asyncmachine-go

Introduction

asyncmachine-go is a general purpose state machine for managing complex asynchronous workflows in a safe and structured way

asyncmachine-go abstracts everything and all (but only if necessary) as a state. Many states can be active at the same time, some of them can mutually exclude each other, some can be divided into several smaller states, some states can be nested sub-machines, and finally, some states can aggregate groups of nested sub-machines.

The purpose of **asyncmachine-go** is never to block (ie always be synchronous), which is achieved by splitting long-running actions into steps and orchestrating blocking calls according to a predefined convention. Another goal is to give structure to non-determinism, by embracing it.

Comparison

Common differences from other state machines:

- many states can be active at the same time
- transitions between all the states are allowed
- states are connected by relations
- every transition can be rejected
- error is a state

Legend

Examples here use a string representations of machines in the format of `(ActiveState:\d)[InactiveState:\d]` , eg `(Foo:1)[Bar:0 Baz:0]`.

Machine and States

Defining States

States are defined using `am.Struct`, a string-keyed map of `am.State` struct, which consists of **properties and relations**. List of **state names** have a readability shorthand of `am.S` and can be combined using `am.SMerge`.

```go
am.Struct{
    "StateName": {

        // properties
        Auto:    true,
        Multi:   true,

        // relations
        Require: am.S{"AnotherState1"},
        Add:     am.S{"AnotherState2"},
        Remove:  am.S{"AnotherState3", "AnotherState4"},
        After:   am.S{"AnotherState2"},
    }
}
```

State names have a predefined **naming convention** which is `CamelCase`.

Example - synchronous state

```
Ready: {},
```

Asynchronous States

If a state represents a change from `A` to `B`, then it's considered as an **asynchronous state**. Async states can be represented **by 2 to 4 states**, depending on how granular information we need from them. More than 4 states representing a single abstraction in time is called a Flow.

Example - asynchronous state (double)

```
DownloadingFile: {
    Remove: groupFileDownloaded,
},
FileDownloaded: {
    Remove: groupFileDownloaded,
},
```

Example - asynchronous boolean state (triple)

```
Connected: {
    Remove:  groupConnected,
},
Connecting: {
    Remove:  groupConnected,
},
Disconnecting: {
    Remove: groupConnected,
},
```

Example - full asynchronous boolean state (quadruple)

```
Connected: {
    Remove:  groupConnected,
},
Connecting: {
    Remove:  groupConnected,
},
Disconnecting: {
    Remove: groupConnected,
},
Disconnected: {
    Auto: true,
    Remove: groupConnected,
},
```

Machine Init

There are two ways to initialize a machine - using `am.New` or `am.NewCommon` . The former one always returns an instance of `Machine`, but it's limited to only initializing the states structure and basic customizations via `Opts`. The latter one is more feature-rich and provides states verification, handler binding, and debugging. It may also return an error.

```go
import am "github.com/pancsta/asyncmachine-go/pkg/machine"
// ...
ctx := context.Background()
states := am.Struct{"Foo":{}, "Bar":{}}
mach := am.New(ctx, states, &am.Opts{
    ID: "foo1",
    LogLevel: am.LogChanges,
})
```

Each machine has an ID (via `Opts.ID` or a random one) and the build-in Exception state.

State Clocks and Context

Every state has a clock. State clocks are logical clocks which increment ("tick") every time the state gets activated or de-activated. **Odd ticks mean active, while even ticks mean inactive.** The sum of all the state clocks forms the **machine time**.

The purpose of a state clocks is to distinguish different instances of the same state. It's the most commonly used by being abstracted as `context.Context` via `Machine.NewStateCtx(state string)`.

Other related methods and functions:

- `Machine.Clock(state string) uint64`
- `Machine.Time(states S) T`
- `Machine.TimeSum(states S) T`
- `Machine.HasStateChangedSince(clocks Clocks) bool`
- `IsTimeAfter(t1 T, t2 T)`
- `IsActiveTick(tick uint64)`

Example - clocks

```go
// ()[Foo:0 Bar:0 Baz:0 Exception:0]
mach.Clock("Foo") // ->0

mach.Add1("Foo", nil)
mach.Add1("Foo", nil)
// (Foo:1)[Bar:0 Baz:0 Exception:0]
mach.Clock("Foo") // ->1
```

```go
mach.Remove1("Foo", nil)
mach.Add1("Foo", nil)
// (Foo:3)[Bar:0 Baz:0 Exception:0]
mach.Clock("Foo") // ->3
```

Example - state context

```go
func (h *Handlers) DownloadingFileState(e *am.Event) {
    // open until the state remains active
    ctx := e.Machine.NewStateCtx("DownloadingFile")
    // fork to unblock
    go func() {
        // check if still valid
        if ctx.Err() != nil {
            return // expired
        }
    }()
}
```

Checking Active States

Each state can be **active** or **inactive**, determined by its state clock. You can check the current state at any time, without a long delay, which makes it a dependable source of decisions.

