Preamble

As of today the Anoma specification is on version v0.1.3, and although we've made great strides in making the specification readable I believe we can once again greatly improve the situation by simplifying the theoretical framing we use on the specs.

The Anoma specification (hereby referred to as specs) currently aims to serve two purposes. The main purpose is to prop up engineering in their endeavor to implement Anoma. The second purpose is to serve as a base to do proofs about theoretical properties of Anoma.

However, I believe that the current specs has aimed to satisfy the second purpose at a higher priority to avoid locking themselves out of the path at a later date, this is all well and good however I believe our ideas on Anoma itself are still immature and prioritizing such a goal has made the specs lackluster at serving it's primary focus and has made it paradoxically harder to achieve the second goal in the long run.

Theoretically Framing of the Specs

Before we suggest a new way the specs should be formed let us familiarize ourselves with the framework the specs use to explore Anoma.

The model is an actor models, with our specific actors called Engines

, here are the types in Juvix.

type Engine (C S B H A AM AC AE : Type) := mkEngine@{ cfg : EngineCfg C; // Static Information env : EngineEnv S B H AM; // Dynamic Information behaviour : EngineBehaviour C S B H A AM AC AE; };

type EngineCfg (C : Type) := mkEngineCfg@{ node : NodeID; name : EngineName; cfg : C; };

type EngineEnv (S B H AM : Type) := mkEngineEnv@{ localState : S; mailboxCluster : Map MailboxID (Mailbox B AM); acquaintances : Set EngineName; timers : List (Timer H); };

type EngineBehaviour (C S B H A AM AC AE : Type) := mkEngineBehaviour@{guards : GuardEval C S B H A AM AC AE};

type GuardEval (C S B H A AM AC AE : Type) := | First (List (Guard C S B H A AM AC AE)) | Any (List (Guard C S B H A AM AC AE));

Guard (C S B H A AM AC AE : Type) : Type := (trigger : TimestampedTrigger H AM) -> (cfg : EngineCfg C) -> (env : EngineEnv S B H AM) -> Option (GuardOutput C S B H A AM AC AE);

These types are what define out an Engine

I did not post the GuardOutput

, but this is what theoretically defines out the life cycle (we will get back to this in a bit), however for now I want to emphasize a few important details about this model:

1. AM

is the type of all Messages

1. AC

is the type of all Configurations (Static State)

1. AE

is the type of all Environments (Dynamic State)

Meaning that all messages, configurations, and environments must be known upfront and added to a sum type that all other engines must also know about. This implies that the processing of Engines and their updates will be done with all of them mixed together in some capacity (I have tried to think about how this would be done, and I can't see it personally as the sum types aren't there for storing said engines thus meaning it's very confused).

To better illustrate my point, let us look at how the standard Anoma

message type that all engines in the specs put for their AM

generic.

type Msg := | MsgIdentityManagement IdentityManagementMsg ...;

type IdentityManagementMsg := ... | MsgIdentityManagementDeleteIdentityRequest RequestDeleteIdentity | MsgIdentityManagementDeleteIdentityResponse ResponseDeleteIdentity;

type RequestDeleteIdentity := mkRequestDeleteIdentity@{ externalIdentity : EngineID; backend : Backend; };

type ResponseDeleteIdentity := mkResponseDeleteIdentity@{ err : Option String; };

With this we can see a few things:

1. Wee must have 2 layers of constructor wrappers, the first to uniquely identify what message it is and then a second identifying what engine should(???) process this message
2. This gets complicated actually as the common Response

data types should be processed by the actor that requested it, so this is very confusing for me for trying to determine a life cycle.

1. This gets complicated actually as the common Response

data types should be processed by the actor that requested it, so this is very confusing for me for trying to determine a life cycle.

1. We must have 2 types, one for the message request

then another for the response

.

The issue with 1.

is that the response has no easy way for me to determine who to send it to properly, and the problem with 2.

is that now where do we put the documentation about the semantics of a message? Do we put it in the request

or do we describe it in the response.

Further since this is all convention it's not always request

, response

, sometimes it's request

, reply

. Which means that the specs authors must be careful and keep this in mind when writing specs for their Engine.

Lastly this complicates the sidebar when browsing the specs

[

433×983 61.2 KB

](https://europe1.discourse-cdn.com/flex013/uploads/anoma1/original/1X/23e3977c3408e0da42829fb69583657eb703a863.png)

making it very cumbersome from an UI perspective.

