Sandia National Laboratories

Exceptional service in the national interest

Attribute Trees

Fred Rothganger

Sandia National Labs / 1421 Cognitive & Emerging Computing

NICE 2026

SAND2026-19078C

Motivation

- Why do neural computing?
  - Better AI
  - Understand how brains (and people) work – just because it is a scientific mystery

- Whole-brain modeling
  - Complex parts
    - Biological synapses have hundreds of moving parts (different proteins) that form complex signaling networks. They hold multiple chemical reservoirs, implying multiple state variables.
    - Biological neurons have dynamics that extend over multiple time scales.
  - Many part types: n = 100 to 1000
  - O(n) connection rules

Modeling frameworks

- How to get there?
  - Collaboration among thousands of scientists
  - AI-based analysis of literature and data

- Requires a common framework for the model
  - Central repository that everyone can edit (like Wikipedia or an OSS project)
  - Single coherent model, not just a stamp collection of one-off models
    - → Common language
  - Most frameworks are not designed to integrate a model at this scale
    - 1000s of part types, each with 10s of dynamic variables
    - 1000s of connectivity patterns
    - shared by 1000s of developers

Modeling frameworks

- But why do I care about that? I just want to implement my app.
  - Practical neuromorphic algorithms do require some complexity
    - 10s of populations, 10s of connectivity patterns
  - We can have both
    - A framework that supports large scale can also be convenient for small scale
    - A framework that supports complex neuroscience can also be convenient for simple neuromorphic
- Desirable features
  - User-defined dynamics
  - Shareable – Marketplace of parts
  - Recursive composition – No limit on depth of component structure
    - → Part definition, part use and network construction must all be in the same language!
  - Concise
  - Readable

Modeling frameworks

Many attempts

- Fugu (in its present form)
- ONNX
- NIR
- NeuroML / LEMS
- BRIAN
- PyNN
- NestML
- NEURON NMODL
- …

Attribute trees can achieve all the desired features (user-defined dynamics, recursive composition, share/re-use, etc.)

However, there is a learning curve involved in using them.
The following presentation introduces the basic concepts, step by step.

Attribute trees

Examples of attribute trees:

- Windows registry
- Schema-free databases

An attribute tree consists of one type of node:

key : value
    children – a list of nodes with locally-unique keys

Keys and values are usually strings.
Value can be undefined, which is slightly different than “” (empty string).
The list of children can be empty.

Concrete realizations

YAML

```
A: eh
  1: one
  2: two
B: bee
```

JSON

```
{
  "A": {
    "": "eh",
    "1": "one",
    "2": "two"
  },
  "B": "bee"
}
```

XML – More verbose and complicated.

Since YAML is the most concise, we keep using it in this presentation.

Translating math to tree form

$$\frac{dV}{dt} = \begin{cases} 0 & if\ V \geq Vspike \\ (G(E-V)+I)/C & otherwise \end{cases}$$

$$V \leftarrow \begin{cases} E & if\ start \\ Vreset & if\ V \geq Vspike \\ V & otherwise \end{cases}$$

$$\Rightarrow$$

```
LIF:
    fire: V>=Vspike
    V ':
        @: (G*(E-V)+I)/C
        @fire: 0
    V:
        @$init: E
        @fire: Vreset
    C: 10pF
    E: -70mV
    G: 3nS
    Vreset: -70mV
    Vspike: -50mV
```

The tree is organized so that keys are part names (LIF), variables (V) or conditions (@…).

Parameters (C, E, G, Vreset, Vspike) are the same as variables.

The prime notation (V ') indicates a time derivative.

Any variable can act as a Boolean. Zero is false. Nonzero is true.

Translating math to tree form

$$\frac{dV}{dt} = \begin{cases} 0 & if\ V \geq Vspike \\ (G(E-V)+I)/C & otherwise \end{cases}$$

$$V \leftarrow \begin{cases} E & if\ start \\ Vreset & if\ V \geq Vspike \\ V & otherwise \end{cases}$$

```
LIF:
	fire: V>=Vspike
	V ′:
		@: (G*(E-V)+I)/C
		@fire: 0
	V:
		@$init: E
		@fire: Vreset
	C: 10pF
	E: -70mV
	G: 3nS
	Vreset: -70mV
	Vspike: -50mV
```

A bare @ translates to “otherwise”. If no condition matches and there is no bare @, the value is not changed.

