JSML Documentation

JSML Cheatsheet

Edit

Defining Types

Generally speaking, you'll want to define types for physical quantities. This will include the units associated with that physical type along with things like limits. Here are a few basic types for electrical systems (although the goal will be to eventually create a comprehensive library of SI types).

```
type Voltage = Real(units="V")
type Current = Real(units="A")
type Resistance = Real(units="Ω", min=0)
type Capacitance = Real(units="F", min=0)
type Inductance = Real(units="H", min=0)
type VoltageRate = Real(units="V/s")
```

There is no MTK equivalent of `type`, but we'll see the effect of these types in the next section.

Creating a Connector

In JSML, a connector is defined with matching pairs of `potential` and `flow` variables and then any number of `stream` and `singleton` fields. The latter two are not discussed here, but a basic electrical pin connector could be defined as:

```

connector Pin
potential v::Voltage
flow i::Current
end
```

...and the generated MTK code would be:

```
@connector Pin begin
  v(t), [unit = u"V"]
  i(t), [unit = u"A", connect = Flow]
end
export Pin
Pin
```

Note how the variables have units? Those units are based on the `type` used in the JSML source.

Creating a Model

In JSML, creating an acausal component model looks like this:

```
component Resistor
  parameter R::Resistance
  p = Pin()
  n = Pin()
  variable v::Voltage
  variable i::Current
relations
  v = p.v - n.v
  i = p.i
  p.i + n.i = 0
  v = i * R
end
```

The generated Julia code will then look like this:

```
@component function Resistor(; name, R=nothing)
  systems = @named begin
    p = Pin()
    n = Pin()
  end
  vars = @variables begin
    v(t), [unit = u"V", guess = 0]
    i(t), [unit = u"A", guess = 0]
```

JSML **does not have comments**. But it does have descriptive strings that are part of the JSML grammar and, therefore, bind the descriptions to specific entities. So, for example, we could add these descriptive strings to our JSML model:

```
# A basic linear [resistor](https://en.wikipedia.org/wiki/Resistor)
component Resistor
  # The resistance of the resistor
  parameter R::Resistance
  p = Pin()
  n = Pin()
  variable v::Voltage
  variable i::Current
relations
  v = p.v - n.v
  i = p.i
  p.i + n.i = 0
  # Ohm's law
  v = i * R
end
```

...and we'd get the following updated Julia code:

```
"""
A basic linear [resistor](https://en.wikipedia.org/wiki/Resistor)
"""
@component function Resistor(; name, R=nothing)
```

System Models

A system level model in JSML would look something like:

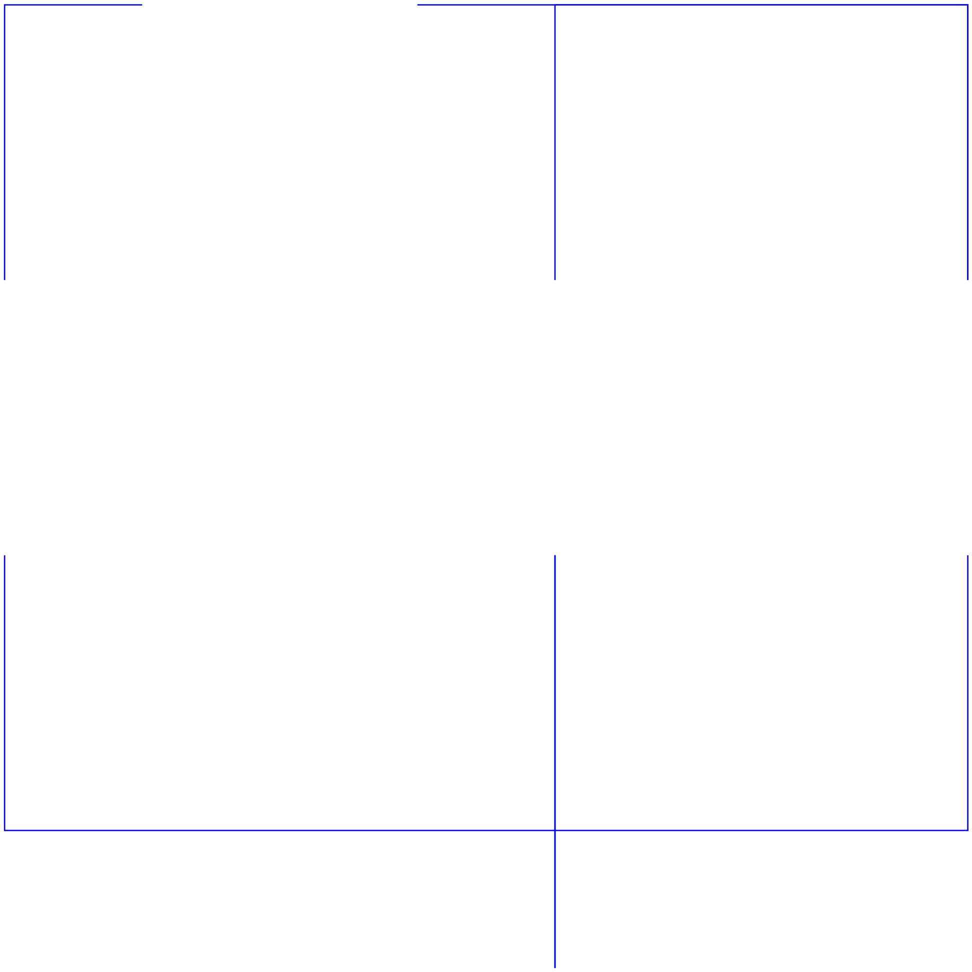
```
component RLCModel
  resistor = Resistor(R=100)
  capacitor = Capacitor(C=1m)
  inductor = Inductor(L=1)
  source::TwoPin = ConstantVoltage(V=30)
  ground = Ground()
relations
  initial inductor.i = 0
  connect(source.p, inductor.n)
  connect(inductor.p, resistor.p, capacitor.p)
  connect(resistor.n, ground.g, capacitor.n, source.n)
end
```

