

State Machine Generator

User Manual

System Documentation

February 24, 2026

Contents

| | | |
|----------|---|-----------|
| 1 | Overview | 3 |
| 2 | Generating the Code | 3 |
| 2.1 | Command Line Options | 3 |
| 2.2 | Outputs | 3 |
| 3 | SMB File Structure | 3 |
| 3.1 | Language-Specific Syntax | 4 |
| 4 | Integration & API | 4 |
| 4.1 | Rust Integration | 4 |
| 4.2 | C Integration | 5 |
| 4.3 | Context Struct | 6 |
| 5 | Writing Logic: Handlers vs. Hooks | 6 |
| 5.1 | Local Handlers (Per State/Transition) | 6 |
| 5.2 | Global Hooks (System-wide) | 6 |
| 5.3 | Available Variables in Code Blocks | 7 |
| 6 | Transitions | 7 |
| 6.1 | Basic Transition Syntax | 7 |
| 6.2 | Transition Execution Order | 7 |
| 6.3 | Using Actions (The Mealy Aspect) | 8 |
| 6.4 | Path Syntax | 8 |
| 6.5 | Transition Scenarios | 8 |
| 6.6 | Termination | 9 |
| 7 | History | 9 |
| 8 | Orthogonal States (Parallelism) | 9 |
| 8.1 | Safety: <code>transition_fired</code> | 10 |
| 8.2 | Fork Transitions (Drilling Down) | 10 |
| 8.3 | Cross-Limb Transitions | 10 |
| 8.4 | Joining (Synchronization) | 10 |
| 9 | Decisions (Logic Trees) | 11 |
| 9.1 | Defining Decisions | 11 |
| 9.2 | Using Decisions | 11 |

1 Overview

This tool generates a safety-critical Hierarchical State Machine (HSM) from a YAML/SMB definition file. It supports two output languages — **Rust** and **C** — and includes advanced features such as orthogonal regions (parallelism), history patterns, decision trees, and fork transitions.

The generator also produces a **Graphviz DOT** file for visualizing the state machine diagram.

2 Generating the Code

The generator is a Python script (`sm-compiler.py`) that validates your model and produces code in the selected target language.

```
# Language is read from the 'lang:' key in the .smb file
uv run python sm-compiler.py model.smb

# Override the language
uv run python sm-compiler.py model.smb --lang rust
uv run python sm-compiler.py model.smb --lang c

# Custom output base path (extensions added automatically)
# Produces build/myfsm.rs and build/myfsm.dot
uv run python sm-compiler.py model.smb -o build/myfsm
```

Listing 1: Command Line Usage

2.1 Command Line Options

| Option | Default | Description |
|--------------|-----------------------------|--|
| file | (required) | Path to the input <code>.smb</code> file. |
| --lang | from file | Output language: <code>rust</code> or <code>c</code> . If omitted, reads the <code>lang:</code> key from the SMB file (defaults to <code>rust</code> if absent). The <code>lang:</code> key may be a list, in which case code is generated for all listed languages. |
| -o, --output | <code>./statemachine</code> | Output base path without extension. Language-specific extensions (<code>.rs</code> , <code>.c</code> , <code>.h</code> , <code>.dot</code>) are appended automatically. Parent directories are created if needed. |

2.2 Outputs

Given `-o path/to/name`, the following files are produced:

| Language | Output Files | Description |
|----------|---|---|
| Rust | <code>name.rs</code> | Complete Rust implementation (single file). |
| C | <code>name.h</code> , <code>name.c</code> | Header with struct/function declarations and source with implementation. The <code>.c</code> file includes <code>name.h</code> automatically. |
| Both | <code>name.dot</code> | Graphviz visualization of the state machine. |

3 SMB File Structure

The input file uses YAML syntax with the extension `.smb`. A complete model has this top-level structure:

```
lang: rust                # Target language (rust or c)

includes: |                # Code placed before the Context struct
  // imports, helper functions, etc.

context: |                 # User-defined fields for the Context struct
  pub counter: i32,        # (Rust syntax)
  pub flag: bool,

context_init: |            # Initialization code for context fields
  counter: 0,              # (Rust: struct init syntax)
  flag: true,

hooks:                    # Global code injected into all states
  entry: |
    // runs on every state entry
  exit: |
    // runs on every state exit
  do: |
    // runs on every state tick
  transition: |
    // runs on every transition

initial: first_state      # Name of the initial child state

states:
  first_state:
    # ... state definition ...
```