With some idea on how the Engine life cycle works let me wrap up this section with the following comment. I believe the architecture around Engines in the specs is overly complex and it does not easily form into a cohesive model that will be useful for proving nor explaining Anoma. I will now try to explain a plan that I came up with to try satisfy both requirements. The approach I'll take put simplicity and engineering first however from simplicity I believe we will find an easier model to prove. I will also give a full life cycle that is flawed

, and I open the floor to the specs team to refine the hypothetical life cycle if it is useful for proofs or maybe use the flexibility in my model to cut complexity and model the engine architecture part separately from the actual engines!

On a simple model

In coming up with a simpler to reason about model of the specs I had two thoughts:

1. If we defined the specs in Java, what shape would things take?
2. Can we model our messages as functions for known documentation techniques?

In coming to these questions I came up with the following model.

for the *DeleteIdentity

types above, model them as if they are a function

delete-identity : engine-id -> backend -> option string

Further for life cycle reasons we'll want to wrap the output type in a Monad

.

delete-identity : engine-id -> backend -> monad (option string)

This would look very similar to what our Elixir code does

@spec handle_tx({Backends.backend(), Noun.t()}, binary(), t()) :: t() defp handle_tx(tx = {backend, code}, tx_id, state = %Mempool{}) do ... Executor.launch(node_id, tx, tx_id) ...

Now to properly scaffold our engine-id

we should think of engine-id

as an object

that accepts a message like Executor

that locates the Executor we care about. In concrete Juvix terms we'd define it like this:

type Engines := { executor : Executor identity : Identity workers : [List Workers] ... }

Thus we can actually say

some-engine-function : EngineId -> ... | engine-id ... := do Identity.delete engine-id ...

This gets rid of the concrete wrappers like Msg

because instead of having a list where they all mix we have a concrete data that disambiguates all the kinds of engines. Note that this technique even works for multiple Engines in the structure, we just have to make it a list or a mapping with extra disambiguation information.

The next step is to follow in Erlang's step and separate this nice interface from how the actors actually work.

In our Elixir code we have

@spec tx(String.t(), {Backends.backend(), Noun.t()}, binary()) :: :ok def tx(node_id, tx_w_backend, id) do GenServer.cast(Registry.via(node_id, **MODULE**), {:tx, tx_w_backend, id}) end

@impl true def handle_cast({:tx, tx, tx_id}, state) do {:noreply, handle_tx(tx, tx_id, state)} end

Likewise, for our delete

function we'd have a distinction between the interface and the actual handle code. The particular methods for handling this may vary depending on the implementation techniques chosen, so let us discuss some benefits of this method before procceeding.

This approach allows us to document delete

in a single place and document it like a normal function using the normal doc tools. Further we can use familiar UX that is know to be effective for this kind of documentation, see [1][2][3][4].

defmodule Identity

--- deletes an identiy from a given backend --- argument ... --- returns ... delete : EngineId -> Backend -> Monad (Option String) | engine-id backend := do ...;

Further we can cut having the AC

, AE

and AM

generics, along with the double layer of wrappers that comes with concretely filling out these types. Also we've cut having to

have 2 types defined for every message we'd like to define.

Now for various strategies on how delete

could work:

1. We fall back on the sum type that we saw before in IdentityManagementMsg

, meaning that it looks something like this

delete : EngineId -> Backend -> Monad (Option String) | engine-id backend := invoke-message (MsgIdentityManagementDeleteIdentityRequest engine-id backend)

The downside of this approach is that besides the sum types coming back, the flow is very unclear still!

1. We process things immediately with the option of removing engine scaffolding. This would look something like this

delete : EngineId -> Backend -> Option String | engine-id backend := handle-delete (EningeId.identity engine-id) backend

handle-delete : (State, Configuration) -> Backend -> Option String | (state, config) backend := ...;

with EngineId.identity

fetching the state for the real computation.

1. Adopting a call

cast

system with a monad life cycle to handle this all

delete : EngineId -> Backend -> Option String | engine-id backend := call (EningeId.identity engine-id) (Delete backend)

This requires a greater wrapper type like we saw before in IdentityManagementMsg

and in the first approach, however using it in this way gives us some advantages.

However before we continue I want to compare the approaches and their benefits and downsides before we explain how 3

's life cycle works to any degree.

I believe we can mostly throw out 1, as it has a confused life cycle and isn't going anywhere if we wish to model a full life cycle.