$init is true during the first cycle of simulation, then false the rest of the time. Provided by simulator.

Special keys, such as $init, start with $ to distinguish them from user-defined variables. ($ stands for “special”.)

Notice that V’ uses I, but I is not defined here. It is provided by the synapse part (post-synaptic current).

Inheritance

Imagine a library with many parts in it.

```
Synapse:
    …
Synapse with STDP:
    …
LIF:
    …
Izhikevich:
    …
Et cetera:
    …
```

And we want to derive a new part.

```
LIF with refractory period:
    $inherit: LIF
    V':
        @$t<refractoryEnd: 0
    refractoryEnd:
        @fire: $t + refractoryPeriod
    refractoryPeriod: 2ms
```

$inherit pulls an attribute tree from the library.

(Next slide shows how attributes get combined.)

$t is current simulation time.

Inheritance

```
LIF:
    fire: V>=Vspike
    V ’:
        @: (G*(E-V)+I)/C
        @fire: 0
    V:
        @start: E
        @fire: Vreset
    C: 10pF
    E: -70mV
    G: 3nS
    Vreset: -70mV
    Vspike: -50mV
```

∪

```
LIF with refractory period:
    $inherit: LIF
    V’:
        @$t<refractoryEnd: 0
    refractoryEnd:
        @fire: $t + refractoryPeriod
    refractoryPeriod: 2ms
```

→

```
LIF with refractory period:
    $inherit: LIF
    fire: V>=Vspike
    V’:
        @: (G*(E-V)+I)/C
        @fire: 0
        @$t<refractoryEnd: 0
    V:
        @start: E
        @fire: Vreset
    refractoryEnd:
        @fire: $t + refractoryPeriod
    C: 10pF
    E: -70mV
    G: 3nS
    Vreset: -70mV
    Vspike: -50mV
    refractoryPeriod: 2ms
```

Inheritance

To use a library part, inherit it and override the parameter(s).

```
Specific Neuron 1:
    $inherit: LIF
    Vspike: -20mV
```

```
Specific Neuron 1:
    $inherit: LIF
    fire: V>=Vspike
    V':
        @: (G*(E-V)+I)/C
        @fire: 0
    V:
        @start: E
        @fire: Vreset
    C: 10pF
    E: -70mV
    G: 3nS
    Vreset: -70mV
    Vspike: -20mV
```

The math of a part is fully specified after inherit.

(Most modeling systems have a separate language for defining part dynamics.)

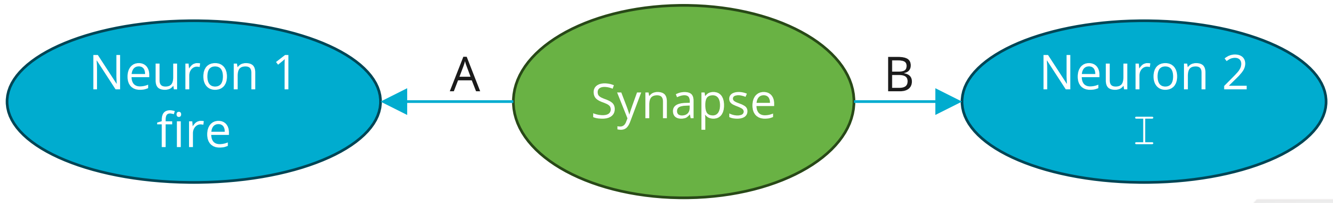
Defining a network

"Simple network" connects two neurons.

A and B are reference variables (like pointers).
They are bound to Neuron 1 and Neuron 2.

```
Simple network:
    Neuron 1:
        $inherit: LIF
    Neuron 2:
        $inherit: LIF
    Connection:
        $inherit: Synapse
        A: Neuron 1
        B: Neuron 2

Synapse:  (in library)
    B.I:+
        @A.fire: weight
    weight: 1
```



Defining a network

```
Simple network:
    Neuron 1:
        $inherit: LIF
    Neuron 2:
        $inherit: LIF
    Connection:
        $inherit: Synapse
        A: Neuron 1
        B: Neuron 2
```

Simple network:

```
Neuron 1:
    $inherit: LIF
    fire: V>=Vspike
    V':
        @: (G*(E-V)+I)/C
        @fire: 0
    V:
        @start: E
        @fire: Vreset
    C: 10pF
    E: -70mV
    G: 3nS
    Vreset: -70mV
    Vspike: -50mV
```

```
Neuron 2:
    $inherit: LIF
    fire: V>=Vspike
    V':
        @: (G*(E-V)+I)/C
        @fire: 0
    V:
        @start: E
        @fire: Vreset
    C: 10pF
    E: -70mV
    G: 3nS
    Vreset: -70mV
    Vspike: -50mV
```

```
Connection:
    $inherit: Synapse
    A: Neuron 1
    B: Neuron 2
    B.I:+
        @A.fire: weight
    weight: 1
```

Defining a network

The Synapse part accesses/modifies values in the neurons via the references.

