Categorical Epidemic Modelling

This notebook showcases the robustness of a categorical approach to modelling epidemics in their ability to accelerate the composition of various existing standard models, and stratifying the models to study their impact on different groups in the populace.

A preliminary understanding of categorical concepts such as Morphisms, Cospans, Structured Multicospans, Pullbacks and Natural Transformations, and a preliminary understanding of Petri-net modelling is assumed.

Setting up the Environment

We begin by importing the necessary modules for creating our system. Catlab provides support for Category Theoretical constructs and Graphviz, AlgebraicPetri is our friendly neighbourhood Petri-net maker, AlgebraicDynamics aids in composing dynamical systems via UWDs, the remaining modules have self-explanatory names.

If you face any issues in importing the modules in the following cell, make sure to resolve them using your package manager in Julia.

```
begin
    using Catlab, Catlab.CategoricalAlgebra, Catlab.Programs,
    Catlab.WiringDiagrams, Catlab.Graphics
    using AlgebraicPetri
    using AlgebraicDynamics.UWDDynam

    using DifferentialEquations

    using LabelledArrays
    using Plots
end
```

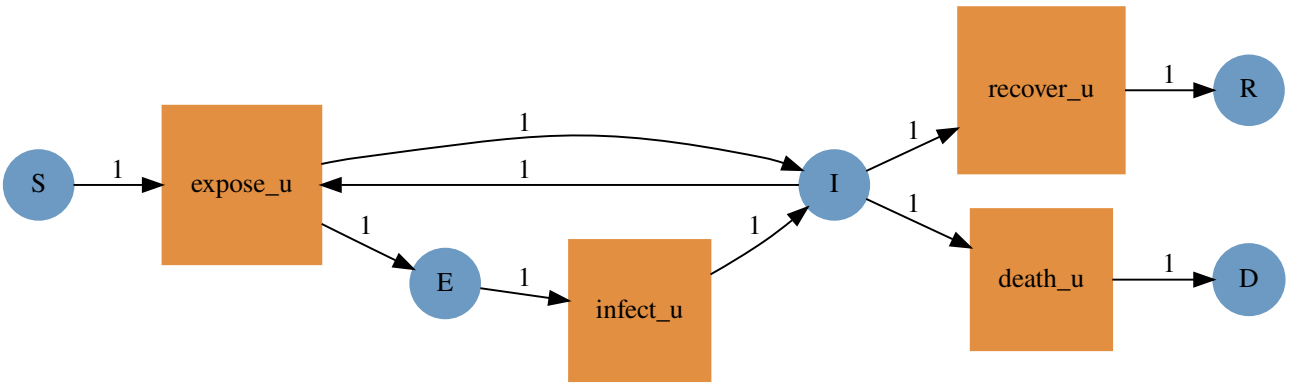
Our first Model (SEIRD)

The SEIRD model is a compartmental model involving the following compartments:

- S (Susceptible) - people who can potentially be infected by the disease through contact with an infected individual
- E (Exposed) - people who are infected with the disease, but are not infectious
- I (Infected) - people who are infected with the disease, and are infectious
- R (Recovered) - people who have gained permanent immunity to the disease
- D (Deceased) - people who died on account of the disease

We model this behaviour as a LabelledPetriNet (labelled because we wish to add 'names' to each transition and population compartment) and use the Open syntax to convert it to an 'open' petri-net in the form of a Structured Multicospan.

You are encouraged to use the Live Docs functionality to understand the syntax for creating a petri net. In short, here we have specified the list of comparments, the list of transitions along with their input and output arcs to create the petri net.




```
begin
    # An SIERD Model (Susceptible, Exposed, Infected, Recovered, Deceased)
    SEIRD_opetri_net = Open(LabelledPetriNet(

        [:S, :E, :I, :R, :D],

        :expose_u => ((:S, :I) => (:E, :I)),
        :infect_u => ((:E) => (:I)),
        :recover_u => ((:I) => (:R)),
        :death_u => ((:I) => (:D)),
        )
    )

    to_graphviz(SEIRD_opetri_net)
end
```

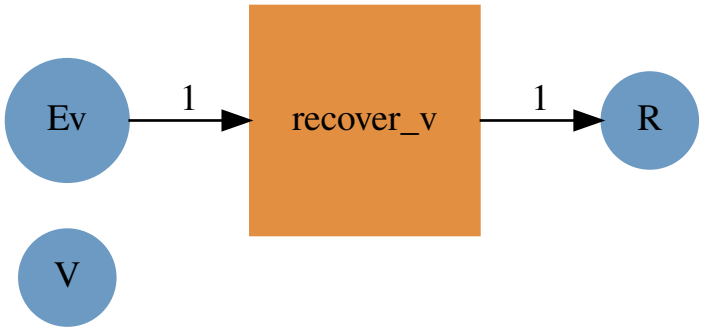
The tool graphviz can be used to visualize the petri-net created above.