Methods to check the active states:

- `Machine.Is(states)`
- `Machine.Is1(state)`
- `Not(states)`
- `Not1(state)`
- `Any(states1, states2...)`
- `Any1(state1, state2...)`

Methods to inspect / dump the currently active states:

- `Machine.String()`
- `Machine.StringAll()`
- `Machine.Inspect(states)`

`Is` checks if all the passed states are active.

```go
// ()[Foo:0 Bar:0 Baz:0 Exception:0]

mach.Add1("Foo", nil)
// (Foo:1)[Bar:0 Baz:0 Exception:0]

mach.Is1("Foo") // true
mach.Is(am.S{"Foo", "Bar"}) // false
```

`Not` checks if none of the passed states is active.

```go
// ()[A:0 B:0 C:0 D:0 Exception:0]

mach.Add(am.S{"A", "B"}, nil)
// (A:1 B:1)[C:0 D:0 Exception:0]

// not(A) and not(C)
mach.Not(am.S{"A", "C"}) // false
// not(C) and not(D)
mach.Not(am.S{"C", "D"}) // true
```

`Any` is group call to `Is`, returns true if any of the params return true from `Is`.

```go
// ()[Foo:0 Bar:0 Baz:0 Exception:0]

mach.Add1("Foo", nil)
// (Foo:1)[Bar:0 Baz:0 Exception:0]

// is(Foo, Bar) or is(Bar)
mach.Any(am.S{"Foo", "Bar"}, am.S{"Bar"}) // false
// is(Foo) or is(Bar)
mach.Any(am.S{"Foo"}, am.S{"Bar"}) // true
```

Inspecting States

Being able to inspect your machine at any given step is VERY important. These are the basic method which don't require any additional debugging tools.

Example - inspecting active states and their clocks

```go
mach.StringAll() // ->()[Foo:0 Bar:0 Baz:0 Exception:0]
mach.String() // ->()

mach.Add1("Foo")

mach.StringAll() // ->(Foo:1)[Bar:0 Baz:0 Exception:0]
mach.String() // ->(Foo:1)
```

Example - inspecting relations

```go
// From examples/temporal-fileprocessing/fileprocessing.go
mach.Inspect()
// Exception:
//    State:   false 0
//
// DownloadingFile:
//    State:   false 1
//    Remove:  FileDownloaded
```

```
//
// FileDownloaded:
//   State:   true 1
//   Remove:  DownloadingFile
//
// ProcessingFile:
//   State:   false 1
//   Auto:    true
//   Require: FileDownloaded
//   Remove:  FileProcessed
//
// FileProcessed:
//   State:   true 1
//   Remove:  ProcessingFile
//
// UploadingFile:
//   State:   false 1
//   Auto:    true
//   Require: FileProcessed
//   Remove:  FileUploaded
//
// FileUploaded:
//   State:   true 1
//   Remove:  UploadingFile
```

Auto States

Automatic states (`Auto` property) are one of the most important concepts of **asyncmachine-go**. After every transition with a clock change (tick), `Auto` states will try to active themselves via an auto mutation.

- `Auto` states can be set partially (within the same mutation)
- auto mutation is **prepended** to the queue
- `Remove` relation of `Auto` states isn't enforced within the auto mutation

Example - log for FileProcessed causes an `Auto` state UploadingFile to activate

```
// [state] +FileProcessed -ProcessingFile
// [external] cleanup /tmp/temporal_sample1133869176
// [state:auto] +UploadingFile
```

Multi States

Multi-state (`Multi` property) describes a state which can be activated many times, without being de-activated in the meantime. It always triggers `Enter` and `State` transition handlers, plus the clock is always incremented. It's useful for describing many instances of the same event (e.g. network input) without having to define more than one transition handler. `Exception` is a good example of a

`Multi` state (many errors can happen, and we want to know all of them). The downside is that `Multi` states don't have state contexts.

