ChatGPT

Supporting External Event Enums in a Procedural Macro (Ergonomic & Const-Correct)

Problem Overview

In the `lit-bit` statechart system, we want to allow developers to use **plain Rust `enum` types** (even those defined in other modules or crates) as events in a statechart DSL macro. The macro should automatically generate **pattern-matching logic** (e.g. `match` arms for each event variant) and possibly **dummy values** for those variants' payloads, all while remaining **const-correct** (suitable for compile-time evaluation where needed). Crucially, we'd like to achieve this *without* burdening users with extra annotations like `#[derive(...)]` or `#[statechart_event]` on their event enums. In short, the goal is *zero boilerplate* for the user: they define a normal `enum` (which may have payload data in its variants) and the statechart macro "just works" with it.

This is a challenging goal due to Rust's macro system and type rules. We need to examine what is possible on **stable Rust**, identify any limitations, and determine the minimal user-side compromise if truly no-annotation integration isn't feasible. We will also compare our findings to how existing libraries handle similar problems (such as Strum, Serde, and enum-dispatch) to balance ergonomics versus macro complexity.

Limitations of Procedural Macros for External Enums

Rust's procedural macros operate purely on the **tokens** they receive at compile time – they do not have built-in reflection or type introspection capabilities on stable Rust. This means a macro cannot directly "look into" an arbitrary type (like an enum) unless that type's definition is provided to the macro. If the user only passes the name or path of an enum (especially one defined in another crate or module), the macro sees **no information about its variants or fields**. As one Rust expert succinctly put it:

> "Macros only have access to syntactical tokens, they don't have any type information. If you need to access the variants of an `enum`, your macro must be applied to the definition of the enum, not just the name of the enum." [1]

In other words, without some cooperation at the enum's definition site, a procedural macro **cannot enumerate an enum's variants** or know about its payload types. This is a fundamental limitation on stable Rust. There is no stable API for a macro to query the compiler's knowledge of a type's structure in another module or crate.

Implications for our statechart macro: If the event enum is external and unannotated, the macro cannot automatically generate exhaustive pattern matches for its variants, because it simply doesn't know what those variants are. It would also be unable to fabricate "dummy" values of that enum, since constructing an enum variant requires knowing its variant identifiers and the types of any payload fields.

Some specific technical challenges include:

- **Parsing the Enum Definition:** Procedural macros can parse an enum's definition (using `syn` or similar) *only if the enum is passed into the macro invocation*. In practice, this means using an **attribute macro or derive macro on the enum itself**. If the macro is invoked elsewhere (like as a DSL in another module) and only given a type name (e.g. `MyEvent`), it cannot pull in the source of `MyEvent` from another file or crate on stable Rust [1]. There is no reflection to get variant names or field types from just the type identifier.

- **Cross-Crate Orphan Rules:** One might consider having the user implement a trait (provided by the statechart library) on the event enum to supply variant info. However, if the enum comes from an external crate that the user does not control, implementing a foreign trait for it is often disallowed by Rust's orphan rule (the user's crate would be defining neither the trait nor the type in that case). This means even a manual trait impl isn't an option for truly external types unless the user wraps or newtypes them (which is essentially duplicating the enum – not ergonomic).

- **Enums with Payloads:** Enums that carry data (e.g. `enum Event { Foo(i32), Bar(String) }`) pose an extra challenge. Generating a pattern match arm for such variants is straightforward if you know the variant name and field list (you can produce `Event::Foo(x)` in the `match`). But generating a **dummy instance** of such variants at compile time is hard – what value should we use for the `i32` or `String`? Ideally, we might use something like `Default::default()` for payload fields, but that requires those types to implement `Default` (and for the Default impl to be usable in const context if we need a `const`). Not all types implement `Default` or have obvious "zero" values. Without user help, the macro cannot guess how to construct arbitrary types. Generating uninitialized dummy data is unsafe and not possible in const contexts (e.g. using `MaybeUninit` is not const-friendly and would be unsound if executed).

- **Const Context Requirements:** If we aim to do certain computations at compile time (for example, assign each event variant a constant ID, or build a static array of all variants), we run into Rust's const evaluation rules. Constructing complex or non-`Copy` types in a `const` is restricted. For instance, creating a `const EVENT_LIST: [Event; N]` array of all variants would require each variant (and its payload) to be constructible in a const context. Many payload types (e.g. `String`, or user-defined structs without `const fn` constructors) won't allow that. Thus, a fully const-initialized list of variants is often not feasible unless we impose additional bounds (like requiring payloads to be `ConstDefault` or something, which adds more burdens on the user).

- **Exhaustiveness vs. Extensibility:** Pattern matching on an enum can be *exhaustive* (list every variant) or include a catch-all `_`. For const-correctness and reliability, we might prefer exhaustive matches so that adding a new variant causes a compile error (ensuring the statechart handles it explicitly). But if the enum is external (especially from a third-party crate), it could be marked `#[non_exhaustive]`, meaning we **must** include a wildcard arm when matching it outside its defining crate [2]. Non-exhaustive enums can gain new variants in patch releases [2], so our macro would have to either always insert a wildcard (sacrificing compile-time checking of new variants) or somehow force users to update matches when the enum changes. This is a design trade-off to consider.

