ChatGPT

Designing a Parallel Statecharts Event Dispatch Algorithm for lit-bit

SCXML: Preventing Duplicate Transitions and Conflicts

The W3C **SCXML** (State Chart XML) standard provides a formal algorithm for event dispatch in statecharts, which ensures that each transition is handled **exactly once** per event. SCXML's event selection logic processes each active *leaf state* and its ancestor hierarchy, but **avoids duplicate transitions** and resolves conflicts:

- **Selecting Transitions (No Duplicates):** SCXML iterates over all active atomic (leaf) states in the current configuration (in document order) and for each, it searches up its ancestry for the first matching transition on the event [1] [2]. If a transition is found, it is added to the set of enabled transitions *only if* it's not already in the set [2]. This means if two active leaves share an ancestor with a transition for that event, that **same transition** (defined on the common ancestor) will be added only once. In other words, an ancestor-level transition is not duplicated for each region – it's treated as a single enabled transition, preventing double execution.

- **Multiple Transitions in Parallel Regions:** In a purely hierarchical (non-parallel) state machine, only one transition can be taken for a given event. However, in a **parallel** state (with orthogonal regions), it's possible for an event to enable *multiple independent transitions* – e.g. one in each region. SCXML allows this: after gathering all candidate transitions, there may be multiple in the list (one per region) when in a parallel state configuration [3]. These represent independent, simultaneously enabled transitions. SCXML will attempt to fire all of them in a single *macrostep*, provided they don't conflict.

- **Conflict Resolution (Remove Conflicting Transitions):** SCXML defines that two transitions **conflict** if the sets of states they would exit overlap (i.e. they compete to leave the same state) [4]. Before executing transitions, SCXML prunes conflicts by choosing the *most specific* (deepest) transition in any conflict set. Essentially, if one transition exits a state that is an ancestor of the state exited by another transition, the transition from the deeper (more nested) state is given priority [5]. The algorithm sorts candidate transitions by document order, then for each, removes any other transition that conflicts with it [4]. The criterion is: *prefer narrower scope* – a transition from a deeper state will be chosen over one from a higher-level state in the hierarchy [6]. Only one transition per region (or per conflict set) survives this phase. This ensures a **consistent set** of transitions that can all fire together without interfering.

- **Macrostep Execution:** After selecting the final set of non-conflicting transitions (the *ActiveTransitions*), SCXML executes them as one macrostep. It performs an **Exit phase**, **Transition effects phase**, and **Entry phase** encompassing *all* these transitions together [7]. Notably, this means all states to be exited (from all selected transitions) exit before any new states are entered. Duplicate processing is avoided because each transition in the set is unique, and ancestor transitions

(if selected) appear only once. SCXML's selection logic and conflict removal together prevent the same transition from firing twice across parallel branches.

XState's Approach to Event Dispatch in Parallel Regions

The **XState** library (JavaScript) follows semantics very similar to SCXML for dispatching events, including support for parallel states. On receiving an event, XState's interpreter will evaluate **each active branch** of the statechart and determine which transitions to take, while avoiding duplication:

- **Hierarchical Checking:** XState checks each active state configuration for a matching transition. The event is first offered to the deepest active states; if no handler is found, it bubbles up to parent states, and so on up to the root [8]. This means XState effectively searches the state's ancestry for a transition, just like SCXML. A parent-state transition (defined in an ancestor) will fire if none of the deeper states handled the event. In a parallel statechart, this process happens for each orthogonal region's active substate.

- **Multiple Enabled Transitions:** If an event triggers transitions in multiple regions of a parallel state, XState will enable **all** those non-conflicting transitions. The documentation notes that in parallel (orthogonal) state nodes, it is possible for multiple transitions to be enabled at once, and in such cases *"all enabled transitions to these regions will be taken."* [9]. For example, if two parallel regions each have a transition on event `"E"`, both will fire in response to `"E"` – assuming they don't target states that interfere with each other. This aligns with the goal that **independent transitions** in different regions can occur simultaneously for one event.