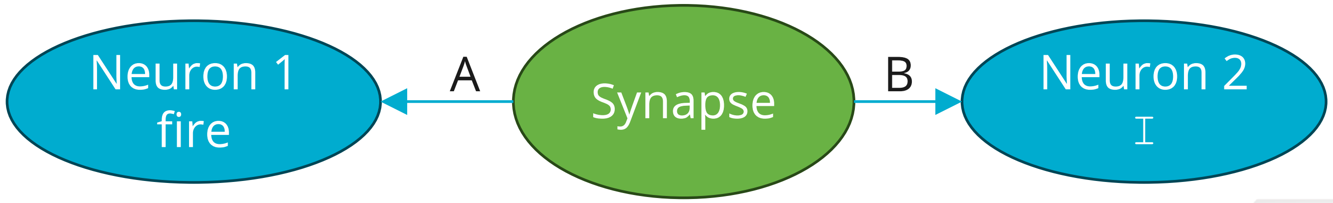
The bare + after B.`I` is a reduction. It says add the value to whatever is already in Neuron 2.`I`

It is called a "reduction" because it can gather values from multiple synapses.

Reduction types include:

    Sum (+)
    Product (* or /)
    Minimum (<)
    Maximum (>)

```
Simple network:
    Neuron 1:
        $inherit: LIF
        fire: V>=Vspike
        …
    Neuron 2:
        $inherit: LIF
        I:+          <----------- created by
                                  assignment
        …
    Connection:
        $inherit: Synapse
        A: Neuron 1
        B: Neuron 2
        B.I:+
            @A.fire: weight
        weight: 1
```



Defining a network

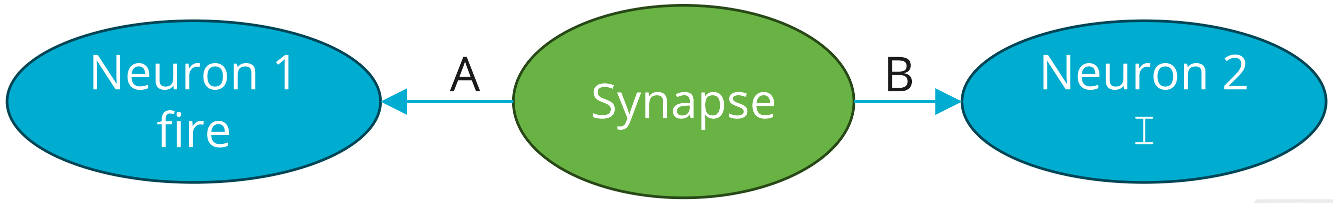
"A=Neuron 1" and "B=Neuron 2" are resolved by first searching in the children of Connection, then in Simple network.

"A.fire" and "B.`I`" are paths.

A is bound to Neuron 1, like a symbolic link, so search for "fire" occurs in Neuron 1.

"`I`" is not found in Neuron 2, but created at compile time because it is the target of an assignment.

```
Simple network:
    Neuron 1:
        $inherit: LIF
        fire: V>=Vspike
        …
    Neuron 2:
        $inherit: LIF
        I:+          <-------------  created by
                                     assignment
        …
    Connection:
        $inherit: Synapse
        A: Neuron 1
        B: Neuron 2
        B.I:+
            @A.fire: weight
        weight: 1
```



Name resolution rules

Path elements are separated by dot.

Paths are resolved left to right.