In summary, on stable Rust we **cannot magically introspect** an external enum's variants without some form of **user opt-in**. The next sections will explore what forms that opt-in could take and how to minimize the burden, as well as examine how other libraries tackle similar problems.

Techniques for Enum Introspection in Macros

Given the limitations, here are known techniques and workarounds to enable a macro to work with an enum's variants:

1. Attribute/Derive Macros on the Enum (Preferred)

The most robust solution is to ask the user to annotate their event enum with a procedural macro from our library. This could be a derive (e.g. `#[derive(StatechartEvent)]`) or an attribute macro (e.g. `#[statechart_event]`). By doing so, the macro gets to **parse the enum's definition** and generate helper code. This approach is essentially what popular crates like Strum, Serde, and enum_dispatch do:

- **Strum (Variant Iteration and Names):** The Strum crate provides derives that enumerate enum variants. For example, `#[derive(EnumIter)]` generates an iterator over all variants by literally injecting each variant into an array or match. Strum can't do this unless you derive on the enum because it must see all the variant identifiers. (Notably, Strum's `EnumIter` only works for enums without data associated with variants; if variants have payloads, Strum recommends deriving a separate discriminant enum – more on that below [3].) Users of Strum accept adding a `derive` as a minor inconvenience in exchange for auto-generated methods like `MyEnum::iter()` or variant name lookups.

- **Serde (Serialize/Deserialize):** Serde's derive macros (`Serialize`, `Deserialize`) generate code that matches on each variant to convert it to/from data formats. Under the hood, Serde's macros parse the enum and produce `match self { Enum::Variant1 => ..., Enum::Variant2(ref fields) => ... }` arms for every variant. Again, this is only possible because the user opts in with `#[derive(Serialize, Deserialize)]`. If the enum isn't annotated, Serde has no way to implement those traits automatically. (Serde even provides a special `remote` derive mode for external types – requiring the user to write a mirror definition – which underscores that without cooperation, it can't introspect external types either [4].)

- **enum_dispatch (Trait Delegation):** The `enum_dispatch` crate lets an enum delegate trait implementations to its variant types. The usage involves marking the enum with `#[enum_dispatch(MyTrait)]` and tagging the trait as well [5]. The macro then reads the enum's variants and generates `impl MyTrait for MyEnum { ... }` that internally matches each variant and calls the corresponding variant's implementation. This requires the macro to know each variant name (to call the correct variant's method), hence the need for the attribute on the enum definition [1].

In our case, a `#[statechart_event]` macro could do the following when applied to an enum: - **Gather variant names and payload types** via the `syn` crate. - Generate an internal **trait impl** or constants that the statechart runtime or macro can use. For example, it might implement a trait `StatechartEvent` that provides: - An associated constant or function returning the **number of variants** (like `const

`VARIANT_COUNT: usize`). Since stable Rust doesn't yet have a built-in for this, we would compute it by counting the parsed variants (similar to Strum's `EnumCount` derive, which provides a const for variant count [6]). - Perhaps an **array of variant IDs or names**, or a method to iterate variants. If variants have data, this could be an array of a parallel *discriminant enum* (discussed below) or simply an array of `&'static str` names for debugging. - Methods like `fn variant_index(&self) -> usize` or `fn as_discriminant(&self) -> EventKind` to map an enum value to some identifier. This could be implemented with a `match` on `self` covering all variants. This `match` would be exhaustive and thus const-friendly (as long as it only returns, say, a number or simple value) – it ensures at compile time we didn't miss a variant. For example:

~~~~
```rust
impl StatechartEvent for Event {
    fn variant_index(&self) -> u32 {
        match self {
            Event::Foo(_) => 0,
            Event::Bar(_) => 1,
            Event::Baz    => 2,
        }
    }
    // ...
}
```
~~~~

- Optionally, define **dummy constructors** for each variant if needed. For instance, the macro could emit something like `fn dummy_variant_Foo() -> Event { Event::Foo(Default::default()) }` for payload-carrying variants (requiring their types implement `Default`). This would provide a way to instantiate each variant in a default way. However, introducing such requirements (all payload types must be `Default`) or using `Default::default()` in const context (which is only allowed if the `Default` impl is const) can be limiting. We would weigh whether this is necessary or if there's a better approach (often, a separate discriminant enum obviates the need for dummy values – see below).

