Composing an SIR model from individual transitions using AlgebraicPetri.jl

Simon Frost (@sdwfrost), 2023-06-13

## Introduction

This example serves as a ‘Hello World’ to composing Petri net models, where models are ‘glued’ together using the shared states of the individual submodels.

## Libraries

using AlgebraicPetri,AlgebraicPetri.TypedPetri  
using Catlab, Catlab.CategoricalAlgebra, Catlab.Programs  
using Catlab.WiringDiagrams  
using AlgebraicDynamics.UWDDynam  
using LabelledArrays  
using OrdinaryDiffEq  
using Plots

## Transitions

We first define a labelled Petri net that has the different types of transition in our models. The first argument is an array of state names as symbols (here, a generic :Pop), followed by the transitions in the model. Transitions are given as transition\_name=>((input\_states)=>(output\_states)).

epi\_lpn = LabelledPetriNet(  
 [:Pop],  
 :infection=>((:Pop, :Pop)=>(:Pop, :Pop)),  
 :recovery=>(:Pop=>:Pop)  
);

Labelled Petri nets contain four types of fields; S, states or species; T, transitions; I, inputs; and O, outputs.

Next, we define the transmission model (from susceptibles, S to infecteds, I) as an undirected wiring diagram using the @relation macro, referring to the transitions in our labelled Petri net above (infection). We include a reference to Pop in the definition of the state variables to allow us to do this.

si\_uwd = @relation (S, I) where (S::Pop, I::Pop) begin  
 infection(S, I, I, I)  
end;

We then use oapply\_typed, which takes in a labelled Petri net (here, epi\_lpn) and an undirected wiring diagram (si\_uwd), where each of the boxes is labeled by a symbol that matches the label of a transition in the Petri net, in addition to an array of symbols for each of the rates in the wiring diagram. This produces a Petri net given by colimiting the transitions together, and returns the ACSetTransformation from that Petri net to the type system.

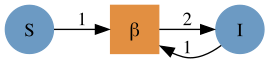
si\_acst = oapply\_typed(epi\_lpn, si\_uwd, [:β]);

To obtain the labelled Petri net, we extract the domain of the ACSetTransformation using dom.

si\_lpn = dom(si\_acst);

We can obtain a GraphViz representation of the labelled Petri net using Graph.

Graph(si\_lpn)



We repeat for the recovery (I to R) transition.

ir\_uwd = @relation (I, R) where (I::Pop, R::Pop) begin  
 recovery(I, R)  
end  
ir\_acst = oapply\_typed(epi\_lpn, ir\_uwd, [:γ])  
ir\_lpn = dom(ir\_acst)  
Graph(ir\_lpn)



To glue the SI and IR models together to make an SIR model, we first define an undirected wiring diagram which contains all our states, and two transitions.

sir\_uwd = @relation (S, I, R) where (S::Pop, I::Pop, R::Pop) begin  
 si(S, I)  
 ir(I, R)  
end;

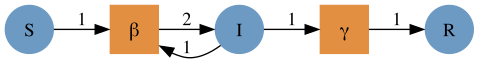
We then create a StructuredMulticospan using this wiring diagram, telling oapply that si in the wiring diagram corresponds to the si\_lpn labelled Petri net, etc.. Open converts a PetriNet to an OpenPetriNet where each state is exposed as a leg of the cospan, allowing it to be composed over an undirected wiring diagram.

sir\_smc = oapply(sir\_uwd, Dict(  
 :si => Open(si\_lpn),  
 :ir => Open(ir\_lpn),  
))