Let “W” be working location in tree. W moves as resolution proceeds.

```
for P in path elements:
    while W exists:
        search for P in W
        if found
            if P is a reference, move W to referenced node
            else if P is a child, move W to child node
            break
        W ← parent(W)
```

```
Simple network:
    Neuron 1:
        $inherit: LIF
        fire: V>=Vspike
        …
    Neuron 2:
        $inherit: LIF
        I:+

        …
    Connection:
        $inherit: Synapse
        A: Neuron 1
        B: Neuron 2
        B.I:+
            @A.fire: weight
        weight: 1
```

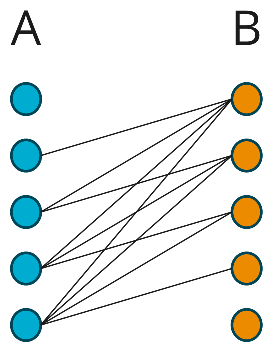
Populations

$n specifies the number of instances in a population. Default is 1 (a single neuron).

$p indicates whether a given pair of neurons should connect. Can be any expression, including functions or weight data from a file. Numbers in (0,1) cause a uniform random draw. Default is 1 (always connect).

$index is position in a given population.

$$\begin{array}{c|ccccc|} & & & B & & \\ \hline & 0 & 0 & 0 & 0 & 0 \\ & 1 & 0 & 0 & 0 & 0 \\ A & 1 & 1 & 0 & 0 & 0 \\ & 1 & 1 & 1 & 0 & 0 \\ & 1 & 1 & 1 & 1 & 0 \\ \hline \end{array}$$



```
Network:
    Population 1:
        $inherit: LIF
        $n: 5
    Population 2:
        $inherit: LIF
        $n: 5
    Connection:
        $inherit: Synapse
        A: Population 1
        B: Population 2
        $p: A.$index > B.$index
```

Inner Populations

```
Network of networks:
    Outer 1:
        $inherit: Network
        $n: 2
    Outer 2:
        $inherit: Network
        $n: 3
```

Network of networks:

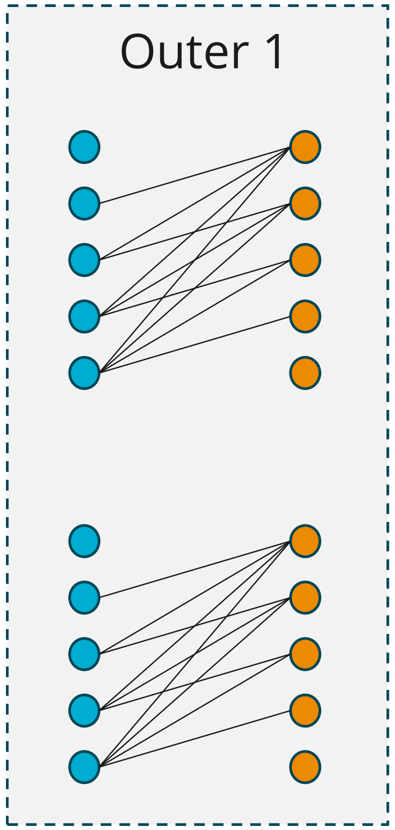
```
Outer 1:
    $inherit: Network
    $n: 2
    Population 1:
        $inherit: LIF
        $n: 5
    Population 2:
        $inherit: LIF
        $n: 5
    Connection:
        $inherit: Synapse
        A: Population 1
        B: Population 2
        $p: A.$index > B.$index
```

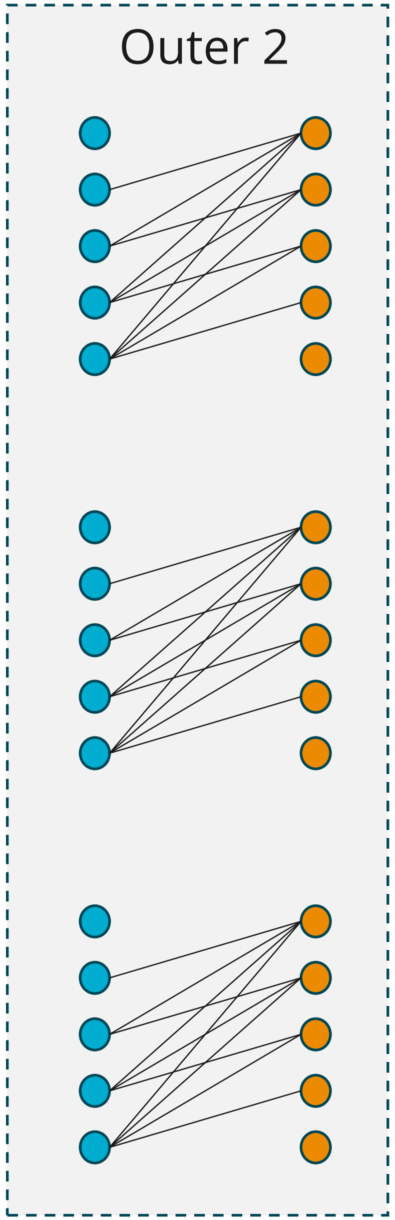
```
Outer 2:
    $inherit: Network
    $n: 3
    Population 1:
        $inherit: LIF
        $n: 5
    Population 2:
        $inherit: LIF
        $n: 5
    Connection:
        $inherit: Synapse
        A: Population 1
        B: Population 2
        $p: A.$index > B.$index
```