Categories of States

States usually belong to one of these categories:

1. Input states (e.g. RPC msgs)
2. Read-only states (e.g. external state / UI state / summaries)
3. Action states (e.g. Start, ShowModal, public API methods)

Action states often de-activate themselves after they are done, as a part of their final handler.

Example - self removal

```
func (h *Handlers) ClickState(e *am.Event) {
    // add removal to the queue
    e.Machine.Remove1("Click")
}
```

Changing State

State Mutations

Mutation is a request to change the currently active states of a machine. Each mutation has a list of states knows as **called states**, which are different from **target states** (calculated during a transition).

Mutations are queued, thus they are never nested - one can happen only after the previous one has been processed. Mutation methods return a `Result`, which can be:

- `Executed`
- `Canceled`
- `Queued`

You can check if the machine is busy executing a transition by calling `Machine.DuringTransition()` (see queue) or wait until it's done with `<-Machine.WhenQueueEnds()`.

There are 3 types of mutations:

- add
- remove
- set

`Machine.Add(states, args)` is the most common method, as it preserves the currently active states. Each activation increases the states' clock to an odd number.

Example - Add mutation

```go
// ()[Foo:0 Bar:0 Baz:0 Exception:0]

mach.Add(am.S{"Foo"}, nil)
// (Foo:1)[Bar:0 Baz:0 Exception:0]

mach.Add(am.S{"Bar"}, nil)
// (Foo:1 Bar:1)[Baz:0 Exception:0]

mach.Add1("Bar", nil)
// (Foo:1 Bar:1)[Baz:0 Exception:0]
```

`Machine.Remove(states, args)` deactivates only the specified states. Each de-activation increases the states' clock to an even number.

Example - Remove mutation

```go
// ()[Foo:0 Bar:0 Baz:0 Exception:0]

mach.Add(am.S{"Foo", "Bar"}, nil)
// (Foo:1 Bar:1)[Baz:0 Exception:0]

mach.Remove(am.S{"Foo"}, nil)
// (Bar:1)[Foo:2 Baz:0 Exception:0]

mach.Remove1("Bar", nil)
// [Foo:2 Bar:2 Baz:0 Exception:0]
```

`Machine.Set(states, args)` de-activates all but the passed states and activates the remaining ones.

Example - Set mutation

```go
// ()[Foo:0 Bar:0 Baz:0 Exception:0]

mach.Add1("Foo", nil)
// (Foo:1)[Bar:0 Baz:0 Exception:0]

mach.Set(am.S{"Bar"}, nil)
// (Bar:1)[Foo:2 Baz:0 Exception:0]
```

Mutation Arguments

Each mutation has an optional map of arguments of type `am.A`, passed to handlers via the `am.Event` struct.

Example - passing arguments to handlers

```go
args := am.A{"val": "key"}
mach.Add1("Foo", args)
```

```go
// ...

// wait for a mutation like the one above
<-mach.WhenArgs("Foo", am.A{"val": "key"}, nil)
```

Transition Lifecycle

Transition is created from a mutation and tries to execute it, which can result in the changing of machine's active states. Each transition has several steps and (optionally) calls several handlers (for each of the bindings).

Once a transition begins to execute, it goes through the following steps:

1. Calculating Target States - collecting target states based on relations, currently active states and called states. Transition can already be `Canceled` at this point.
2. Negotiation handlers - methods called for each state about-to-be activated or deactivated. Each of these handlers can return `false`, which will cause the mutation to be `Canceled` and `Transition.Accepted` to be `false`.
3. Apply the **target states** to the machine - from this point `Is` (and other checking methods) will reflect the target states.
4. Final handlers - methods called for each state about-to-be activated or deactivated, as well as self handlers of currently active ones. Transition cannot be canceled at this point.

Transition Handlers

State handler is a struct method with a predefined suffix or prefix, which receives an Event struct. There are negotiation handlers (returning a `bool`) and final handlers (with no return). Order of the handlers depends on currently active states and relations of active and target states.

Example - handlers for the state Foo

```go
func (h *Handlers) FooEnter(e *am.Event) bool {}
func (h *Handlers) FooState(e *am.Event) {}
func (h *Handlers) FooExit(e *am.Event) bool {}
func (h *Handlers) FooEnd(e *am.Event) {}
```

List of handlers during a transition from `Foo` to `Bar`, in the order of execution:

- `FooExit` - negotiation handler
- `FooBar` - negotiation handler
- `FooAny` - negotiation handler
- `AnyBar` - negotiation handler
- `BarEnter` - negotiation handler
- `FooEnd` - final handler
- `BarState` - final handler

All handlers execute in a series, one by one, thus they don't need to mutually exclude each other for accessing resources. This reduces the number of locks needed. No blocking is allowed in the body of a handler, unless it's in a goroutine. Additionally, each handler has a limited time to complete (**100ms** with the default handler timeout), which can be set via `am.Opts`.