Listing 2: Top-level SMB structure

3.1 Language-Specific Syntax

Since `includes`, `context`, `context_init`, guards, and actions contain raw target-language code, they must use the syntax of the selected output language:

| Element | Rust | C |
|----------------|----------------------------------|--|
| Context access | <code>ctx.field</code> | <code>ctx->field</code> |
| Context fields | <code>pub field: Type,</code> | <code>Type field;</code> |
| Context init | <code>field: value,</code> | <code>sm->ctx.field = value;</code> |
| Print | <code>println!("{}", ...)</code> | <code>printf("...\n")</code> |
| String type | <code>&str</code> | <code>const char*</code> |
| Boolean | <code>bool</code> | <code>bool (via stdbool.h)</code> |

4 Integration & API

4.1 Rust Integration

The generated code provides a `StateMachine` struct. You must initialize it and call `tick()` cyclically.

```

mod statemachine;
use statemachine::StateMachine;

fn main() {
    let mut sm = StateMachine::new();

    while sm.is_running() {
        sm.ctx.now += 0.1; // Advance logical time
        sm.tick();
        println!("State: {}", sm.get_state_str());
        std::thread::sleep(std::time::Duration::from_millis(100));
    }
}

```

Listing 3: Example Rust driver (main.rs)

Rust API

| Function | Location | Description |
|---------------------|-------------|---|
| StateMachine::new() | constructor | Creates and starts the state machine. |
| sm.tick() | sm | Executes one cycle of logic (do-activities & transitions). |
| sm.is_running() | sm | Returns false if the machine has terminated. |
| sm.get_state_str() | sm | Returns a string like <code>/run/active</code> or <code>/p/[a/x,b/y]</code> . |
| ctx.in_state_X() | ctx | Returns true if state X is currently active. |

4.2 C Integration

The generated code provides a `StateMachine` struct with an `SM_Context` inside. Use the provided API functions.

Listing 4: Example C driver (main.c)

```

#include "statemachine.h"
#include <stdio.h>
#include <unistd.h>

int main(void) {
    StateMachine sm;
    sm_init(&sm);

    char state_buf[256];

    while (sm_is_running(&sm)) {
        sm_tick(&sm);
        sm_get_state_str(&sm, state_buf, sizeof(state_buf));
        printf("State: %s\n", state_buf);
        usleep(100000);
    }

    return 0;
}

```

Compile with:

```
gcc main.c statemachine.c -o my_program
```

C API

| Function | Description |
|--|---|
| <code>sm_init(StateMachine* sm)</code> | Initializes context (zeroes memory, applies initial state). |
| <code>sm_tick(StateMachine* sm)</code> | Executes one cycle of logic. |
| <code>sm_is_running(StateMachine* sm)</code> | Returns <code>true</code> if the machine has not terminated. |
| <code>sm_get_state_str(StateMachine* sm, char* buf, size_t max)</code> | Writes the current state path into <code>buf</code> . |
| <code>IN_STATE_X</code> | Macro: evaluates to true if state <code>X</code> is current inside guards and actions where <code>ctx</code> is in scope. |

4.3 Context Struct

Both backends generate a `Context` struct (Rust: `Context`, C: `SM_Context`) containing:

- `now`: Logical time (`f64/double`). You update this before each tick.
- `state_timers[]`: Per-state entry timestamps (set automatically on entry).
- `transition_fired`: Safety flag for orthogonal regions (managed automatically).
- `terminated`: Set when a `null` transition fires (managed automatically).
- Hierarchy pointers: Function pointers tracking the active state at each level (managed automatically).
- User fields: Whatever you declare in the `context:` block.

5 Writing Logic: Handlers vs. Hooks

5.1 Local Handlers (Per State/Transition)

These are defined inside specific states or transitions in the YAML. They apply only to that specific element.

- `entry`: Runs once when entering the specific state.
- `do`: Runs every tick while in the specific state.
- `exit`: Runs once when leaving the specific state.
- `action`: Runs during a specific transition (see Section 6).