Model 2 is the simplest by far as we need no monads to model this, when we "send a message" to another actor, that function is switched to right away and process happens linearly. Note under this approach we can keep the engine framework or toss it completely in this context as that just changes the "run time" of how the processing is actually done rather than a forced baked in idea.

This approach works the best if you want to model properties around message sends separately from the actual details of Engines. This gives you huge flexibility there along with making way for a trivial implementation that we can try our proofs out on.

Approach 3 makes the most sense if you want to prove the properties of engines along side specifics of particular engine executions together. This comes at a great cost however, as the engine model itself is no longer optional and we must model everything in a very specific monad. The rest of this post will cover my attempts at characterizing this monad, I hope future posters can simplify what I came up with, however I'm unsure how simple this can be made as we are effectively recreating doing what Javascript did when creating async

computation and call back hell

, just with better concurrency ideas.

On an engine life cycle

If we really take approach 3, then I came up with the following life cycle that ought to work but is overly complicated.

1. Anoma is a Solid State Interpreter

(SSI), meaning that we can realize Anoma in steps of computation. We apply our update function which begets messages for the next step to then process so on until the messages are empty. Each step is a monad

(haskell not the mathematical kind) in which our computation will happen.

1. On each step engines look at their mailbox to process any new messages.
2. Engines can send two kinds of messages. 1.

synchronous messages and 2.

asynchronous messages. If a synchronous

messages is sent, then the engine defers all control until their request is handled. * 2

. is easy to realize as we can buildup a list of Asynchronous messages

that gets fed into mailboxes on the next step.

- 1.

on the other hand is much more complicated and requires us to serialize our current function into a continuation and offer up our next step as a response to whenever the Engine finishes the computation.

1. 2

. is easy to realize as we can buildup a list of Asynchronous messages

that gets fed into mailboxes on the next step.

1. 1.

on the other hand is much more complicated and requires us to serialize our current function into a continuation and offer up our next step as a response to whenever the Engine finishes the computation.

1. When a synchronous message

is handled by another engine, it builds up a monad of the asynchronous actions` from it’s own processing along with applying the continuation on the return, taking a step for both itself and for the engine that it’s responsible for continuing.

To give a better feeling for this, I will give a brief example of a life cycle in action.

Imagine we have a network where engine A

is processing Foo

, and no other engines have any messages queued.

On step 1, all engines besides A

have no messages to process so quietly return with no new actions, engine A

finds it must process Foo

and does so with handle-foo

.

defmodule EngineA

Engine : Types := Anoma.Engine Configuration State; type Configuraiton := ...; type State := ...;

handle-foo : State -> Argument1 -> ... -> LifeCycle (Output) | state argument1 ... := do Identity.delete argument1 new-identity <- Identity.create argument2 return 5; end;

Upon reaching handle-foo

, it’s ran within the overarching LifeCycle

Monad the SSI

wraps around each step.

when we compute Identity.delete

it simply adds it to the Identity

mailbox for computation next LifeCycle

as it’s a cast

, however when we see Identity.create

, this is a call

and thus we must stop here, adding Identity.create

to the the mailbox. A

returns these actions, with the rest of the logic (return 5) in a continuation in the message Identity.create

. Further we set the processing logic of A

to be busy as it can not currently handle any computations.

cycle 2 starts, and Identity

has messages, where as everyone else either has no messages or is considered busy (A).

Identity sees the delete message, and then processes it. Let’s assume it does no further calls, and sends no messages, however this updates the state of the Identity engine to no longer have argument1

.

Cycle 2 is now done, in cycle 3, Identity

has a message, whereas everyone else is waiting.

Identity

processes create

. Like previously let us assume create

only does casts

not calls

, meaning that all it’s processes finish. Since this is a call, the result return by the create

is sent into the continuation that it was called with and A

resumes the computation returning 5

with no extra affects to append.

With that there are no more messages to append and each engine successfully executed their logic.

This isn’t the most elegant way to do this, however this is the approach that works given a fully functional system with no real concepts of threads.

Conclusion

I hope this post has been helpful in understanding in what way the current Engine

approach is inadequate for both explaining and proving Anoma. Further this post outlines a few techniques that let us simplify the approach without having to give up provability. Lastly the approach displayed utilizes what primitives we have in Juvix for making the specs into a full life cycle implementation utilizing a single step interpreter

approach to system design.

References

[1] Juvix Documentation

[2] String (Java Platform SE 8)

[3] sets vocabulary - Factor Documentation

[4] adjoin (elt set --) - Factor Documentation