Self handlers

Self handler is a final handler for states which were active **before and after** a transition (all no-change active states). The name is a doubled name of the state (eg `FooFoo`).

List of handlers during a transition from `Foo` to `Foo Bar`, in the order of execution:

- `AnyBar` - negotiation handler
- `BarEnter` - negotiation handler
- `FooFoo` - final handler and **self handler**
- `BarState` - final handler

Self handlers provide a simple alternative to `Multi` states, while fully maintaining state clocks.

Defining Handlers

Handlers are defined as struct methods. Each machine can have many handler structs bound to itself using `Machine.BindHandlers`, although at least one of the structs should embed the provided `am.ExceptionHandler` (or provide its own). Any existing struct can be used for handlers, as long as there's no name conflict.

Example - define `FooState` and `FooEnter`

```go
type Handlers struct {
    // default handler for the build in Exception state
    *am.ExceptionHandler
}

func (h *Handlers) FooState(e *am.Event) {
    // final activation handler for Foo
}

func (h *Handlers) FooEnter(e *am.Event) bool {
    // negotiation activation handler for Foo
    return true // accept this transition by Foo
}

func main() {
    // ...
```

```go
    err := mach.BindHandlers(&Handlers{})
}
```

Log output:

```
[add] Foo
[handler] FooEnter
[state] +Foo
[handler] FooState
```

Event Struct

Every handler receives a pointer to an `Event` struct, with `Name`, `Machine` and `Args`.

```go
// definition
type Event struct {
    Name    string
    Machine *Machine
    Args    A
}
// send args
mach.Add(am.S{"Foo"}, A{"test": 123})
// ...
// receive args
func (h *Handlers) FooState(e *am.Event) {
  test := e.Args["test"].(string)
}
```

Calculating Target States

Called states combined with currently active states and a relations resolver result in **target states** of a transition. This phase is **cancelable** - if **any** of the called states gets rejected, the **transition is canceled**. This isn't true for Auto states, which can be partially rejected.

Transition exposes called, target and previous states using:

- `e.Transition.StatesBefore`
- `e.Transition.TargetStates`
- `e.Transition.ClocksBefore`
- `e.Transition.ClocksAfter`
- `e.Transition.Mutation.CalledStates`

```go
// machine
m := am.New(ctx, am.Struct{
    "Foo": {
        Add: am.S{"Bar"},
    },
```

```go
    "Bar": {}
}, nil)

// ...

// handlers
func (h *Handlers) FooEnter(e *am.Event) bool {
    e.Transition.StatesBefore // ()
    e.Transition.TargetStates // (Foo Bar)
    e.Machine.Transition.CalledStates() // (Foo)

    e.Machine.Is(am.S{"Foo", "Bar"}) // false
    return true
}
func (h *Handlers) FooState(e *am.Event) {
    e.Transition.StatesBefore // ()
    e.Transition.TargetStates // (Foo Bar)
    e.Machine.Transition.CalledStates() // (Foo)

    e.Machine.Is(am.S{"Foo", "Bar"}) // true
}

// ...

// usage
m.Add1("Foo", nil)
```

```
[add] Foo
[implied] Bar
[handler] FooEnter
FooEnter
| From: []
| To: [Foo Bar]
| Called: [Foo]
()[Bar:0 Foo:0 Exception:0]
[state] +Foo +Bar
[handler] FooState
FooState
| From: []
| To: [Foo Bar]
| Called: [Foo]
(Bar:1 Foo:1)[Exception:0]
end
(Bar:1 Foo:1)[Exception:0]
```

Negotiation Handlers

```go
func (h *Handlers) FooEnter(e *am.Event) bool {}
func (h *Handlers) FooExit(e *am.Event) bool {}
```

Negotiation handlers `Enter` and `Exit` are called for every state which is going to be activated or de-activated. They are allowed to cancel a transition by optionally returning `false`. **Negotiation handlers** are limited to read-only operations, or at least to side effects free ones. Their purpose is to make sure that final transition handlers are good to go.

```go
// negotiation handler
func (h *Handlers) ProcessingFileEnter(e *am.Event) bool {
    // read-only ops
    // decide if moving fwd is ok
    // no blocking
    // lock-free critical zone
    return true
}
```