Our second Model (VEvR)

The VEvR model is a non-standard compartmental model involving the following compartments:

- V (Vaccinated) - people who have been vaccinated
- Ev (Exposed-Vaccinated) - people who were once vaccinated, but came in contact with an infected individual. These people can never die from the disease and can only recover from it to gain immunity later.
- R (Recovered) - people who have gained permanent immunity to the disease

We use a similar syntax as the one in the previous cells to create and visualise the petri-net corresponding to this model.

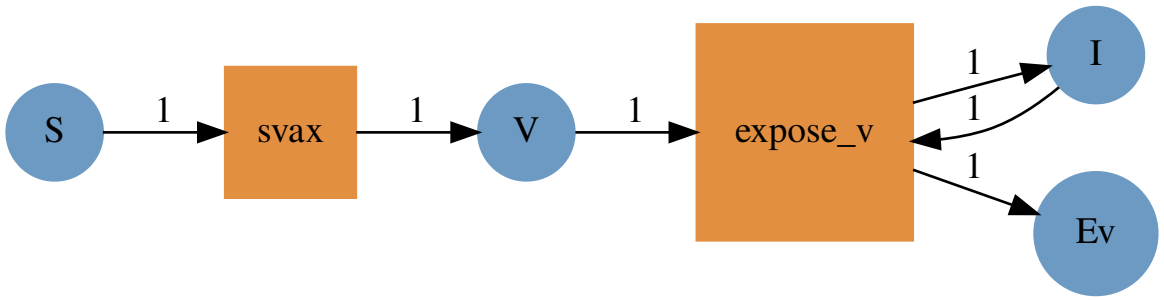



```
begin
    # An VEvR Model (Vaccinated, Exposed_Vaccinated, Recovered)
    VEvR_opetri_net = Open(LabelledPetriNet(

        [:V, :Ev, :R],

        :recover_v => ((:Ev) => (:R)),
        )
    )
    to_graphviz(VEvR_opetri_net)
end
```

Dictating interaction between the two Models

Now we define a cross exposure model which dictates the interaction between the populations in the first and the second model.

- The susceptible people (S) can get vaccinated (svax) and change their compartment from S to V
- The vaccinated people (V) on coming into contact with the infected people (I) can turn into exposed vaccinated (Ev)

