Many fruitful discussions where happening at the hacker house in Berlin, this summer. My personal highlights for getting this post going are

- the postulation of identities for all concrete engines (and there is good progress in what an Idendity is, just shy of providing a fully fledged abstract data type)
- a general composition mechanism for engines and the idea to design Anoma as a composition of engines.

We have not yet converged on the details of how to compose engines though this is important—probably also for the denotational specs. How important it really is will become hopefully clear at the end of this post.

This title mentions two “dimensions” we have to take into account

(and let us ignore the dynamic “rewriring” of engines for the moment to keep this post to a digestible length):

- abstract vs. concrete:

mathematical models being the entities of the abstract “Platonic” world and concrete processes the agents of Anoma instances;

- synchronous vs. asynchronous:

let us consider these notions for the abstract and the concrete in separation * abstract synchronous:

here we have the string diagrammatic composition (see, e.g.,Signal Flow Graphs: Props, Presentations, and Proofs) for both parallel and sequential composition

- abstract asynchronous:

for this, it seems necessary to introduce a network model with reliable delivery and the state of the system always has this network state of messages in transit present;

- concrete synchronous:

for sequential composition, this amounts to synchronous function calls—the first engine calling the second with their outpus—while for parallel composition this is a “wrapper function”—the wrapper function calling the parallel components (in any serializable order) and returning the pair of results;

- concrete asynchronous:

we have a concrete messaging mechanism that sends messages between engine instances and the parallel / sequential composition is not really that pronounced any more as messages are just passed around based on a connectivity graph where each node has in-ports and out-ports which are suitably connected.

- abstract synchronous:

here we have the string diagrammatic composition (see, e.g.,Signal Flow Graphs: Props, Presentations, and Proofs) for both parallel and sequential composition

- abstract asynchronous:

for this, it seems necessary to introduce a network model with reliable delivery and the state of the system always has this network state of messages in transit present;

- concrete synchronous:

for sequential composition, this amounts to synchronous function calls—the first engine calling the second with their outpus—while for parallel composition this is a “wrapper function”—the wrapper function calling the parallel components (in any serializable order) and returning the pair of results;

- concrete asynchronous:

we have a concrete messaging mechanism that sends messages between engine instances and the parallel / sequential composition is not really that pronounced any more as messages are just passed around based on a connectivity graph where each node has in-ports and out-ports which are suitably connected.

The first point to make is that we can have corresponding notions of sequential and parallel composition in the abstract and the concrete; the crucial idea is that there are two readings of

- engine functions $\mathrm{on\_msg} \colon I \times S \to S \times \wp_{\mathrm{fin}}(O)$

and

- engine implementations of type (State, InMsg) -> (State, Set OutMsg)

as per

type Engine State InMsg OutMsg = (State, InMsg) -> (State, Set OutMsg)

namely

1. synchronously (like typical for APIs in programming languages)
2. asychronously, where the outputs are assumed to be delivered eventually (over a network).

Let us go through the example of sequential and parallel composition for the engine functions to make the difference clear.

Composing Sequentially

Synchronously

To execute two machines M

and M'

in sequential order,

we first execute M

on the input,

and then iterate over the outputs,

calling the second machine M'

and collecting the outputs in a list.

We might then actually need to impose orders on the output sets

as the order matters and we are interested in deterministic outputs.

Assuming single inputs and outputs for the sake of simplicity,

let us spell out what it means to compose machines sequentially.

So, given

- sets S,S'

, I,I'

, O,O'

with O = I'

- an engine $\mathrm{on\_msg} \colon I \times S \to S \times O$

and

- another engine $\mathrm{on\_msg}' \colon I' \times S' \to S' \times O'$

,

the synchronous sequential composition

is the function

$\mathrm{on\_msg}''_\otimes : I \times (S\times S') \to (S\times S') \times O'$

such that for all $i\in i, s \in S, s'\in S'$

$\mathrm{on\_msg}''_\otimes(i, (s,s')) = \bigl((\tilde s , \tilde s'), \tilde o \bigr)$

where for some (unique) $o \in O$

- $(\tilde s, o) = \mathrm{on\_msg}(i,s)$

- (\tilde s', \tilde o) = \mathrm{on_msg}'(\tilde s,o)

.

This is spelling out what feeding the output of the first machine into the second machine means in full detail.

Asynchronously

Recall that in the asynchronous setting, we can think of the outputs of a machine arriving in some arbitrary order (due to network delays). Thus, for the asynchronous sequential composition, we have to consider all orders in which the outputs of the first machine arrive as inputs to the second machine; then, we loop over the list and collect the outputs of the sequence of machine invocations of the second machine. Note that this means that the output state is no longer a single set, i.e., also the state becomes non-deterministic as a consequence of the re-ordering, in general.

Thus, we would actually need to consider engines to be of non-deterministic type

- engine functions \mathrm{on_msg} \colon I \times S \to \wp_{\mathrm{fin}}(S \times \wp_{\mathrm{fin}}(O))

and

- engine implementations of type

type Engine State InMsg OutMsg = (State, InMsg) -> Set (State, Set OutMsg)

{}*{{}*{\gets}\text{(A research question is lurking there as to how we can fix this seeming necessity, in the spirit of serializability.)}}

The deterministic functions would then be those that return a singleton, i.e., an element of the form {(s,{o_1, \ldots, o_n})}

.

We postpone the combinatorial juggling to the future—or maybe@AHart can write this up using a single line with some linear logic magic or Kleisli maps of the powerset monad? In any case, nobody want to read the set theoretic details here.

Parallel composition

Synchronously

The parallel composition is essentially the Cartesian product extended to functions.

Asynchronously

This is maybe the only point that requires some seemingly arbitrary definition. Given two machines with inputs I

and I'

, the composed engine will have as input the disjoint sum I + I'

. The outputs are O + O'

,

i.e., on a single input, we either execute the one or the other, depending on which engine was “addressed”.

Conclusion

Given the above considerations, there are two “easy classes”.

- synchronous deterministic with single value I/O:

this is the usual world of synchronous API calls (on the concrete side) and abstract data types naturally live here as mathematical objects

- asynchronous non-deterministic, composed or not, finite number of outputs:

this is the world of event-driven state machines in the concrete and while there is an “obvious” abstract counterpart, it unfortunately leads to a combinatorial explosion of the state space as we have to take into account the reordering of message as part of the mathematical model.

Now, the upshot is: there will be no simple mathematical model of Anoma if we blindly apply the asynchronous composition

and things don’t become easier if we add the partial synchrony assumption. It might pay off greatly to have as many order preserving connections between engines. @tg-x mentioned this just some days ago.