On the Value of Consensus

Patrick Bouchaud, 2019-03-11

Introduction

A question which came up time and again along with the development of Consensus, over the past few years, is: what is it for - or, as per the established Business terminology: what is your Value Proposition?

To answer the question, I have made many presentations, each of them trying to look at my interlocutor's problems from their own standpoint, and showing how Consensus could help in - take your pick: democratizing computer knowledge and allowing each of us to make better informed decisions; utilizing computer sciences as a tool to educate ourselves and develop human intelligence and intuition; reducing the existing, unnecessary complexity in the IT tool landscape as it is, today; completing and supporting the Industry 4.0 vision; addressing climate change; etc.

None of these worked. That is: none of these perspectives seemed compelling enough to motivate my interlocutor(s) to invest, if not their time and money, at least their active support, in this development. No hard feelings: I am sincerely grateful for the opportunities I had to exchange, and the occasional valuable suggestion I received regarding this development. I understand that the approach I propose, on the one hand: challenges some of the most deeply rooted Business practices and fundamental IT Design & Architecture principles - and even, to some extents, the way we relate to ourselves; and on the other: did not promise enough short-term concrete benefits beyond what can already be accomplished today, with our existing tools, practices, and based on the established IT architecture & design principles. In short, its added value comes down *in the short term* to "better quality" - which, by definition, automatically falls into the category "Nice To Have" / "If Time Allows".

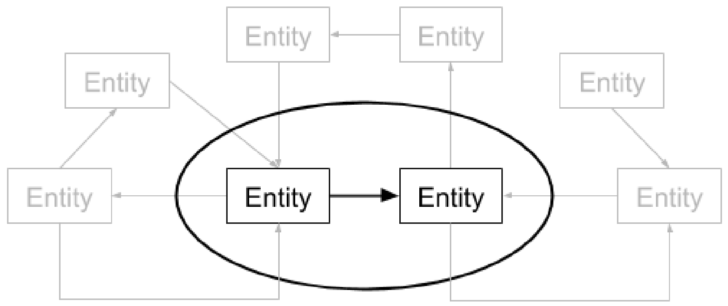
One thing everybody agrees on, however, is that it doesn't hurt to take a look.

The purpose of this article, therefore, is not to convince so much as to *share* the value that I see, with my own mind’s eye, in the Consensus software implementation and design. That is, at this stage: not so much *what* it does, as *how* it does it. Feel free to contact me, either by messaging or commenting on this post, should you wish to know more. In the meantime, my greatest hope is that you will enjoy the ride, at least as much as I do.

I. Consensus: About & Visualization

Consensus, like our brains, relies on simple entities to achieve complex results, and this, simply by allowing to form associations.

These are called, in Consensus lingo, *Relationship Instances*.



Consensus
Relationship Instance

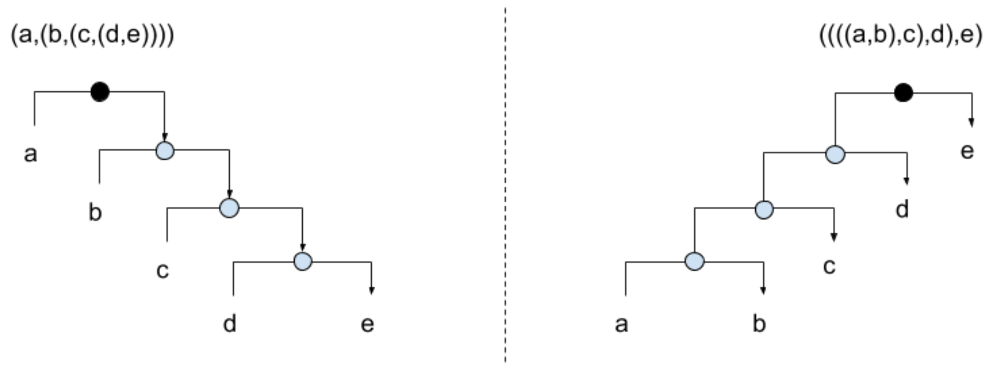
As corroborated by the established model in Neurosciences, information - such as memories - do not reside in the neurons of our brains, but in the states of their connections: in the *circuits*, more precisely, which are activated when information signals - such as sensory impulses - propagate from neuron to neuron.

Each neuron features thousands of connections with others, and their firing, which allows us to produce, and project, frame by frame, our own image of the World, also results in our most fascinating creation: the language - fascinating not least for the fact that it is, between our language and our brains, *as above, so below*.

Hence, where the brain uses the Neuron as its base entity, Consensus uses the Word. Where the brain uses Synapses, Consensus uses Relationship Instances. And, when it comes to describing these relationships in writing, which is a linear representation, Consensus uses parentheses - which is where, in order to keep supporting our intuition, we need a higher-dimensional representation, a drawing, the key to which is to allow to treat relationship instances as entities in their own right.

For this, rather than drawing a circle around its components, to denote the instance, we simply place a dot, which provides a handler, on their connection.