- **Avoiding Duplicate Ancestor Transitions:** XState prevents executing the same transition more than once per event across parallel branches. Internally, when an ancestor-level transition is eligible (one defined on a parent or the parallel state itself), the interpreter will recognize it as a single transition. In other words, XState won't fire the same transition action twice just because two regions share that ancestor. This is analogous to SCXML's approach – the transition is tied to a specific state (the ancestor), and XState will handle it once, resulting in one set of exit/entry actions for that state. The selection algorithm inherently merges such cases, so a transition defined on a parallel state (or any common ancestor) is taken only once even if multiple child states received the event. (This behavior is implied by statechart semantics and confirmed by the lack of duplicate actions in practice.)

- **Internal vs. External Transitions:** One nuance in XState is how it handles *self-transitions*. By default, a transition that stays in the **same state** (self-target) is considered an "external" transition in SCXML terms – meaning the state will exit and re-enter. XState v4/v5 allow control of this via the `internal` vs `external` (or `reenter`) setting. By default, XState treats explicit self-targets as external (re-entering) [10], but if you use a relative target (like `target: .someState`) or specify `internal: true`/`reenter: false`, XState will *not* exit the state. This distinction becomes important for parallel states (discussed below), where re-entering a parallel state can restart all its regions. XState's designers have noted that blindly re-entering a parallel parent on a transition can produce surprising results (e.g. exiting unrelated regions) [11], so XState tends to favor internal transitions by default when the intent is to transition within the same state hierarchy without resetting it. Nonetheless, for standard external transitions, XState's behavior remains consistent with SCXML: it will exit affected states and avoid double-firing any one transition.

Proposed Event Dispatch Algorithm for lit-bit

To correctly handle events in the **lit-bit** statechart library (with parallel states and hierarchical transitions), the generated `send` method should follow a procedure inspired by the SCXML algorithm. The goal is to allow multiple independent transitions (one per orthogonal region, if applicable) while ensuring each distinct transition is executed only once. Below is a recommended algorithm breakdown:

1. **Gather Active Leaf States:** Obtain the current active state configuration – i.e. the set of all active *leaf* states (atomic states) in the statechart. In a parallel state, there will be one active leaf in each region. For a non-parallel (simple composite) state, there is just one active leaf. These are the starting points for evaluating the event.

2. **Collect Candidate Transitions:** For each active leaf state, attempt to find a transition that can handle the incoming event:

3. Start at the leaf state and check if it has an outgoing transition whose trigger matches the event (and whose guard condition, if any, is true). If none, move to its parent state and check there, continuing up the hierarchy toward the root [8].
4. Take **the first matching transition** found in this upward search (the innermost transition has priority for that branch). This mirrors SCXML's "first match in ancestry" rule [1].
5. If a matching transition is found, add it to the list of enabled transitions **if it's not already present** [2]. (Use an identity check, e.g. a pointer or unique ID for the transition definition, to avoid duplicates.) This step is crucial for not double-counting a transition that may be reachable from multiple active leaves. For example, if two active leaves share a parent state that defines a transition on this event, that transition will be discovered from both leaves – but you should add it to the list only once.

6. If no transition is found in the leaf or any of its ancestors, that branch simply contributes no transition for this event (the event is unhandled in that region).

7. **Handle No Transition Case:** If after checking all active leaves the list of enabled transitions is empty, no state handled the event. In this case, the event is ignored or treated as unhandled (unless your design specifies an error or default handler). No state changes occur. (This is analogous to SCXML doing nothing when `SelectTransitions(event)` returns an empty set.)