Inner Populations

Network of networks:

```
Outer 1:
    $inherit: Network
    $n: 2
Outer 2:
    $inherit: Network
    $n: 3
```

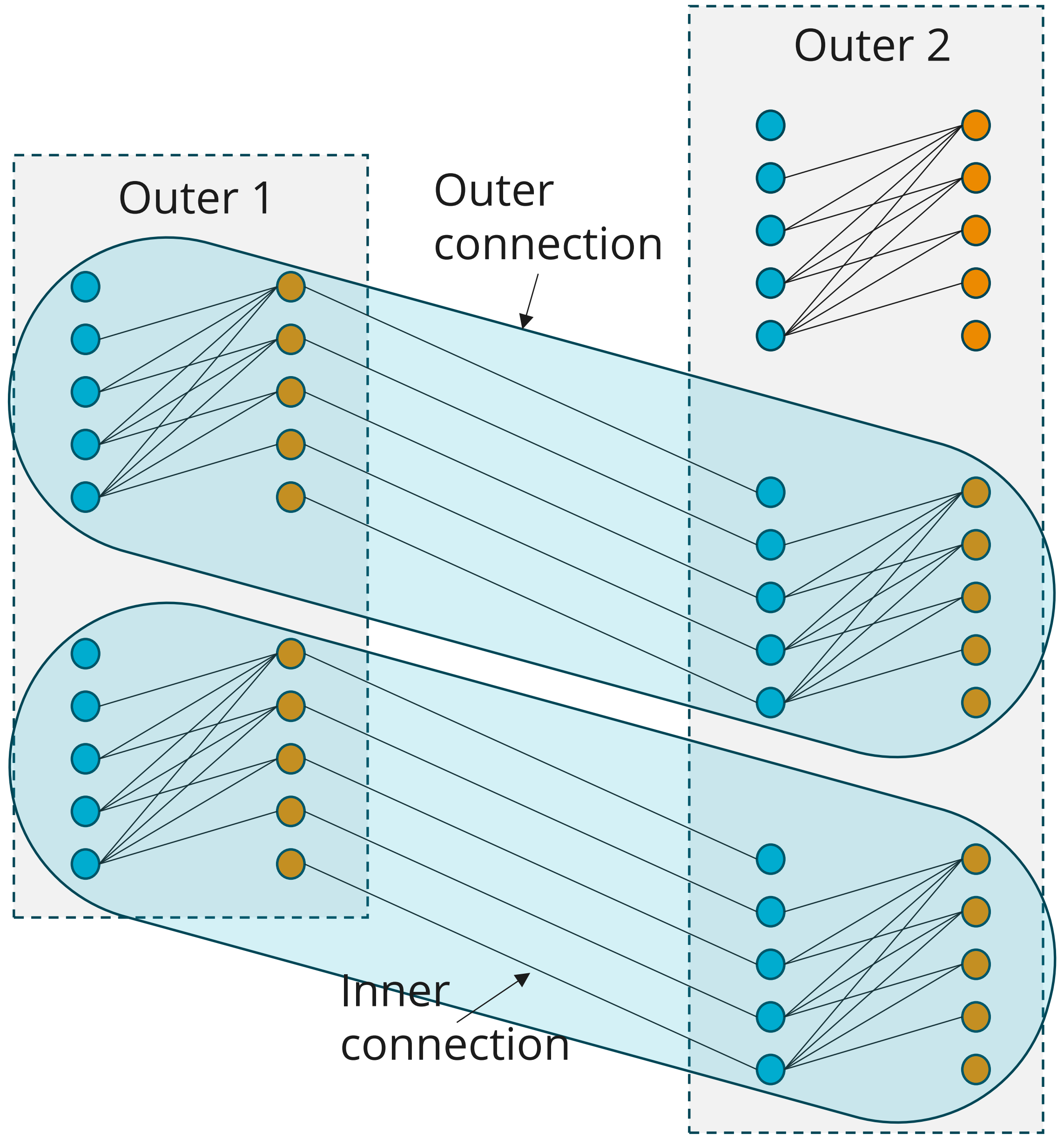




Inner Connections

```
Network of networks:
    Outer 1:
        $inherit: Network
        $n: 2
    Outer 2:
        $inherit: Network
        $n: 3
    Outer Connection:
        A: Outer 1
        B: Outer 2
        $p: A.$index+1 == B.$index
        Inner Connection:
            $inherit: Synapse
            A: $up.A.Poulation 2
            B: $up.B.Poulation 1
            $p: A.$index == B.$index
```

$up is a path element equivalent to Posix ".."
Necessary so "A" does not resolve to itself.