Example - rejected negotiation

```go
// machine
m := am.New(ctx, am.Struct{
    "Foo": {
        Add: am.S{"Bar"},
    },
    "Bar": {},
}, nil)

// ...

// handlers
func (h *Handlers) FooEnter(e *am.Event) bool {
    return false
}

// ...

// usage
m.Add1("Foo", nil) // ->am.Canceled
// ()[Bar:0 Foo:0 Exception:0]
```

```
[add] Foo
[implied] Bar
[handler] FooEnter
[cancel:ad0d8] (Foo Bar) by FooEnter
()[Bar:0 Foo:0 Exception:0]
```

Final Handlers

```go
func (h *Handlers) FooState(e *am.Event) {}
func (h *Handlers) FooEnd(e *am.Event) {}
func (h *Handlers) FooBar(e *am.Event) {}
func (h *Handlers) BarFoo(e *am.Event) {}
func (h *Handlers) AnyFoo(e *am.Event) {}
func (h *Handlers) FooAny(e *am.Event) {}
```

Final handlers `State` and `End` are where the main handler logic resides. After the transition gets accepted by relations and negotiation handlers, final handlers will allocate and dispose resources, call APIs, and perform other blocking actions with side effects. Just like negotiation handlers, they are called for every state which is going to be activated or de-activated. Additionally, the Self handlers are called for states which remained active.

Like any handler, final handlers cannot block the mutation. That's why they need to start a goroutine and continue their execution within it, while asserting the state context is still valid.

```go
func (h *Handlers) ProcessingFileState(e *am.Event) {
    // read & write ops
    // no blocking
    // lock-free critical zone
    mach := e.Machine
    // tick-based context
    stateCtx := mach.NewStateCtx("ProcessingFile")
    go func() {
        // block in the background, locks needed
        if stateCtx.Err() != nil {
            return // expired
        }
        // blocking call
        err := processFile(h.Filename, stateCtx)
        if err != nil {
            mach.AddErr(err)
            return
        }
        // re-check the tick ctx after a blocking call
        if stateCtx.Err() != nil {
            return // expired
        }
        // move to the next state in the flow
        mach.Add1("FileProcessed", nil)
    }()
}
```

Dynamic Handlers

`OnEvent` returns a channel that will be notified with `*Event`, when any of the passed events happen. It's quick substitute for a predefined transition handler, although it does not guarantee a deterministic order of execution (**dynamic handlers** are triggered in bulk at the end of a transition). It also accepts an optional context for an earlier disposal. Usage of **dynamic handlers** is discouraged and may cause problems. To subscribe to events other than handlers, there's Tracer API.

Example - bind to `FooEnter`

```
// machine
m := am.New(ctx, am.Struct{
    "Foo": {},
    "Bar": {},
})

// ...

// usage
fooEnter := m.OnEvent([]string{"FooEnter"}, nil)
go func() {
    <-fooEnter
    println("OnEvent(FooEnter)")
})
```

```
[add] Foo
[state] +Foo
OnEvent(FooEnter)
```

Advanced Topics

State's Relations

Each state can have 4 types of **relations**. Each relation accepts a list of state names. Relations are handled by RelationsResolver, which should be extendable and potentially even replaceable.

`Add` relation The `Add` relation tries to activate listed states, whenever the owner state gets activated.

Their activation is optional, meaning if any of those won't get accepted, the transition will still be `Executed`.

```
// machine
m := am.New(ctx, am.Struct{
    "Foo": {
        Add: am.S{"Bar"},
```

```
    },
    "Bar": {},
}, nil)

// usage
m.Add1("Foo", nil) // ->Executed
// (Foo:1 Bar:1)[Exception:0]
```

```
[add] Foo
[implied] Bar
[state] +Foo +Bar
(Foo:1 Bar:1)[Exception:0]
```

`Remove` relation The `Remove` relation prevents from activating, or deactivates listed states.

If some of the called states `Remove` other called states, or some of the active states `Remove` some of the called states, the transition will be `Canceled`.