This allows us to visualize e.g. the instances `(a,(b,(c,(d,e))))` and `((((a,b),c),d),e)` as

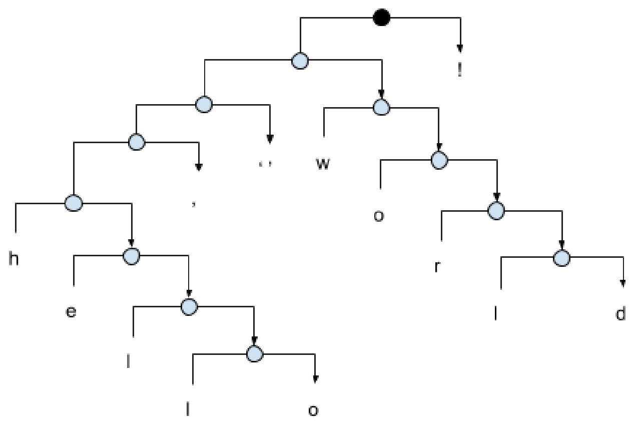


Note that, as per Consensus current implementation, any letter in the illustration above can be replaced with a *name*. However, to familiarize ourselves further with the representation, let us suppose that we want to remove the parentheses altogether from the language.

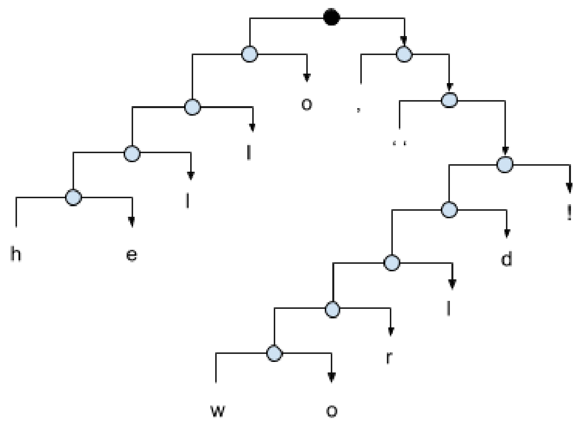
A way to proceed would be to rely on a convention whereby any sequence of non-separators characters converts internally to the expression on the left (resp. right), while the representation on the right (resp. left) is used when coupling with separators.

This would allow, for instance, the sequence `hello, world!` typed in by the user, to convert internally to

`((((h,(e,(l,(l,o)))),','),' '),(w,(o,(r,(l,d))))),'!')`



resp. `(((((h,e),l),l),o),(',',(' ',(((((w,o),r),l),d),'!' ))))`



Note how the sequence structure reveals itself as the main branch of either graph, to which each word connects as a self-contained, reusable instance.

II. Consensus Algebra

Now that we know how to visualize complex Relationship Instances, the key to bringing these representations to life is to allow the user to traverse, query and modify their states, according to the Consensus programming language:

```
in condition
        on event
                do action
```

Where

`condition`, `event` and `action` are *expressions* representing instances.[1]

To solve this problem efficiently, and in a non-recursive manner, we need to switch to an algebraic representation, whereby an instance is entirely defined by the set of its connections:

Instance := (sub[0], sub[1]) × (as_sub[0], as_sub[1])

Where

- sub[i] are instances
- as_sub[i] are the sets of instances in which *instance* participates, as sub i

Let us assume, for example, that the CNDB contains only the base entities source and target and the relationship instance (source, target). We have

- (source, target).sub[0] = source
  (source, target).sub[1] = target
  (source, target).as_sub[0] = {}
  (source, target).as_sub[1] = {}

- source.sub[0] = NULL
  source.sub[1] = NULL
  source.as_sub[0] = { (source, target) }
  source.as_sub[1] = {}

- target.sub[0] = NULL
  target.sub[1] = NULL
  target.as_sub[0] = {}
  target.as_sub[1] = { (source, target) }

[1] See https://github.com/Eyescale/Consensus/tree/master/Release/Version-1.0

In fact, `condition`, `event` and `action` are also relationship instances, and so is the Consensus Narrative itself, but this will be exposed in the next version.

Most importantly we have

(source, target) = source.as_sub[0] ∩ target.as_sub[1]

Which we can generalize as follow:

let I be a set of identifiers (aka. base entities)
let I' := I U { '?', '.', '*' }
let E({ ai }) be an expression of ai where ai in I'

Assuming no expression term is negated in E,
We call

E(ai) := E({ aij }) where aij = ((i==j) ? ai : '.')
E(.) := E({ ai }) where ai = '.'

We have

E({ ai }) = E(a0) ∩ E(a1) ∩ ... ∩ E(an)
Where, for ai not in { '?', '.' }
E(ai) = E(ai) ∩ E(.)
And therefore
~E(ai) = ~E(ai) ∪ ~E(.)

And therefore

~E({ ai }) = (~E(a0)∪~E(.)) ∪ (~E(a1)∪~E(.)) ∪ ... ∪ (~E(an)∪~E(.))
= ~E(a0) ∪ ~E(a1) ∪ ... ∪ ~E(an) ∪ ~E(.)