5.2 Global Hooks (System-wide)

Hooks are defined at the root level under the `hooks:` key. They inject code into **every** occurrence of an event type across the entire machine (e.g., for logging).

| Hook Key | Trigger Event |
|-------------------|---|
| hooks: entry | Injected into all states' entry handlers. |
| hooks: do | Injected into all states' do handlers. |
| hooks: exit | Injected into all states' exit handlers. |
| hooks: transition | Injected into all transitions (equivalent to a global action). |

Note: There is no `hooks: action`. Use `hooks: transition` instead.

5.3 Available Variables in Code Blocks

The following variables are injected into the scope of your code snippets. Use the appropriate syntax for your target language.

| Variable | Scope | Description |
|------------------------------|-------------|--|
| <code>ctx</code> | All | The shared context struct. Rust: <code>&mut Context</code> . C: <code>SM_Context*</code> . |
| <code>time</code> | All | Time elapsed (seconds) since entering the current state (<code>f64/double</code>). |
| <code>state_name</code> | All | Short name of the current state (e.g., <code>"idle"</code>). |
| <code>state_full_name</code> | All | Full path of the current state (e.g., <code>"/run/idle"</code>). |
| <code>t_src</code> | Transitions | The full path of the source state. |
| <code>t_dst</code> | Transitions | The full path/description of the target. |

6 Transitions

A transition determines how the machine moves between states.

6.1 Basic Transition Syntax

```
transitions:
- guard: ctx.counter == 5      # Condition (target language expression)
  action: |                    # Code executed during the transition
    // runs BEFORE exit/entry sequence
  to: target_state              # Where to go
```

Listing 5: Transition syntax

All three fields are optional:

- Omitting `guard` makes the transition unconditional (`true`).
- Omitting `action` means no transition-time code.
- `to: null` triggers machine termination.

6.2 Transition Execution Order

When a transition fires:

1. Evaluate the `guard` condition.

2. Set `transition_fired = true`.
3. Execute `action` code (if any).
4. Execute the exit sequence (from current state up to the Least Common Ancestor, bottom-up).
5. Execute the entry sequence (from LCA down to target state, top-down).

6.3 Using Actions (The Mealy Aspect)

The `action` keyword allows you to execute code *during* the transition, before the old state is exited.

```
states:
  reading:
    transitions:
      - guard: ctx.value > 100
        action: |
          println!("Moving_{}_->{}", t_src, t_dst);
          ctx.log_alert("Threshold_Exceeded");
        to: alert_mode
```

Listing 6: Rust example using action variables

```
states:
  reading:
    transitions:
      - guard: ctx->value > 100
        action: |
          printf("Moving_%s->%s\n", t_src, t_dst);
          log_alert(ctx, "Threshold_Exceeded");
        to: alert_mode
```

Listing 7: C example using action variables

6.4 Path Syntax

The `to:` keyword supports several addressing modes:

| Syntax | Description |
|----------------------|---|
| /absolute/path | From the root. Use for long jumps across the hierarchy. |
| sibling | A state in the same parent (lateral transition). |
| ./child | A direct child of the current state (drill down). |
| ../uncle | Up one level, then a sibling of the parent. |
| . | Self-transition: the state exits and immediately re-enters. |
| null | Machine termination. Exits all states up to root and stops. |
| @decision_name | Delegate to a named decision tree (see Section 9). |
| /path/[a/tgt, b/tgt] | Explicit fork into an orthogonal state (see Section 8). |

6.5 Transition Scenarios

Lateral (Sibling to Sibling)

- Exits: A
- Enters: B

Drill Down (Parent to Child)

- **Exits:** None (P remains active).
- **Enters:** C

Move Up (Child to Parent/Uncle)

- **Exits:** C, then Parent P.
- **Enters:** U

6.6 Termination

Setting `to: null` causes the machine to exit all states from the current state up through root, then stop. After termination, `is_running()` (Rust) or `sm_is_running()` (C) returns false.

7 History

A composite state with `history: true` remembers its last active child. When re-entered, it resumes from that child instead of the `initial` state.

```
process:
  history: true
  initial: step_a
  states:
    step_a:
      transitions:
        - guard: ctx.counter == 3
          to: step_b
    step_b:
      transitions:
        - guard: ctx.counter == 5
          to: /other_state
```

Listing 8: History example

If the machine leaves `process` while in `step_b`, and later transitions back to `process`, it will re-enter `step_b` directly (skipping `step_a`).