Example of an accepted transition involving a `Remove` relation:

```
// machine
m := am.New(ctx, am.Struct{
    "Foo": {
        Remove: am.S{"Bar"},
    },
    "Bar": {},
}, nil)

// usage
m.Add1("Foo", nil) // ->Executed
m.Add1("Bar", nil) // ->Executed
println(m.StringAll()) // (Foo:1)[Bar:0 Exception:0]
```

```
[add] Foo
[state] +Foo
[add] Bar
[cancel:reject] Bar
(Foo:1)[Bar:0 Exception:0]
```

Example of a canceled transition involving a `Remove` relation - some of the called states `Remove` other Called States.

```
// machine
m := am.New(ctx, am.Struct{
    "Foo": {},
    "Bar": {
        Remove: am.S{"Foo"},
    },
```

```go
}, nil)

// usage
m.Add(am.S{"Foo", "Bar"}, nil) // ->Canceled
m.Not1("Bar") // true
// ()[Foo:0 Bar:0 Exception:0]
```

```
[add] Foo Bar
[cancel:reject] Foo
()[Exception:0 Foo:0 Bar:0]
```

Example of a canceled transition involving a `Remove` relation - some of the active states `Remove` some of the called states.

```go
// machine
m := am.New(ctx, am.Struct{
    "Foo": {},
    "Bar": {
        Remove: am.S{"Foo"},
    },
}, nil)

// usage
m.Add1("Bar", nil) // ->Executed
m.Add1("Foo", nil) // ->Canceled
m.StringAll() // (Foo:1)[Bar:0 Exception:0]
```

```
[add] Bar
[state] +Bar
[add] Foo
[cancel:reject] Foo
(Bar:1)[Foo:0 Exception:0]
```

`Require` relation The `Require` relation describes the states required for this one to be activated.

Example of an accepted transition involving a `Require` relation:

```go
// machine
m := am.New(ctx, am.Struct{
    "Foo": {},
    "Bar": {
        Require: am.S{"Foo"},
    }
}, nil)

// usage
m.Add1("Foo", nil) // ->Executed
```

```
m.Add1("Bar", nil) // ->Executed
// (Foo:1 Bar:1)[Exception:0]
```

```
[add] Foo
[state] +Foo
[add] Bar
[state] +Bar
(Foo:1 Bar:1)[Exception:0]
```

Example of a canceled transition involving a `Require` relation:

```
// machine
m := am.New(ctx, am.Struct{
    "Foo": {},
    "Bar": {
        Require: am.S{"Foo"},
    },
}, nil)

// usage
m.Add1("Bar", nil) // ->Canceled
// ()[Foo:0 Bar:0 Exception:0]
```

```
[add] Bar
[reject] Bar(-Foo)
[cancel:reject] Bar
()[Foo:0 Bar:0 Exception:0]
```

`After` relation The `After` relation decides about the order of execution of transition handlers. Handlers from the defined state will be executed **after** handlers from listed states.

```
// machine
m := am.New(ctx, am.Struct{
    "Foo": {
        After: am.S{"Bar"},
    },
    "Bar": {
        Require: am.S{"Foo"},
    },
}, nil)

// ...

// handlers
func (h *Handlers) FooState(e *am.Event) {
    println("Foo")
}
```

```go
func (h *Handlers) BarState(e *am.Event) {
    println("Bar")
}

// ...

// usage
m.Add(am.S{"Foo", "Bar"}, nil) // ->Executed
```

```
[add] Foo Bar
[state] +Bar +Foo
[handler] BarState
Bar
[handler] FooState
Foo
```

Waiting

```go
// wait until FileDownloaded becomes active
<-mach.When1("FileDownloaded", nil)

// wait until FileDownloaded becomes inactive
<-mach.WhenNot1("DownloadingFile", args, nil)

// wait for EventConnected to be activated with an arg ID=123
<-mach.WhenArgs("EventConnected", am.A{"ID": 123}, nil)

// wait for Foo to have a tick >= 6 and Bar tick >= 10
<-mach.WhenTime(am.S{"Foo", "Bar"}, am.T{6, 10}, nil)

// wait for DownloadingFile to have a tick increased by 2 since now
<-mach.WhenTicks("DownloadingFile", 2, nil)
```

Almost all "when" methods return a share channel which closes when an event happens (or the optionally passed context is canceled). They are used to wait until a certain moment, when we know the execution can proceed. Using "when" methods creates new channels and should be used with caution, possibly making use of the early disposal context. In the future, these channels will be reused and should scale way better.

