ChatGPT

Extending the Rust Statechart DSL Macro for Timers and Guard Rules

Supporting Timer Transitions (`after(Duration)` Syntax)

- **Parsing the `after(...)=` Syntax:** To introduce timer-based transitions, the procedural macro's parser should recognize the `after` keyword and an associated duration expression. The best practice is to define a custom keyword in `syn` (e.g. `syn::custom_keyword!(after)`) and add a parse branch for it [1]. The grammar (from the project spec) would allow a construct like:

```
after DURATION => TARGET_STATE [action ...];
```

For example, `after 5s => TimedOut [action .handle_timeout];` triggers a delayed transition [2]. Instead of a custom literal (`5s`), we want to use a Rust expression for the duration. This means after parsing the `after` token, we can parse a `syn::Expr` (enclosed in parentheses) that should evaluate to a `core::time::Duration`. For instance, the user could write `after(Duration::from_secs(5)) => StateB;`. Using normal Rust `Duration` APIs keeps the DSL in line with Rust syntax (no special `.ms()` methods or suffixes), simplifying parsing and leveraging the type system. We can use `ParseStream` to expect `after`, then a parenthesized expression. For example:

```
input.parse::<keywords::after>()?;
let content; parenthesized!(content in input);
let dur_expr: syn::Expr = content.parse()?;
let arrow: Token![=>] = input.parse()?;
let target: Path = input.parse()?;
```

This treats everything inside `after(...)` as a normal Rust expression, so constructs like `Duration::from_millis(250)` are accepted. The macro's AST can store this duration expression (as a `syn::Expr`) in a new variant (e.g. an `AfterTransitionAst`). This new variant would be included in the state's body alongside event transitions. By using Rust-native duration expressions, we avoid implementing a custom unit parser, and we keep the DSL **consistent with Rust's syntax** (a design goal) [2].

- **Code Generation with Conditional Timer Calls:** When expanding the macro, we need to generate code that triggers the transition after the specified delay. A zero-cost way to do this is to leverage conditional compilation and an **abstract timer interface** rather than baking in one specific runtime. We can introduce a `TimerService` trait in the library that provides a platform-agnostic API for sleeping. For example:

```rust
pub trait TimerService {
    type SleepFuture: core::future::Future<Output = ()>;
    fn sleep(duration: core::time::Duration) -> Self::SleepFuture;
}
```

Then, provide **feature-gated implementations** of this trait for each supported runtime. Under the `async-tokio` feature, implement it to call `tokio::time::sleep(duration)` (which returns a `Future` that implements `Future<Output=()>`). Under `async-embassy`, implement it to call `embassy_time::Timer::after(duration)` (Embassy's async timer) [3]. This pattern ensures that when the user enables Tokio support, `TimerService::sleep` uses Tokio's clock, and for Embassy it uses the hardware timer, all under a unified interface. Crucially, if the user is in a `no_std` context without `async`, these implementations are excluded, so there's **no cost or dependency on those frameworks** (the trait can be empty or omitted). This design is similar to how the Backoff/Retry crate *BackON* abstracts sleeping: it defines a `Sleeper` trait with an associated `Sleep` future, and provides different impls (Tokio, Embassy, etc.) behind feature flags [4] [5].

- **Integrating into the Macro Expansion:** With `TimerService` in place, the macro can generate code that calls it. One approach is to insert an **async block or future** in the state machine's transition logic. For example, when an `after(duration) => S` transition is present in a state, the macro could generate something akin to:

```rust
#[cfg(feature = "async")]
{
    let dur = #duration_expr;  // from the parsed syn::Expr
    // Use the TimerService to sleep asynchronously, then trigger
transition:
    lit_bit::TimerService::sleep(dur).await;
    // After await, fire an internal event or call transition function:
    self.process_event(&InternalEvent::TimerFired(#state_ident));
}
```

This pseudocode assumes the state machine is being run inside an async context (e.g., as part of an actor or an async task). We use `#[cfg(feature="async")]` or more specific flags (`async-tokio`, `async-embassy`) around the code so that if the async feature is disabled, this block is omitted entirely (no overhead for sync builds). The **conditional compilation** approach ensures zero cost for those not using timers, and allows the macro to emit different code depending on the enabled runtime. For example, under `async-tokio` the expansion might literally call `tokio::time::sleep(duration).await`, whereas under `async-embassy` it would call Embassy's API – this can be achieved either via the trait or by inlining `#[cfg]` guards in the quoted output. The key is that only one of these gets compiled in. This pattern of cfg-driven code generation is standard for cross-platform Rust; it allows us to target Tokio's and Embassy's timers without forcing a runtime choice on the user.