Graphics

In order to represent the layout the components and connections in a system level mo we add metadata to the model. So our previous `RLCModel` with graphical layout information would look like this:

```
component RLCModel
  resistor = Resistor(R=100) [
      { "JuliaSim": { "placement": { "icon": { "x1": 700, "y1": 400, "x2": 900, "y2":
    ]
  capacitor = Capacitor(C=1m) [
      { "JuliaSim": { "placement": { "icon": { "x1": 400, "y1": 400, "x2": 600, "y2":
    ]
  inductor = Inductor(L=1) [
      { "JuliaSim": { "placement": { "icon": { "x1": 200, "y1": 100, "x2": 400, "y2":
    ]
  source::TwoPin = ConstantVoltage(V=30) [
      { "JuliaSim": { "placement": { "icon": { "x1": 0, "y1": 400, "x2": 200, "y2": 6
    ]
  ground = Ground() [
      { "JuliaSim": { "placement": { "icon": { "x1": 400, "y1": 900, "x2": 600, "y2":
    ]
relations
  initial inductor.i = 0
  connect(source.p, inductor.n) [{
    "JuliaSim": {
        "route": [[{"x": 100, "y":200}]]
    }
  }]
  connect(inductor.p, resistor.p, capacitor.p) [{
    "JuliaSim": {
        "route": [
            [{"x": 500, "y": 200}, {"x": 800, "y":200}],
            [{"x": 800, "y": 200}, {"x": 500, "y": 200}]
        ]
    }
  }]
  connect(resistor.n, ground.g, capacitor.n, source.n) [{
    "JuliaSim": {
        "route": [
            [{"x": 800, "y": 800}, {"x": 500, "y": 800}],
            [{"x": 500, "y": 800}],
            [{"x": 500, "y": 800}, {"x": 100, "y": 800}]
        ]
    }
  }]
```

...will have the same generated Julia code. But it is also possible to generate an SVG of the system:



RLC Model Diagram

Test Cases

A component definition can also include experiments and test cases. For example, the same `RLCModel` could be augmented with metadata as follows:

```
component RLCModel
  resistor = Resistor(R=100)
  capacitor = Capacitor(C=1m)
  inductor = Inductor(L=1)
  source::TwoPin = ConstantVoltage(V=30)
  ground = Ground()
relations
  initial inductor.i = 0
  connect(source.p, inductor.n)
  connect(inductor.p, resistor.p, capacitor.p)
  connect(resistor.n, ground.g, capacitor.n, source.n)
metadata {
  "JuliaSim": {
    "experiments": {
      "simple": { "start": 0, "stop": 10.0, "initial": { "capacitor.v": 10, "inductor
    },
    "tests": {
      "case1": {
        "stop": 10,
        "initial": { "capacitor.v": 10, "inductor.i": 0 },
        "expect": {
            "initial": {
                "t": 0,
                "capacitor.v": 10.0
            },
            "final": {
                "t": 10.0
            }
        }
      }
    }
  }
}
end
```

...and the generated Julia code would be augmented with additional functions, `@test`s and `@testset`s as a result:

```
"""Run model RLCModel from 0 to 10"""
```

Expressions

```
expression
  | simple_expression
  | ternary_expression

simple_expression
  | logical_expressions (':' logical_expression (`:` logical_expression)?)?

ternary_expression
  | 'if' expression 'then' expression ('elseif' expressions 'then' expression)* 'else

logical_expression
  | logical_term ('or' logical_term)*

logical_term
  | logical_factor ('and' logical_factor)*
```

Edit this page

Previous
JSML Tutorial

Next
Library Design