The attribute/derive macro approach is **minimal from the user perspective**: they add one annotation to their enum. This single annotation can generate everything needed behind the scenes. Compared to requiring users to manually list out variants in multiple places, one derive is relatively ergonomic and in line with common Rust practice. For example, a user might write:

```rust
#[statechart_event]
pub enum MyEvent {
    Started,
    Stopped(u32),
    Error(String),
}
```

And our proc-macro would generate (conceptually) something like:

```rust
impl StatechartEvent for MyEvent {
    const VARIANT_COUNT: usize = 3;
    fn variant_index(&self) -> usize {
        match self {
            MyEvent::Started => 0,
            MyEvent::Stopped(_) => 1,
            MyEvent::Error(_) => 2,
        }
    }
    // possibly: const NAMES: [&'static str; 3] = ["Started", "Stopped",
"Error"];
    // possibly: type Discriminant = MyEventKind; (if using a separate enum)
}
```

This data can then be used by the statechart macro expansion to drive pattern matching or to build lookup tables, etc. The key is that **by parsing the enum**, we offload all the variant enumeration work to the macro (not the user) while preserving type correctness.

Why prefer one custom attribute over zero? Because, as established, zero is not realistic on stable Rust for external types. The one attribute/derive gives us full insight into the enum with minimal effort from the user. This is essentially the route taken by Strum, Serde, enum_dispatch, and many others – a reasonable compromise between ergonomics and functionality.

2. Implementing a Trait Manually (User-Driven)

An alternative (when avoiding custom derives) is to ask the user to implement a known trait for their event type, enumerating variants themselves. For example, we could define in `lit_bit` something like:

```rust
trait StatechartEvent {
    const VARIANT_COUNT: usize;
    fn variant_index(&self) -> usize;
    // ... perhaps default methods or associated data
}
```

A user could manually implement this for their enum, essentially writing the same kind of match we showed above. However, this has several downsides:

- It's **boilerplate-heavy and error-prone**: the user must manually keep the implementation in sync with the enum definition. If they add a new variant but forget to update the trait impl, the code may compile but be incorrect (if we provided a default wildcard arm) or fail to compile if we required exhaustive matching. It's exactly this sort of repetitive task that procedural macros are meant to automate.
- For external enums (in a different crate), the user cannot implement our trait at all if both the trait and enum are foreign (orphan rule, as discussed). So this approach only works for enums in the

user's crate. If the event enum comes from a third-party crate, the user's hands are tied unless the third-party crate itself provides an implementation or they use a newtype wrapper.
- Requiring a manual impl doesn't actually remove the need for macro support; it just shifts the work to the user. Given that one of our goals is ergonomics, this is a step backward. We want *less* work for users, not more.

In practice, we would prefer an automated solution (like the derive above) over expecting users to hand-write trait impls. It's worth noting that some crates do use trait implementations as a form of interface (for example, some state machine libraries have the user implement a trait with an `on_event` method for each state instead of using a macro). But in our context of generating pattern matches for variants, a trait alone wouldn't solve the enumeration problem without the user doing exactly what a macro would do.

3. Explicit Variant Listing in the Macro Invocation

If we insisted on no attributes on the enum, another workaround is to have the user supply the variant information **to the macro directly**. For example, the statechart DSL could require the user to list the event variants in some form. This could look like:

```
statechart! {
    // Hypothetical syntax where the user enumerates variants:
    events: MyExternalEvent { VariantA, VariantB(u32), VariantC },
    // ... state transitions ...
}
```

Here, the user essentially duplicates the enum's variant names (and payload types) inside the macro. The macro could parse this and thereby know about all variants. It could then generate code accordingly (pattern matches, etc.). This achieves the technical goal, but at a high cost: - **Duplication**: The user must repeat the enum definition (at least the variant signatures) in the macro. This is tedious and risks mismatches. If the actual enum is changed, the user must remember to update the macro invocation as well. - **Ergonomics**: This is a pretty poor developer experience. One might almost prefer to just mark the enum with an attribute (which is less work and single-source-of-truth) than to have to maintain two parallel definitions. - **Error Handling**: The macro could potentially verify at compile time that the listed variants exist on the given enum (by generating references to them and seeing if it compiles), but error messages might be confusing if there's a typo. It also can't easily verify the payload types match exactly without attempting to use them. This could lead to cryptic compile errors if, say, the user writes `VariantB(u64)` in the macro but the real enum has `u32`. In contrast, a derive on the actual enum would directly see the true definition and never have such discrepancy.

Some older Rust libraries and macros (before proc-macros became powerful) did require manual listing of things, but this has fallen out of favor due to the maintenance burden. Therefore, while this approach avoids modifying the enum, it's typically not worth the hassle. It's essentially the opposite of the DRY principle (Don't Repeat Yourself).

