ASKEM Model Documentation

The ASKEM Project

Table of contents

Introduction

This document compiles some of the informal contracts across different implementations of the ASKEM modeling framework, as broadly construed.

Here we try to answer some questions like

- What does a modeling framework consist of?
- What is syntax for models?
- How do we go from syntax to semantics?

Obviously this is a work in progress; we're going to start by "working in the small" and specifying examples before getting to the bigger picture.

This document will contain some code samples, but the technical core will specified via mathematics rather than via implementation in any specific language. The reason for this is that the purpose of ASKEM is not just to develop libraries in specific languages for modeling work; the purpose is to understand how to create modeling abstractions that can work across many different tools. Thus, the core of ASKEM must rest on language-independent foundations, and mathematics is the lingua franca of formalization.

TODO

- Comprehensive references for Petri nets
  - Syntax
  - Semantics
- Comprehensive references for regulatory nets
- Ontological overview of program

1 Modeling Ontology

On this page, we define core terms for the ASKEM project, like "modeling framework", "syntax", and "semantics".

We start with syntax.

1.1 Syntax

The tricky thing about these definitions is that what they capture is not the "intrinsic" nature of what syntax is, but rather how syntax relates to other parts of the ontology.

We can define an antelope to be an animal with certain genotypical and phenotypical attributes. However, we can't define "manager" in the same way. Managers are defined by how they relate to other entities within a context, for instance other employees, purchase orders, the CEO, etc. Similarly we can't define syntax to be a certain concrete thing; we define it by how it relates to other parts of the ontology. Of course, this results in necessarily circular definitions, because we must reference other parts of the ontology before they are built.

We can "close this loop" by formalizing everything within some appropriate logical framework (such as category theory). However, this is meant to be a document that conveys an *intuitive* sense for what various words mean, and this intuitive sense is formalism-independent. That is, we might build different formalisms and identify parts of them as "syntax". The ability to do this is predicated on a consistent understanding of the *role* that syntax plays, and as said before, defining roles requires of necessity reference to other roles which may not be defined yet.

With that out of the way, I will attempt to define the syntax role.

The role of syntax within the ASKEM paradigm is to *build* and *store* models. In order to perform this role, it must have the following characteristics.

1. It must be serializable and deserializable in a programming-language independent way. This is because ASKEM is a multilingual program, and syntax has to be interpreted in (at least) Javascript, Python, and Julia. This rules out the possibility of simply writing Julia or Python code in a string, as is the current practice in industry for "saving" models.

2. It must be formally structured, so that operations of composition and augmentation can be performed on it, and so it can be analysed without running a model (which is called "static analysis").

A particular "syntax" is a some data type that supports the relevant operations of serialization/deserialization, analysis, composition and augmentation. Within ASKEM we have many different syntaxes, and in fact we might also consider small variations on a syntax (for instance, allowing or disallowing custom rates) to define different syntaxes. Ideally, instead of having a fixed list of syntaxes and manually implementing every operation from scratch for each syntax, we can build the syntaxes we need by composing different features together. The degree of reuse that we can obtain in practice is yet to be discovered, however.

1.2 Semantics

A semantic for a given syntax is a way of turning instances of that syntax into some mathematical model. Each semantic has a mathematical specification, which should be written down somewhere (hopefully here), but we also may have one or more computer implementations of that mathematical specification.

Because syntaxes are, by design, "just data", there is not a canonical way to turn them into mathematical models. Of course, there might be a way that is natural for certain scientific or logical reasons, but that is an aesthetic judgment.

One special case of a semantic might be compiling one syntax into another syntax! For instance, we could compile a Petri net into the syntax of a symbolic differential equation.

1.3 Model

A model is a mathematical description of an abstracted part of nature. Models specify the *behavior* of a *system*. What this specifically means varies on the type of model. For instance:

1. A model might simply be a collection of propositions that must be satisfied: "either gene A or gene B is activated"
2. A model might tell you how some state evolves over time, either discretely (i.e., the next step is X), or continuously (the derivative is X)
3. A model could do 1 or 2 stochastically, in that we only get the probability that a certain law is satisfied, or that a state evolves in a certain way.

Models are produced by the application of a semantic to a syntax.

1.4 Modeling framework

A modeling framework consists simply of a choice of a syntax and a semantic.

2 ODE Semantics

There are various frameworks within ASKEM that have "ODE semantics". What does this mean?

Formally speaking, this has something to do with category theory, functors, etc. But we can get at the core of the matter in a fairly first-principles way.

Before we get into the technical details, a review of notation/terminology.

- A **finite set** is a collection of things. The two important questions a finite set answers are:
  - How many things there are?
  - How do we refer to those things?

  Technically speaking, we could get away with only numbering things 1 to n. But it's convenient to give things human-readable names instead.