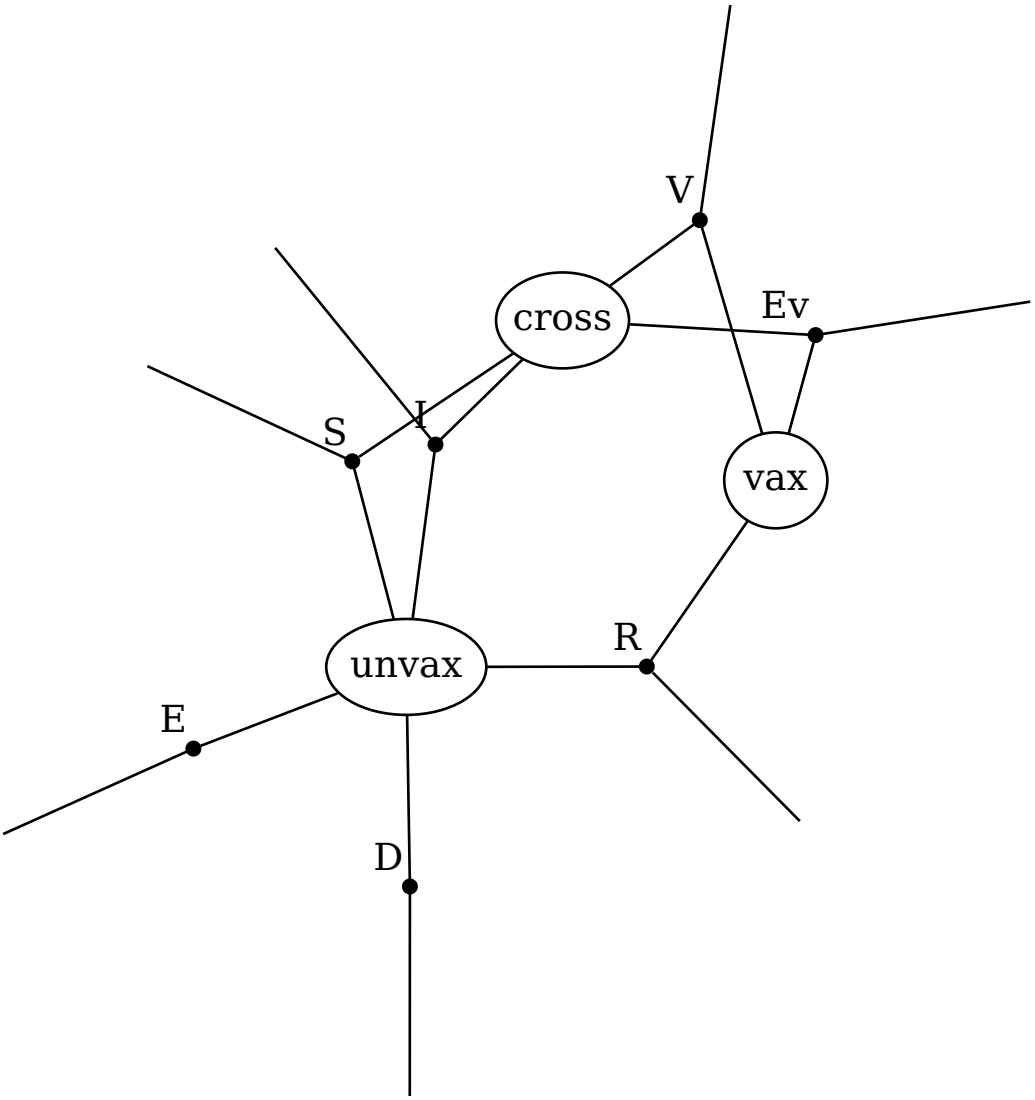



```
begin
    # Dicating cross exposure rules
    cross_exposure_opetri_net = Open(LabelledPetriNet(

        [:S, :I, :V, :Ev],

        :expose_v => ((:V, :I) => (:I, :Ev)),
        :svax => ((:S) => (:V)),
        )
    )
    to_graphviz(cross_exposure_opetri_net)
end
```

Bringing it all Together

Now, we are ready to compose the open petri-nets made so far using operad algebra. An undirected wiring diagram serves as a visual representation of the composition of the structured multicospans.

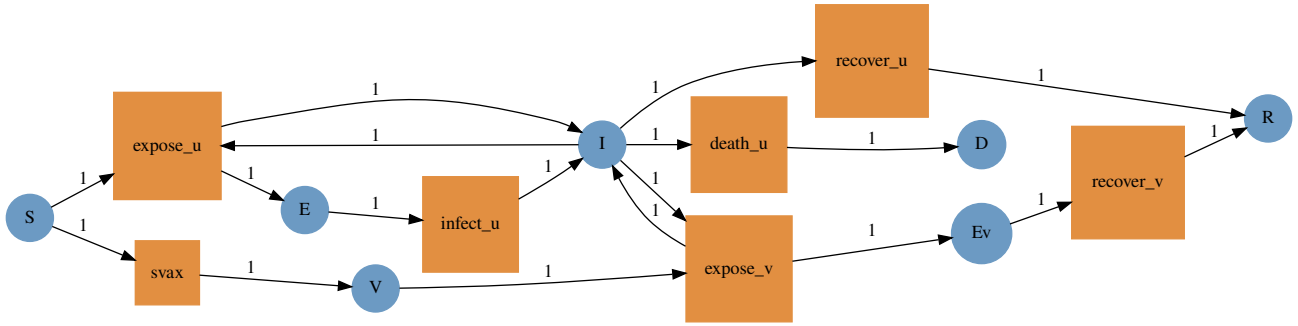
We use the relation notation in Catlab to construct the UWD.




```
begin
    SVIEEvRD_composition_uwd = @relation (S, V, I, E, Ev, R, D) begin
        unvax(S, E, I, R, D)
        vax(V, Ev, R)
        cross(S, I, V, Ev)
    end

    to_graphviz(SVIEEvRD_composition_uwd, box_labels=:name,
    junction_labels=:variable, edge_attrs=Dict(:len => "1"))
end
```

We use Catlab's oapply to compose the models using our UWD. We use the SIERD_opetri_net for unvax, VEvR_opetri_net for vax, cross_exposure_opetri_net for cross.