8. **Resolve Conflicts (One Transition per Region):** If multiple transitions were collected, they may need filtering to ensure they don't conflict:

9. Two transitions conflict if they would cause exiting of any common state [4]. In practical terms, this usually means they originate from states in the *same region or hierarchy*. For example, one transition might be from a parent state and another from one of its substates – obviously both cannot fire, since taking the inner transition precludes also taking the parent's transition.
10. To resolve conflicts, determine the *least common ancestor (LCA)* of the source states of each transition and see if any LCAs overlap. A simpler method: for each candidate transition, compute the set of states that would be exited if that transition fires (all active descendant states of its source up to the transition's LCA) [12]. If any two transitions have an overlap in those exit sets, they conflict.

11. Remove conflicting transitions by prioritizing the deeper (more specific) transition. If Transition A exits a state that is also part of Transition B's exit path, then A is coming from a deeper nested state while B is from a higher-level state. Prefer A and discard B [6]. This ensures, for example, that a transition from a specific substate will win over a transition defined on a parent state for the same event.

12. If two transitions are in completely separate parallel regions, they will not conflict at all (their exit sets are disjoint), so both are kept. The result of conflict resolution is that you have **at most one transition per orthogonal region** (and generally one per independent branch of the state hierarchy). This set is analogous to SCXML's final *ActiveTransitions* list.

13. **Execute Selected Transitions (Macrostep):** Now process the chosen set of transitions in a controlled sequence, ensuring each transition's effects (exits and enters) happen once:

14. **Exit Phase:** For each transition in the set, calculate which states will be exited when it fires. This includes the transition's source state and any active descendant states of that source down the hierarchy (essentially everything in that branch below the LCA of source and target) [12]. Collect all such states to exit. Then exit all these states *in the proper order*. The usual order is from inner states outward. If multiple regions are being exited, each region's active states should exit, and then the composite (parallel) state itself if it's also being exited. (SCXML specifies that when a parallel composite is exited, all its child states' `onexit` actions run before the parent's `onexit` [13].)
15. **Transition Action Phase:** Next, execute the effect or action associated with each transition (the code in the transition's effect, if any). In an implementation, you might execute them in some defined order (e.g. document order or the order of the transitions list) for deterministic behavior. These actions run after exits but before entering new states [13].
16. **Entry Phase:** Enter the target states of each transition. For each transition, perform the appropriate entry actions and state activations:
    - If a transition targets a **simple state**, enter that state (and run its `onentry` behavior).
    - If it targets a **composite state**, enter it and then immediately enter its initial child state (and so on down to a leaf).
    - If it targets a **parallel state**, re-enter the parallel state and activate all of its regions' initial states concurrently. (This means entering each region's initial substate and running their `onentry` actions). The entry sequence should mirror exit: first the higher-level state's entry, then each region's entry, unless the semantics dictate the reverse; in SCXML, entering a parallel means all child regions are entered "simultaneously" (conceptually) [14].

17. **Update Active State Set:** Update the record of active states to the new configuration produced by these transitions. At this point, the statechart has moved to a new stable configuration.

18. **Conclude Event Processing:** Mark the event as processed. The dispatch (`send`) method call for this event ends. Any further transitions will be in response to new events (or internal events, timers, etc., handled in separate calls or loop iterations). In SCXML terms, one external event triggers one **macrostep** (which might comprise multiple transitions as above), and then the machine is ready for the next event.

This algorithm ensures that for a single event, **each enabled transition is executed exactly once**. If a transition is reachable from multiple active states (due to a shared ancestor), our collection step and uniqueness check guarantee it ends up in the set only a single time [2]. And by resolving conflicts, we avoid

attempting two transitions that would step on each other (you choose the appropriate one). The above approach matches the intended design goals: supporting multiple independent transitions in parallel regions, without duplicating effects for transitions on shared ancestors.