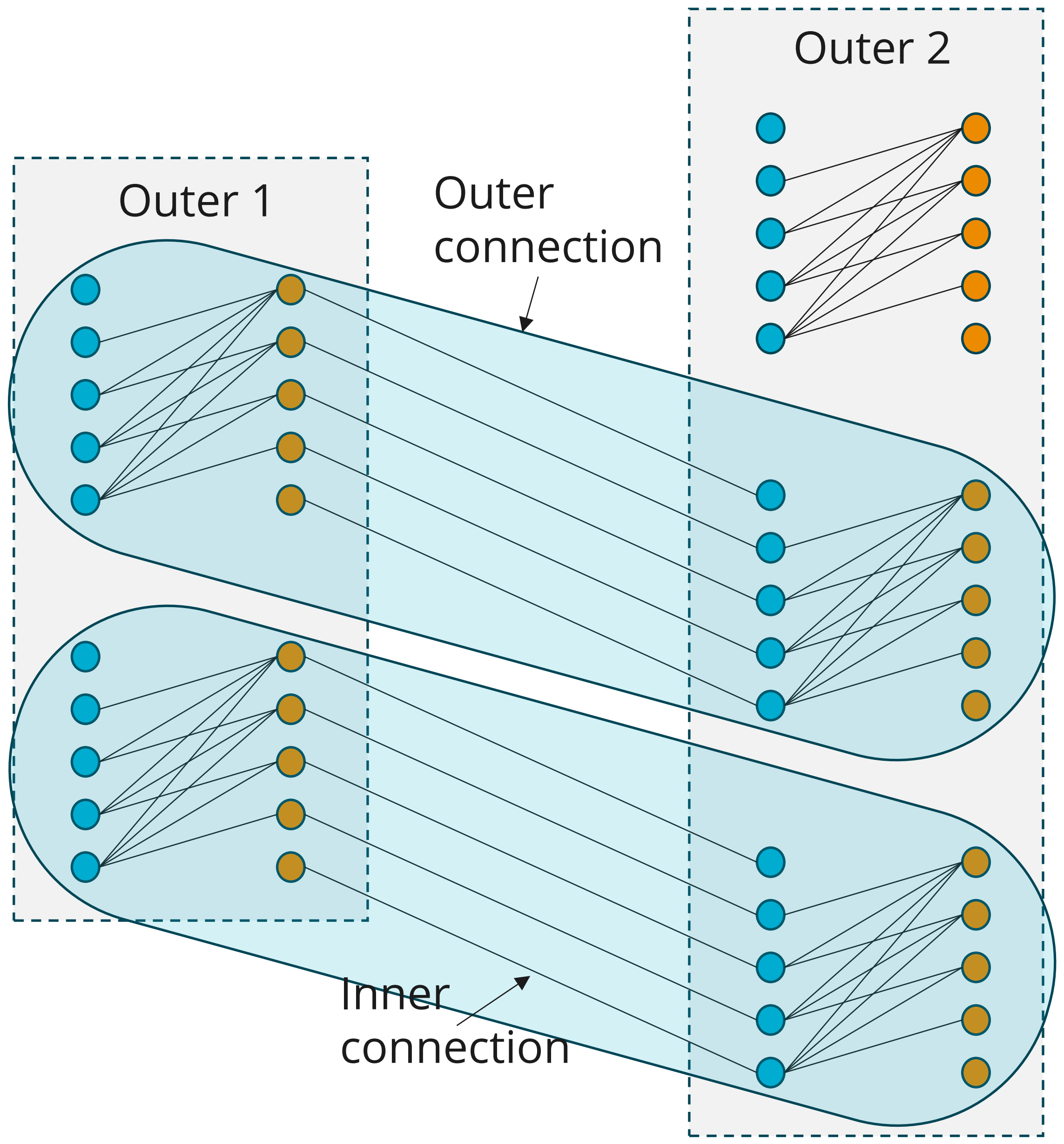
Inner Connections

Outer connections are formed first. Then for each outer connection that actually exists, the inner connection is evaluated.

An outer connection can also contain inner populations (not illustrated here).

There is no limit to depth.

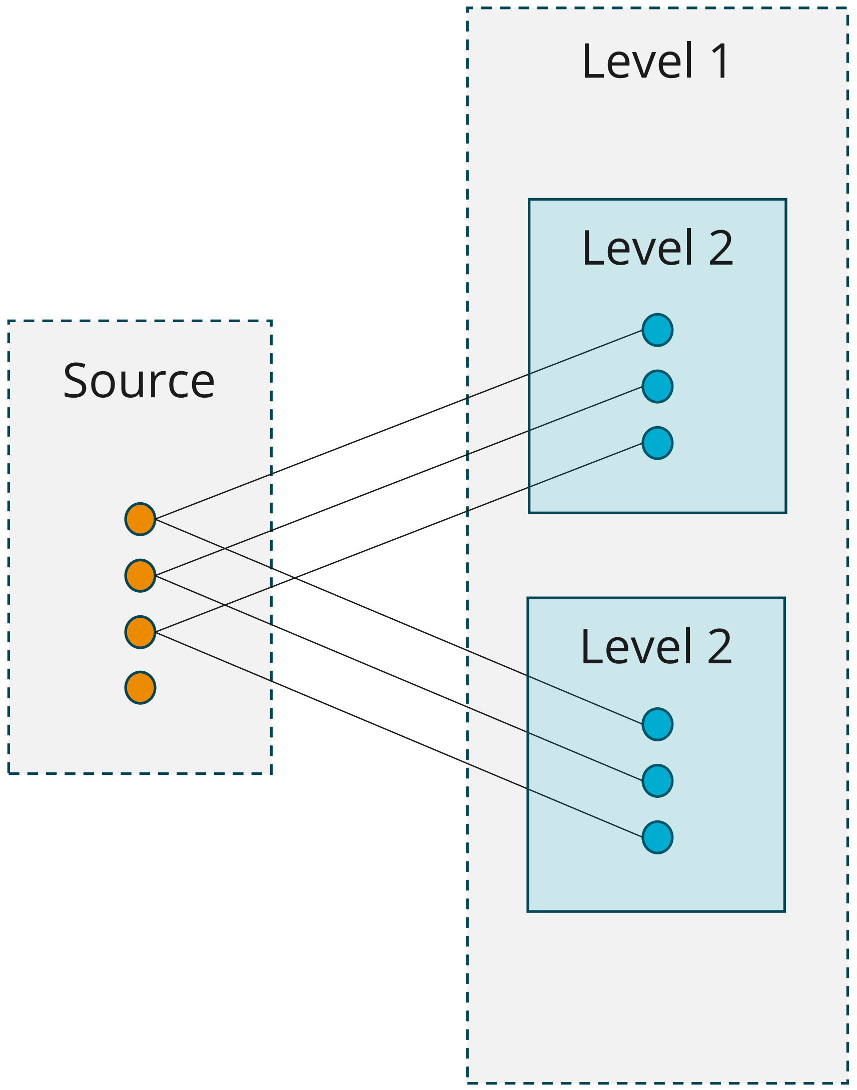
→ **Recursive composition**



Deep connections

A connection iterates over all referenced objects in deep populations.

```
Network iterated:
    Level 1:
        $n: 2
        Level 2:
            $inherit: LIF
            $n: 3
    Source:
        $inherit: LIF
        $n: 4
    Connection:
        A: Source
        B: Level 1 . Level 2
        $p: A.$index == B.$index
```



Deep overrides

Only need to specify tree nodes along path to the overridden attribute.

$inherit is evaluated recursively – First the top-level $inherit is evaluated. Then $inherit in each child node.