- **TimerService Trait Design:** The `TimerService` trait as outlined uses an associated `SleepFuture` to encapsulate the future type (Tokio's `Sleep` type, Embassy's timer future, etc.). This means the compiler knows the exact future type at compile-time for each runtime, and it can optimize accordingly (no dynamic dispatch or box allocation). For example, an implementation for Tokio might be:

```
#[cfg(feature = "async-tokio")]
pub struct TokioTimer;
#[cfg(feature = "async-tokio")]
impl TimerService for TokioTimer {
    type SleepFuture = tokio::time::Sleep;
    fn sleep(duration: core::time::Duration) -> Self::SleepFuture {
        tokio::time::sleep(duration)
    }
}
```

and similarly for Embassy. The statechart code could use a type alias (via feature flags) like `type Timer = TokioTimer;` so that `Timer::sleep(dur)` calls the correct one. Another design is to make `TimerService::sleep` a **free function** (or associated function on a struct) chosen by cfg—this is simpler but less extensible. The trait approach, however, makes it easier to inject custom timer behavior (for example, a mock timer in tests) and follows the zero-cost abstraction ethos. In summary, model the timer similar to how one would model a hardware abstraction: a trait with implementations for each platform. This avoids heap allocation and keeps the API `no_std`-compatible by default (since the futures from tokio/embassy are `!Unpin` but can be polled without heap, and on bare-metal one might use a different mechanism) [3].

- **Cancelation and Runtime Considerations:** One subtle aspect of delayed transitions is canceling the timer if the state exits early. This typically means if we spawn an async task to handle the delay, we need to cancel it on state exit. A possible implementation strategy is to schedule an **internal event** instead of spawning a detached task. For example, in an actor model, one could post a delayed message to self. In Tokio, this could be done with `tokio::spawn` plus `sleep().await` then send an event, or by using `tokio::time::sleep_until` with a deadline and selecting between external events and the timer. In Embassy (which is `no_std`), one might use `embassy_time::Timer::after` directly in an `async fn` that is `await`ed within the state machine's own async handling. The spec suggests that for pure `no_std` (no async runtime), the user might call a `tick()` method to advance timers [3]. Implementing that is more involved (it requires the state machine to track remaining durations and decrement them), but the macro could support it by generating code under a `#[cfg(not(async))]` that registers the timer in a list and a `StateMachine::tick(delta)` method. For now, focusing on Tokio/Embassy async cases, we rely on their schedulers to handle the wait. The main point is to keep **sync usage unaffected**: if the `async` feature (or timer feature) is off, the macro should either disallow `after(...)` syntax or no code for timers is generated. This ensures a binary that doesn't use timers has *no latent overhead* (no timer thread, no extra state) – fulfilling the zero-cost requirement.

- **Testing Timer Transitions:** Testing such macro-generated async behavior requires covering multiple feature combinations and using the proper runtime. Good strategies include:

- **Compile-time tests**: Use conditional test modules or separate crates to compile a statechart with `async-tokio` enabled and one with `async-embassy` enabled, ensuring that the code compiles and links against the intended runtime. This can catch issues where, say, a Tokio-specific call is emitted while Embassy is enabled.
- **Integration tests on runtime**: For Tokio, one can write an async test (`#[tokio::test]`) that creates the state machine, enters the state with an `after` transition, and uses `tokio::time::pause()` or a deterministic timer to advance time. Tokio's ability to manipulate the clock or simply waiting a bit (with a small buffer) can assert that the transition occurs. For Embassy, you might simulate an Embassy executor or use a controlled environment (Embassy has a `Duration` and timer that can sometimes be polled in tests). If direct testing on hardware is not feasible, at least compile the Embassy variant to ensure it doesn't regress.
- **Time abstraction in tests**: Since we introduced `TimerService`, we can also introduce a **test timer**. For example, implement `TimerService` for a mock that records the requested duration or immediately returns a ready future. The project's testing utilities already hint at such capabilities – there is a `TimeController` and other tools in `lit_bit_core::actor::test_utils` [6]. These could allow a test to simulate "advancing" time or to verify that a timer was set. By swapping in a dummy `TimerService` (maybe via a feature like `test-clock`), we can avoid waiting for real time in unit tests.
- **Cancelation test**: A crucial behavior to test is that if the state is exited before the timer fires (e.g., an external event triggers a transition sooner), the scheduled transition is canceled. In a Tokio scenario, if we spawned a task for the timer, cancellation could be achieved by dropping the JoinHandle. We should test that no timer transition occurs if the state changed early. This might involve internal instrumentation (like each `after` could record a token that gets cleared on exit) – tests should verify no spurious transition happens. This ensures the semantics match the spec (timer transitions are only taken if still in the state after the delay) [7].

Compile-Time Guard Restrictions (Disallowing `async` in Guards)

- **Why Disallow Async Guards:** Guard conditions in statecharts are meant to be quick, pure boolean checks (often just checking context data or event payloads). They should **not perform awaits or side effects**. Allowing an `async` guard (e.g. `[guard async { ... }.await]`) would imply pausing state evaluation mid-transition, which complicates the state machine's execution model and could block the event loop. To enforce this, we want a compile-time error if the user accidentally writes an async closure or attempts to `.await` inside a guard.

- **Detection via AST Inspection:** Using `syn`, after parsing the guard expression (which the macro already parses as a `syn::Expr` inside the `[guard ...]` brackets [8]), we can analyze its form. We specifically look for any `Expr::Async` or `Expr::Await` nodes. Syn's full AST provides these variants: an `async { }` block appears as `Expr::Async(ExprAsync)` and an `.await` on a future is `Expr::Await(ExprAwait)` [9]. We can implement a small check either by matching directly (if we expect the guard expression not to be too complex) or by walking the expression tree. For example:

```
fn reject_async_in_expr(expr: &syn::Expr) -> syn::Result<()> {
    match expr {
        syn::Expr::Async(async_block) => {
```

```
            return Err(syn::Error::new(async_block.async_token.span(),
                    "Guard conditions cannot be `async`. Guards must
return a bool without awaiting."));
        }
        syn::Expr::Await(await_expr) => {
            return Err(syn::Error::new(await_expr.dot_token.span(),
                    "Guard conditions cannot use `.await`. Guards must be
synchronous."));
        }
        // Recursively check inside blocks or closures, if needed:
        syn::Expr::Block(block) => {
            for stmt in block.block.stmts.iter() {
                if let syn::Stmt::Expr(e) = stmt {
                    reject_async_in_expr(e)?; // recurse
                }
            }
        }
        syn::Expr::Closure(closure) => {
            if let Some(body) = &closure.body {
                reject_async_in_expr(body)?;
            }
        }
        // ... handle other nested expression types as needed ...
        _ => {}
    }
    Ok(())
}
```

This function (or similar logic using `syn::visit::Visit`) will traverse the guard expression. In most cases, a guard will be a simple expression (maybe a comparison or a function call). The most likely async-related pattern to catch is an `async move { ... }` block or a call to an async function (which, if not awaited, yields a future instead of bool — that would fail to type-check later, but the error would be confusing). By explicitly catching `.await`, we also handle the scenario where a user *does* put the guard in an async context by mistake. Since `.await` is only legal inside an async block or function, seeing it in the macro input likely means the whole statechart is inside an async function (or they wrote `#[tokio::main]` on `main`). That would still parse, so we should flag it.

