ChatGPT

Correct Entry/Exit Sequence in Hierarchical & Parallel State Machines (Rust `#![no_std]`)

1. Standard Algorithms for Entry Sequence (LCA-Based)

Modern statechart semantics (UML, SCXML, XState) use the **Lowest Common Ancestor (LCA)** of the source and target states to determine which states to exit and enter during a transition [1] [2]. In practice, the algorithm is:

- **Find the LCA:** Identify the lowest composite state that is an ancestor of both the source and target. The LCA is *not* exited or re-entered during the transition [1].
- **Exit Phase:** Exit all active states from the source state **up to but not including** the LCA. Perform each state's exit actions in order from the deepest state up towards the LCA [1] [3]. This ensures no parent state is exited unless it's below the LCA (preventing improper re-execution of a still-active superstate's entry later).
- **Transition Actions:** Execute any actions associated with the transition itself (if using a model that supports transition-specific actions).
- **Entry Phase:** Enter the chain of states from the LCA down to the target state. Starting from the **immediate child of the LCA** that lies on the path to the target, execute each state's entry action, working down the hierarchy to the target [2] [4]. Entry actions are done in order from outer (higher-level) to inner (leaf) so that parent contexts are entered before children. Importantly, the LCA itself is neither exited nor re-entered in a local transition [5] [6] (for an external transition, the LCA *would* also be exited and re-entered, see below).
- **Complete Entry into Composite/Parallel States:** If the *target state* is *non-atomic* (i.e. a composite or parallel state), you must **drill down into its substates** after entering it [7] [8]. In UML terms, after performing the target state's on-entry action, the machine automatically takes the target's initial transition (or default initial state) to enter its child state(s). This recursive initialization continues until a leaf (atomic) state is reached [7] [8]. In a **parallel state**, entering it means simultaneously entering the initial substate of *each* region of that parallel composite [9] (each region may have its own initial state). All regions become active concurrently once the parallel state's entry is complete.

Example (UML/SCXML): Suppose state machine hierarchy `S` with children `s1, s2`; `s1` has child `s11`; `s2` has child `s21`. If the machine transitions from `s11` (inside `s1`) to `s21` (inside `s2`), the LCA is `S`. The sequence will be: exit `s11`, exit `s1` (stop at `S`); then execute transition actions; then enter `s2`, enter `s21`. The LCA `S` is *not* exited or re-entered [10] [11]. If `s2` or `s21` were composite, the machine would next enter their initial substates automatically [8]. This aligns with the W3C SCXML algorithm and UML statechart rules. The Zephyr RTOS HSM framework (SMF) summarizes similarly: *"All exit() functions from the current state up to but not including the LCA are called... then all entry() functions from (but not including) the LCA down to the next state are called."* [3].

External vs. Internal Transitions: By default, UML/SCXML assume **external** transitions, meaning the source state is exited even if the target is a substate or superstate. UML 2 introduced *local (internal)*

transitions that do not exit/re-enter a state if the transition stays within it [12]. For example, a self-transition marked "internal" will not exit or re-enter the state, whereas an external self-transition will trigger the state's exit and entry (and reinitialize its children) [13]. SCXML's default is external, but XState (a JS statecharts library) chooses internal by default for certain relative targets to avoid unnecessary exits [14]. This ensures that, for instance, transitioning between two substates of a parallel parent doesn't unintentionally tear down and rebuild the entire parallel state context (more on this in Pitfalls). In summary, the standard entry/exit sequencing uses the LCA method to avoid duplicate entry actions and correctly manage hierarchical transitions.

2. Coordinating Path-Based vs. Recursive Entry Actions

In an implementation like `lit-bit-core` (inspired by XState), you typically split the work between a *path-based* traversal (e.g. `execute_entry_actions_from_lca`) and a *recursive entry* helper (e.g. `enter_state_recursive_logic`). The question is how to divide responsibility to **avoid duplicating entry actions**, especially for the target state.

Best Practice: Let the *path-based function* handle entering all states **down to and including the target**, then let the *recursive function* initialize any child regions of the target. In other words, `execute_entry_actions_from_lca` should perform on-entry actions for each intermediate state on the LCA→target path and also the target state's own entry (since the target is part of that path) [15]. Once the target state is entered (now active), you hand off to `enter_state_recursive_logic` to enter any *initial substates or parallel regions* of the target. This approach mirrors the formal description: the transition's entry sequence ends with the target state's entry action, and *then* if the target is composite, the machine enters its default substate(s) **recursively until reaching a leaf state** [16]. By performing the target's entry exactly once (in the path-based sequence), you ensure its entry action occurs in the correct order relative to ancestor entries and the transition action [15], and you avoid calling it twice.

