member does not have to be named _fsm. Add the following line if `Push` transitions are used:

      ```
      FSM_STACK(&network->_fsm, AppStack);
      ```

where `AppStack` is a memory location used to store the state stack.

2. If the start state has entry actions and these actions need to be executed before any transitions are issued, then add the following code:

    ```
    NetworkIFContext_EnterStartState(&network->_fsm);
    ```

3. Whenever you want to issue a state machine transition, add the following line of code:

    ```
    NetworkIFContext_<transition>(&network->_fsm, ...);
    ```

    where `<transition>` is the transition name. If the transition takes arguments, then pass them in the transition call:

    ```
    NetworkIFContext_Connect(&network->_fsm, "192.168.3.100", 80);
    ```

Changes to Makefile

Add `NetworkIF_sm.c` to the source file list and link `NetworkIF_sm.o` into your application.

#includes in NetworkIF.sm

If you need to include header files in your .sm file, use the `%include` keyword:

```
%include <util/logger.h>
```

Assumptions

- Your application class is named `NetworkIF`.
- Your class is stored in NetworkIF.h and NetworkIF.cpp.
- The state machine is in NetworkIF.sm.
- As explained in section 1, you have added the lines:

  ```
  %class NetworkIF
  %header NetworkIF.h
  %start MainMap::Start
  ```

  at the appropriate location in NetworkIF.sm. Note: your .sm file does not need to use the map name `MainMap` nor have a state named `Start`. This is for example purposes only.

- You have successfully compiled NetworkIF.sm.

Changes to NetworkIF.h

1. Add the following #include to NetworkIF.h:

   ```
   #include "NetworkIF_sm.h"
   ```

2. Add the data member to `NetworkIF`:

   ```
   NetworkIFContext _fsm;
   ```

   This data member can be either `public`, `protected`, or `private` and can have any name (I use "_fsm" but you can use another name).

3. All state machine actions must be implemented as NetworkIF *public* methods. For an explanation as to why these methods must be public, see the answer to the FAQ question: Why do actions have to be declared as public?

Changes to NetworkIF.cpp

1. In all NetworkIF constructors, add the following initialization:

   ```
   _fsm(*this)
   ```

   Again, the data member does not need to be named `_fsm`.

2. If the start state has entry actions which must be executed before any transitions are issued, add the following code *outside* any `NetworkIF` constructors:

```
_fsm.enterStartState();
```

3. Whenever you want to issue a state machine transition, add the following line of code:

```
_fsm.<transition>(...);
```

where *`<transition>`* is the transition name. If the transition takes arguments, then pass them in the transition call:

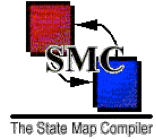```
_fsm.Connect("192.168.3.100", 80);
```

## Changes to Makefile

Add `NetworkIF_sm.cpp` to the source file list and link `NetworkIF_sm.o` into your application.

## #includes in NetworkIF.sm

If you need to include header files in your .sm file, use the `%include` keyword:

```
%include <util/logger.h>
```

(using -crtp: Curiously Recurring Template Pattern)

Assumptions

Same as C++ assumptions.

Changes to NetworkIF.h

1. Add the following `#include` to NetworkIF.h:

   ```
   #include "NetworkIF_sm.h"
   ```

2. Inherit the context template:

   ```
   class NetworkIF : public NetworkIFContext<NetworkIF>
   ```

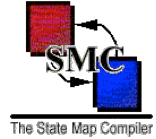   This must be a `public` inheritance, otherwise the SMC-generated code will not compile.

3. All state machine actions must be implemented as NetworkIF *public* methods. For an explanation as to why these methods must be public, see the answer to the FAQ question: Why do actions have to be declared as public?

Changes to NetworkIF.cpp

1. If the start state has actions which must be executed before any transitions are issued, add the following code *outside* any `NetworkIF` constructors:

   ```
   enterStartState();
   ```

2. Wherever you want to issue a state machine transition, add the following line of code:

   ```
   <transition>(...);
   ```

   where `<transition>` is the transition name. If the transition takes arguments, then pass them in the transition call:
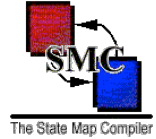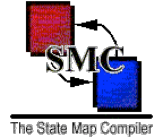
   ```
   Connect("192.168.3.100", 80);
   ```

Changes to Makefile

Add `NetworkIF_sm.cpp` to the source file list and link `NetworkIF_sm.o` into your application.

#includes in NetworkIF.sm
Same as in C++.

[Objective-C]

Assumptions:

- Your application class is named `NetworkIF`.
- Your class is stored in NetworkIF.h and NetworkIF.cpp.
- The state machine is in NetworkIF.sm.
- As explained in section 1, you have added the lines:

      %class NetworkIF
      %header NetworkIF.h
      %start MainMap::Start

   at the appropriate location in NetworkIF.sm. Note: your .sm file does not need to use the map name

   `MainMap` nor have a state named `Start`. This is for example purposes only.

- You have successfully compiled NetworkIF.sm

Changes to NetworkIF.h

1. Forward declare the FSM class by adding this line to NetworkIF.h:

        @class NetworkIFContext;

2. Add the data member to class `NetworkIF`:

        NetworkIFContext *_fsm;

Changes to NetworkIF.m

1. Import the FSM header:

        #import "NetworkIF_sm.h"

2. When allocating the FSM, pass in the NetworkIF instance reference:

        _fsm = [[NetworkIF alloc] initWithOwner:self];

3. If the start state has entry actions which must be executed prior to issuing any transitions, add the following code *outside* any `NetworkIF` constructors:

        [_fsm enterStartState];

4. When you want to issue a state machine transition, add the following line of code:

        [_fsm <transition>];

    where `<transition>` is the transition name.

Assumptions

- You have a Java class named `NetworkIF` defined in the file NetworkIF.java.
- The state machine is in NetworkIF.sm.
- You have successfully compiled NetworkIF.sm

Changes to NetworkIF.java

1. Add the following data member to class `NetworkIF`:

   `private NetworkIFContext _fsm;`

   The data member does not have to be `private` but may be `public` or `protected`. Also, the variable name does not have to be `_fsm`.

2. Guarantee that the following line is executed by all NetworkIF constructors:

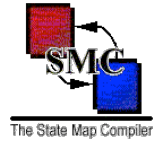   `_fsm = new NetworkIFContext(this);`

3. If the start state has entry actions which must be executed prior to issuing any transitions, add the following code *outside* the `NetworkIF` constructors:

   `_fsm.enterStartState();`

4. Wherever you want to issue a transition, all you need to do is:

   `_fsm.<transition>(...);`

   where <transition> is the transition name. If the transition takes arguments then pass them in the transition call:

   _fsm.Connect("192.168.3.100", 80);

5. All state machine actions may be implemented in NetworkIF as either `public` or `package private` methods so that the SMC-generated code may access them.

package in NetworkIF.sm

If you need to place the SMC-generated classes into a Java package, then see section 8.
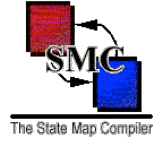
import in NetworkIF.sm

If you need to use `import` statements in you .sm file, use the `%import` keyword at the top of the .sm file:

```
%import java.net.InetAddress
%import java.util.Calendar
```

Assumptions:

- You have an [incr Tcl] class named `NetworkIF` defined in the file NetworkIF.tcl
- The state machine is in NetworkIF.sm.
- You have successfully compiled NetworkIF.sm.

Changes to NetworkIF.tcl

Adding the state machine to NetworkIF requires the following changes to NetworkIF.tcl:

1. At the top of the file, add:

    ```
    source NetworkIF_sm.tcl
    ```

2. Add the following data member declaration to class `NetworkIF`:

    ```
    private variable _fsm
    ```

3. In the NetworkIF constructor, create the state machine object and place the name into your previously declared data member:

    ```
    set _fsm [NetworkIFContext #auto $this];
    ```

    You don't have to use `#auto` to name the object but you do have to pass in the NetworkIF object's name to the NetworkIFContext constructor.

4. If the start state has entry actions which must be executed prior to issuing any transitions, add this code *outside* the `NetworkIF` constructors:

    ```
    $_fsm enterStartState
    ```

5. Wherever you want to issue a transition, all you need to do is:

    ```
    $_fsm <transition> ...
    ```
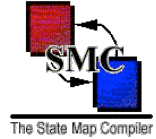
    Where `<transition>` is the transition name. If the transition takes arguments, pass them in the transition call:

    ```
    $_fsm Connect "192.168.3.100" 80;
    ```

6. All state machine actions must be implemented in NetworkIF as *public* methods. For an explanation as to why these methods must be public, see the answer to the FAQ question: <u>Why do actions have to be declared as public?</u>

"namespace eval" in NetworkIF.sm

If you need to place the SMC-generated classes into a Tcl namespace, then see Section 8.

"package require" in NetworkIF.sm

If you need to use `package require` or `source` statements in your .sm file, place these statements inside the verbatim code section at the top of the .sm file:

```
%{
    package require NetworkUtil;
%}
```

VB.net

Assumptions

- You have an VB.net class named `NetworkIF` defined in the file NetworkIF.vb
- The state machine is in NetworkIF.sm.
- You have successfully compiled NetworkIF.sm.

Changes to NetworkIF.vb

Adding the state machine to NetworkIF requires the following changes to NetworkIF.vb:

1. Add the following data member to class `NetworkIF`:

    ```
    Private _fsm As NetworkIFContext
    ```

    You can use any variable name you want. I use _fsm.

2. In the `New()` constructors add the line:

    ```
    _fsm = New NetworkIFContext(Me)
    ```

3. If the start state has entry actions which must be executed prior to issuing any transitions add the following code outside the `NetworkIF` constructors:

    ```
    _fsm.EnterStartState()
    ```
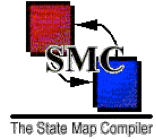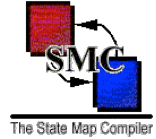
4. Whenever you want to issue a transition all you need to do is:

    ```
    _fsm.<transition>(...)
    ```

    where *`<transition>`* is the transition name. If the transition takes arguments then pass them in the transition call:

    ```
    _fsm.Connect("192.168.3.100", 80)
    ```

5. All state machine actions must be implemented in NetworkIF as Public methods so that the SMC generated code can access them.

Imports in NetworkIF.sm

f you need to use `Imports` in your .sm file, use the `%import` keyword in the top of your .sm file:

```
%import System.Drawing
```

C#

Assumptions

- You have an C# class named `NetworkIF` defined in the file NetworkIF.cs
- The state machine is in NetworkIF.sm.
- You have successfully compiled NetworkIF.sm.

Changes to NetworkIF.cs

Adding the state machine to NetworkIF requires the following changes to NetworkIF.vb:

1. Add the following data member to class `NetworkIF`:

   ```
   private NetworkIFContext _fsm;
   ```

   You can use any variable name you want. I use _fsm.

2. In the `New()` constructors add the line:

   ```
   _fsm = new NetworkIFContext(this);
   ```

3. If the start state has entry actions which must be executed prior to issuing any transitions add the following code outside the `NetworkIF` constructors:

   ```
   _fsm.enterStartState()
   ```

4. Whenever you want to issue a transition all you need to do is:

   ```
   _fsm.<transition>(...);
   ```

   where *`<transition>`* is the transition name. If the transition takes arguments then pass them in the transition call:
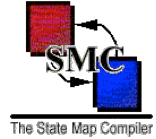
   ```
   _fsm.Connect("192.168.3.100", 80);
   ```
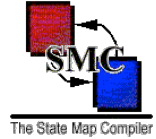
5. All state machine actions must be implemented in NetworkIF as Public methods so that the SMC generated code can access them.

Imports in NetworkIF.sm

f you need to use `import` in your .sm file, use the `%import` keyword in the top of your .sm file:

```
%import System.Drawing
```

Assumptions

- You have a Groovy class named `NetworkIF` defined in the file NetworkIF.groovy
- The state machine is in NetworkIF.sm.
- You have successfully compiled NetworkIF.sm.

Changes to NetworkIF.groovy

Adding the state machine to NetworkIF requires the following changes to NetworkIF.groovy:

1. Add the following data member to class NetworkIF:

   ```
   private def _fsm
   ```

   The data member does not have to be private. Also, the variable name does not have to be `_fsm`.

2. Guarantee that all constructors execute the following line:

   ```
   _fsm = new NetworkIFContext(this)
   ```

3. If the start state has entry actions which must be executed prior to issuing any transitions, add the following code *outside* `NetworkIF` constructors:

   ```
   _fsm.enterStartState()
   ```

4. Wherever you want to issue a transition, all you need to do is:

   ```
   _fsm.<transition>(...)
   ```

   where <transition> is the transition name. If the transition takes arguments then pass them in the transition call.

   ```
   _fsm.Connect("192.168.3.100", 80)
   ```

5. All state machine actions may be implemented in NetworkIF as either public or package private methods, allowing the SMC-generated code to access said methods.
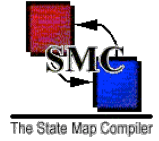
package in NetworkIF.sm

If you need to place the SMC-generated classes into a Groovy package, then see Section 8.

import in Network.sm

If you need to use import statements in your .sm file, use the %import keyword at the top of the .sm file:

```
%import groovy.util
%import java.net.InetAddress
%import java.util.Calendar
```

Assumptions

- You have an Lua class named `NetworkIF` defined in the file NetworkIF.lua
- The state machine is in NetworkIF.sm.
- You have successfully compiled NetworkIF.sm.

Changes to NetworkIF.lua

Adding the state machine to `NetworkIF` requires the following changes to NetworkIF.lua:

1. In the `NetworkIF new` methods, add the line:

   ```
   o._fsm = NetworkIF_sm.sm:new({_owner = 0})
   ```

2. If the start state has entry actions which must be executed prior to issuing any transitions add the following code *outside* the `NetworkIF` constructors:

   ```
   self.fsm:enterStartState()
   ```

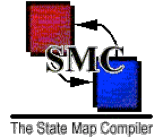3. Wherever you want to issue a transition, all you need to do is:

   ```
   self._fsm:<transition>(...)
   ```

   where `<transition>` is the transition name. If the transition takes arguments then pass them in the transition call:

   ```
   self._fsm:Connect("192.168.3.100", 80)
   ```

Imports in NetworkIF.sm

If you need to use `require` in your .sm file, use the `%import` keyword in the top of your .sm file:

```
%import System.Drawing
```

Python

Assumptions

- You have an Python class named `NetworkIF` defined in the file NetworkIF.py
- The state machine is in NetworkIF.sm.
- You have successfully compiled NetworkIF.sm.

Changes to NetworkIF.py

Adding the state machine to `NetworkIF` requires the following changes to NetworkIF.py:

1. In the `NetworkIF __init__` methods, add the line:

    ```
    self._fsm = NetworkIF_sm.NetworkIF_sm(self)
    ```

2. If the start state has entry actions which must be executed prior to issuing any transitions, add the following code *outside* the `NetworkIF` constructors:

    ```
    self._fsm.enterStartState()
    ```

3. Wherever you want to issue a transition, all you need to do is:

    ```
    self._fsm.<transition>(...)
    ```
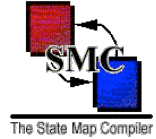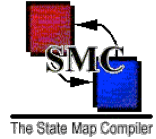
    where `<transition>` is the transition name. If the transition takes arguments, then pass them in the transition call:

    ```
    self._fsm.Connect("192.168.3.100", 80)
    ```

Imports in NetworkIF.sm

If you need to use `import` in your .sm file, use the `%import` keyword in the top of your .sm file:

```
%import System.Drawing
```

Assumptions

- You have an PHP class named `NetworkIF` defined in the file NetworkIF.php
- The state machine is in NetworkIF.sm.
- You have successfully compiled NetworkIF.sm.

Changes to NetworkIF.php

Adding the state machine to `NetworkIF` requires the following changes to NetworkIF.php:

1. In the `NetworkIF __construct` methods, add the line:

    ```
    $this->_fsm = new NetworkIF_sm($this);
    ```

2. If the start state has entry actions which must be executed prior to issuing any transitions, add the following code *outside* the `NetworkIF` constructors:

    ```
    $this->_fsm->enterStartState();
    ```

3. Wherever you want to issue a transition, all you need to do is:

    ```
    $this->_fsm-><transition>(...);
    ```

    where `<transition>` is the transition name. If the transition takes arguments, then pass them in the transition call:

    ```
    $this->_fsm->Connect("192.168.3.100", 80);
    ```

Imports in NetworkIF.sm

If you need to use `require_once` in your .sm file, use the `%import` keyword in the top of your .sm file:

```
%import Event/Dispatcher.php
```

Perl

Assumptions

- You have an Perl module named `NetworkIF` defined in the file NetworkIF.pm
- The state machine is in NetworkIF.sm.
- You have successfully compiled NetworkIF.sm.

Changes to NetworkIF.pm

Adding the state machine to `NetworkIF` requires the following changes to NetworkIF.pm:

1. In the `NetworkIF new` methods, add the line:

   ```
   $self->{_fsm} = new NetworkIF_sm($self);
   ```

2. If the start state has entry actions which must be executed prior to issuing any transitions, add the following code *outside* the `NetworkIF` constructors:

   ```
   $self->{_fsm}->enterStartState();
   ```

3. Wherever you want to issue a transition, all you need to do is:

   ```
   $self->{_fsm}-><transition>(...);
   ```
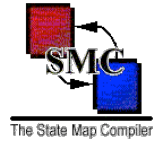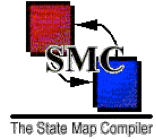
   where `<transition>` is the transition name. If the transition takes arguments, then pass them in the transition call:

   ```
   $self->{_fsm}->Connect("192.168.3.100", 80);
   ```

Imports in NetworkIF.sm

If you need to use `use` in your .sm file, use the `%import` keyword in the top of your .sm file:

```
%import System.Drawing
```

Ruby

Assumptions

- You have an Ruby class named `NetworkIF` defined in the file NetworkIF.rb
- The state machine is in NetworkIF.sm.
- You have successfully compiled NetworkIF.sm.

Changes to NetworkIF.rb

Adding the state machine to `NetworkIF` requires the following changes to NetworkIF.rb:

1. In the `NetworkIF initialize` methods, add the line:

   ```
   @_fsm = NetworkIF_sm::new(self)
   ```

2. If the start state has entry actions which must be executed prior to issuing any transitions, add the following code *outside* the `NetworkIF` constructors:

   ```
   @_fsm.enterStartState
   ```

3. Wherever you want to issue a transition, all you need to do is:

   ```
   @_fsm.<transition>(...)
   ```

   where `<transition>` is the transition name. If the transition takes arguments, then pass them in the transition call:

   ```
   @_fsm.Connect("192.168.3.100", 80)
   ```

Imports in NetworkIF.sm

If you need to use `require` in your .sm file, use the `%import` keyword in the top of your .sm file:

```
%import System.Drawing
```

Assumptions

- You have an Scala class named `NetworkIF` defined in the file NetworkIF.scala
- The state machine is in NetworkIF.sm.
- You have successfully compiled NetworkIF.sm.

Changes to NetworkIF.scala

Adding the state machine to `NetworkIF` requires the following changes to NetworkIF.scala:

1. Add the following data member to class `NetworkIF`:

   ```
   private val _fsm = new NetworkIFContext(this)
   ```

   The data member does have to be `private` but may be `public` or `protected`. Also the variable name does not have to be `_fsm`.

2. If the start state has entry actions which must be executed prior to issuing any transitions, add the following code *outside* the `NetworkIF` constructors:

   ```
   _fsm.enterStartState()
   ```

3. Wherever you want to issue a transition, all you need to do is:

   ```
   _fsm.<transition>(...)
   ```
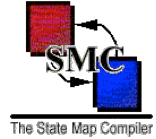
   where `<transition>` is the transition name. If the transition takes arguments, then pass them in the transition call:

   ```
   _fsm.Connect("192.168.3.100", 80)
   ```

4. All state machine actions may be implemented in `NetworkIF` as either `public` or package-private methods, so long as the SMC-generated code can access these methods.

package in NetworkIF.sm

If you need to place the SMC-generated classes into a Scala a package, then see Section 8.
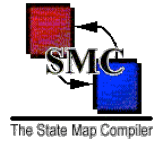
import in NetworkIF.sm

If you need to use `import` in your .sm file, use the `%import` keyword in the top of your .sm file:

```
%import scala.concurrent
%import java.net.InetAddress
%import java.util.Calendar
```

JavaScript

Assumptions

- You have an JavaScript class named `NetworkIF` defined in the file NetworkIF.js
- The state machine is in NetworkIF.sm.
- You have successfully compiled NetworkIF.sm.

Changes to NetworkIF.js

Adding the state machine to `NetworkIF` requires the following changes to NetworkIF.js:

1. In the `NetworkIF` constructor, add the line:

   ```
   this._fsm = new NetworkIF_sm(this);
   ```

2. If the start state has entry actions which must be executed prior to issuing any transitions, add the following code *outside* the `NetworkIF` constructors:

   ```
   this._fsm.enterStartState();
   ```

3. Wherever you want to issue a transition, all you need to do is:

   ```
   this._fsm.<transition>(...);
   ```

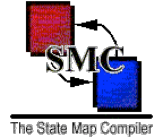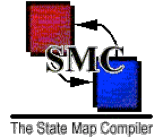   where *`<transition>`* is the transition name. If the transition takes arguments, then pass them in the transition call:

   ```
   this._fsm.Connect("192.168.3.100", 80);
   ```
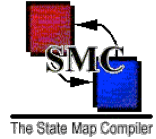
Section 4: Compiling a .sm

These instructions assume that:

- Java 1.7 or newer is properly installed, `javac`, `java` and `jar` executables are in you `PATH` environment variable. The standard Java Development Kit (JDK) may be obtained for *free* from Oracle.
- The `SMC_HOME` environment variable contains the path to where SMC is installed.

The following table explains the default file name and suffix used for each target language supported by SMC. The default file name suffix may be changed by using the `-suffix` and `-hsuffix` command line parameters. By default, SMC places the generated file(s) in the current working directory. This may be altered using the `-d` and `-headerd` command line parameters.

The following table assumes that the input file is named `AppClass.sm`:

Target Language	Command Line Option	Base File Name	Default File Name Suffix	Complete File Name
C	-c	AppClass_sm	.h, .c	AppClass_sm.h AppClass_sm.c
C++	-c++ (including -crtp)	AppClass_sm	.h, .cpp	AppClass_sm.h AppClass_sm.cpp
C#	-csharp	AppClass_sm	.cs	AppClass_sm.cs
GraphViz	-graph	AppClass_sm	.dot	AppClass_sm.dot
Groovy	-groovy	AppClassContext	.groovy	AppClassContext.groovy
Java	-java, -java7	AppClassContext	.java	AppClassContext.java
JavaScript	-js	AppClass_sm	.js	AppClass_sm.js
Lua	-lua	AppClass_sm	.lua	AppClass_sm.lua

Target Language	Command Line Option	Base File Name	Default File Name Suffix	Complete File Name
Objective-C	-objc	AppClass_sm	.h, .m	AppClass_sm.h AppClass_sm.m
Perl	-perl	AppClass_sm	.pm	AppClass_sm.pm
PHP	-php	AppClass_sm	.php	AppClass_sm.php
Python	-python	AppClass_sm	.py	AppClass_sm.py
Ruby	-ruby	AppClass_sm	.rb	AppClass_sm.rb
Scala	-scala	AppClassContext	.scala	AppClassContext.scala
HTML Table	-table	AppClass_sm	.html	AppClass_sm.html
Tcl	-tcl	AppClass_sm	.tcl	AppClass_sm.tcl
VB.net	-vb	AppClass_sm	.vb	AppClass_sm.vb

Table 2: SMC target languages.

The steps in compiling a .sm file are:
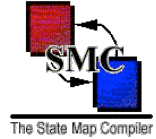
1. From a shell (Unix or Windows CMD), go to the directory containing the .sm file.
2. `$java -jar $SMC_HOME/bin/Smc.jar <target language option> <fsm_source_file>.sm`

If you are using `ant` to build your application, then you can use `smc-ant.jar` to compile you .sm file using the following target:

```
<taskdef classname="net.sf.smc.ant.SmcJarWrapper" ⇒
    classpath="${smc-ant.jar}" name="smc-compile" />
<smc-compile smcjar="${smc.jar}" classpath="${smc.classpath}" ⇒
    target="java7" destdir="${src.dir}/appclass/" smfile="${AppClass.sm}" />
```

where `${smc-ant.jar}` references the location of `smc-ant.jar`, `${smc.jar}` references the location of `Smc.jar`, and `${smc.classpath}`is a Java classpath containing the path to `statemap.jar`. The attribute `target="<target language>"` must be one of the support target languages listed in the table above.

At a minimum, the SMC command line must contain either `-<target language>`, `-help`, or `-version`. All other command line options are optional. These optional options are:

Option	Target Language	Description
`-access <level>`	`-java` `-java7`	Generate SM class access level.
`-suffix <suffix>`	All	Use this suffix for the output file.
`-g, -g0`	All	Add level 0 debugging output to generated code.
`-g1`	All	Add level 1 debugging output to generated code.
`-nostreams`	`-c++`	Do not use C++ iostreams for debugging output.
`-crtp`	`-c++`	Generate state machine using CRTP.
`-verbose`	All	Output compiler messages (SMC is silent by default)
`-vverbose`	All	Output more compiler messages.

`-sync`	`-csharp`
	`-groovy`
	`-java`
	`-java7`
	`-scala`
	`-vb`
		Synchronize access to transition methods. Necessary to make generated code thread safe.

`-noex`	`-c++`	Do not generate C++ exception throws.
`-nocatch`	All	Do not generate try/catch/re-throw code (not recommended).
`-stack <max-depth>`	`-c++`	Specifies a fixed-size state stack, using no dynamic memory allocation.
`-protocol`	`-objc`	FSM context extends a `@protocol` and referenced via protocol.

Option	Target Language	Description
-serial	All	Generate serialization code.
-return	All	`Smc.main()` returns instead of exiting. Use this option with `ant`.
-reflect	-csharp, -groovy, -java, -java7, -js, -lua, -perl, -php, -python, -ruby, -scala, -tcl, -vb	Generate reflection code.
-generic	-csharp, -java, -java7, -vb	Use generic collections.
-generic7	-java7	Use Java 7 generic collections.
-cast *<type>*	-c++	Use this C++ cast type.
-d *<directory>*	All	Placed generated files in *<directory>*.
-headerd *<directory>*	-c, -c++, -objc	Placed generated header files in *<directory>*.
-hsuffix *<suffix>*	-c, -c++, -objc	Add this suffix to output header files.
-glevel *<n>*	-graph	Detail level from 0 (least) to 2 (greatest).

Table 3: SMC command line options.

Section 5: Behind the Curtain

SMC is so straight forward and easy to use, that it is unnecessary to know anything about the code it generates in order to use it. But for the curious, here is an explanation.

The SMC-generated code follows the State pattern as described in Gamma, Helm, Johnson and Vlissides "Design Patterns" book (pp. 305 to 313):
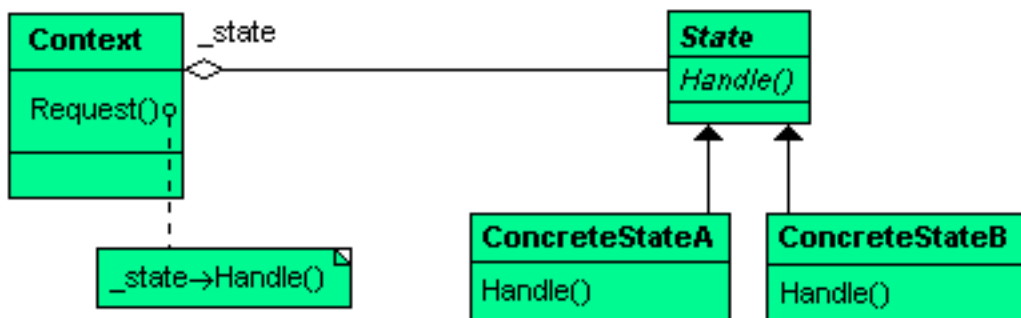
The SMC State pattern is different due to support for multiple machines, a default state, default transition and push/pop transitions:
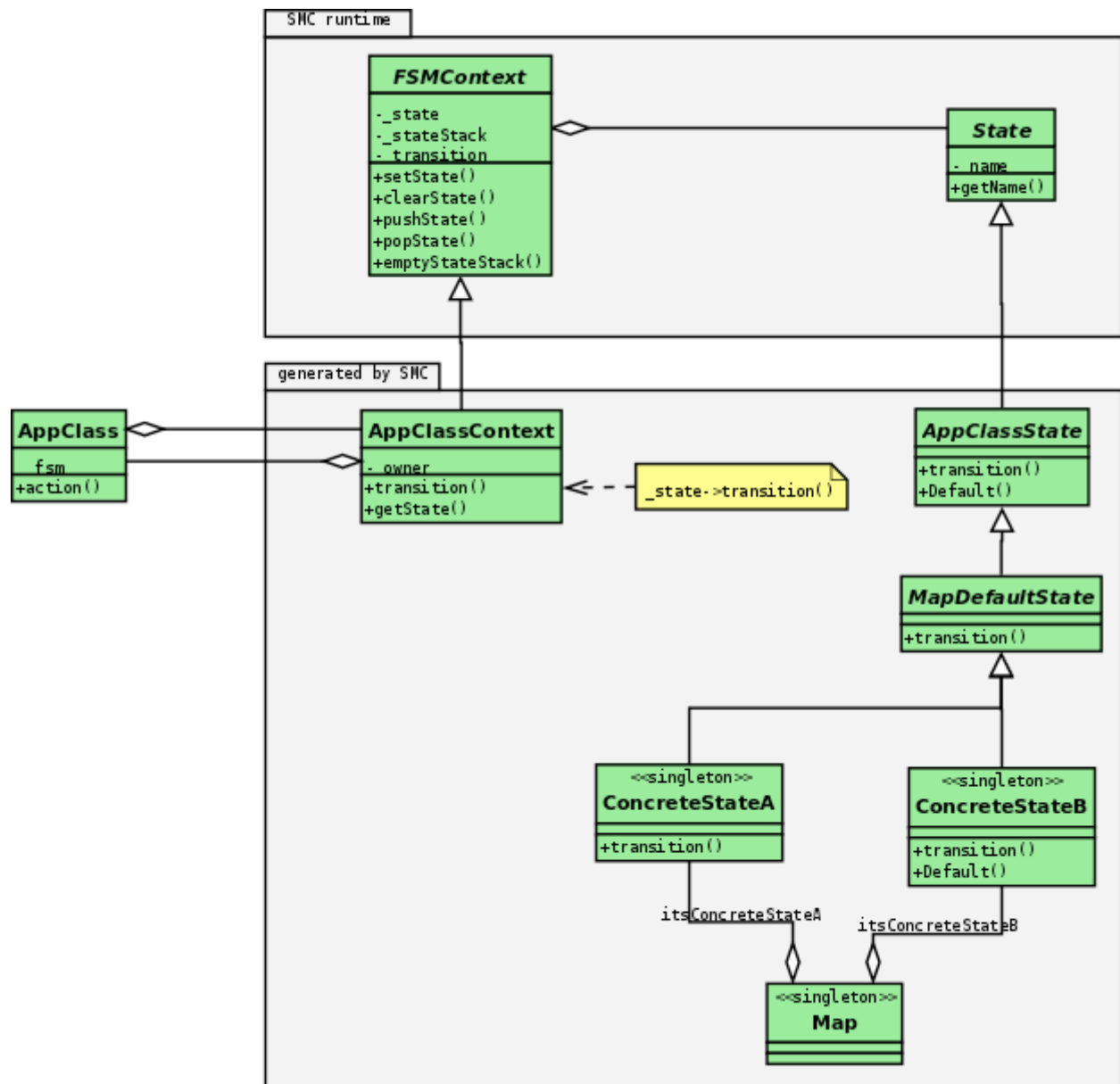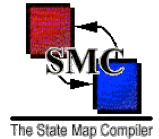
SMC deviates from the State pattern as follows:

- The Context class was broken into two classes: an abstract FSMContext class (not generated but provided with the SMC compiler) and `<AppClass>Context` class.

FSMContext stores both the current state and the state stack (used for pushing and popping states). FSMContext also defines methods for setting the state, and pushing and popping states.

*<AppClass>*Context inherits from FSMContext, defines the getState() (which returns the current state as a *<AppClass>*State object and not a State object - that is why this method is not in FSMContext), provides access to all transitions defined in all machines and maintains a reference back to its owner *<AppClass>* object.

- The state pattern has an abstract State class and ConcreteState classes which inherit from State. SMC expands this hierarchy to four levels: State, *<AppClass>*State, Map Default state and concrete states.

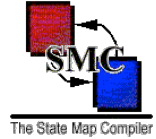 This hierarchy is used to support SMC's default transitions. *<AppClass>*State has a virtual method for each transition appearing in all state machines. These transition methods call *<AppClass>*State's Default transition method. This global Default transition throws a "Transition Undefined" exception when called.

  The map default state class contain the Default state's transitions.

  Each concrete state is a class which inherits from its map's default state class. The state class methods implement state machine transitions.

  Each map class statically declares one instance of each of its states since concrete state classes are singletons. The map class has no methods and is itself a singleton. The map class' purpose is to gather a map's state instances into a single location.

While SMC generates many classes, they take up little run time space. There is only one instance of each concrete state class. Only one *<AppClass>*Context class need be instantiated for each *<AppClass>* class instance.

Finally, the SMC State pattern is hidden from the application class. All a developer needs to do is instantiate *<AppClass>*Context, passing to it the *<AppClass>* object. After that, it's only a matter of calling the Context object's transition methods.

For further examples on using the State Machine Compiler, see <u>Section 6: For Example ...</u> .

Section 6: For Example …

The State Machine Compiler download contains both simple and complex examples using all SMC features - simple transitions, default states, default transitions, state entry and exit actions and pushing and popping. Each example is presented in C, C++, Java, [incr Tcl], VB.net, C#, Groovy, Lua, Python, Perl, PHP, Ruby and Scala. The first examples are the simplest and gradually become more complex.

These examples are in SMC's "example" directory. All examples contain README text files describing what the example does, how to build it and run it.

Example 1

Describes a system that determines whether a string is acceptable or not. The intent is to process a string of bits having a specific pattern "0*1*". That is:

- Anything other than a 0 or a 1 will trigger an `Unknown` transition, producing an Unacceptable result.
- Once a 1 is encountered, a 0 may no longer appear.

The `Error` state is designed specifically to collect unacceptable strings. Unknown transitions (regardless of state) and transitions from 1 to 0 both end up in the final `Error` state.

We can walk through the bit string 0000111 which is acceptable to gain some understanding of the FSM's function:

1. Start in the `Start` state and walk the bits from left to right.
2. 0 is the first transition, moving to the `Zero` state.
3. Subsequent 0's keep the FSM in the `Zero` state.
4. The first 1 transitions the FSM to the `One` state.
5. Additional 1's keep the FSM in the `One` state.
6. Finally the `EOS` transition is issued when the bit string end is reached. `EOS` transitions the FSM to the final `OK` state.

The `EOS` transition has an `Acceptable` action which sets `AppClass`'s boolean `acceptable` property to true. Conversely, the `EOS` transition in `Error` state sets `acceptable` to false (unacceptable).

Both `OK` and `Error` states are final. Once entered, there is no way to transition out of the error state.

Example 2

This example refactors example 1. Notice that the same `Unknown` transition applies for all states except `OK` state. Rather than define `Unknown` repeatedly for every state, this transition is placed in the `Default` state which acts as a fallback mechanism. No if an `Unknown` transition is encountered in any state, the `Default` state's `Unknown` transition is taken. It doesn't matter is this means adding an `Unknown` transition to the `OK` state because we have already reached the string's end at that point. Technically the `OK` state is no longer a final state but practically it still is.

Example 3

This example is the textbook demonstration of a push-down finite state machine. This FSM checks for palindromes using a three character alphabet {0, 1, c}. Palindromes read the same way from left to right as they do from right to left (case-insensitively):

Able was I ere I saw Elba

Each side is a mirror image of the other. The 'c' is used to demarcate the middle point (palindromes are odd).

This examples shows SMC push and pop transitions in action. It contains three `%maps`. The basic behavior of this FSM is to keep pushing the next bit encountered onto the stack until the middle point is reached. The stack acts as a sort of memory for the bits encountered. If you're in `Map1` that means you've been encountering zeros. If you're in `Map2`, that means you've been encountering ones. When you get to the center ('c'), it's a signal to start popping or consuming the stack. When we were pushing, we were always in a map's `PushIt` state; now that we're popping we move to the map's `PopIt` state.

To see this in action is to turn on debugging by uncommenting the `setDebugFlag` line in AppClass.java. For strings that at least are in the alphabet (0, 1, c) and that at least have the same number of characters to each side of 'c', you should see that number of pushes follows by the same number of pops.

So this dictates the transitions for both maps:

- In `ZerosMap` if a zero is popped, that is OK. Otherwise, it is an error.
- In `OnesMap` if a one is popped, that is OK. Otherwise, it is an error.

That is the gist of this exercise.

Notice a small nuance in the code: there's an action called `emptyStack`. Typically all action methods are defined in `AppClass` but you'll see no such method in `AppClass`. `emptyStack` is a reserved action method name defined in `FSMContext`. SMC looks for such reserved names and makes sure to generate source code that reroutes such calls to context (the `FSMContext` class) instead of `ctxt` (the `AppClass`).

Example 7

Java, Ant, Python, Perl and Ruby example EX7 uses the push/pop transition in a more realistic way. `%map PhoneNumber` is used to collect the dialed telephone number. If the dialed number was acceptable, then the `PhoneNumber` map pops the transition:

```
pop(DialingDone, callType, areaCode, exchange, local)
```

where `DialingDone` is transition name and `callType`, `areaCode`, `exchange`, and `local` are transition parameters. The push transition itself is different:

```
Dialing/push(PhoneNumber::DialTone)
```

This has the FSM first go to the `Dialing` state and then push to the `PhoneNumber::DialTone` state. The popped `DialingDone` or `InvalidDigit` transition is taken from the `Dialing` state.

PHP Example

PHP example web is an example for building stateless dynamic web pages with a state machine. It implements a simple RPN calculator which preserves its state and calculation stack across HTTP requests with a hidden input field.

Section 7: Queuing Up

SMC does not allow a transition to issue a transition itself. The reason for this is explained in SMC's FAQ. Firstly, it is suggested that your `AppClass` issue transitions from a single thread. Doing so means that a second transition call cannot be issued until the first transition call returns.

But if AppClass will be accessed by multiple threads and so issue transition calls from multiple threads, then you must synchronize the transition calls so that only one transition is taken at a time. The simplest way to do this is by SMC's `-sync` command line option. This uses a programming language feature to synchronize access to the FSM transition method. This option is only available for C#, VB.net , Java (-java and -java7), Groovy, and Scala.

For all other languages, the developer must implement a multi-threaded protection scheme for transition method access.

Section 8: Packages and Namespaces

Putting the SMC-generated classes into a specific Java package/C++ namespace/Tcl namespace/Groovy package/Perl package/Ruby module/Scala package is as easy placing the `%package` keyword at the top of your .sm file along with the `%class` keyword.

(Aside: Remember to place `namespace import <::namespace::*>` after sourcing in a *_sm.tcl file.)

To import a C++ namespace, Java class, Tcl package, VB.net, C# namespace, Groovy class, Lua module, Python module, Perl module, PHP file, Ruby module or Scala class, use the `%import` keyword:

`%import <name>`

(**Note:** There is *no* semicolon at the line's end.)

The `%import` is translated into the following syntax:

- **C++:** `using namespace <name>;`
- **Java:** `import <name>;`
- **Tcl:** `package require <name>`
- **VB.net:** `Imports <name>`
- **C#:** `using <name>`
- **Groovy:** `import <name>`
- **Lua:** `require '<name>'`
- **Python:** `import <name>`
- **Perl:** `use <name>;`
- **PHP:** `require_once '<name>';`
- **Ruby:** `require '<name>'`
- **Scala:** `import <name>`

See examples/Java/EX4, C++/EX4, Tcl/EX4, VB/EX4, CSharp/EX4, Perl/EX4 and Ruby/EX4 for sample code which uses packages/namespaces with SMC.

Fully Qualified Class Names

If you are not able to place the SMC-generated classes into the same package/namespace as the `%class` context class, then you will need to specify the context class' fully qualified name. The SMC `%class` key supports Java, C++ and Tcl fully qualified name syntax. For example, the following are accepted:

```
// For a Java, VB.net and C# application
%class com.acme.project.AppClass
```

and

```
// For a C++ or Tcl application
%class ::acme::project::AppClass
```

Section 9: Be Persistent!

This section describes how to persist an SMC-generated finite state machine and restore it at a later date. In the following examples I persist the FSM to a flat file but they can be modified to work with other storage types. The focus is only capturing the FSM's current state and state stack. There is no other data to persist in the SMC-generated code.

The examples use the class `AppClass` which has an associated finite state machine stored in its data member _fsm.

Note: the following sample code requires the .sm file be compiled with the `-serial` option.

If you are *not* using push/pop transitions, then the use the following code to persist the current state:

```cpp
int
AppClass::serialize(const char *filename) const {
    int fd;
    int stateId(_fsm.getState().getId());
    int retcode(-1);

    fd = open(filename,
            (O_WRONLY | O_CREAT | O_TRUNC),
            (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH));
    if (fd >= 0) {
        retcode = write(fd, &stateId, sizeof(int));

        (void) close(fd);
        fd = -1;
    }

    return (retcode);
}
```

Deserializing is the mirror:

```
int AppClass::deserialize(const char *filename) const {
    int fd;
    int stateId;
    int retcode(-1);

    // This code assumes _fsm is already instantiated.

    fd = open(filename, O_RDONLY);
    if (fd >= 0) {
        retcode = read(fd, &stateId, sizeof(int));
        if (retcode >= 0) {
            _fsm.setState(_fsm.valueOf(stateId));
        }

        (void) close(fd);
        fd = -1;
    }

    return (retcode);
}
```

If you *are* using push/pop transitions, then the serialization will be require your to persist the state stack in reverse order (bottom to top) followed by the current state.

Warning! Reading in the state stack results in emptying the stack and corrupting the FSM. This should not be a problem because you are persisting the FSM for use later when you will restore the state stack.

```
int AppClass::serialize(const char *filename) const {
    int fd;
    int retcode(-1);

    fd = open(filename,
              (O_WRONLY | O_CREAT | O_TRUNC),
              (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH));
    if (fd >= 0) {
        int size(_fsm.getStateStackDepth() + 1);
        int bufferSize(size * sizeof(int));
        int buffer[size + 1];
        int i;

        // Copy the states into the buffer in reverse order:
        // from the state stack bottom to the top and the current
        // state last. The first element is the number of states.
        buffer[0] = size;
        buffer[size] = (_fsm.getState()).getId();
        for (i = (size - 1); i > 0; --i) {
```

```
            _fsm.popState();
            buffer[i] = (_fsm.getState()).getId();
        }

        retcode = write(fd, buffer, (bufferSize + sizeof(int)));

        (void) close(fd);
        fd = -1;
    }

    return (retcode);
}
```

When reading in the persisted FSM, first read the state count and then read in the states:

```
int AppClass::deserialize(const char *filename) {
    int fd;
    int size;
    int retcode(-1);


    // The FSM's current state is probably set to the start state. Clear it
    // out because it is not correct.
    _fsm.clearState();

    // Open the file for reading and then read in the number of persisted
    // states.
    fd = open(filename, O_RDONLY);
    if (fd >= 0 && read(fd, &size, sizeof(int)) >= 0) {
        int bufferSize(size * sizeof(int));
        int buffer[size];

        if (read(fd, buffer, bufferSize) >= 0) {
            int i;

            // Note: Do not call setState for the final current state because
            // pushState actually sets the current state while pushing the
            // next state on the stack.
            for (i = 0; i < size; ++i) {
                _fsm.pushState(_fsm.valueOf(buffer[i]));
            }
        }
    }

    // Make sure the file is closed before leaving.
    if (fd >= 0) { (void) close(fd); }

    return (retcode);
}
```

SMC makes full use of Java's object serialization. Assuming that `AppClass` declares `implements java.io.Serializable` and that the `_fsm` member data is *not* marked as `transient`, then serializing the `AppClass` instance also serializes the FSM.

SMC considers the code it generates to be subservient to the application code. For this reason the SMC code does not serialize its references to the FSM context owner or property listeners. The application code after deserializing the FSM *must* call the `setOwner` method to re-establish the application/FSM link. If the application listens for FSM state transitions `addStateChangeListener` must also be called to put the listeners in place.