- **Emitting a Helpful Compile Error:** After detecting an async usage in a guard, we use `syn::Error` to stop compilation with a clear message. The macro can attach the error to the exact location of the misuse. For example, using `syn::Error::new_spanned(expr, "message")` will underline the guard condition in the user's code. The error message should guide the user to the solution. Something like: *"Guards must be synchronous (cannot contain `async` or `.await`). Consider moving this logic out of the guard: e.g., perform async work in an entry action or external event, and have the guard use a boolean flag/result of that work."* This turns a potentially confusing compiler error (or runtime bug) into an actionable compile-time feedback. The design in **lit-bit** puts heavy emphasis on clear errors for invalid definitions [10], so we'd integrate with that style. For instance, elsewhere the

macro already provides user-friendly errors (e.g., it rejects unsupported “dot notation” in actions with a custom message) [11] . We’d follow suit by using the same approach for guard errors.

- **Preventing `async fn` Calls in Guards:** One tricky scenario is calling an `async fn` without `.await` in a guard. For example, if the context has an `async fn check_something() -> bool`, and the user writes `[guard ctx.check_something()]`. This will **compile** the macro (since `ctx.check_something()` is parsed as a normal method call expression), but it actually returns a `Future`, not a `bool`. Therefore, the generated code will likely fail to compile with a type error (expecting bool, found impl Future). We can improve the error by detecting this pattern. It’s hard to know an arbitrary call is to an async fn without type info, but a clue is that an async function’s name might not itself indicate it’s async. We might rely on the compiler’s type error in this case. However, we can catch a common pattern: if the guard expression is just a path or method call, we could **optionally** allow the compiler to handle it. The compiler error would say something like “expected `bool`, found opaque type `impl Future<Output=bool>`”. To make this clearer, we could augment our error message if we detect an `Expr::Path` or `Expr::MethodCall` and we know the `async` feature is on – but this is speculative. Simpler is: our explicit checks for `Expr::Async` / `Await` handle the obvious cases that the user writes `async` in the macro input. If they call an async fn by mistake, the error will happen, just not with our custom message. We can document that “guards cannot call async functions unless the call is awaited beforehand”.

- **Testing Guard Restrictions:** We should add **UI tests** (using something like the `trybuild` crate) to ensure that forbidden guard usages produce the expected errors. For example, a test case with:

```
statechart! {
    // ... state definitions ...
    state A {
        on Event [guard async { true }] => B;
    }
}
```

should fail to compile. Using trybuild, we capture the compiler output and assert that it contains our custom message about async guards. Similarly, a case with `[guard future.await]` (in an async context) should be caught. We also test that a normal guard (no async) still compiles and works, both with and without the `async` feature enabled (to ensure our check doesn’t erroneously forbid things). These tests will guard our implementation (no pun intended) against regressions.

By performing these checks at compile time, we maintain the library’s guarantee that guards are pure and fast. The user will get an error early, rather than a runtime issue or a confusing type mismatch deeper in the codegen.

Parsing and Architectural Considerations

- **Extending the Macro Parser:** The introduction of `after(...)` transitions is a new grammar branch for the macro. We have to integrate this with the existing parsing of state bodies. Currently,

transitions are parsed when the parser sees an `on` keyword (for event-triggered transitions) [12]. We have two main options:

- **Unified Transition AST:** Extend the `TransitionDefinitionAst` to include both event transitions and timed transitions. This could be done by giving it an enum field or option for the trigger (either an event pattern or a duration). For example, `TransitionDefinitionAst` could gain a field like `trigger: TransitionTriggerAst` where `TransitionTriggerAst` is an enum `{ OnEvent(syn::Pat), AfterDuration(syn::Expr) }`. Then in the `parse` impl, we peek for `keywords::on` vs `keywords::after` and populate accordingly. This keeps all transitions in one list/struct, but some fields (like guard) would only apply to certain triggers (we might decide guards are not allowed on `after` transitions in this DSL). We'd have to ensure that if `trigger` is `AfterDuration`, we don't allow a guard clause in the syntax (the parser can error if it encounters `[guard ...]` after an `after` transition).
- **Separate AST for Timers:** Introduce a new struct, say `AfterTransitionAst { after_token: keywords::after, duration: syn::Expr, arrow_token: Token![=>], target: Path, action: Option<TransitionActionAst> }`. Then the state body AST can have an enum like `StateBodyItemAst::TimerTransition(AfterTransitionAst)` in addition to the existing `Transition(TransitionDefinitionAst)` and `NestedState`. This cleanly separates the parsing logic: a different parse function for `AfterTransitionAst` triggered when we see `after`. The codegen or IR phase can then handle these two kinds of transitions slightly differently. This separation might make the code more readable, at the cost of having two transition types in the IR. Given that timer transitions have some unique semantics (no event pattern, possibly no guard, and they need special runtime handling), treating them distinctly is reasonable.

