State Machine Generator
User Manual

System Documentation

January 24, 2026

Contents

1 Overview

This tool generates a safety-critical Hierarchical State Machine (HSM) in Rust from a YAML definition. It supports advanced features like orthogonal regions (parallelism), history patterns, and complex decision trees.

2 Generating the Code

The generator is a Python script (`sm-builder.py`) that validates your model and produces Rust code.

```
# Basic Usage
python3 sm-builder.py model.yaml

# Specify Output Language (Default is Rust)
python3 sm-builder.py model.yaml --lang rust
```

Listing 1: Command Line Usage

2.1 Outputs

- `statemachine.rs`: The complete Rust implementation.
- `statemachine.dot`: A Graphviz visualization of your logic.

3 Integration & API

3.1 The Main Loop

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();

    // Initialize inputs (accessing public context)
    sm.ctx.counter = 0;

    // Execution Loop
    while sm.is_running() {
        // 1. Advance logical time (if using timers)
        sm.ctx.now += 0.1;

        // 2. Run logic
        sm.tick();

        // 3. Debug Output
        println!("Current State: {}", sm.get_state_str());

        std::thread::sleep(std::time::Duration::from_millis(100));
    }
}
```

Listing 2: Example main.rs

3.2 Public API Methods

These methods are available on the `StateMachine` instance (`sm`) or its context (`ctx`).

Function	Location	Description
`tick()`	`sm`	Executes one cycle of logic (Do-activities & Transitions).
`is_running()`	`sm`	Returns `false` if the machine has terminated.
`get_state_str()`	`sm`	Returns a string representation (e.g., `/run/active`).
`in_state_X()`	`ctx`	Returns `true` if state `X` is currently active.

4 Writing Logic: Handlers vs. Hooks

4.1 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 5).

4.2 Global Hooks (System-wide)

Hooks are defined at the root level under the `hooks:` key. They allow you to 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.

4.3 Available Variables in Code Blocks

The following variables are injected into the scope of your Rust snippets:

Variable	Type	Scope	Description
`ctx`	`&mut Context`	All	The shared context struct holding your data.
`time`	`f64`	All	Time elapsed (seconds) since entering the current state.
`t_src`	`&str`	Action/Trans.	The full path of the state being left.
`t_dst`	`&str`	Action/Trans.	The full path of the target state.

5 Transitions

A transition determines how the machine moves.

5.1 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
        # This code runs BEFORE reading exits
        action: |
          println!("Moving␣{}␣->␣{}", t_src, t_dst);
          ctx.log_alert("Threshold␣Exceeded");
        to: alert_mode
```

Listing 3: Example using Action variables

5.2 Path Syntax

The `to:` keyword supports several addressing modes:

- `/absolute/path`: Starts from the root. Use for long jumps.
- `sibling`: Looks for a state in the same parent.
- `./child`: Looks for a direct child of the current state.
- `../uncle`: Moves up one level, then finds a sibling.
- `.`: **Self-Transition**. The state exits and immediately re-enters.

5.3 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
- **Exits:** Parent P.
- **Enters:** U

6 Orthogonal States (Parallelism)

An orthogonal state runs multiple child regions simultaneously. **Keyword:** `orthogonal: true`

6.1 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 normally.
2. **Explicit Fork:** `to: /run/g/[a/sub_state, b/other]`
   Explicitly sets the target for multiple regions at once.

6.2 Joining (Synchronization)

To "join" parallel states, transition from the **Parent** based on the status of the **Children**.

```
# Parent State
main_task:
  orthogonal: true
  states:
    A:
      states:
        working:
          transitions:
            - guard: done
              to: finished
        finished: {} # Idle wait state
    B:
      # ... similar logic ...

  # Transition on the Parent Container
  transitions:
    - guard: ctx.in_state_A_finished() && ctx.in_state_B_finished()
      to: next_step
```

7 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.

1. Define the Decision (Root Level):

```
decisions:
  validate_input:
    - guard: ctx.input < 0
      to: /error/negative
    - guard: ctx.input > 100
      to: check_permissions # Chaining to another decision
    - to: /process/calc     # Default

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

2. Use the Decision:

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