Alternatively, some designs choose to **exclude** the target's on-entry from the path-based loop and let the recursive logic handle it. For example, one could have `execute_entry_actions_from_lca` only enter up to the target's parent, then call `enter_state_recursive_logic(target)` which would execute the target's entry and then descend. This can also work, but you must then ensure intermediate composites on the path don't get their children initialized prematurely. The key is consistency: whichever function is responsible for a given state's entry action, the other should not repeat it. If `enter_state_recursive_logic` assumes responsibility for executing a state's entry, then the path function should stop right before that state. Conversely, if the path function already did a state's entry, the recursive function should detect that and skip re-running it.

Intermediate Composite/Parallel States on the Path: When transitioning **within the same parent state** (e.g., from `Region1.StateA` to `Region1.StateB` in the same region), the LCA will be that parent. The parent's entry should *not* be re-run (it wasn't exited), so the path function will only exit the source (`StateA`) and enter the target (`StateB`), leaving the parent active [17]. If an intermediate state on the path is a *composite* that you are entering on the way to a deeper target, you do **not** perform its initial transition, because you have an explicit deeper target. You simply execute its entry action and continue down the path. Initial actions are only taken when entering a composite **without a specified substate** (i.e. when it *is* the target or being activated as a whole). For a *parallel* state on the path, consider that entering it may require initializing *all* its regions. If the transition's target lies in one region of a parallel state that was previously

inactive, upon entering the parallel state you must also start its other regions to fulfill the semantics of parallel states (all regions active) [9]. One strategy is: when the path function enters a parallel state that wasn't active, immediately call a recursive initializer that activates each region's initial state **except** the region containing the deeper target, which will be entered by continuing the path. This is a complex scenario, and an implementation must carefully ensure that other regions aren't left uninitialized. In many cases, it's simpler to treat a transition into a deep target of a parallel as two steps: (1) enter the parallel state (with all regions' defaults), then (2) internally transition within the target's region to the specified substate. (The SCXML spec allows a single transition to target multiple states for parallel entry, but in code it may be easier to split it.) The bottom line is that the **entry sequence is "complete" for the target state itself once its onentry ran**, and at that point you hand off to recursion to finalize any nested/orthogonal activation. Ensuring a clean split of duties (no double-calls) between the path-based and recursive logic is crucial to avoid redundant actions.

Summary: It's acceptable for `execute_entry_actions_from_lca` to invoke the target state's entry **once** as part of the linear path (as UML/SCXML do), with `enter_state_recursive_logic` then handling child initialization. Just be cautious that `enter_state_recursive_logic` doesn't call the same onentry again. Some implementations add flags or state markers to track active states, so if a state is already active (just entered), its recursion will skip repeating the entry. The goal is one entry action per state per transition. If in doubt, delegating the *target* state's entry entirely to the recursive function can simplify logic, but then the transition action ordering needs to be managed so that the target's entry still occurs before its children. Many battle-tested frameworks (e.g. SCXML, Yakindu, QP) effectively perform the target's entry in the transition step and then do a separate step for entering descendants [16].

3. Common Pitfalls in Hierarchical State Entry/Exit

Implementing hierarchical and parallel state machines is tricky. Here are common pitfalls and how to avoid them:

- **Redundant Entry or Exit Actions:** This often happens if the LCA is miscomputed or if internal vs. external transitions are handled incorrectly. For example, re-entering a state that's already active leads to duplicate entry actions. Ensure that you do *not* exit or re-enter the LCA state (or any common parent) during a transition [5]. Likewise, on self-transitions, differentiate between internal (no exit/entry) and external (full exit+entry) to avoid double-calling onentry/onexit when not needed [13]. A self-transition on a composite state should typically be external (exit substates and the state, then re-enter it and its initial child), unless explicitly defined as an internal self-event that stays in state.

- **Re-entering Active Parents:** Incorrectly treating a transition as external when it should be local can cause a parent state to exit and re-enter even though the transition target is within that parent. This leads to "restarting" the parent state unnecessarily (and repeating its entry action). The UML local transition semantics exist to prevent this: if the target is a child of the source state, the source (parent) state should not exit [18]. Conversely, if the target is a superstate of the source, a local transition means you don't re-enter the target (avoiding re-entry of an ancestor) [18]. Always check if your transition's source and target are in the same region of a composite – if so, exit only down to their parent, and enter from that parent down. Tools like XState default many transitions to *internal* (local) to avoid unwanted exits [14]. Failing to account for this can trigger redundant onexit/onentry calls and disrupt expected state persistence.