Managing Multiple Transitions and Event Consumption

A key consideration in the dispatch loop is whether to **stop after the first transition** or to continue searching for others. In a hierarchical (non-parallel) context, as soon as one transition is taken, that typically consumes the event – you wouldn't look for another because only one path can handle it. However, in a parallel context you *must not stop* at the first match; the algorithm should find all enabled transitions across different regions before concluding the event:

- **No Global Break on First Match:** The dispatch logic should not globally break out as soon as a transition is found, because other regions might also have transitions for the same event. Instead, gather all candidates (as described above) and then pick the set that will fire. For each individual leaf state, you *do* stop going up its ancestry once you find a match (you don't continue to higher ancestors for that branch) [1], but you continue this process independently for each active leaf. This way, all regions get a chance to handle the event. The event is effectively "broadcast" to all active states, and each region can capture it with at most one transition.

- **Consume the Event after Macrostep:** Once the selected transitions execute, the event is considered handled/consumed. Libraries like SCXML and XState do not re-process the same event again. SCXML goes as far as removing the event from the event queue after the macrostep, and XState's state transition function returns the new state immediately after handling the event. If additional transitions become enabled *as a result* of the state change (for example, an *eventless transition* or a queued internal event), those are handled in subsequent microsteps or event turns, but not as part of the same dispatch loop for the external event.

- **Tracking Processed Transitions:** Both SCXML and XState internally maintain data structures to track which transitions have been selected in the current step. In our algorithm, we use a list/set of enabled transitions to track this. This prevents double-processing the same transition. For instance, SCXML's `SelectTransitions` explicitly checks membership before adding a transition to the list [2]. lit-bit's `send` implementation can use a boolean flag or set for each transition (or transition ID) to mark that it's already handled for the current event, so if another active state finds the same transition, it won't be added again. After the event dispatch is finished, this structure can be cleared or discarded, since the next event will have its own new selection process.

- **Looping Behavior:** Typically, the `send` method as generated would perform the above steps once per event call. It doesn't loop indefinitely. It might use a loop internally only to gather transitions or to handle internal cascade events. For example, after executing transitions, there could be a check for any *immediate (eventless) transitions* that became enabled; if so, it might loop to handle those without a new external event (this is akin to an internal microstep loop in SCXML). But for the scope of a single external event dispatch, you gather the set of transitions and execute them, then you're done. You would **not** keep searching for new transitions in the new state for the *same* event – that event is done. (Any new event would be a separate call to `send`.)

In summary, the dispatch algorithm should **process all relevant transitions in parallel branches in one go**, and then stop. This ensures maximal responsiveness (all regions react to the event) while preventing any one event from being handled twice by the same transition logic.

Self-Transition Semantics in Parallel States

Self-transitions (a state transitioning to itself) in the context of a parallel state require careful consideration, as they can imply re-entering a complex state. The expected semantics differ slightly between SCXML (which defaults to external transitions) and how one might design lit-bit's behavior (potentially with options for internal transitions):

- **SCXML External Self-Transition on a Parallel:** By default, a self-transition in SCXML is **external**, meaning the state exits and re-enters fully [10]. If the state in question is a **parallel composite**, an external self-transition will cause the entire parallel state to exit and re-enter. In effect, all active child states in *all regions* will be exited, and then the parallel state (as a whole) is re-entered, which entails entering each region's initial state again. The SCXML specification confirms this behavior: if a transition causes a parallel state to be left (even by a self-loop), *all* child states are exited (child `onexit` handlers run first, then the parallel state's `onexit`) before the transition action executes, and then the state (and all its regions) are entered afresh [13]. This means the parallel state's own `onexit` and `onentry` will fire, and every region resets to its start. The order of operations in such a self-transition would be:
- Exit all active substates of the parallel (each region's states) – e.g. `Exit R1`, `Exit R2`, etc., in some defined order (SCXML uses document order for regions) [13].
- Exit the parallel state itself (`Exit Parallel`) [13].
- Execute the transition's action (if any) [13].
- Re-enter the parallel state (`Enter Parallel` onentry).
- Enter the initial state of each region (`Enter R1.initial`, `Enter R2.initial`, etc.).
- Continue into deeper initial states if those regions have nested structure, invoking their `onentry` actions.