StructuredMulticospan{Catlab.CategoricalAlgebra.StructuredCospans.DiscreteACSet{AnonACSet{TypeLevelBasicSchema{Symbol, Tuple{:S}, Tuple{}, Tuple{:Name}, Tuple{(:sname, :S, :Name)}}, Tuple{Symbol}, Catlab.LVectors.LVector{(:S,), Int64}, NamedTuple{(:sname,), Tuple{Catlab.ColumnImplementations.DenseColumn{Symbol, Vector{Symbol}}}}}, LabelledPetriNet}, Multicospan{LabelledPetriNet, StructTightACSetTransformation{TypeLevelBasicSchema{Symbol, Tuple{:T, :S, :I, :O}, Tuple{(:it, :I, :T), (:is, :I, :S), (:ot, :O, :T), (:os, :O, :S)}, Tuple{:Name}, Tuple{(:tname, :T, :Name), (:sname, :S, :Name)}}, NamedTuple{(:T, :S, :I, :O), NTuple{4, Catlab.CategoricalAlgebra.FinSets.FinDomFunctionVector{Int64, Vector{Int64}, Catlab.CategoricalAlgebra.FinSets.FinSetInt}}}, LabelledPetriNet, LabelledPetriNet}, Vector{StructTightACSetTransformation{TypeLevelBasicSchema{Symbol, Tuple{:T, :S, :I, :O}, Tuple{(:it, :I, :T), (:is, :I, :S), (:ot, :O, :T), (:os, :O, :S)}, Tuple{:Name}, Tuple{(:tname, :T, :Name), (:sname, :S, :Name)}}, NamedTuple{(:T, :S, :I, :O), NTuple{4, Catlab.CategoricalAlgebra.FinSets.FinDomFunctionVector{Int64, Vector{Int64}, Catlab.CategoricalAlgebra.FinSets.FinSetInt}}}, LabelledPetriNet, LabelledPetriNet}}}, Vector{AnonACSet{TypeLevelBasicSchema{Symbol, Tuple{:S}, Tuple{}, Tuple{:Name}, Tuple{(:sname, :S, :Name)}}, Tuple{Symbol}, Catlab.LVectors.LVector{(:S,), Int64}, NamedTuple{(:sname,), Tuple{Catlab.ColumnImplementations.DenseColumn{Symbol, Vector{Symbol}}}}}}}(Multicospan{LabelledPetriNet, StructTightACSetTransformation{TypeLevelBasicSchema{Symbol, Tuple{:T, :S, :I, :O}, Tuple{(:it, :I, :T), (:is, :I, :S), (:ot, :O, :T), (:os, :O, :S)}, Tuple{:Name}, Tuple{(:tname, :T, :Name), (:sname, :S, :Name)}}, NamedTuple{(:T, :S, :I, :O), NTuple{4, Catlab.CategoricalAlgebra.FinSets.FinDomFunctionVector{Int64, Vector{Int64}, Catlab.CategoricalAlgebra.FinSets.FinSetInt}}}, LabelledPetriNet, LabelledPetriNet}, Vector{StructTightACSetTransformation{TypeLevelBasicSchema{Symbol, Tuple{:T, :S, :I, :O}, Tuple{(:it, :I, :T), (:is, :I, :S), (:ot, :O, :T), (:os, :O, :S)}, Tuple{:Name}, Tuple{(:tname, :T, :Name), (:sname, :S, :Name)}}, NamedTuple{(:T, :S, :I, :O), NTuple{4, Catlab.CategoricalAlgebra.FinSets.FinDomFunctionVector{Int64, Vector{Int64}, Catlab.CategoricalAlgebra.FinSets.FinSetInt}}}, LabelledPetriNet, LabelledPetriNet}}}(LabelledPetriNet:  
 T = 1:2  
 S = 1:3  
 I = 1:3  
 O = 1:3  
 it : I → T = [1, 1, 2]  
 is : I → S = [1, 2, 2]  
 ot : O → T = [1, 1, 2]  
 os : O → S = [2, 2, 3]  
 tname : T → Name = [:β, :γ]  