- **Cross-Region (Parallel) Transitions:** Parallel states add complexity – a transition that crosses from one region to another (within the same parallel composite) can inadvertently affect sibling regions. According to the SCXML semantics, an *external* transition whose LCA is above a parallel will exit all active states below that LCA [19]. This means if you have parallel regions and you transition from a state in Region A to a state in Region B (same parallel), a naive external transition would exit *both* regions' states (because the LCA might be the parallel's parent or the root, causing the entire parallel to reset). This is often undesirable. **Pitfall:** unintentionally resetting orthogonal regions. **Solution:** use internal transitions or explicit separate transitions. For example, XState found it surprising that SCXML's default would exit unrelated region states, and thus when a transition target is in another region, XState tries to treat it in a way that doesn't exit the parallel unless necessary [20] [14]. When implementing, be mindful: if the parallel state itself isn't being exited, you should only exit states in the source region and enter states in the target region, leaving other regions untouched. Failing to do so either leaves orphan active states or clears too much state. Handling this correctly often requires identifying that the LCA is the parallel state (or above) and adjusting exit sets accordingly (exit source region's states, but not the parallel or other regions). Testing transitions in parallel setups is important to catch these issues.

- **Not Initializing New Active Regions:** When entering a composite or parallel state, forgetting to enter its initial substate(s) is a common mistake. A composite state with no active child is an incomplete configuration. Always follow up a state's entry by initializing its default child. For parallel states, *all* regions must be active upon entry [9]. A pitfall is to enter a parallel and only activate one region (e.g., the one you care about) – the other regions would remain unentered, which violates the statechart semantics. Make sure your `enter_state_recursive_logic` covers all regions of a parallel (you might loop over `state.regions`). Likewise, if you have final states, reaching a final state in a region may trigger events (like completing the parent) – be sure your design accounts for that, though that's beyond basic entry sequence.

- **Incorrect Order of Entry/Exit:** The order must be strictly maintained: all exits (deepest first) then transition actions, then entries (outermost first) [15]. A pitfall is interleaving entry/exit incorrectly or doing a child's entry before a parent's entry (violating hierarchy). Sticking to the LCA algorithm prevents this. Also, be wary of performing actions during exit/entry that might send new events – since in a run-to-completion model, those should usually queue until the transition is done. In a Rust `no_std` context (no OS), this typically means being careful with interrupts or reentrant calls during state transitions.

- **Handling of History or Self-Transitions:** (Advanced) If you implement history pseudostates (shallow or deep history), ensure that re-entering a state via history doesn't also re-execute its entry if you intend to restore the last substate. Similarly, a self-transition on a parallel state (exiting and re-entering the same parallel) should exit all regions and re-enter them. These are edge cases where duplication or omission of actions can occur if not explicitly coded. Following formal specs can guide these behaviors (e.g., SCXML's history handling [21]).

In summary, many pitfalls boil down to **exiting or entering too much or too little**. Avoid redundant calls by precisely computing the exit set and entry path via LCA. Use guards (or transition types) to handle local vs. external transitions. And thoroughly test scenarios like sibling transitions, transitions between parallel regions, and self-transitions on composite states. The references (UML, SCXML) provide the formal ground

truth for these behaviors, which battle-tested frameworks (Yakindu, QP, statechart libraries) have adopted – leveraging those can help avoid common bugs.

4. Pseudocode and Implementation Examples

To solidify these concepts, here's a simplified pseudocode outline for a transition in a hierarchical state machine, incorporating the discussed algorithm (suitable for an embedded/Rust `no_std` runtime):

```
fn transition(source: StateID, target: StateID) {
    let lca = find_lowest_common_ancestor(source, target);
    // 1. Exit phase: exit all states from source up to (but not including) LCA
    for state in active_state_stack_from(source) {
        if !state.is_ancestor_of(lca) {
            execute_exit_action(state);
            mark_inactive(state);
        } else {
            break;
        }
    }
    // 2. [Optionally] execute actions associated with this transition
    // (e.g., effect actions defined on the transition arrow)

    // 3. Entry phase: enter states from LCA down to target
    let path = get_state_path(lca, target);  // list of states from LCA-
>child ... -> target
    for (i, state) in path.iter().enumerate() {
        execute_entry_action(state);
        mark_active(state);
        // If this state is parallel and not the final target, initialize all
its regions:
        if state.is_parallel() && i < path.len() - 1 {
            for region in state.regions() {
                if ! region.contains(path[i+1]) {
                    // initialize regions that are not along the target path
                    let init = region.initial_state();
                    transition_to_initial(region.parent_state(),
init); // enter initial substate (recursive call or loop)
                }
            }
        }
    }

    // 4. Recursive initialization if target is composite/parallel
    let target_state = target;
    if target_state.is_composite() {
        // Enter initial child state of target
```

```
        let init = target_state.initial_state();
        if init != None {
            // This will call entry on the child and possibly its descendants
            enter_state_recursive_logic(init);
        }
    } else if target_state.is_parallel() {
        // Enter initial state of each region of the parallel state
        for region in target_state.regions() {
            let init = region.initial_state();
            enter_state_recursive_logic(init);
        }
    }
}
```

This pseudocode illustrates one way to break up the tasks. In practice, `enter_state_recursive_logic(state)` would perform the state's entry (if not already done) and then its descendants. The above code assumes the path loop already did the target's entry; thus, when handling a composite target, we immediately dive into its initial substate. If instead we hadn't yet entered the target, `enter_state_recursive_logic` would first call the target's entry then proceed to children – the logic needs to be consistent with how you split responsibilities (as discussed in section 2).