The remarkable power in categorical modelling in composing various models has been demonstrated here.




```
begin
    SVIEEvRD_opetri_net = oapply(
        SVIEEvRD_composition_uwd,
        Dict(
            :unvax => SEIRD_opetri_net,
            :vax => VEvR_opetri_net,
            :cross => cross_exposure_opetri_net
        )
    )

    to_graphviz(SVIEEvRD_opetri_net)
end
```

Stratification of our Models

The first step to stratification of our model is to create a typing system. Following this, we turn our petri-nets into "typed" petri-nets by specifying the nature of each transition and compartment.

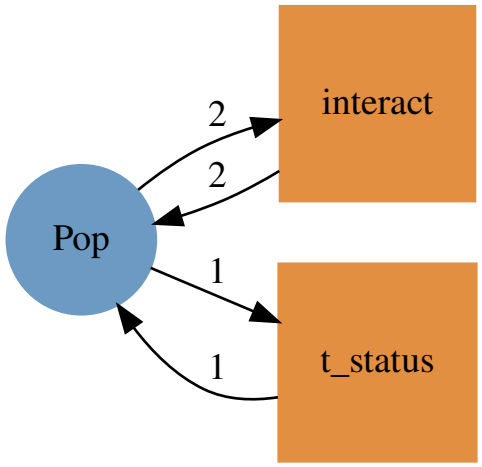
Adding Typing

We begin by creating a general population type and specifying the various kinds of interactions they can go through.

- "interact" - represents a contact between two individuals
- "t_status" - represents a change in the status of the individual (say from vaccinated to unvaccinated for example)

We explicitly extract the TISO componenents of the petri-net in order to associate them later to the various parts of our petri-nets which are to be typed. This association happens in the form of an C-Set Transformation.

We also remove the names of each part to create a loose transformation, removing the structure of the name of the parts of the petri-net.




```
begin
    # General population type
    infectious_type = LabelledPetriNet(
        [:Pop],

        :interact => ((:Pop, :Pop) => (:Pop, :Pop)),
        :t_status => ((:Pop) => (:Pop))
    )

    m = to_graphviz(infectious_type)                          # with names

    s, = parts(infectious_type, :S)
    t_interact, t_status = parts(infectious_type, :T)
    i_interact1, i_interact2, i_status = parts(infectious_type, :I)
    o_interact1, o_interact2, o_status = parts(infectious_type, :O)

    infectious_type = map(infectious_type, Name=name->nothing)  # without names

    m
end
```