 sname : S → Name = [:S, :I, :R], StructTightACSetTransformation{TypeLevelBasicSchema{Symbol, Tuple{:T, :S, :I, :O}, Tuple{(:it, :I, :T), (:is, :I, :S), (:ot, :O, :T), (:os, :O, :S)}, Tuple{:Name}, Tuple{(:tname, :T, :Name), (:sname, :S, :Name)}}, NamedTuple{(:T, :S, :I, :O), NTuple{4, Catlab.CategoricalAlgebra.FinSets.FinDomFunctionVector{Int64, Vector{Int64}, Catlab.CategoricalAlgebra.FinSets.FinSetInt}}}, LabelledPetriNet, LabelledPetriNet}[ACSetTransformation((T = FinFunction(Int64[], 0, 2), S = FinFunction([1], 1, 3), I = FinFunction(Int64[], 0, 3), O = FinFunction(Int64[], 0, 3)), LabelledPetriNet {T = 0, S = 1, I = 0, O = 0}, LabelledPetriNet {T = 2, S = 3, I = 3, O = 3}), ACSetTransformation((T = FinFunction(Int64[], 0, 2), S = FinFunction([2], 1, 3), I = FinFunction(Int64[], 0, 3), O = FinFunction(Int64[], 0, 3)), LabelledPetriNet {T = 0, S = 1, I = 0, O = 0}, LabelledPetriNet {T = 2, S = 3, I = 3, O = 3}), ACSetTransformation((T = FinFunction(Int64[], 0, 2), S = FinFunction([3], 1, 3), I = FinFunction(Int64[], 0, 3), O = FinFunction(Int64[], 0, 3)), LabelledPetriNet {T = 0, S = 1, I = 0, O = 0}, LabelledPetriNet {T = 2, S = 3, I = 3, O = 3})]), AnonACSet{TypeLevelBasicSchema{Symbol, Tuple{:S}, Tuple{}, Tuple{:Name}, Tuple{(:sname, :S, :Name)}}, Tuple{Symbol}, Catlab.LVectors.LVector{(:S,), Int64}, NamedTuple{(:sname,), Tuple{Catlab.ColumnImplementations.DenseColumn{Symbol, Vector{Symbol}}}}}[AnonACSet{TypeLevelBasicSchema{Symbol, Tuple{:S}, Tuple{}, Tuple{:Name}, Tuple{(:sname, :S, :Name)}}, Tuple{Symbol}, Catlab.LVectors.LVector{(:S,), Int64}, NamedTuple{(:sname,), Tuple{Catlab.ColumnImplementations.DenseColumn{Symbol, Vector{Symbol}}}}}:  
 S = 1:1  
 sname : S → Name = [:S], AnonACSet{TypeLevelBasicSchema{Symbol, Tuple{:S}, Tuple{}, Tuple{:Name}, Tuple{(:sname, :S, :Name)}}, Tuple{Symbol}, Catlab.LVectors.LVector{(:S,), Int64}, NamedTuple{(:sname,), Tuple{Catlab.ColumnImplementations.DenseColumn{Symbol, Vector{Symbol}}}}}:  
 S = 1:1  
 sname : S → Name = [:I], AnonACSet{TypeLevelBasicSchema{Symbol, Tuple{:S}, Tuple{}, Tuple{:Name}, Tuple{(:sname, :S, :Name)}}, Tuple{Symbol}, Catlab.LVectors.LVector{(:S,), Int64}, NamedTuple{(:sname,), Tuple{Catlab.ColumnImplementations.DenseColumn{Symbol, Vector{Symbol}}}}}:  
 S = 1:1  
 sname : S → Name = [:R]])

We extract the labelled Petri net by extracting the object that is the codomain of all the legs, using the apex function.

sir\_lpn = apex(sir\_smc)  
Graph(sir\_lpn)



## Running the model

To run an ODE model from the labelled Petri net, we generate a function that can be passed to SciML’s ODEProblem using vectorfield.

sir\_vf = vectorfield(sir\_lpn);

The initial conditions and parameter values are written as labelled arrays.

u0 = @LArray [990.0, 10.0, 0.0] Tuple(snames(sir\_lpn))

3-element LArray{Float64, 1, Vector{Float64}, (:S, :I, :R)}:  
 :S => 990.0  
 :I => 10.0  
 :R => 0.0

p = @LArray [0.5/1000, 0.25] Tuple(tnames(sir\_lpn))

2-element LArray{Float64, 1, Vector{Float64}, (:β, :γ)}:  
 :β => 0.0005  
 :γ => 0.25

tspan = (0.0, 40.0);

We can now use the initial conditions, the time span, and the parameter values to simulate the system.

sir\_prob = ODEProblem(sir\_vf, u0, tspan, p)  
sir\_sol = solve(sir\_prob, Rosenbrock32())  
plot(sir\_sol)