"When" methods are:

- `Machine.When(states, ctx)`
- `Machine.WhenNot(states, ctx)`
- `Machine.When1(state, ctx)`
- `Machine.WhenNot1(state, ctx)`
- `Machine.WhenArgs(state, args, ctx)`
- `Machine.WhenTime(states, ctx)`

- `Machine.WhenTicks(state, ctx)`
- `Machine.WhenTicksEq(state, ctx)`
- `Machine.WhenQueueEnds(state, ctx)`
- `Machine.WhenErr(state, ctx)`

Example - waiting for states `Foo` and `Bar` to being active at the same time:

```go
// machine
m := am.New(ctx, am.Struct{
    "Foo": {
        Add: am.S{"Bar"},
    },
    "Bar": {},
})

// ...

// usage
select {
    case <-mach.When(am.S{"Foo", "Bar"}, nil):
        println("Foo Bar")
    }
}

// ...

// state change
m.Add1("Foo", nil)
m.Add1("Bar", nil)
// (Foo:1 Bar:1)[Exception:0]
```

```
[add] Foo
[implied] Bar
[state] +Foo +Bar
Foo Bar
(Bar:1 Foo:1)[Exception:0]
```

Error Handling

Considering that everything meaningful can be a state, so can errors. Every machine has a predefined `Exception` state (which is a `Multi` state) and an optional `ExceptionHandler`, which can be embedded into handler structs. Creating more detailed error states which have an `Add` relation to the `Exception` state is one way of handing errors.

- `Machine.AddErr(error)`
- `Machine.AddErrStr(string)`
- `Machine.WhenErr(ctx)`

- `Machine.Err`

Example - timeout flow with error handling

```go
select {
case <-time.After(10 * time.Second):
    // timeout
case <-mach.WhenErr(nil):
    // error or machine disposed
    fmt.Printf("err: %s\n", mach.Err)
case <-mach.When1("Bar", nil):
    // state Bar active
}
```

```
[add] Foo
[state] +Foo
[add] Exception
[state] +Exception
err: fake err
(Foo:1 Exception:1)[Bar:0]
```

Panics In Handlers

Panics are automatically caught and transformed into the `Exception` state. This can be disabled using `Machine.PanicToException` or `Opts.DontPanicToException`. Same goes for `Machine.PrintExceptions` and `DontPrintExceptions`, which manipulate printing to the log sink.

In case of a panic inside a transition handler, the recovery flow depends on the type of the erroneous handler.

Panic in a negotiation handler

1. Cancels the whole transition.
2. Active states of the machine stay untouched.
3. Add mutation for the `Exception` state is prepended to the queue.

Panic in a final handler

1. Transition has been accepted and target states has been set as active states.
2. Not all the final handlers have been executed, so the states from non-executed handlers are removed from active states.
3. Add mutation for the `Exception` state is prepended to the queue and the integrity should be restored manually (e.g. relations, resources involved).

TODO example

Queue and History

The purpose of **asyncmachine-go** is to synchronize actions, which results in only one handler being executed at the same time. Every mutation happening inside the handler, will be queued and the mutation call will return `Queued`.

Queue itself can be accessed via `Machine.Queue()` and checked using `Machine.IsQueued()`. After the execution, queue creates history, which can be captured using a dedicated package `pkg/history`. Both sources can help to make informed decisions based on scheduled and past actions.

- `Machine.Queue()`
- `Machine.IsQueued(mutationType, states, withoutArgsOnly, statesStrictEqual, startIndex)`
- `Machine.DuringTransition()`
- `History.ActivatedRecently(state, duration)`
- `Machine.WhenQueueEnds()`

```go
// machine
m := am.New(ctx, am.Struct{
    "Foo": {},
    "Bar": {}
}, nil)

// ...

// handlers
func (h *Handlers) FooState(e *am.Event) {
    e.Machine.Add1("Bar", nil) // ->Queued
}

// ...

// usage
m.Add1("Foo", nil) // ->Executed
```

```
[add] Foo
[state] +Foo
[handler] FooState
[queue:add] Bar
[postpone] queue running (1 item)
[add] Bar
[state] +Bar
```

TODO describe duplicate detection rules

Logging

Besides inspecting methods, **asyncmachine-go** offers a very verbose logging system with 4 levels of details:

- `LogNothing` (default)
- `LogChanges` state changes and important messages
- `LogOps` detailed relations resolution, called handlers, queued and rejected mutations
- `LogDecisions` more verbose variant of Ops, explaining the reasoning behind
- `LogEverything` useful only for deep debugging

Example of all log levels for the same code snippet:

```go
// machine
m := am.New(ctx, am.Struct{
    "Foo": {},
    "Bar": {
        Auto: true,
    },
    // disable ID logging
}, &am.Opts{DontLogID: true})
m.SetLogLevel(am.LogOps)

// ...

// handlers
func (h *Handlers) FooState(e *am.Event) {
    // empty
}
func (h *Handlers) BarEnter(e *am.Event) bool {
    return false
}

// ...

// usage
m.Add1("Foo", nil) // Executed
```