  Examples of finite sets: $\{1, 2, 3\}, \{S, I, R\}, \{\texttt{susceptible}, \texttt{infected}, \texttt{recovered}\}$
- $\mathbb{R}$ is the set (or type, if you prefer) of real numbers. On a computer, these are represented by floating point numbers.
- If I is a finite set, then $\mathbb{R}^I$ is the set (or type) of *assignments of a real number to each element of* I. So for instance, if $I = \{a, b, c\}$, then $\mathbb{R}^I$ is the set of three-dimensional vectors.

Finally, we will talk about functions $\mathbb{R}^A \to \mathbb{R}^B$. We use "function" as a physicist would; we assume that the function is well-behaved enough to do what we want with it (i.e., solve an ODE). Functions are a tricky subject, because one cannot serialize in a language-independent way an arbitrary function; more on this later.

With this out of the way, an ODE semantics for a modeling frameworks means a systematic way of assigning to each model the following data.

1. A finite set X called the set of state variables. By this we mean a finite set of *names* for state variables. The fact that we think of these as state variables has no mathematical or technical meaning; this is just a set of names. These names could either be descriptive, or could be simply $\{1, \ldots, n\}$.

2. A finite set B called the set of parameter variables. One thing to note is that calling these parameters doesn't mean that they are simply held fixed. They could just as well be dynamic and determined by the output of some other system.
3. A finite set A called the set of output variables.
4. A function $v\colon \mathbb{R}^X \times \mathbb{R}^B \to \mathbb{R}^X$ called the vector field. The associated differential equation to this is written as
$$\dot{x} = v(x, u)$$
where $x \in \mathbb{R}^X$ and $u \in \mathbb{R}^B$
5. A function $f\colon \mathbb{R}^X \to \mathbb{R}^A$ called the *output map*. This part is often neglected, because often $A = X$ and this is the identity map $f(x) = x$. But in cases where we need to do model comparison across models with different sets of state variables, this becomes important.

See (Petri net base semantics), (Petri net mass action semantics), (RegNet Lotka-Volterra semantics) for examples.

After a certain ODE semantics has been applied to a model, we can perform certain operations at the level of just ODEs. These include

1. Reparameterization.
2. Modifying the output.
3. Composition with other ODE models.

3 Petri nets

3.1 What is a Petri net?

TODO

3.2 ODE semantics

There are two ODE semantics for Petri nets, which share much in common.

1. A state variable for each species (representing the population of that species)
2. A parameter variable for each transition (what this represents is different across the two semantics)
3. An output variable for each species
4. (different vector fields)
5. An output function that is the identity.

We now discuss what is different between the two semantics.

3.2.1 Push-pull network

Note: I am not sure exactly what to name this semantics. I have settled on push-pull for now.

In the push-pull network semantics for a Petri net, the parameter for a transition corresponds to the rate at which that transition converts its inputs to its outputs.

Note that this gives no guarantee that populations will never go negative. If you have a Petri net with species A and B and one transition that has input A and output B, and you set the rate of that transition to be a constant, then the population of A will eventually go negative.

Thus, the intended use of the push-pull semantics is to be used with a custom reparameterization that chooses some sensible rate laws.

```
"""
  Computes the vector field for the push_pull_network semantics
"""
function push_pull_network(pn::PetriNet, _populations::Vector{Float64}, rates::Vector{Floa
  # TODO
end
```

```
def push_pull_network(...):
  """
    Computes the vector field for the push_pull_network semantics
  """
  pass
```

3.2.2 Mass-action

One way of thinking about mass-action semantics is that it's a reparameterization of push-pull semantics in the simplest possible way in order to make sure that no population ever goes negative. However, it is a common reparameterization that can be derived uniformly from the structure of the Petri net itself, and thus it makes sense for it to be its own semantics.

The basic idea is to multiply the rate of each transition by the product of the populations of the input species. This ensures that transitions slow down when the input is close to being depleted, in such a way that it doesn't force the input below zero.

```
"""
  Computes the vector field for the mass_action semantics
"""
function mass_action(pn::PetriNet, populations::Vector{Float64}, rates::Vector{Float64})
  # TODO
end
```

```
def mass_action(...):
  """
    Computes the vector field for the mass_action semantics
  """
  pass
```

3.3 References

A good slow-paced reference on Petri nets is (Baez and Biamonte 2018). It has an intimidating title, but chapters 1 and 2 cover the rate equation (i.e. the mass-action semantics) and are purely ODE-based (zero category theory, zero probability theory, zero quantum mechanics).

4 Regulatory networks

4.1 What is a regulatory network?

A regulatory network is a signed graph (todo: expand this)

4.2 ODE semantics

4.2.1 Lotka-Volterra

When interpreting Regnets with Lotka-Volterra semantics, variables of the system are the vertices and interactions are edges.