One scenario where not listing all variants *might* be acceptable is if the statechart transitions themselves implicitly enumerate the events of interest. For example, if the DSL is something like:

```
statemachine! {
    transitions: {
        State1 + MyEvent::Foo => State2,
        State2 + MyEvent::Bar(u32) => State1,
        // ...
    }
}
```

In this case, the macro sees `MyEvent::Foo` and `MyEvent::Bar(_)` as patterns in transitions. The macro could collect those variant names and generate match arms for exactly these variants. If an event variant is never mentioned in any transition, it would effectively be ignored (unhandled) by the state machine. This is workable (similar to how the `smlang` DSL only concerns itself with listed transitions). However, it means unmentioned variants are silently dropped or unhandled. If the goal is to ensure all variants are accounted for (const-correct exhaustive handling), this approach falls short – a variant could be added and the statechart would not know about it until the user adds a transition or wildcard. We could incorporate a default catch-all for “unhandled” events in each state, but then we lose exhaustive matching (and a newly added variant might just hit the default handler without notice).

In summary, letting the DSL usage enumerate events is somewhat more ergonomic than an explicit separate list, but it only covers variants that have transitions. It might be acceptable for a design where any event without a defined transition is automatically ignored. But if we want to flag unhandled events or use events for other purposes (like generating documentation or diagrams of the statechart), we still benefit from knowing the full set of events.

Given these trade-offs, the **most practical minimal compromise** is still to ask for one annotation on the enum itself (approach #1), rather than forcing the user to replicate information in the macro invocation or implement large traits.

4. Using a Discriminant Enum or IDs for Payloaded Variants

A special mention is warranted for handling enums with data in their variants (payloads). As noted, creating actual dummy values of each variant can be problematic. A common technique in libraries is to separate the **variant identity** from the payload. This means generating a parallel enum (or constants) that represent just the **discriminants** (variant names), with no payloads attached.

For example, the `enum-kinds` crate will generate a new enum with the same variant names but without fields, for any given enum you annotate [7]. Strum offers a similar feature via `EnumDiscriminants` derive, which produces `MyEventDiscriminant` enum alongside your `MyEvent` [3]. Each variant in `MyEventDiscriminant` corresponds to a variant in `MyEvent`, but carries no data. Strum even lets you derive `EnumIter` on the discriminant enum, so you can easily get a static array of all variant identifiers [3]. This approach neatly sidesteps the need to create dummy payload values: you can iterate or match on the discriminant enum exhaustively, and if needed, convert from the original enum to the discriminant (e.g. via an auto-generated method or `From` impl).

How we can apply this idea: - Our `#[statechart_event]` macro could generate an internal **const mapping** of variant discriminants to, say, numeric IDs or a new enum. For instance, produce:

```
#[derive(Copy, Clone, Eq, PartialEq)]
enum MyEventKind { Started, Stopped, Error }
impl From<&MyEvent> for MyEventKind {
    fn from(e: &MyEvent) -> Self {
        match e {
            MyEvent::Started => MyEventKind::Started,
            MyEvent::Stopped(_) => MyEventKind::Stopped,
            MyEvent::Error(_) => MyEventKind::Error,
        }
    }
}
const ALL_EVENTS: [MyEventKind; 3] = [MyEventKind::Started,
MyEventKind::Stopped, MyEventKind::Error];
```

Here, `MyEventKind` is essentially the "dummy" enum without data. We can iterate `ALL_EVENTS` at compile time, use it for exhaustiveness checks, etc., without worrying about constructing a `String` or other payload. The statechart macro might use something like `MyEventKind` behind the scenes for things like building transition tables or printing variant names.

- Alternatively, we could assign each variant a **const integer ID**. Since Rust 1.66, it's even possible to put explicit discriminant values on data-carrying enum variants (e.g. `Foo(String) = 1, Bar(u32) = 2`) [8] [9]. With `#[repr(u8)]` or similar, one could ensure a stable representation and theoretically cast to get the discriminant at runtime. However, there is no stable, safe way to extract that discriminant as a number without either using the unstable `variant_count`/`discriminant_value` intrinsics or doing unsafe pointer casts [8] [10]. So while assigning explicit discriminants can make for nice IDs (and we might do it for consistency), we would still generate our own mapping function or table for those IDs in safe code. In short, using our own discriminant enum or match is simpler and safer than relying on reading enum layout bytes.

Using a parallel "kind" enum is effectively what Strum and enum-kinds do to handle payload enums. It does mean we're generating an extra type under the hood, but this is entirely transparent to the user (unless we choose to expose it for their use). The benefit is that we can get a **const list of all variant identifiers** easily (since they are just simple unit-like values) and we can pattern-match on these in const contexts or at runtime with ease. This helps achieve the "const-correctness" goal, for example by allowing a `const fn is_valid_event_kind(x: u8) -> bool` that checks against the ID range or by building compile-time structures for transitions keyed by event kind.

5. Unstable Introspection (Future Possibilities)

Currently, stable Rust lacks any direct reflection API, but there are ongoing discussions and proposals to improve this in the future. A couple of relevant developments:

- `std::mem::variant_count` **Intrinsic:** As of Rust 1.67+, an intrinsic `variant_count::<T>()` exists (nightly only at the moment) to get the number of variants of an enum type [11]. If this becomes stable, a macro-less approach could know how many variants an external enum has, but *not* what they are named or what data they carry. It's a partial solution – useful for allocating arrays, perhaps, but insufficient for generating pattern matches or identifying each variant. (Also, if an enum is non-exhaustive, `variant_count` could change when new variants are added, and the function itself notes that it's not a substitute for exhaustive matching in those cases [2].)