Either approach can work. Many projects choose an approach that doesn't overly generalize disparate syntax forms, to keep parsing straightforward. Here, option 2 (separate AST) is attractive for maintainability: the `TransitionDefinitionAst` remains focused on event triggers, and we bolt on a small new parser for `after` triggers. In either case, the initial parse happens in one pass. We don't need a completely separate "phase" to parse timers; we just extend the grammar rules. It's important to handle ordering or lookahead carefully – e.g., ensure that `after` as an identifier can't be misinterpreted as something else. Since `after` is now a reserved keyword in this macro's context, we should forbid state names or events named "after" (the macro's grammar likely already treats unrecognized keywords as errors, so that's fine).

- **Semantic Analysis Phase:** The project uses a multi-phase macro expansion: first building an AST, then constructing an intermediate representation (IR) for semantic checks, then generating code [13]. We should leverage this to enforce more complex rules and to inject the timer handling code. For example, once we have the IR (perhaps a `TmpTransition` struct for each transition), we can:
- Assign each `after` transition a unique internal event or ID so that we can refer to it in generated code (e.g., generate an enum variant like `Event::(state_name)_TimerExpired` or use a special event ID).
- Validate that the target states of `after` transitions exist (the existing code likely already checks that all transition target paths match a defined state [14]).
- Potentially, ensure that `after` transitions are only used in states that are not parallel/composites if that matters (the spec doesn't forbid it explicitly, but one might consider interactions with parallel states).
- Ensure that if the `no_std` environment is used without any async runtime, the user is aware that they must manually drive the timers (perhaps by a compile-time error or warning if `after` is used

with no supported timer features – the spec suggests a possible tick-based fallback, which could be future work).

Performing these checks in the IR stage, rather than all in the parsing stage, keeps parsing simpler and confines logic to the appropriate stage. This mirrors how frameworks like **Nom** handle parsing vs. validation – parse the structure first, then apply any context-sensitive rules in a second pass.

- **Balancing Feature Richness and Maintainability:** Adding features like timers and async integration can bloat a procedural macro if not managed well. Some strategies employed by similar projects include:
- **Modularize by Feature:** Clearly separate code related to optional features. For example, have a module or functions that generate the timer-handling code, only called when needed. Use `cfg` attributes to include/exclude blocks of the quote! output. This prevents the macro code from becoming a tangled maze of `if async_feature { ... } else { ... }`. In our case, since the statechart macro should work with or without async, we'll likely have `quote!` fragments guarded by `#[cfg(feature = "async")]` inside the generated implementation. This approach is analogous to how one might write manual code with conditional `#[cfg]` blocks; `quote` will faithfully include those attributes in the output tokens. The key is to make it **declarative**: e.g., embed `#[cfg(feature="async-tokio")]` in the output around tokio-specific code [3].
- **Offload Complexity to Library Code:** Whenever possible, do the heavy lifting in the runtime library rather than in the macro. For instance, by introducing the `TimerService` trait and possibly some helper functions in `lit-bit-core`, the macro's job is just to call `TimerService::sleep()`, rather than inlining a bunch of `#[cfg]` calls to different timer APIs all over. This makes the generated code cleaner and the macro easier to maintain. If later another runtime (say, async-std or smol) needs support, one can add a new impl of `TimerService` without modifying the macro's parsing logic – just gating the use via feature. This separation of concerns is important for maintainability.
- **Learn from Other DSL Macros:** The `statig` crate (a state machine library) is a good example: it uses an attribute macro to generate state machine code, and it automatically adapts to async or sync based on the user's state functions. It achieves this by detecting `async fn` in the input and then generating an `async` version of the state machine if needed [15] [16]. This is done without separate user syntax; the macro internally switches the output. We can mirror this philosophy: the presence of `after(...)` transitions could implicitly require an async context for that state machine. We then generate the necessary code behind the scenes. But we also must ensure that if the user hasn't enabled the `async` feature, using `after` yields a clear error (e.g., "enable the `async` feature to use delayed transitions") or a compile failure. Designing the macro to **fail gracefully** in unsupported scenarios is part of balancing complexity (it's better than generating incorrect code).
- **Testing and Documentation:** To keep things maintainable, write thorough documentation for these new features (the spec's section 4.6 on delayed transitions is a great start, and we should update it if needed to reflect any adjustments in syntax). Document any limitations (for example, "guards on `after` transitions are not allowed" or "using `after` in no_std requires calling `tick()` manually"). This saves future developers (and users) from misusing the features or introducing breaking changes unintentionally. Additionally, having extensive tests (as discussed) will act as a guardrail for future refactoring – if someone modifies the macro internals, the tests for timers and guard restrictions will catch deviations in behavior or performance (like ensuring a sync build truly has no async code paths).