This allows $inherit in deeper children to be overridden by the primary tree, with well-defined results.

```
My derived network:
    Network 1:
        $inherit: Network of networks
        Outer 1:
            Population 1:
                $inherit: Izhikevich    ← Change the
                                          neuron model
        Outer 2:
            Population 2:
                Vspike: -20mV
    Network 2:
        $inherit: Another network
```

Conclusions

Modeling framework features

- **User-defined dynamics** – Parts are always fully specified as a set of equations. No separate language for part definition.
- **Shareable** – Library parts can be stored in YAML files and exchanged over Git.
- **Recursive composition** – Inner populations, inner connections, deep override
- **Concise** – Individual parts only need to specify the tree nodes they actually modify.
- **Readable** – Attribute trees can be opaque until you learn the semantics.

Whole-brain modeling – Integrates many different models into a single large model. Features like deep override and deep connections allow incompatible models to be adapted.

The N2A tool

- Visualizes the tree, identifying local versus inherited attributes.
- Compiles models for specific simulators and hardware.
- Manages the library of parts.
- Connects to Git repositories.

Multiple inheritance

$inherit can list more than library part.

Precedence is left to right. The first listed part dominates the others.

```
Base:
    Ann: 1
    Dorothy: 4

One:
    $inherit: Base
    Bob: 2
    Dorothy: 1

Two:
    $inherit: Base
    Cecilia: 3
    Dorothy: 2
```

```
Diamond:
    $inherit: Two, One

Diamond 2:
    $inherit: One, Two
```

→

```
Diamond:
    $inherit: Two, One
    Ann: 1
    Bob: 2
    Cecilia: 3
    Dorothy: 2

Diamond 2:
    $inherit: One, Two
    Ann: 1
    Bob: 2
    Cecilia: 3
    Dorothy: 1
```

Pruning

So far, inherited tree structure has all been additive.
Sometimes it is useful to remove a tree node.

$kill removes an inherited tree node from the final model.

“$kill: 0” restores the node in a descendant.

Node with <undefined> allows parent value to be visible.

```
Parent:
    x: 1
    y: 2
    z: 3
```

```
Child:
    $inherit: parent
    y: <undefined>
        $kill

Grandchild:
    $inherit: child
    y: <undefined>
        $kill=0
```

$\Rightarrow$

```
Child:
    $inherit: parent
    x: 1
    z: 3

Grandchild:
    $inherit: child
    x: 1
    y: 2
    z: 3
```

Evaluation order

Two kinds of variables:

- Temporary – Value is used only during current simulation step.
- State – Value is remembered between steps.

Within a given equation set

- Temporaries are evaluated before they are used (dependency order).
- State variables separate current value from next value
  - Current (read) value does not change during step.
  - Written value becomes visible at next step.
  - Exception: During init (step 0), assignments are immediately visible to maximize information spread.

Rules for determining state or temporary

- Variables are temporary by default.
- Some variables are always state: derivatives, integrated values, referenced values
  - Because they are read/written asynchronously.
- Compiler breaks cycles by promoting some variable(s) to state.
- User may force state by marking a variable with “:”

Evaluation order

State variables are effectively orderless.

Cyclic Dependency 1:

```
a: :
    @: c+1
    @$init: 1
b: :
    @: a+1
    @$init: 1
c: :
    @: b+1
    @$init: 1
```

User marks variables as state.

No cycles.

$t	**a**	**b**	**c**
0	1	1	1
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4

Before init, all variables are 0.

Explicit initial value.

Previous values are visible.
All are assigned 2.
2 becomes visible in next cycle

Evaluation order

Nothing is marked as state.

{a, b, c} form a cycle

Compiler chooses c to break cycle (completely arbitrary).

Cyclic Dependency 2:

```
a:
    @: c+1
    @$init: 1
b:
    @: a+1
    @$init: 1
c:
    @: b+1
    @$init: 1
```

$t	a	b	c
0	1	1	1
1	2	3	1
2	5	6	4
3	8	9	7

Without init.

Cyclic Dependency 3:

```
a: c+1
b: a+1
c: b+1
```

Temporaries add to visible value of c

$t	a	b	c
0	1	2	3
1	4	5	3
2	7	8	6
3	10	11	9

Before init, all variables are 0.

Sum propagates through temporaries, ending at c.

Previous value of c (3) is visible. Assigned 6.

6 becomes visible in next cycle