- **Compile-Time Reflection RFCs:** There's an active interest in Rust for *safe compile-time reflection*. A proposed concept (sometimes called "Mirror" or `core::introspect`) would allow code to introspect type structure at compile time [12]. For example, one could imagine a future where a const function could iterate over an enum's variants or retrieve metadata about them. Such features are complex and not yet available. If they materialize, it might become possible for our macro (or even a library function) to generically handle external enums without explicit user annotations. However, this is speculative and likely years away. As of mid-2025, the pragmatic approach is still to use procedural macros for these tasks.

- **Const Trait Implementations:** Rust is gradually allowing traits to be implemented in a const context (`~const` trait bounds and const functions). If our `StatechartEvent` trait had a const method to get variant info, and if we could somehow impl it generically for any enum, that could be another path. Unfortunately, without some reflection, a generic impl for all enums can't know their variants. There was an idea of using specialization or marker traits per variant, but that quickly becomes unwieldy and still needs macro generation of those impls somewhere. So this doesn't solve the core problem either – it just might let us call trait methods in const contexts if we have them.

In summary, **upcoming Rust features might eventually reduce the need for custom codegen**, but for now, none of them fully address our requirements on stable Rust. We can keep an eye on features like `variant_count` (if it stabilizes) to possibly simplify a bit of our macro's job (like automatically getting the variant count for error-checking), but they are not a replacement for the information a proc-macro derive can provide today.

Comparison to Existing Libraries

It's useful to see how similar problems are handled in the ecosystem, to validate that requiring a small annotation is acceptable and to possibly borrow techniques:

- **Strum:** This crate focuses on enum introspection (variant names, counts, iterations, etc.). It **always requires a derive macro** (from `strum_macros`) on the enum. For instance, to iterate variants you do `#[derive(EnumIter)]`, to get variant names you do `#[derive(EnumVariantNames)]`, and so on. Strum's derive will generate code that lists out each variant. It doesn't attempt any "magic" to avoid that – because it can't on stable Rust. Notably, Strum splits functionality into multiple derives (for modularity), but in our case we can provide one macro that does everything needed for statecharts. Strum's handling of payload-carrying variants is to not support iteration on them directly; instead it suggests deriving an `EnumDiscriminants` (which generates a new enum with the same variant names but no data) so you can iterate those [3]. This is a clear precedent for using a parallel enum to avoid dummy data issues.

- **Serde:** As mentioned, Serde requires derives for automated (de)serialization. If you want to implement Serde for an external type, you must either write it by hand or use the *remote derive* pattern (where you create a dummy representation in your crate and instruct Serde's macro to implement the trait for the external type via that dummy) [4] . This remote derive is an advanced tactic that still involves a lot of user-provided detail (essentially describing the type's fields in your own code). It's analogous to listing variants manually, just slightly more guided. The existence of remote derive shows that even one of Rust's most advanced macro ecosystems (Serde) cannot escape needing user help for foreign types. The minimal compromise there is the user writes a mirror of the type. For our use-case, we could theoretically offer a similar pattern (e.g. the user writes out a mirror enum inside our macro invocation), but as discussed in Technique #3, that's essentially the same as requiring them to list everything – not much better than remote derive in terms of labor.

- **enum_dispatch:** This library's approach is to keep user effort low but still require annotations. You tag your enum and the trait, and it generates the delegation. It does not try to inspect anything unless you mark it. The user benefit is significant (no need to write repetitive impls for each variant), and one attribute is a trivial cost. Our statechart event support is analogous – by adding one attribute, the user avoids writing a lot of boilerplate match logic for events in each state.

- **State Machine DSLs (smlang, statig, etc.):** The `smlang` crate provides a DSL for state machines via a proc macro. In its design, the user defines the events as a separate enum (manually) and then references them in the state machine description. `smlang` doesn't introspect the enum unless it's part of the macro input. In practice, `smlang` doesn't generate code to handle every event variant universally – it relies on the transitions listed. Any event not mentioned just isn't handled (or can be caught by a wildcard if you write one). This is a simpler model that avoids needing a derive on the enum, but at the cost of not having a concept of "all events" known to the machine. Our goals for `lit-bit` seem to include const-correctness (which implies we might want to pre-compute things like event indices or ensure exhaustive handling). If so, the attribute macro on the enum is a more powerful approach. If we were willing to live with the "mentioned events only" approach, we could potentially do without an attribute, as discussed earlier. But that introduces the risk of silently ignoring new/unhandled events.