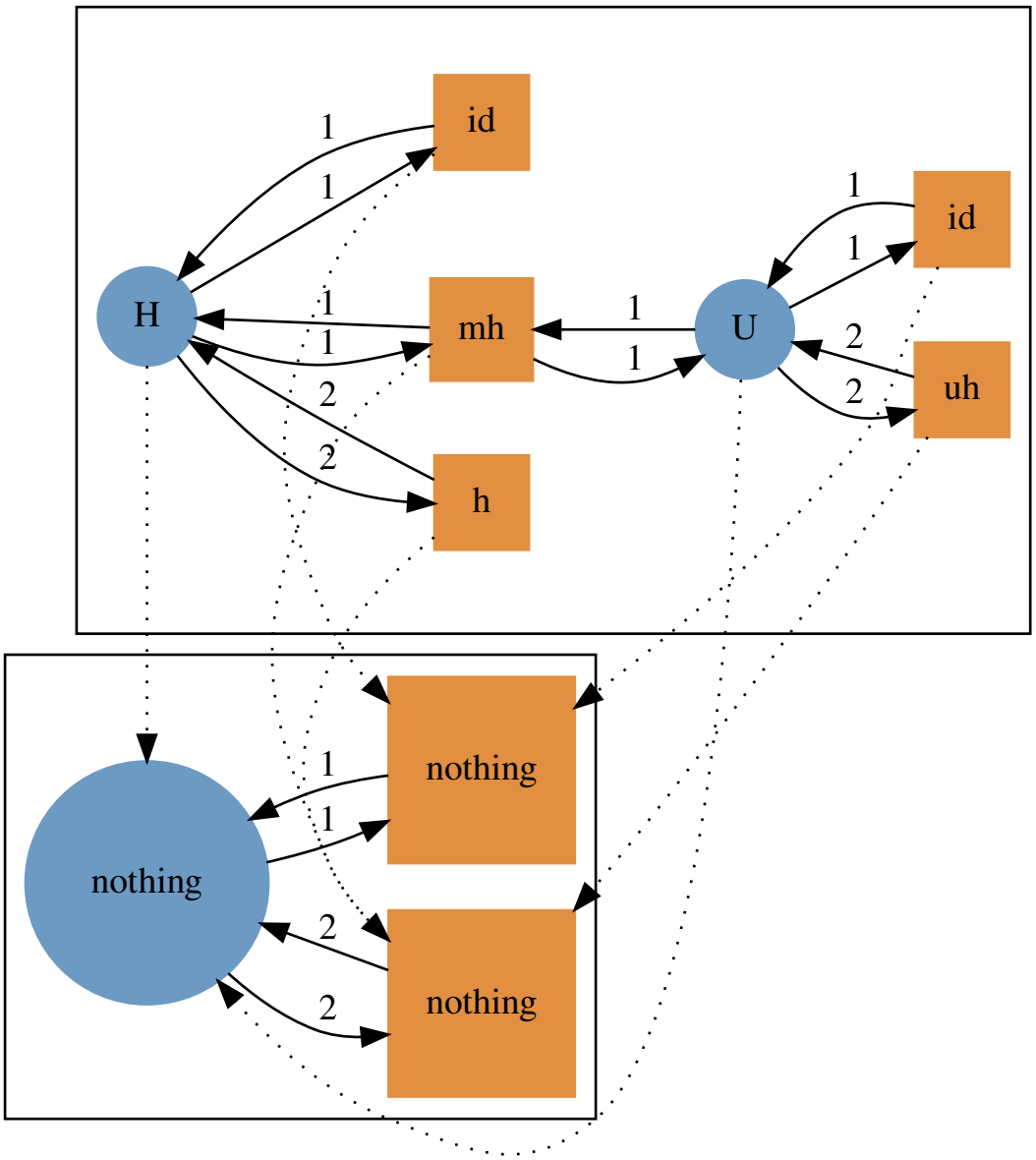
Hygiene Stratification

Now, we create a **hypothetical** "Hygiene" stratifictaion, associating a hygiene class to each individual. An individual can either begin to one of the two classes:

- H: "hygienic" - uses masks, safety precautions, etc. - has a lower chance of infection, faster recovery rate and lower mortality rate
- U: "unhygienic" - does not take safety precautions - has a higher chance of infection, slower recovery rate and higher mortality rate

We have not incorporated mobility across stratifications, however, this addition is trivial and only requires additional typing in our general model. It becomes important to add identity transitions, which in our case aid in association with status transitions enabling the pullback operation to work later on.

Notice how each interaction in our hygiene scheme are "typed" with an interaction from our general population type.


id
1
1
H
1
1
mh
1
1
U
1
1
id
2
2
uh
2
2
h
nothing
1
1
nothing
2
2
nothing

```
begin
    # Hygiene Stratification

    hygiene_petri_net = LabelledPetriNet(
        [:H, :U],

        :h => ((:H, :H) => (:H, :H)),   # interaction between two hygienic people
        :uh => ((:U, :U) => (:U, :U)),  # interaction between two unhygienic
    people
        :mh => ((:U, :H) => (:U, :H)),  # interaction between one from each kind

        :id => (:H => :H),
        :id => (:U => :U),
    )

    typed_hygiene_petri_net = ACSetTransformation(
        hygiene_petri_net, infectious_type,
        S = [s, s],
            # expose,    infect,   recover,  die,      id begins
        T = [t_interact, t_interact, t_interact, t_status, t_status],
        I = [i_interact1, i_interact2, i_interact1, i_interact2, i_interact1,
    i_interact2, i_status, i_status],
        O = [o_interact1, o_interact2, o_interact1, o_interact2, o_interact1,
    o_interact2, o_status, o_status],
        Name = name -> nothing
    )

    to_graphviz(typed_hygiene_petri_net)

end
```

Written below is a helper function to add identity transitions for typing with state transitions in our pre-made SVIEEvRD model

```
begin

    function add_id(petrinet::LabelledPetriNet)
        n_comp = ns(petrinet)
        ts = add_transitions!(petrinet, n_comp, tname = :id)
        add_inputs!(petrinet, n_comp, ts, 1:n_comp)
        add_outputs!(petrinet, n_comp, ts, 1:n_comp)
        return petrinet
    end

    # Adding identity transitions to the compartments
    SVIEEvRD_petri_net = add_id(apex(SVIEEvRD_opetri_net))

    nothing # suppressing output
end
```

Now that we have added the identity transitions, we add typing to the model. Adding typing is quite cumbersome, unless you discover an intelligent system to associate types to the different species and transitions in the petri-net without subjective considerations. Note that the typing of the parts must occur in the same order as they were defined in the original petri-net. If the

petri-net was born out of the composition of other petri-nets, then the specification must follow the order in which they appear in the UWD.