Yakindu & QP Patterns: Tools like Yakindu Statechart Generator and Quantum Leaps' QP framework follow essentially this algorithm. Yakindu generates code that on a transition will call all required exit handlers, then entry handlers, in proper order (often using switch-case or function tables). QP (active object framework in C/C++) uses macros to implement transitions: you specify the source and target states in code, and the framework computes the exit/entry sequence at runtime (or even compile-time with templates in some implementations). QP strictly adheres to UML semantics, e.g., it will not exit the LCA state on a local transition [5]. It provides utility functions (like `QHsm::tran(target)`) that internally perform the LCA algorithm, ensuring no duplicate entry calls. These frameworks typically don't support *orthogonal* (parallel) states in the same state machine (QP treats concurrency via separate state machines), so your case with parallel regions is more akin to SCXML or statechart libraries like XState.

Rust Embedded Examples: The `statig` crate is a recent Rust library that supports hierarchical state machines with no heap usage (compatible with `#![no_std]`) [22]. It uses a macro to generate state transition code that manages superstate entry/exit. While its internal code isn't shown here, one can infer it finds the LCA of states and calls the appropriate entry/exit handlers just as outlined. Another is the experimental `rustate` (inspired by XState) and `sfsm` (state machine for embedded) – all of these implement the essence of the LCA-based transition algorithm.

UML-like Visualization: Imagine a state diagram with nested states and maybe a parallel region. A transition from one nested state to another far away will have a *fork* at the LCA – above that point, nothing changes; below it, one branch of the hierarchy is exited and another branch entered. Tools like UML, Yakindu, or SCXML graphs often depict this by showing an arrow exiting up out of the source state(s) to the common ancestor, then into the target branch. If needed, you can draw the hierarchy as a tree and mark the exit path and entry path. The key is that the "turnaround" point is the LCA. (If the transition is external vs. internal can shift that point up or down one level.)

In conclusion, a robust implementation should mirror these well-established algorithms: find the correct LCA boundary, call exit actions upward, then entry actions downward, and initialize any composite/parallel children. By leveraging formal definitions (UML Superstructure spec, SCXML spec) and studying reference implementations (XState's interpreter, Yakindu's generated code, Miro Samek's QP patterns), you can avoid the common pitfalls. All these approaches are feasible in a Rust `no_std` environment since they require no OS features – just careful state management in memory. **State machine correctness comes from respecting the hierarchy relationships**: never enter or exit a state out of order. Following the above guidelines and patterns will ensure your `lit-bit-core` handles hierarchical and parallel transitions with correct entry/exit sequencing every time.

Sources:

- UML Statecharts semantics (entry/exit order via LCA) [1] [7]
- W3C SCXML Spec (algorithm for exiting to LCCA and entering target + initial states) [15] [16]
- XState and SCXML transition types (internal vs external default) [14]
- Zephyr SMF (embedded HSM rules and examples) [3] [23]
- Parallel state entry in XState docs [9] and internal vs self-transition note [13].

---

[1] [7] [12] [18] UML state machine - Wikipedia
https://en.wikipedia.org/wiki/UML_state_machine

[2] [6] [8] [10] [11] [15] [16] [21] State Chart XML (SCXML): State Machine Notation for Control Abstraction
https://www.w3.org/TR/scxml/

[3] [4] [17] [23] The Zephyr State Machine Framework (SMF) | mbedded.ninja
https://blog.mbedded.ninja/programming/operating-systems/zephyr/state-machine-framework/

[5] QP Real-Time Event Frameworks & Tools / Forum / Free Support: Transition triggered by a a super state
https://sourceforge.net/p/qpc/discussion/668726/thread/5f30d49a/

[9] Parallel states | Stately
https://stately.ai/docs/parallel-states

[13] QM: Working with Transitions
https://www.state-machine.com/qm/sm_tran.html

[14] [19] [20] Surprising external transitions behavior for parallel states · statelyai xstate · Discussion #1829 · GitHub
https://github.com/statelyai/xstate/discussions/1829

[22] GitHub - mdeloof/statig: Hierarchical state machines for designing event-driven systems
https://github.com/mdeloof/statig