- log level `LogChanges`

```
[state] +Foo
```

- log level `LogOps`

```
[add] Foo
[state] +Foo
[handler] FooState
```

```
[auto] Bar
[handler] BarEnter
[cancel:4a0bc] (Bar Foo) by BarEnter
```

- log level `LogDecisions`

```
[add] Foo
[state] +Foo
[handler] FooState
[auto] Bar
[add:auto] Bar
[handler] BarEnter
[cancel:2daed] (Bar Foo) by BarEnter
```

- log level `LogEveryting`

```
[start] handleEmitterLoop Handlers
[add] Foo
[emit:Handlers:d7a58] AnyFoo
[emit:Handlers:d32cd] FooEnter
[state] +Foo
[emit:Handlers:aa38c] FooState
[handler] FooState
[auto] Bar
[add:auto] Bar
[emit:Handlers:f353d] AnyBar
[emit:Handlers:82e34] BarEnter
[handler] BarEnter
[cancel:82e34] (Bar Foo) by BarEnter
```

Arguments logging TODO

Custom logging

```go
// max out the log level
mach.SetLogLevel(am.LogEverything)
// level based dispatcher
mach.SetLogger(func(level LogLevel, msg string, args ...any) {
    if level > am.LogChanges {
        customLogDetails(msg, args...)
        return
    }
    customLog(msg, args...)

})

// include some args in the log and traces
mach.SetLogArgs(am.NewArgsMapper([]string{"id", "name"}, 20))
```

Debugging

TODO

```
$ am-dbg
```

```go
import "github.com/pancsta/asyncmachine-go/pkg/telemetry"
// ...
err := telemetry.TransitionsToDBG(mach, "")
```

```bash
# use AM_DEBUG to increase timeouts of NewCommon machines
AM_DEBUG=1
```

Typesafe States

TODO

Example - using am-gen

```go
// generate using either
// $ am-gen states-file Foo,Bar
// $ task gen-states-file Foo,Bar
package states

import am "github.com/pancsta/asyncmachine-go/pkg/machine"

// S is a type alias for a list of state names.
type S = am.S

// States map defines relations and properties of states.
var States = am.Struct{
    Start:     {},
    Heartbeat: {Require: S{Start}},
}

//#region boilerplate defs

// Names of all the states (pkg enum).

const (
    Start             = "Start"
    Heartbeat         = "Heartbeat"
)

// Names is an ordered list of all the state names.
var Names = S{Start, Heartbeat}

//#endregion
```

Tracing and Metrics

TODO

- `pkg/telemetry` - Otel and am-dbg
- `pkg/telemetry/prometheus`

Optimizing Data Input

TODO

Example - batch data into a single transition

```go
var queue []*Msg
var queueMx sync.Mutex
var scheduled bool

func Msg(msgTx *Msg) {
    queueMx.Lock()
    defer queueMx.Unlock()

    if !scheduled {
        scheduled = true
        go func() {
            // wait some time
            time.Sleep(time.Second)

            queueMx.Lock()
            defer queueMx.Unlock()

            // add in bulk
            mach.Add1("Msgs", am.A{"msgs": queue})
            queue = nil
            scheduled = false
        }()
    }
    // enqueue
    queue = append(queue, msgTx)
}
```

Cheatsheet

- **State**: main entity of the machine, higher-level abstraction of a meaningful workflow step
- **Active states**: states currently activated in the machine, `0-n` where `n == len(states)`
- **Called states**: states passed to a mutation method, explicitly requested

- **Target states**: states after resolving relations, based on previously active states, about to become new active states
- **Mutation**: change to currently active states, created by mutation methods
- **Transition**: container struct for a mutation, handles relations and events
- **Accepted transition**: transition which mutation has passed negotiation and relations
- **Canceled transition**: transition which mutation has NOT passed negotiation or relations
- **Queued transition**: transition which couldn't execute immediately, as another one was in progress, and was added to the queue instead
- **Transition handlers**: methods defined on a handler struct, which are triggered during a transition
- **Negotiation handlers**: handlers executed as the first ones, used to make a decision if the transition should be accepted
- **Final handlers**: handlers executed as the last ones, used for operations with side effects

Other sources

Please refer to the cookbook and examples.