8 Orthogonal States (Parallelism)

An orthogonal state runs multiple child regions simultaneously.

Keyword: `orthogonal: true`

```
parallel_task:
  orthogonal: true
  states:
    region_a:
      initial: idle_a
      states:
        idle_a: {}
        working_a: {}
    region_b:
      initial: idle_b
      states:
        idle_b: {}
        working_b: {}
```

Listing 9: Orthogonal state definition

When `parallel_task` is entered, **all** child regions are entered simultaneously. Each region ticks independently. When exiting, all regions are exited.

The state path for orthogonal states uses bracket notation: `/parallel_task/[region_a/idle_a,region_b/]`

8.1 Safety: `transition_fired`

When a transition fires in one region, the other regions' ticks are skipped for that cycle to prevent inconsistent state. This is handled automatically via the `transition_fired` flag.

8.2 Fork Transitions (Drilling Down)

You can enter an orthogonal state by targeting its children directly.

1. **Implicit Fork:** `to: /run/g/a/sub_state`

The generator forces region `a` to `sub_state`. Other regions start at their `initial` states.

2. **Explicit Fork:** `to: /run/g/[a/sub_state, b/other]`

Explicitly sets the target for multiple regions at once.

8.3 Cross-Limb Transitions

A transition from one region to another region within the same orthogonal parent is a **cross-limb transition**. The generator handles this by exiting the target region's current state and entering the new target, while the source region remains unaffected.

8.4 Joining (Synchronization)

To "join" parallel states, place a transition on the **parent** that checks the status of the **children**.

```
main_task:
  orthogonal: true
  states:
    A:
      initial: working
      states:
        working:
          transitions:
            - guard: ctx.done_a
              to: finished
        finished: {}
    B:
      initial: working
      states:
        working:
          transitions:
            - guard: ctx.done_b
              to: finished
        finished: {}

# Transition on the parent container
transitions:
  - guard: IN_STATE(root_main_task_A_finished) && IN_STATE(
    root_main_task_B_finished)
    to: next_step
```

Listing 10: Join pattern (Rust guards)

Note: Use the `IN_STATE(X)` macro in guards for portable state checks. In Rust, it expands to `ctx.in_state_X()`. In C, it expands to the `IN_STATE_X` macro.

9 Decisions (Logic Trees)

Decisions allow you to externalize branching logic. A decision node can point to states, or **chain to other decisions**.

Restriction: You cannot use `action` or `hooks` inside a decision block. Actions must be placed on the transition *leading to* the decision, or on the state transitions following it.

9.1 Defining Decisions

Decisions can be defined at the root level or within any state. They are collected into a flat namespace, so names must be unique across the entire model.

```
decisions:
  validate_input:
    - guard: ctx.input < 0
      to: /error/negative
    - guard: ctx.input > 100
      to: "@check_permissions" # Chain to another decision
    - to: /process/calc # Default (no guard)

  check_permissions:
    - guard: ctx.is_admin
      to: /admin/override
    - to: /error/denied
```

Listing 11: Decision tree definition

9.2 Using Decisions

Reference a decision with the `@` prefix in the `to:` field:

```
transitions:
- guard: ctx.data_ready
  to: "@validate_input"
```

Listing 12: Using a decision in a transition

Decisions can also be defined locally within a state:

```
my_state:
  decisions:
    local_check:
      - guard: ctx.x > 0
        to: positive
      - to: negative
  initial: waiting
  states:
    waiting:
      transitions:
        - to: "@local_check"
    positive: {}
    negative: {}
```

10 Complete Example

Below is a minimal but complete SMB file demonstrating the key features, with Rust as the target language.

```
lang: rust

includes: |
  fn log(msg: &str) { println!("{}", msg); }

context: |
  pub counter: i32,

context_init: |
  counter: 0,

hooks:
  entry: |
    println!("Entering_{}", state_full_name);
  exit: |
    println!("Exiting_{}", state_full_name);

initial: active

states:
  active:
    initial: idle
    states:
      idle:
        transitions:
          - guard: ctx.counter >= 3
            to: working
      working:
        do: |
          ctx.counter += 1;
        transitions:
          - guard: ctx.counter >= 10
            to: null # Terminate

error:
  entry: |
    log("ERROR_STATE");
```

Listing 14: Complete SMB example (Rust)