```java
public final class AppClass implements java.io.Serializable {
    private AppClassContext _fsm;


    public AppClass() { _fsm = new AppClassContext(); }

    // Restore the FSM context's reference to this object. This is necessary
    // because AppClass and AppClassContext reference each other. When
    // AppClass is serialized, its AppClassContext instance is serialized.
    // But when AppClassContext is serialized, its AppClass instance is not
    // serialized which breaks the circular reference. When AppClassContext
    // is deserialized, its AppClass reference is null. Hence the need to
    // implement readObject and reset AppClassContext's AppClass reference.
    // The state change property listeners list is empty for the same reason.
    private void readObject(java.io.ObjectInputStream istream)
        throws java.io.IOException, ClassNotFoundException
    {
        // Do the default read first which sets _fsm to null.
        istream.defaultReadObject();

        // Now set the FSM's owner.
        _fsm.setOwner(this);

        // State change listeners must also be added back.
        _fsm.addStateChangeListener(_stateListener);

        return;
    }
```
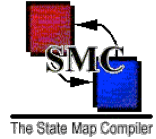
```
    public static void main(String[] args) {
        Serialize();
        Deserialize();
    }

    public static void Serialize() {
        AppClass appInstance = new AppClass();
        ObjectOutputStream ostream =
            new ObjectOutputStream(new FileOutputStream("./fsm_serial.bin"));

        try {
            ostream.writeObject(appInstance);
        } catch (java.io.IOException ioex) {
            // Handle serialization exception.
        } finally {
            ostream.close();
        }
    }

    ...
}
```

Recreating the persisted AppClass instance is equally simple:

```
    public static void Deserialize() {
        AppClass appInstance = null;
        ObjectInputStream istream =
            new ObjectInputStream(new FileInputStream("./fsm_serial.bin"));

        try {
            appInstance = (AppClass) istream.readObject();
        } catch (java.io.IOException ioex) {
            // Handle deserialization exception.
        } finally {
            istream.close();
        }
    }
```
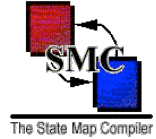
Note: the following sample code requires the .sm file is compiled with the `-serial` option.

Tcl persistence is similar to C++. If you are not using push/pop transitions, then the use the following code to persist the current state:

```tcl
public method serialize {fileName} {
    if [catch {open $fileName w 0644} fileId] {
        set retcode error;
        set retval "Failed to open ${filename} for writing";
    } else {
        puts $fileId [[$_fsm getState] getId];
        close $fileId;

        set retcode ok;
        set retval "";
    }

    return -code ${retcode} ${retval};
}
```

The following deserializes the current state:

```tcl
public method deserialize {fileName} {
    if [catch {open $fileName r} fileId] {
        set retcode error;
        set retval "Failed to open ${filename} for reading";
    } else {
        gets $fileId stateId;
        $_fsm setState [$_fsm valueOf $stateId];

        close $fileId;

        set retcode ok;
        set retval "";
    }

    return -code ${retcode} ${retval};
}
```

If you *are* using push/pop transitions, then the serialization will be require your to persist the state stack in reverse order (bottom to top) followed by the current state.

Warning! Reading in the state stack results in emptying the stack and corrupting the FSM. This should not be a problem because you are persisting the FSM for use later when you will restore the state stack.

```
public method serialize {fileName} {
    if [catch {open $fileName w 0644} fileId] {
        set retcode error;
        set retval "${fileName} open failed";
    } else {
        set state [$_fsm getState];
        set states {};

        lappend states [$state getId];
        while {[catch {$_fsm popState} retcode] == 0} {
            set state [$_fsm getState];
            set states [linsert $states 0 [$state getId]];
        }

        set size [llength $states];
        puts $fileId $size;

        foreach stateId $states {
            puts $fileId $stateId;
        }

        close $fileId;

        set retcode ok;
        set retval "";
    }

    return -code ${retcode} ${retval};
}
```

When reading in the persisted FSM, first read the state count and then read in the states:
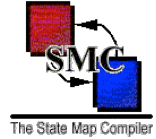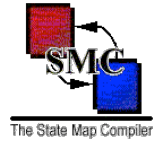
```
public method deserialize {fileName} {
    if [catch {open $fileName r} fileId] {
        set retcode error;
        set retval "${fileName} open failed";
    } else {
        # Clear out the default start state.
        $_fsm clearState;

        gets $fileId size;
        for {set i 0} {$i < $size} {incr i} {
            gets $fileId stateId;
            set state [$_fsm valueOf $stateId];

            $_fsm pushState $state;
        }

        close $fileId;
        set retcode ok;
        set retval "";
    }

    return -code ${retcode} ${retval};
}
```

VB.net

SMC makes full use of .net's object serialization. Assuming that `AppClass` has the `<Serializable()>` attribute and that the _fsm member data is not marked as `<NonSerializable()>`, then serializing the `AppClass` instance will also serialize the FSM.

SMC considers the code it generates to be subservient to the application code. For this reason the SMC code does not serialize its references to the FSM context owner or property listeners. The application code after deserializing the FSM *must* call the `Owner` property setter to re-establish the application/FSM link. If the application listens for FSM state transitions, then event handlers must also be put back in place.

```vbnet
Imports System
Imports System.IO
Imports System.Runtime.Serialization
Imports System.Runtime.Serialization.Formatters.Binary

<Serializable()> Public Class AppClass
    Implements IDeserializationCallback

    Private _fsm As AppClassContext

    Public Sub New()
        _fsm = New AppClassContext(Me)
    End Sub

    ' Restore the FSM context's reference to this object. This is necessary
    ' because AppClass and AppClassContext reference each other. When
    ' AppClass is serialized, its AppClassContext instance is serialized. But
    ' when AppClassContext is serialized, its AppClass instance is not
    ' serialized which breaks the circular reference. When AppClassContext is
    ' deserialized, its AppClass reference is Nothing. Hence the need to
    ' implement IDeserialization and reset AppClassContext's AppClass
    ' reference. Event handlers must also be put back in place for the same
    ' reason.
    Private Sub OnDeserialization(ByVal send As Object) _
        Implements IDeserializationCallback.OnDeserialization

        _fsm.Owner = Me
        AddHandler _fsm.StateChange, handler
    End Sub

    Shared Sub Main()
        Serialize()
```
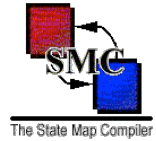
```vb.net
        Deserialize()
    End Sub

    Shared Sub Serialize()
        Dim appInstance As New AppClass()
        Dim stream As Stream = File.Open("fsm_serial.bin", FileMode.Create)
        Dim formatter As New BinaryFormatter()

        Try
            formatter.Serialize(stream, appInstance)
        Catch serialex As SerializationException
            ' Handle serialization failure.
        Finally
            stream.Close()
        End Try
    End Sub

    ...
End Class
```

Recreating the persisted `AppClass` instance is equally simple:

```vb.net
    Shared Sub Deserialize()
        Dim appInstance As AppClass = Nothing
        Dim stream As Stream = File.Open("fsm_serial.bin", FileMode.Open)
        Dim formatter As New BinaryFormatter()

        Try
            appInstance = CType(formatter.Deserialize(stream), AppClass)
        Catch serialex As SerializationException
            ' Handle deserialization failure.
        Finally
            stream.Close()
        End Try
    End Sub
```
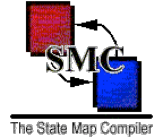
C#

SMC makes full use of .net's object serialization. Assuming that `AppClass` has the `[Serializable]` attribute and that the _fsm member data is not marked as `[NonSerialized]`, then serializing the `AppClass` instance will also serialize the FSM.

SMC considers the code it generates to be subservient to the application code. For this reason the SMC code does not serialize its references to the FSM context owner or property listeners. The application code after deserializing the FSM *must* call the `Owner` property setter to re-establish the application/FSM link. If the application listens for FSM state transitions, then event handlers must also be put back in place.
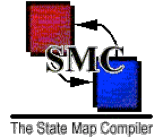
```csharp
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

[Serializable] public class AppClass : IDeserializationCallback
{
    private AppClassContext _fsm;

    public AppClass() {
        _fsm = new AppClassContext(this);
    }

    // Restore the FSM context's reference to this object. This is necessary
    // because AppClass and AppClassContext reference each other. When
    // AppClass is serialized, its AppClassContext instance is serialized.
    // But when AppClassContext is serialized, its AppClass instance is not
    // serialized which breaks the circular reference. When AppClassContext
    // is deserialized, its AppClass reference is null. Hence the need to
    // implement IDeserialization and reset AppClassContext's AppClass
    // reference. Event handlers must also be put back in place for the same
    // reason.
    void IDeserializationCallback.OnDeserialization(Object sender) {
        _fsm.Owner = this;
        _fsm.StateChange += handler;
    }

    static void Main(string [] args) {
        Serialize();
        Deserialize();
    }
```

```
static void Serialize() {
    AppClass appInstance = new AppClass();
    FileStream fstream =
        new FileStream("fsm_serial.dat", FileMode.Create);
    BinaryFormatter formatter = new BinaryFormatter();

    try {
        formatter.Serialize(fstream, appInstance);
    }
    catch (SerializationException serialex) {
        // Handle exception.
    }
    finally {
        fstream.Close();
    }
}

...
}
```

Recreating the persisted `AppClass` instance is equally simple:

```
static void Deserialize() {
    AppClass appInstance = null;
    FileStream fstream = new FileStream("fsm_serial.dat", FileMode.Open);
    BinaryFormatter formatter = new BinaryFormatter();

    try {
        appInstance = (AppClass) formatter.Deserialize(fstream);
    }
    catch (SerializationException serialex) {
        // Handle exception.
    }
    finally
    {
        fstream.Close();
    }
}
```
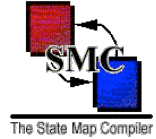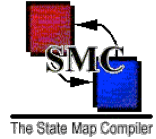
Section 10: Get the Picture

The SMC option **-graph** generates Graphviz DOT files using three detail levels:

◉ **-glevel 0:** Generates the *least* detail:
- state names,
- transition names, and
- pop transition "nodes" only.

◉ **-glevel 1:** Generates all of the above plus:
- entry and exit actions,
- transition guards, and
- transition actions.

◉ **-glevel 2:** Generates the *most* detail, which includes all of the above plus:
- entry and exit action arguments,
- transition parameters,
- pop transition arguments, and
- transition action arguments.

SMC generates a rudimentary DOT file - SMC makes no attempt to produce a "pretty" graph because beauty is in the eye of the beholder. The default Graphviz settings are used. You can then modify the DOT file to your heart's content.

Note: Be careful modifying the SMC-generated DOT file directly. If you have SMC generate a new DOT file, it will overwrite your work.

Stroll through the SMC Picture Gallery to see examples of SMC-generated DOT files and what can be done with them.

Section 11: On Reflection

SMC v. 4.3.0 introduces the **-reflect** command line option for the Java, C#, Perl, PHP, Python, Ruby, Tcl and VB.Net programming languages. This option tells SMC to generate for each state class either a `getTransitions` method or a `Transitions` property for C# and VB.Net. In both cases the method/property returns the state's defined transitions as a map. The map key is a transition name with an integer value.

SMC v. 6.0.1 adds the ability to retrieve a list of all states in the finite state machine.

The method definitions are:

- **C#:** The top-level context class contains the get-only property:

  `public <context>State[] States`

  For each of the states, there is the get-only property:

  `public System.Collections.Dictionary Transitions`

  where the key is a string and the value is an int.

  Note: if the **-generic** option is used, then SMC emits:

  `public System.Collections.IDictionary<string, int> Transitions`

- **Groovy:** The top-level context class contains:

  `final states = [ ... ]`
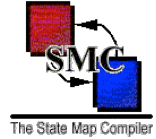
  which is a state array.

  `final transitions = [ ... ]`

  which is a transition name array.

  For each of the states (meaning the following array is contained in the state object), the transitions map is defined:

  `final transitions = [<transition>:<integer value>]`

- **Java:** The top-level context class contains the method:

  `public <context>State[] getStates()`

  For each state, there is the method:

  `public Map getTransitions()`

  where the map key is a `String` and the map value is an `Integer`.

  Note: if the **-generic** option is used, then SMC emits:

  `public Map<String, Integer> getTransitions()`

  When **-generic7** is used, then SmC emits the **<>** braces when instantiating generic collection objects.

  The top-level context class has a method to return the set of all transitions within the finite state machine:

  `public Set getTransitions()` or `public Set<String> getTransitions()`

- **Lua:** The top-level context class contains the methods:

  `function getStates()`

  which returns a state array.

  `function getTransitions()`

  which returns a transition name array.

  For each of the states (meaning the following method is contained in the state object), the transitions map is defined:

  `function getTransitions()`

  which returns a table where the key is a string and the value is an integer.

- **Perl:** The top-level context class contains the methods:

  `sub getStates`

  which returns a state array.

  `sub getTransitions`

  which returns a transition name array.

  For each of the states (meaning the following method is contained in the state object), the transitions map is defined:

  `sub getTransitions`

  which returns a hash where the key is a string and the value is an integer.

- **PHP:** The top-level context class contains the methods:

```
public function getStates()
```
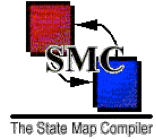
which returns a state array.

```
public function getTransitions()
```

which returns a transition name array.

For each of the states (meaning the following method is contained in the state object), the transitions map is defined:

```
public function getTransitions()
```

which returns an associative array where the key is a string and the value is an integer.

- **Python:** The top-level context class contains the methods:

```
def getStates(self)
```

which returns a state array.

```
def getTransitions(self)
```

which returns a transition name array.

For each of the states (meaning the following method is contained in the state object), the transitions map is defined:

```
def getTransitions(self)
```

which returns a dict where the key is a string and the value is an integer.

- **Ruby:** The top-level context class contains the methods:

```
def getStates()
```
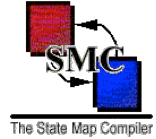
which returns a state array.

```
def getTransitions()
```

which returns a transition name array.

For each of the states (meaning the following method is contained in the state object), the transitions map is defined:

```
def getTransitions()
```

which returns a dict where the key is a string and the value is an integer.

- **Scala:** The top-level context class contains the methods:

  ```
  def getStates(): List[<context>State]
  ```

  which returns a state list.

  ```
  def getTransitions(): List[String]
  ```

  which returns a transition name list.

  For each of the states (meaning the following method is contained in the state object), the transitions map is defined:

  ```
  def getTransitions(): Map[String, int]
  ```

  which returns a map where the key is a string and the value is an integer.

- **[incr Tcl]**: The top-level context class contains the methods:

  ```
  public method getStates{}
  ```

  which returns a state instances array.

  ```
  public method getTransitions {}
  ```

  which returns a transition name array.

- **VB.net:** The top-level context class contains the get-only property:

  ```
  Public Property States() As <context>State()
  ```

  For each of the states, there is the get-only property:

  ```
  Public ReadOnly Property Transitions() As IDictionary
  ```

  where the key is a string and the value is an int.

  Note: if the **-generic** option is used, then SMC emits:

  ```
  Public ReadOnly Property Transitions() As IDictionary(Of String, Integer)
  ```
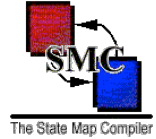
The transition's associated integer value is limited to:

- 0: The transition is undefined in the current state.
- 1: The transition is defined in the current state.
- 2: The transition is defined in the default state.

This allows an application to discover which transitions a state supports.

Note: the returned transition names map includes the `Default` transition.

This feature is most useful for user interface developers who need to activate and deactivate features based on the current state. The user's actions are limited to the current state's transitions.

Reminder: You cannot call `getState()` while in a transition because the state is not set. If need to determine which state you just left while in transition. call `getPreviousState()`.
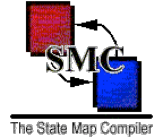
Java Sample

```java
import java.util.Iterator;
import java.util.Map;

public void processEvent(EventObject event) {
    // Transform this event into a FSM transition:
    if (event instanceof OpenEvent) {
        _fsm.openFile((OpenEvent) event);
    }
    else if ... {
    }

    // Now figure out what features are active and inactive based on the
    // current state's returned transition map.
    Map<String, Integer> transitions = (_fsm.getState()).getTransitions();
    String transition;
    int status;

    for (Map.Entry<String, Integer> entry: transitions.entrySet()) {
        transition = entry.getKey();
        status = (entry.getValue()).intValue();

        // Ignore the Default transition.
        if (transitions.equals("Default") == true) {
            // no-op
        }
        // 0 - transition is undefined.
        // 1 - transition is explicity defined in the state.
        // 2 - transition is defined in Default state.
        else if (status == 0) {
            // Deactivate feature.
        }
        else if (status == 1) {
            // Activate feature.
        }
        else {
            // Do something else for default definitions.
        }
    }
}
```

C# Sample

```csharp
using System.Collections.Generic;

public void processEvent(Object event) {
    // Transform this event into a FSM transition:
    if (event is OpenEvent) {
        _fsm.openFile(event);
    }
    else if ...

    // Now figure out what features are active and inactive based on the
    // returned transition map.
    IDictionary<string, int> transitions = (_fsm.getState()).Transitions;
    int status;

    foreach (string name in transitions.Keys) {
        status = transitions[name];

        // Ignore the Default transition.
        if (name == "Default") {
            // no-op
        }
        // 0 - transition is undefined.
        // 1 - transition is explicity defined in the state.
        // 2 - transition is defined in Default state.
        else if (status == 0) {
            // Deactivate feature.
        }
        else if (status == 1) {
            // Activate feature.
        }
        else {
            // Do something else for default definitions.
        }
    }
}
```
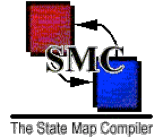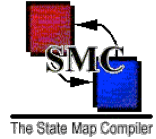
[incr Tcl] Sample

```tcl
Using System.Collections

public method ProcessEvent{event} {
    # Transform this event into a FSM transition:
    if {$event = "open"} {
        _fsm.Open()
    } elseif {...} {
        ...
    }

    # Now figure out what features are active and inactive based on the
    # returned transition map.
    foreach {name status} [[_fsm getState] transitions] {
        # Ignore the Default transition.
        # 0 - transition is undefined.
        # 1 - transition is explicity defined in the state.
        # 2 - transition is defined in Default state.
        if {$name = "Default"} {
            # no-op
        } elseif {$status = 0} {
            # Deactivate feature.
        } elseif {$status = 1} {
            # Activate feature.
        } else {
            # Do something else for default definitions.
        }
    }
}
```

Section 12: Getting Noticed

SMC uses the Java Bean event notification and .Net event raising features to inform listeners when an SMC-generated finite state machine changes state. The following sample code demonstrates how to use this feature in Java, C#, VB.net, Groovy, and Scala.

Java Sample

SMC uses Java Beans package for state change notification. `statemap.FSMContext` defines the `addStateChangeListener` and `removeStateChangeListener` methods. Because the SMC-generated `FSMContext` subclass is privately contained within an application class, it will be necessary to expose the add, remove methods by adding these methods to the application class.

Note: Because applications use multiple state machines and a state change handler may register with multiple FSMs, I have added the methods `FSMContext.getName()` and `FSMContext.setName(String)`. This name should be set and used to distinguish between FSMs.