A *Lotka-Volterra system* of equations has the form

$$\dot{x} * i = \rho_i \, x_i + \sum *j = 1^n \beta\_i, j \, x_i \, x_j, \qquad i = 1, \ldots, n.$$

or equivalently, has logarithmic derivatives that are affine functions of the state variables:

$$\frac{d}{dt}[\log x_i(t)] = \rho * i + \sum *j = 1^n \beta\_i, j \, x_j(t), \qquad i = 1, \ldots, n.$$

The coefficients ρ_i specify baseline rates of growth or decay, according to their sign, and the coefficients $\beta_{i,j}$ rates of activation or inhibition, according to their sign. We construct a functor that sends a signed graph (regulatory network) to a Lotka-Volterra model that constrains the signs of the rate coefficients By working with signed graphs, rather than merely graphs, we ensure that scientific knowledge about whether interactions are promoting or inhibiting is reflected in both the syntax and the quantitative semantics.

In order to comprehend complex biological systems, we must decompose them into small, readily understandable pieces and then compose them back together to reproduce the behavior of the original system. This is the mantra of systems biology, which stresses that compositionality is no less important than reductionism in biology.

Signed Graphs are stored in Catlab using the following schemas:

```
@present SchGraph(FreeSchema)
  (V,E)::Ob
  src::Hom(E,V)
  tgt::Hom(E,V)
end

@present SchSignedGraph <: SchGraph begin
  Sign::AttrType
  sign::Attr(E,Sign)
end

# when you want rates to be stored in the model
@present SchRateSignedGraph <: SchSignedGraph begin
  A::AttrType
  vrate::Attr(V,A)
  erate::Attr(E,A)
end
```

These Catlab Schemas are equivalent to the following SQL

```
CREATE TABLE "Vertices" (
    "id"    INTEGER,
    PRIMARY KEY("id")
);

CREATE TABLE "Edges" (
    "id"    INTEGER,
    "src"   INTEGER,
    "tgt"   INTEGER,
    "sign"  BOOL,
    PRIMARY KEY("id"),
    FOREIGN KEY("src") REFERENCES "Vertices"("id"),
    FOREIGN KEY("tgt") REFERENCES "Vertices"("vid")
);

-- or with rates

CREATE TABLE "Vertices" (
    "id"    INTEGER,
    "vrate" NUMBER,
    PRIMARY KEY("id")
```

```
);

CREATE TABLE "Edges" (
    "id"    INTEGER,
    "src"   INTEGER,
    "tgt"   INTEGER,
    "sign"  BOOL,
    "erate" Number,
    PRIMARY KEY("id"),
    FOREIGN KEY("src") REFERENCES "Vertices"("id"),
    FOREIGN KEY("tgt") REFERENCES "Vertices"("vid")
);
```

The dynamics are given by the following julia program, which iterates over the vertices of the graph and then the neighbors of that vertex (incident edges).

```
function vectorfield(sg::AbstractSignedGraph)
  (u, p, t) -> [
    p[:vrate][i]*u[i] + sum(
        (sg[e,:sign] ? 1 : -1)*p[:erate][e]*u[i]u[sg[e, :src]]
      for e in incident(sg, i, :tgt); init=0.0)
    for i in 1:nv(sg)
  ]
end
```

And they can be drawn using Graphviz using the following snippet that draws positive edges as arrows and negative edges as tees.

```
function Catlab.Graphics.to_graphviz_property_graph(sg::AbstractSignedGraph; kw...)
  get_attr_str(attr, i) = String(has_subpart(sg, attr) ? subpart(sg, i, attr) : Symbol(i))
  pg = PropertyGraph{Any}(;kw...)
  map(parts(sg, :V)) do v
    add_vertex!(pg, label=get_attr_str(:vname, v))
  end
  map(parts(sg, :E)) do e
    add_edge!(pg, sg[e, :src], sg[e, :tgt], label=get_attr_str(:ename, e), arrowhead=(sg[e
  end
  pg
end
```

4.3 References

The Lotka-Volterra semantics described here are from *A compositional account of motifs, mechanisms, and dynamics in biochemical regulatory networks* (Aduddell et al. 2023).

References

Aduddell, Rebekah, James Fairbanks, Amit Kumar, Pablo S. Ocal, Evan Patterson, and Brandon T. Shapiro. 2023. "A Compositional Account of Motifs, Mechanisms, and Dynamics in Biochemical Regulatory Networks."

Baez, John, and Jacob D Biamonte. 2018. *Quantum Techniques in Stochastic Mechanics*. WORLD SCIENTIFIC. https://doi.org/10.1142/10623.