In summary, **most libraries solving this class of problem choose to require a small opt-in from the user (usually a derive)**. This is widely accepted in the Rust community because it's a one-time addition that unlocks a lot of convenience. Our approach should align with this philosophy: provide a procedural macro (derive/attribute) that the user can apply to their event enums. The macro will handle all the heavy lifting of variant introspection and code generation. In return, the user's code remains clean and they don't have to repeat themselves. Compared to Strum or Serde, we wouldn't be asking for anything unusual – just one annotation. In fact, we can aim to make our attribute even more encompassing so the user doesn't need multiple annotations. (For example, Strum might need two derives to get both `VARIANTS` names and iteration; our single `statechart_event` macro can generate whatever is needed for our specific use-case in one go.)

Recommendations for the `lit-bit` Statechart System

1. Use a Procedural Macro Derive on Event Enums: To meet the goals on stable Rust, the most practical solution is to introduce a `#[derive(StatechartEvent)]` **(or attribute macro)** that users apply to their event enums. This derive will parse the enum and generate the necessary impls/structures. It ensures support for enums defined anywhere (user crate or external crate, as long as the user can modify or wrap them). It's a minimal opt-in with maximum gain: - The macro can create an internal **discriminant enum or constants** for variant IDs, enabling pattern matching without needing actual payload values. This covers enums with payloads in an ergonomic way (similar to enum-kinds and Strum's EnumDiscriminants) [7] [3]. - The macro can implement a trait (e.g. `StatechartEvent`) that the statechart runtime or generated code will use. For example, the statechart implementation might call `EventType::variant_index(&event)` to get a quick numeric ID, or compare `event.into(): EventDiscriminant` in const context. - We can design the trait such that it enforces const-correctness. For instance, `variant_index` can be a `const fn` (the macro will implement it with a pure match, which is const-friendly). This means we could, for example, build a `const TRANSITION_TABLE: [[State; EventType::VARIANT_COUNT]; StateCount]` at compile time if we wanted, using those indices. In other words, the combination of a macro-provided exhaustive match and const functions allows a lot of compile-time computation, which aligns with the "const-correct" goal. - The derive can also provide convenience methods like `fn variant_name(&self) -> &'static str` or an array of variant names, if needed for debugging or logging. These would be generated by simply hardcoding the names from the definition. (This is similar to Strum's `VariantNames` trait/derive [13].)

2. Handling External Enums: If an event enum comes from an external crate that the user cannot modify (and it isn't already annotated with our macro), we have a few options: - The **ideal** scenario is to encourage the user to create a local wrapper or alias that is annotated. For example, they could create `enum MyWrappedEvent { ExternalVariant1, ExternalVariant2(...) }` mirroring the external one, or even a one-variant wrapper like `MyEvent(ExternalEvent)`. However, the one-variant wrapper loses the variant distinctions (everything is just "ExternalEvent" variant), so mirroring each variant would be needed to truly preserve behavior. This is admittedly cumbersome. A more realistic approach is to **ask the external crate author to include** `#[derive(StatechartEvent)]` if the crate is under their control and meant to integrate with statecharts. If the external crate is domain-specific (e.g. a set of event types for an application), having them derive our trait is not unreasonable if they plan to use our library. - We could consider providing a **macro to implement the trait for an external enum via a mirror**, similar to Serde's remote derive. For example:

```
statechart_event_remote! {
    type = "other_crate::TheirEvent";
    variants = [ Foo, Bar(u32), Baz ]
}
```

This macro invocation in the user's crate could generate an impl of our `StatechartEvent` for `other_crate::TheirEvent` by internally defining a dummy enum and mapping. This saves the user from writing the match themselves, but they still have to list out the variants (and keep them updated). Essentially, this is a convenience for a scenario we expect to be uncommon. It's an option if supporting external events without modification is a high priority, but it still requires user effort (just slightly less error-prone thanks to the macro doing the mapping).

- If neither wrapping nor remote macros are desirable, the last resort is to handle external events as an opaque variant. For instance, the statechart could treat any external event not derived as a single wildcard case (e.g. "unknown event"). But that defeats the purpose of pattern matching on specific variants. So likely we stick with requiring some user action in these cases.

3. Integrate with Statechart Macro Logic: Once the event enum has the `StatechartEvent` trait (or equivalent) implemented by our derive, the main statechart procedural macro can leverage it. For example: - It can impose a trait bound in generated code: `where Event: StatechartEvent`. This ensures at compile time that the event type used in the statechart has our generated impl (or a manual impl). - The macro can use the associated constants or methods from `StatechartEvent` to build transition handling. One strategy is a **match dispatch** as in the example earlier: generate a nested `match` where the outer match is on the current state, and the inner match is on `event.variant_index()` or on `event_discriminant = event.into(): EventKind`. This becomes something like:

```
match current_state {
    State::State1 => match event.variant_index() {
        0 => { /* handle Variant0 in State1 */ }
        1 => { /* handle Variant1 in State1 */ }
        _ => { /* default for unhandled events */ }
    },
    State::State2 => { ... },
}
```