In summary, the plan is to **extend the macro's grammar** to support `after(Duration::...)` transitions in a way that cleanly compiles down to the right async timer calls, while **preserving zero-cost for non-async use cases**. We achieve this via a feature-flagged `TimerService` trait and conditional code generation, much like abstracting over different timer backends. Simultaneously, we strengthen the macro's compile-time checks to **reject async guards**, maintaining that guards are simple boolean predicates. This is enforced by inspecting the syn AST and emitting user-friendly errors (using `syn::Error` and spanned diagnostics). By parsing and validating in appropriate phases (parsing for syntax, IR for semantic context), and by taking cues from how other frameworks handle optional async features, we ensure the macro grows in capability without becoming unmanageable. These enhancements will enable richer statechart scenarios (e.g. timeouts, delays) while keeping the library aligned with its design goals of clear errors, no_std compatibility, and high performance [10] [3]. The final outcome will allow users to write statecharts that, for example, say **"after 30s of being in state X, transition to Y"** in a natural way, and know that under the hood it will work correctly whether running on a microcontroller with Embassy or a server with Tokio – and if they accidentally write something that isn't allowed (like an async guard), the compiler will guide them to fix it before it becomes a bug.

Sources:

- Lit-bit Project Specification – *Delayed Transitions / Timers* design and grammar [2] [3]
- BackON crate documentation – example of abstracting sleep across runtimes via a trait [4] [5]
- Syn crate documentation – expression AST nodes for `async {...}` blocks and `.await` [9]
- Lit-bit Macro Code – pattern for emitting errors on unsupported syntax (e.g., dot notation in actions) [11]
- Lit-bit Testing Utilities – presence of `TimeController` for controlling time in tests [6]
- *Statig* state machine (mdeloof/statig) – handling of async vs sync state functions in a macro [15] [16]
- Lit-bit Project Goals – emphasis on clear compile-time errors and zero-cost abstractions [10] [3]

---

[1] [8] [11] [12] [13] [14] lib.rs
https://github.com/0xjcf/lit-bit/blob/051250e4287e771030950187c7fc6e406514576b/lit-bit-macro/src/lib.rs

[2] [3] [7] [10] Spec.md
https://github.com/0xjcf/lit-bit/blob/051250e4287e771030950187c7fc6e406514576b/Spec.md

[4] [5] backon - Rust
https://docs.rs/backon/latest/wasm32-unknown-unknown/backon/index.html

[6] test-guide.md
https://github.com/0xjcf/lit-bit/blob/051250e4287e771030950187c7fc6e406514576b/docs/test-guide.md

[9] Expr in syn - Rust
https://docs.rs/syn/latest/syn/enum.Expr.html

[15] [16] GitHub - mdeloof/statig: Hierarchical state machines for designing event-driven systems
https://github.com/mdeloof/statig