```java
import java.beans.PropertyChangeListener;

public class AppClass {
    ...

    public void addStateChangeListener(PropertyChangeListener listener) {
        _fsm.addStateChangeListener(listener);
        return;
    }

    public void removeStateChangeListener(PropertyChangeListener listener) {
        _fsm.removeStateChangeListener(listener);
        return;
    }

    ...

    private final AppClassContext _fsm;
}
```
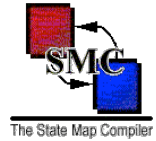
Implement the `java.beans.PropertyChangeListener` interface and pass that implementation to `AppClass.addStateChangeListener`. When the `AppClass` finite state machine changes state, the listener receives a `java.beans.PropertyChangeEvent` containing:

- the `FSMContext` instance as the source,
- the "State" property name,
- the previous state (class `statemap.State`), and
- the new state (also class `statemap.State`).

```java
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import statemap.FSMContext;
import statemap.State;

public class StateChangeListener implements PropertyChangeListener {
    ...

    public void propertyChange(PropertyChangeEvent event) {
        FSMContext fsm = (FSMContext) event.getSource();
        String propertyName = event.getPropertyName();
        State previousStatus = (State) event.getOldValue();
        State newState = (State) event.getNewValue();

        // Handle the state change event.
        System.out.println("FSM " +
                           fsm.getName() +
                           " went from " +
                           previousState +
                           " to " +
                           newState +
                           ".");
        return;
    }
}
```
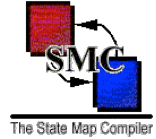
Register the `StateChangeListener` with the FSM as follows:

```java
StateChangeListener listener = new StateChangeListener();
AppClass appInstance = new AppClass();

appInstance.addStateChangeListener(listener);
```

Note: Groovy and Scala also use Java Bean event notification.

C# Sample

.Net events required listeners to register an event directly with the event-raising object. For SMC, the finite state machine class `FSMContext` does the event raising. But the context class should be kept private within the application class. The following code shows how to add and remove state change listeners indirectly, leaving the FSM inaccessible.

The `StateChangeEventArgs` data methods are:

- `FSMName()` - returns the FSM name as a `string`.
- `TransitionType()` - returns one of the following `string`s: "SET", "PUSH", or "POP".
- `PreviousState()` - returns the state which the FSM exited.
- `NewState()` - returns the state which the FSM entered.

Note: Because applications use multiple state machines and a state change handler may register with multiple FSMs, I have added the property `FSMContext.Name`. This name is placed into `StateChangeEventArgs` to allow ready identification of which FSM changed state. The default state name is "FSMContext".
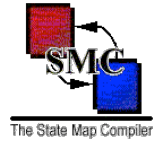
```csharp
use System;
use statemap;

public class AppClass {
    ....

    public void AddStateChangeHandler(StateChangeEventHandler handler) {
        _fsm.StateChange += handler;
        return;
    }

    public void RemoveStateChangeHandler(StateChangeEventHandler handler) {
        _fsm.StateChange -= handler;
        return;
    }
}
```

If a class wishes to receive state change events, then it must implement a method with the following signature. **Note:** the method does not have to be named `StateChanged`.

```
use System;
use statemap;

public class StateChangeHandler {
    public void StateChanged(object sender, StateChangeEventArgs args) {
        //Handle the state change event.
        Console.WriteLine("FSM " +
                         args.FSMName() +
                         " " +
                         args.TransitionType() +
                         " transition from " +
                         args.PreviousState() +
                         " to " +
                         args.NewState() +
                         ".");
        return;
    }
}
```
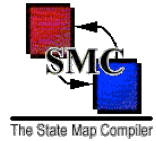
Register the state change handler instance with the FSM as follows:

```
StateChangeHandler handler = new StateChangeHandler();
AppClass appInstance = new AppClass();

appInstance.AddStateChangeHandler(
    new StateChangeEventHandler(handler.StateChange));
```

VB.net Sample

State change event handlers register indirectly via these application class methods (which you must add to your code).

```vb
Public Class AppClass
    ...

    Public Sub AddStateChangeHandler(handler As _
        statemap.StateChangeEventHandler)

        AddHandler _fsm.StateChange, handler
    End Sub

    Public Sub RemoveStateChangeHandler(handler As _
        statemap.StateChangeEventHandler)

        RemoveHandler _fsm.StateChange, handler
    End Sub
End Class
```

If a class wishes to receive state change events, then it must implement a method with the following signature.
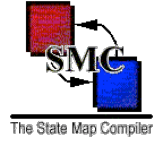
(See the C# sample for more about StateChangeEventArgs.)

```vb
Imports System
Imports statemap

Public Class StateChangeHandler
    ...

    Public Sub StateChange(sender As Object, e As StateChangeEventArgs)

        Console.Write("FSM ")
        Console.Write(e.FSMName())
        Console.Write(" ")
        Console.Write(e.TransitionType())
        Console.Write(" transition from ")
        Console.Write(e.PreviousState())
        Console.Write(" to ")
        Console.Write(e.NewState())
        Console.WriteLine(".")
    End Sub
End Class
```
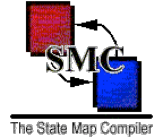
Register the state change handler with the FSM as follows:

```
Dim handler As StateChangeHandler = new StateChangeHandler()
Dim appInstance As AppClass = new AppClass()

appInstance.addStateChangeHandler(AddressOf handler.StateChange)
```

Section 13: Giving Direction

The SMC syntax provides a number of `%` directives. Each directive's use is described in this section.

% Directives

%start - Specifies the state machine start state in the form *<map name>*::*<state name>*

Required: Yes.

Supported target languages: All.

%class - Specifies the associated state machine context class providing the guard and action methods.

Required: Yes.

Supported target languages: All.

%fsmclass - Specifies the generated state machine class name.

Required: No.

Default settings: *<%class name>*Context

Supported target languages: All.

%fsmfile - Specifies the generated state machine class file name. This filename is appended with the appropriate suffix to generate the full file name. The file name suffix may be changed from its default, target language-specific value by using the `-suffix` and `-hsuffix` command line parameters.

Required: No.

Default settings: See [this table](#) for the default FSM class name, FSM file name, and suffix for each supported target programming language.
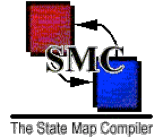
Supported target languages: All.

%package - Puts the generated code into this package/namespace.

Required: No.

Default settings: None.

Supported target languages: All.

%include - Used to include header file needed by the generated state machine code.

Required: No.

Default setting: No includes.

Supported target languages: C, C++, Objective-C

%import - Used to import/use a class/package needed by the generated state machine code.

Required: No.

Default setting: No imports.

Supported target languages: C#, C++, Groovy, JavaScript, Java, Lua, Perl, PHP, Python, Ruby, Scala, Tcl, VB.net

%declare - C/C++/Objective-C forward declaration placed in the generated header file.

Required: No.

Default setting: No declarations.

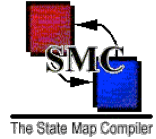Supported target languages: C, C++, Objective-C

%access - Specifies the generated class accessibility level.

Required: No.

Default setting: None.

Supported target languages: C#

%map - SMC states are grouped into maps. The `%map` is followed by two `%%` groups. The state definitions appear between the `%%` groups.

Appendix A: SMC EBNF Grammar

```
FSM := source? start_state class_name header_file? include_file*
       package_name* import* declare* access* map+

source := '%{' raw_code '%}'

start_state := '%start' word

class_name := '%class' word

header_file := '%header' raw_code_line

include_file := '%include' raw_code_line

package_name := '%package' word

import := '%import' raw_code_line

declare := '%declare' raw_code_line

access := '%access' raw_code_line

map := '%map' word '%%' states '%%'

states := word entry? exit? '{' transitions* '}'

entry := 'Entry {' actions* '}'

exit := 'Exit {' actions '}'

transitions := word transition_args? guard? next_state '{' actions '}'

transition_args := '(' parameters ')'

parameters := parameter |
              parameter ',' parameters

parameter := word ':' raw_code

guard := '[' raw_code ']'

next_state := word |
              'nil' |
              push_transition |
              pop_transition

push_transition := word '/' 'push(' word ')' |
                   'nil/push(' word ')' |
                   'push(' word ')'
```
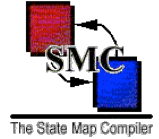
```
pop_transition := 'pop' |
                  'pop(' word? ')' |
                  'pop(' word ',' pop_arguments* ')'

pop_arguments := raw_code |
                 raw_code ',' pop_arguments

actions := dotnet_assignment |
           action |
           action actions

dotnet_assignment := word '=' raw_code ';'

action := word '(' arguments* ');' arguments := raw_code |
          raw_code ',' arguments

word := [A-Za-z][A-Za-z0-9_.]* |
        _[A-Za-z][A-Za-z0-9_.]*

// Reads in code verbatim until end-of-line is reached.
raw_code_line := .* '\n\r\f'

// Read in code verbatim.
raw_code := .*

// Both the // and /* */ comment types are supported.
// Note: SMC honors nested /* */ comments.
comment1 := '//' .* '\n\r\f'
comment2 := '/*' .* '*/'
```