All these steps happen as part of handling one event. In lit-bit's logs, a correct external self-transition of a parallel would appear as a single sequence of exits and entries covering all regions once.

- **Avoiding Duplicate Exits/Entries:** The problem described (duplicate `[ExitR1, ExitR2, ExitParallel, ... EnterParallel...]` appearing twice) was caused by treating the same parallel transition as two separate transitions. Using the improved algorithm, a self-transition on a parallel state would be detected once (added to the enabled set once) and executed once, so its exit/entry actions run a single time. That resolves the duplication. Essentially, the parallel state's transition will be processed as a single unit, so you'll get one set of exits and one re-entry.

- **Internal Self-Transition Option:** If the design allows, one might implement an **internal** self-transition (one that does not exit the state). In SCXML, this is possible by specifying `type="internal"` on the transition. In XState, one can specify `reenter: false` (or use a dot prefix target) to achieve the same. An *internal* self-transition on a parallel state would *not* exit the parallel state or its regions at all – it would simply execute the transition's actions (effect) while the configuration remains active. This effectively "refreshes" nothing; it's just an internal event handling. XState's default behavior for transitions defined at the machine level or using relative targets is often

to treat them as internal to avoid resetting parallel regions unintentionally [15]. In lit-bit, you could consider supporting a similar distinction (perhaps via an attribute in the macro or different method) if you want to allow transitions that don't restart the state. By default, however, you may stick to external semantics for clarity unless explicitly overridden.

- **XState's Semantics for Parallel Self-Transitions:** XState historically defaulted to external for self-transitions, like SCXML, which means a parallel self-transition *would* re-enter all regions. This can surprise users, as noted by XState contributors: for a parallel state, the *least common ancestor* of a transition from one region to itself can be the implicit root, causing *other regions to exit and re-enter as well* [11]. XState v5 addresses this by making the default transition type "internal" when using certain shorthand notations, to avoid unintended re-entry of sibling regions. The net expectation, though, is that unless you explicitly opt into an internal transition, a self-transition on a parallel state *does* mean the whole parallel state recycles (exits/enters). For consistency with statechart semantics, lit-bit should likely do the same by default.

Expected Order of Operations: In summary, for an external self-transition on a parallel state, the expected order is: **exit children → exit parallel → transition action → enter parallel → enter children** [13]. All child regions are effectively "refreshed." This is the behavior you should implement for lit-bit when a parallel state's own transition fires (unless you introduce an explicit internal transition feature to alter it). By adhering to this sequence and the selection algorithm above, lit-bit will correctly handle parallel self-transitions without duplication and with proper entry/exit balancing.

Sources:

- W3C SCXML Specification – Algorithm for transitions selection and conflict resolution [1] [2] [4] [5]
- W3C SCXML Specification – Parallel state semantics (entry/exit order) [13]
- Stately AI XState Docs – Event handling and parallel transitions [8] [9]
- Stack Overflow (XState) – Self-transition (internal vs external) explanation [10]
- XState Discussion – Parallel self-transition and reentry behavior [11]

---

[1] [2] [3] [4] [5] [6] [7] [12] [13] [14] State Chart XML (SCXML): State Machine Notation for Control Abstraction
https://www.w3.org/TR/2007/WD-scxml-20070221/

[8] [9] Events and transitions | Stately
https://stately.ai/docs/transitions

[10] It it possible to define a state that transitions to itself in xstate? - Stack Overflow
https://stackoverflow.com/questions/58901934/it-it-possible-to-define-a-state-that-transitions-to-itself-in-xstate

[11] [15] Surprising external transitions behavior for parallel states · statelyai xstate · Discussion #1829 · GitHub
https://github.com/statelyai/xstate/discussions/1829