This is a more table-driven approach (using numeric IDs). Alternatively, the macro could generate direct pattern matches on the event for each state:

```
match current_state {
    State::State1 => match event {
        Event::Foo(data) => { ... }
        Event::Bar(x) => { ... }
        Event::Baz => { ... }
    },
    ...
}
```

Both are viable. The **direct pattern match** is more readable and leverages Rust's own exhaustiveness checking on the event match arms. However, if many states need to handle the same event variants, that can lead to duplicated pattern-match logic. The numeric dispatch (variant_index) approach might allow using computed tables of transitions. The choice depends on performance and code size considerations. Either way, having the variant list from the trait makes it possible to automate exhaustive matches or build tables. - **Const Dummy Values:** If our derive macro generated a parallel `EventKind` enum (discriminant), we can use that in const contexts freely. For instance, a `const TRANSITIONS: [[Option<StateId>; Event::VARIANT_COUNT]; STATE_COUNT]` could be built where we fill in state transitions keyed by `EventKind`. Constructing such a static would require all indices (events and states) to be known at compile time – which they would if both the states and events are known enums with `VARIANT_COUNT` constants. We'd use something like `EventKind::Foo as usize` as indices. This is advanced usage, but it shows that our design enables highly optimized handling if desired (no runtime match overhead, just array lookups). This is one of the benefits of const-correctness: we can choose a data-driven approach for speed, and the compiler will verify our array sizes and indexing because it knows the variant count at compile time. - The macro can also use the trait to provide better compile-time errors. For example, if a transition is specified for an event variant that doesn't exist, using the variant in the macro code will error out. But with a trait, we could potentially check something like "if variant name in DSL is not in `Event::VARIANT_NAMES`, throw a compile error" – though doing this inside a procedural macro is non-trivial. More straightforward is that if the user spelled a variant wrong in the DSL, Rust itself will error when our generated code tries to reference `Event::ThatVariant` (since our macro, having parsed the DSL, can simply emit the tokens and rely on Rust name resolution to catch invalid ones). - We should also ensure that if an enum is `non_exhaustive`, our derive macro either refuses to derive (forcing the user to handle it manually) or always generates a wildcard for future events. Perhaps the safest route is to detect `#[non_exhaustive]` on the enum (the `syn` AST will have this info) and then implement `StatechartEvent::variant_index` with a `_ =>` arm that maps any unknown variant to a catch-all ID (like returning `usize::MAX` or `None`). But since outside the crate new variants can't even be constructed (they'd be hidden), this may not be a big issue in practice. It's an edge case to document.

4. Balancing Dummy Values vs. Discriminant Approach: If possible, favor generating **discriminant enums or match arms** over trying to create actual dummy instances of the event. Creating dummy instances of each variant (especially with complex payloads) is fraught with difficulties: - The payload types might not be `Copy` or `Default`, making it impossible to fabricate a value without either calling some user-provided const constructor or using unsafe. - Even if payloads are simple (numbers), choosing a dummy (like 0 or -1) could be semantically misleading. It's better to avoid needing a value at all. - Pattern matching on variants doesn't require providing a payload value – you can use `_` to ignore it (`Event::Foo(_) => …`). So for generating match arms, we don't need dummy data, just the variant patterns. - Only if we needed a **collection of Event instances** (e.g. an array of `Event` covering all variants) would dummy values be required. But we likely don't need that; an array of discriminants or just the trait methods should suffice for any compile-time mapping. We can always map from discriminant back to an event at runtime if needed by calling a user-provided constructor or so (though typically, we handle events coming in, not generating new ones arbitrarily). - Therefore, the "discriminant enum + match conversion" approach is the cleanest. It's been proven by crates like enum-kinds (which explicitly advertises generating a no-data enum mirror [7]) and Strum's recommendation [3]. We should implement something similar in our derive macro.

5. User Documentation and Expectations: With this design, we would document that: - If users want to use an enum as a statechart event type, they should derive `StatechartEvent` on it (or use whatever attribute we provide). This is required for the macro to generate the pattern matching logic automatically. In practice, this is a one-line addition. - The derive will impose some trait bounds on payload types if necessary (for example, if we decide to use `Default` for payloads in some context, we'd require that in a bound). We should strive to minimize such bounds. Ideally, we impose none except maybe `Send + Sync` if the statechart needs that for concurrency – but that's unrelated to variant introspection. - We should compare this requirement with what similar libraries require. For instance, if someone is already familiar with Strum or Serde, they'll find our derive approach quite normal. We can even mention in docs "under the hood, this works similarly to how Serde derives or Strum works – it introspects your enum to generate code, since Rust doesn't have runtime enum reflection." This will set correct expectations and justify the design. - If zero-annotation usage is asked about, we can explain that Rust's current capabilities don't allow it on stable, referencing the need for attributes as a general Rust pattern [1]. Most Rust users will accept this given the language constraints.