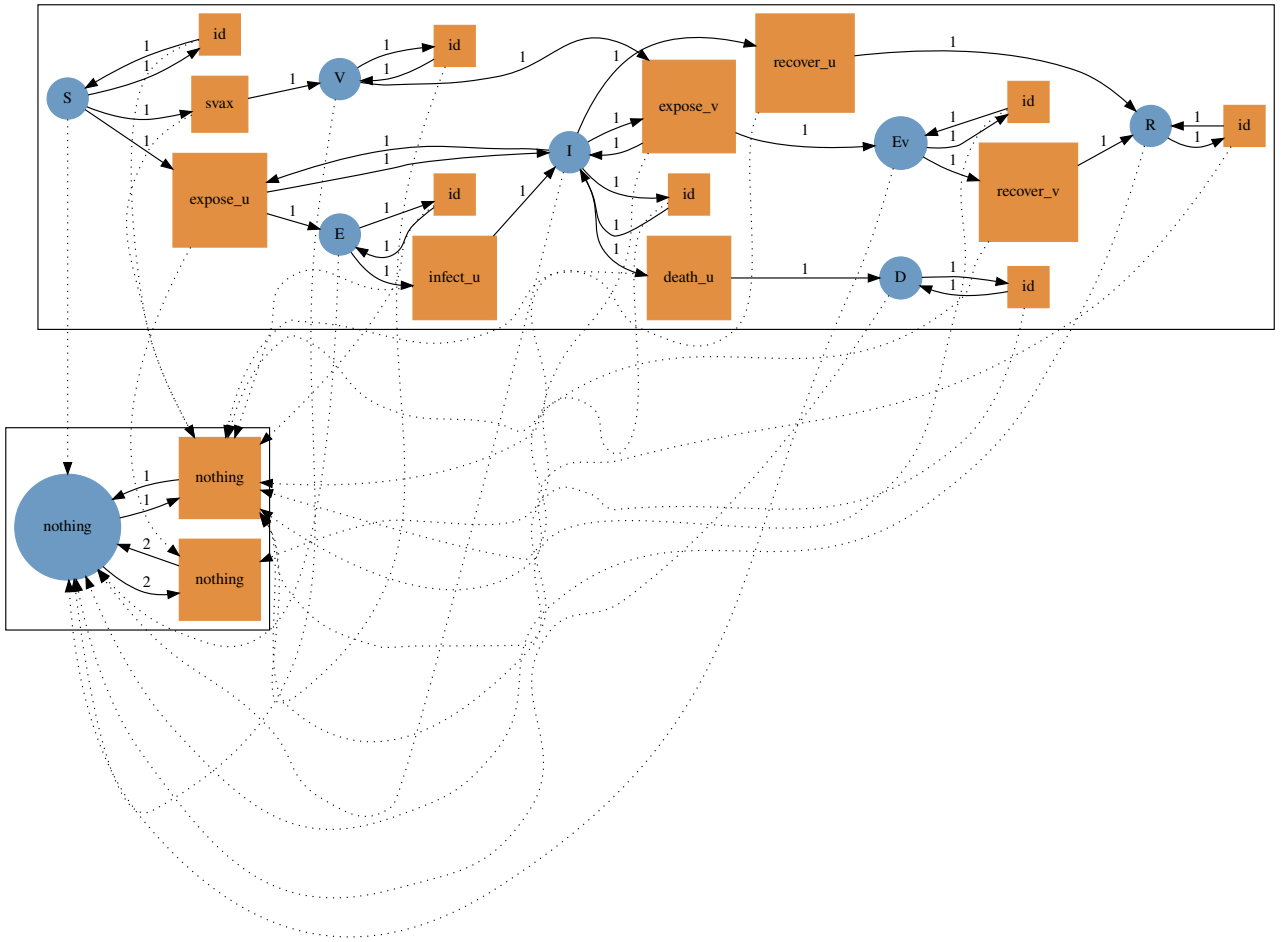
```
typed_SVIEEvRD_petri_net =
ACSetTransformation((T = FinFunction([1, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2], 14, 2
```

```
# Adding typing to the SVIEEvRD_petri_net

typed_SVIEEvRD_petri_net = ACSetTransformation(SVIEEvRD_petri_net,
infectious_type,
    S = repeat([s], ns(SVIEEvRD_petri_net)),
    T = vcat(t_interact, t_status, t_status, t_status, t_status, t_interact,
t_status, repeat([t_status], ns(SVIEEvRD_petri_net))),
    I = vcat(i_interact1, i_interact2, i_status, i_status, i_status, i_status,
i_interact1, i_interact2, i_status, repeat([i_status], ns(SVIEEvRD_petri_net))),
    O = vcat(o_interact1, o_interact2, o_status, o_status, o_status, o_status,
o_interact1, o_interact2, o_status, repeat([o_status], ns(SVIEEvRD_petri_net))),
    Name = name -> nothing
)
```