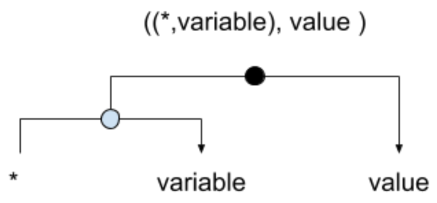
And on this basis we can

- build higher-level expressions F which include negated terms
- reduce F to a boolean logics operation on |F(ai)| and |F(.)|

So that, in conclusion, we can solve any expression without any recursion, by separating each term, with their corresponding as_sub / sub exponents, from the expression, and testing each candidate instance solution (be it the whole CNDB if e.g. F is pure negation) against each term independently.

III. Consensus Value and Change

Consensus supports the notions of *variable* and *value* which have been endorsed by years and years of development practice. Consensus `variable` instances, however, do not "have" values. They are just normal instances, associated with the symbol `*` (aka. the *dereferencing operator*, in traditional programming lingo), associated with a `value` instance.



While `variable` and `value` can be just any instance, and the `(( *, variable ), value )` assignment instance is also just another normal instance, Consensus guarantees that,

> for any `variable` instance, at any given time, there is no more than one `value` instance associated with the `( *, variable )` instance, if it exists.

So, there *is* magic, a developer would say.

Of course there is - but no magic here that cannot be accomplished by simply allowing an operation (or *sub-narrative*[2]) to cause others to hold for a couple of frames, while it performs and keeps track of the changes.

The magic is Change.

And the way Consensus handles changes is by using, hidden from view, a special instance, which here we call *`nil`*, but which really has no *name*, and with which all other instances associate on their way in, and on their way out of, existence, according to the following rules:

1. if `(( instance, nil ), nil )` exists, then
   `instance` is new - literally just came out of non-existence - as of the current frame, and is not yet accessible to the narrative

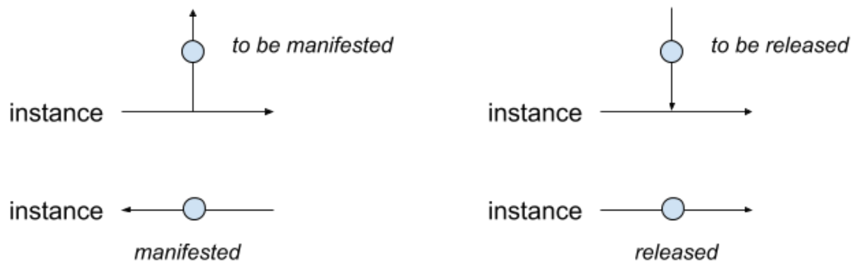
2. if `( nil, instance )` exists, then
   `instance` has just been born, as of last frame, and is now accessible to the narrative

[2] Feature planned for the next release

3. if `( nil, ( instance, nil ))` exists, then
   `instance` must be released, aka. deprecated, at the end of the current frame

4. if `( instance, nil )` exists, then
   `instance` has just been released, as of last frame, and is therefore no longer accessible to the narrative

Since `nil` is, in fact, unnamed, graphically the association with `nil` would be an arrow pointing either to, or away from, the associated instance:



Note however that, as of Version 1.0, neither `nil` nor any of its relationship instances is either visible or directly accessible to the user - although, technically, this would be easily feasible.

Conclusion / What Now?

It is almost impossible to convey the sense of magic, and awe, one feels when everything falls into place, out of chaos.

Hopefully, however, from the snippets above, one can at least get a sense of the beauty I saw with my mind's eye (in the end, I even had to close completely the curtains around me, so I could keep a clear view of all the dependencies) while working on the development of Consensus.

In a way, working on this problem was/is like looking at the Milky Way. And I know that this is not necessarily visible when one looks only at the result - or, to speak poetically: in the light of day.

The prime example is the Turing Machine story[3], i.e. the program written in Consensus language to run a Turing machine.

To my view this program is like a tree whose leaves are rustling in the wind - the wind in this image being the flow of information that streams through the program. I can really see that, because I have been programming for all these years. Actually I could see it even before the completion of this version, and many times I doubted it would really come out. But, now that it is, I understand this may not be so obvious to others.

What may not be immediately apparent here is that the only action this program exerts on its environment is its input/output, i.e. watch, and tell, actions. All other actions only affect its own state, and effects, as far as the program is concerned, occur solely from the fact that it is subscribing to its own changes, as per its writing.

Hopefully, the next version will allow to see not only the tree, but also the forest. Which implies, that each tree not only subscribe to the wind, and the sun, and the earth and the sky, but also to each other; and start to communicate, the way trees do - that is, as far as each tree is concerned, only from watching, and changing, and manifesting their own states.

Which is, also, the way we do.

[3] https://github.com/Eyescale/Consensus/blob/master/Examples/TuringMachine/TM.story