6. Comparison to Not Using An Annotation: The only scenario where we might try to avoid the annotation is if all event handling is explicitly listed in the statechart DSL, and we are okay with not handling unspecified events. If we went that route, we could mirror `smlang` and not require any derive. However, this sacrifices some safety (unhandled events might be forgotten) and prevents us from doing fancy compile-time checks or optimizations with the full event set. Since the question explicitly targets “ergonomic, const-correct” pattern matching, it implies we do want the exhaustive, compile-time-known set of events – which the derive enables.

7. Testing the Approach: We should test our proc-macro on examples with: - Enums with no payload (simple cases). - Enums with several payload types (including non-Default ones, to ensure we handle them via pattern matching or discriminant approach without needing to instantiate). - Enums defined in another module or crate (to ensure our macro path works across crate boundaries; procedural macro crates typically handle this fine as long as the derive is applied). - Ensure that the generated code is const-friendly. For instance, try using a `const fn` with a match on the event – it should compile if our match arms only produce const values. If we use any heap or non-const operation, it wouldn’t, so we’ll avoid those. - Check that if the user adds a variant and forgets to update transitions, either they get a compile error (if we enforce exhaustive match in their code) or at least a warning. Likely, if our generated code uses an exhaustive match in `variant_index`, adding a variant will make that match non-exhaustive and cause a compile failure **inside our macro output**. That’s actually good: it means our derive macro would fail to compile, alerting the user that they need to update to a newer version of our library that handles the new variant (if the enum is from an external crate), or if it’s their own enum, it indicates they might need to regenerate or reconsider their statechart transitions. We might need to manage this carefully (maybe using a wildcard in `variant_index` if we expect variant changes, but if it’s user’s own enum, breaking compilation on new variant might be fine as it reminds them to handle it in state transitions).

8. Performance Considerations: The code generated should be efficient. Pattern matching on enums is a O(1) dispatch (often compiled to jump tables if there are many variants). Using a numeric `variant_index` and a table is also O(1). With const generation, we can push some work to compile time. None of the solutions should introduce undue runtime overhead. The derive macro approach actually helps the compiler see all possibilities explicitly, which can enable optimizations. (For example, a match on a known set of variants can sometimes be optimized to a direct jump or even inlined behaviors.)

In conclusion, **it is not feasible on stable Rust to implement ergonomic, const-correct event pattern matching for external enums without any user annotations**. The minimal compromise is to require a one-time annotation (proc-macro) on the event enum to expose its structure to the macro. This is a small ask that unlocks powerful code generation. Libraries like Strum, Serde, and enum_dispatch all make similar trade-offs, requiring a derive or attribute to do advanced enum manipulations [1] [3]. By following this approach in the `lit-bit` statechart system, we can support enums with payloads, generate exhaustive pattern matches and even const-friendly lookup tables, and do so with minimal friction for the user.

While we should keep an eye on future Rust introspection features, our implementation can proceed with the derive macro strategy today, providing a clear and reliable developer experience. Users will be able to write plain enums (no special supertraits or base classes needed as in other languages) and simply slap on a `#[derive(StatechartEvent)]` to integrate with the statechart DSL. This aligns with Rust’s overarching philosophy: use zero-cost abstractions and opt-in macros to get things done safely at compile time, rather than requiring verbose user code or runtime type reflection.

References:

- Rust forum discussion on enum variant access in macros (why macros need the enum definition) [1]
- Strum documentation on variant iteration and discriminant enums (handling of enums with data) [3]
- Crate `enum-kinds` for generating payload-less “kind” enums from enums with data [7]
- Rust unstable `std::mem::variant_count` (compile-time variant counting) [11] [2]
- Rust RFC 2363 notes (allowing explicit discriminants on data-carrying enums, relevant to assigning IDs) [9]

---

[1] Iterate over enum variants in macro - help - The Rust Programming Language Forum
https://users.rust-lang.org/t/iterate-over-enum-variants-in-macro/114836

[2] [11] variant_count in std::mem - Rust
https://doc.rust-lang.org/beta/std/mem/fn.variant_count.html

[3] [6] [13] strum - Rust
https://docs.rs/strum

[4] Derive for remote crate - Serde
https://serde.rs/remote-derive.html

[5] enum_dispatch - Rust
https://docs.rs/enum_dispatch

[7] enum-kinds - crates.io: Rust Package Registry
https://crates.io/crates/enum-kinds

[8] [10] Enum crimes - getting the discriminant - help - The Rust Programming Language Forum
https://users.rust-lang.org/t/enum-crimes-getting-the-discriminant/88103

[9] 2363-arbitrary-enum-discriminant - The Rust RFC Book
https://rust-lang.github.io/rfcs/2363-arbitrary-enum-discriminant.html

[12] A Mirror for Rust: Compile-Time Reflection Report - Shepherd's Oasis
https://soasis.org/posts/a-mirror-for-rust-a-plan-for-generic-compile-time-introspection-in-rust/