We must also ensure that our transformation is a natural transformation to ensure commutativity of all morphisms. This fortunately serves as a useful tool to check for proper specification of the typing.

```
@assert is_natural(typed_SVIEEvRD_petri_net)
```

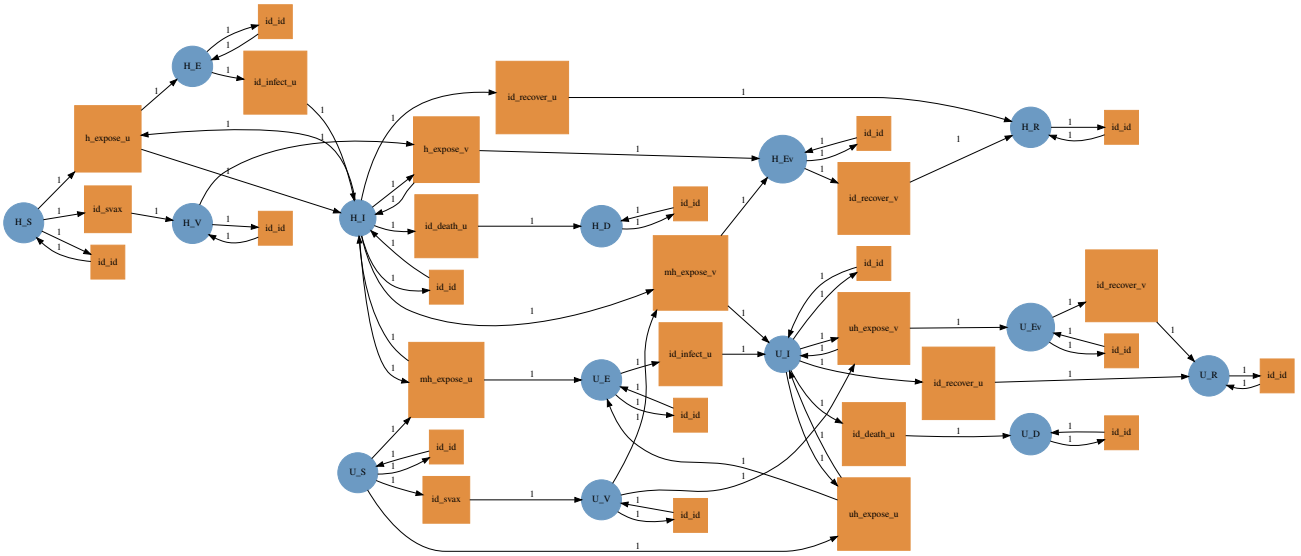
We can once again use graphviz to visualize the typing, that is, the transformation that we are constructing.




```
1 to_graphviz(typed_SVIEEvRD_petri_net, program="dot")
```

Finally, we use a pullback to join together the stratification scheme with our SVIEEvRD model, joined together along the typing scheme, and we extract the object of the pullback which will be our desired petri-net.

By default, the names of the S, T parts in the stratified petri-net occurs as tuples. However, we need to flatten out the names into a single symbol to be able to generate a system of ODEs later on.




```
begin
    # Stratify the model
    UH_SVIEEvRD_petri_net =  ob(pullback(typed_hygiene_petri_net,
    typed_SVIEEvRD_petri_net))

    # Flatten the labels, since our ODE can't handle tuple "names"
    UH_SVIEEvRD_petri_net = flatten_labels(UH_SVIEEvRD_petri_net)
    to_graphviz(UH_SVIEEvRD_petri_net)
end
```

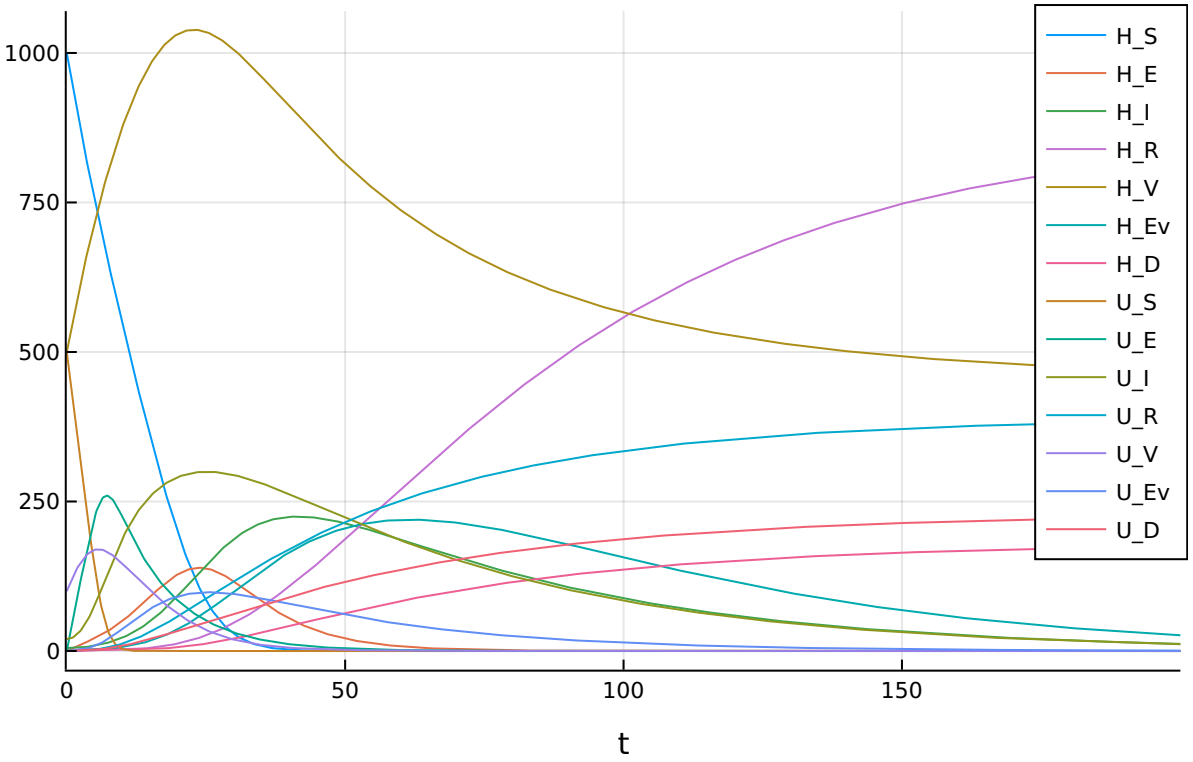
Specifying the Parameters of the System

Now, all that is left to do is to specify the rate parameters for the various transitions and the initial population for the various compartments. The use of LabelledArrays makes it very convenient to keep track of the factors governing our system, and it is encouraged to use them to specify the rate parameters and initial conditions. The names of the rate parameters are the same as the transitions and the names of the compartments is the same as the names of the species in the petri-net.

```
LabelledArrays.LArray{Int64, 1, Vector{Int64}, (:H_S, :H_E, :H_I, :H_R, :H_V, :H_Ev, :H_D,
```

```
begin
    p = LVector(

        h_expose_u  = 0.001,
        uh_expose_u = 0.005,
        mh_expose_u = 0.002,
        id_infect_u = 0.1,
        id_recover_u = 0.010,
        id_death_u = 0.010,

        h_expose_v = 0.00005,
        uh_expose_v = 0.0003,
        mh_expose_v = 0.0002,
        id_recover_v = 0.03,

        id_svax = 0.05,
        id_id = 1
    )

    u0 = LVector(
        H_S = 1000,
        H_E = 0,
        H_I = 5,
        H_R = 0,
        H_V = 500,
        H_Ev = 0,
        H_D = 0,
        U_S = 500,
        U_E = 0,
        U_I = 20,
        U_R = 0,
        U_V = 100,
        U_Ev = 0,
        U_D = 0,
    )
end
```

Finally, we generate a system of ordinary differential equations (in their associated vectorfield) using the vectorfield function, and use julia's DifferentialEquations's ODEProblem to solve it.




```
begin
    soln = solve(ODEProblem(vectorfield(UH_SVIEEvRD_petri_net), u0, (0.0,
    200.0), p))

    plotly()
    plot(soln)
end
```

⚠ Using arrays or dicts to store parameters of different types can hurt performance.
Consider using tuples instead.