# The Neosim interface [ draft ]

Fred Howell, Greg Hood, Nigel Goddard

August 1999

## 1 Background and aims

Neosim is a framework to support large scale multi-level modelling of the nervous system. It:

- allows plug-in of different simulation components for simulation, visualisation and I/O.
- is specified in a way which allows running on workstations as well as networks of workstations and parallel machines without having to write a parallel program.
- is based on discrete event simulation.

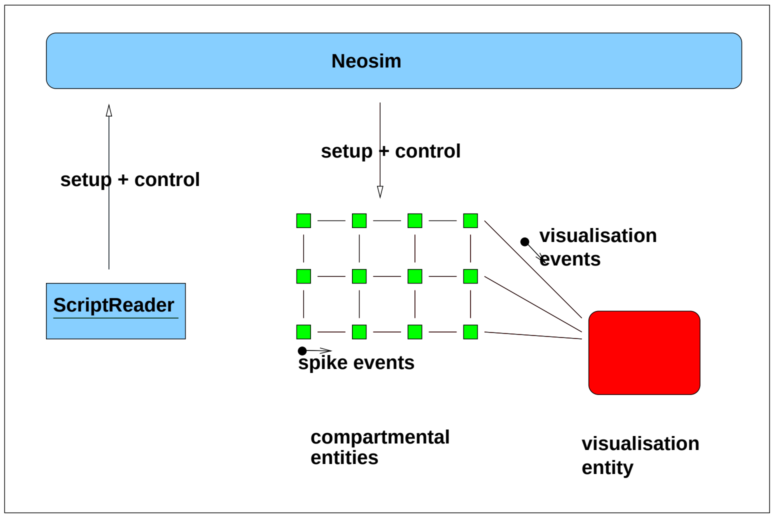


Figure 1: Overview.

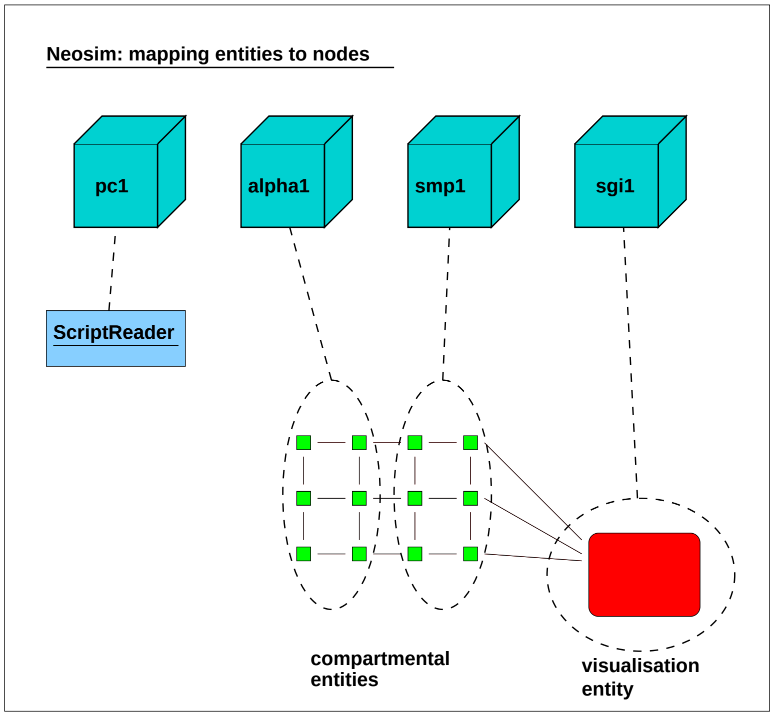


Figure 2: Mapping of entities to processors.

## 1.1 Why plug-in components?

Many independent simulation tools address particular modelling aspects well, such as Neuron and Genesis for compartmental models, MCell for Monte-Carlo diffusion, and various neural network simulators. However, these tools don't interact with one another, so models built with one tool are not accessible to the others. Modellers often resort to writing their own simulations from scratch, when it is possible they could take advantage of some facilities in Neuron or Genesis.

Thus the Neosim kernel will provide mechanisms for coordinating a number of plug-in simulation and visualisation modules.

## 1.2 Why parallel?

The computing power required to model brain subsystems at any level of detail is enormous. Compartmental models of individual neurons, which involve solving sets of differential equations for each compartment make very heavy use of processor time. Network models of even a small region of brain must include large numbers of synapses which make very heavy demands on memory usage. And cell models are likely to become more detailed and biologically realistic as more experimental details of their internal dynamics are discovered.

This suggests the need for parallel or distributed computers to allow larger or more detailed models to be investigated than can fit on a single workstation. This was the motivation for the development of PGENESIS, the parallel version of Genesis, and large models have been run on supercomputers using this tool. The problem with hand coding a parallel simulation using C++ or PGENESIS is the amount of development and debugging time needed to get a parallel program running efficiently, time which would be better spent on modelling issues.

An aim of Neosim is to allow small models developed on workstations to be scaled up and run on networks of workstations and supercomputers without having to write explicitly parallel code.

## 1.3 Why discrete event simulation?

Neurons typically communicate by sending spikes down axons which arrive at their destination some milliseconds later. This fundamental interaction can be modelled as an event which is sent by the source neuron to its destinations with a delay. For compartmental models this delay is usually larger than the integration timestep. This delay allows for a relaxation of the synchronisation constraint; the simulation time of each model neuron does not have to be kept in lockstep, but can lag by up to its output delay. This allows for an amount of latency hiding and performance optimisation of a parallel implementation of the simulation, since if the neurons are on separate processors they do not need to synchronize every timestamp.

Variable timestep integration techniques (such as are used by CVODE in Neuron) provide order of magnitude performance improvements, as when the state variables are changing slowly, larger timesteps can be used, which means less computation. The variable timestep techniques offer particular opportunities for performance gains in large networks where the spiking rate of individual neurons is low (e.g. the granule cell layer of the cerebellar cortex). A fixed timestep technique is wasteful for this type of network, as most of the cells are inactive most of the time, yet they are updated using a timestep determined by their fastest rate of change.

An efficient algorithm for coordinating neuron updates with variable timestep uses an event list sorted in timestamp order; the next neuron to update is pulled off the bottom (this is the technque currently used in Neuron/CVode).

If each neuron has an independent timeline, then a well developed technique to coordinate a simulation is discrete event simulation. Techniques for running discrete event simulations in parallel have been published; to run efficiently on parallel machines they require a degree of *lookahead* or minimum delay for one entity to influence another. Spiking neurons typically provide this lookahead because of the axonal delay.

Tightly coupled systems are not well suited to the discrete event paradigm. The electrical/chemical coupling between compartments of a single neuron is better dealt with within an entity representing that neuron than by using events (note that GENESIS without hsolve used an event-style message mechanism for communicating these properties, hsolve allowed considerable performance improvements).

## 2 The Interface

The neosim interface is a collection of classes specified in Java (but with a C++ binding). The basic classes do very little in themselves, they must be extended in order to provide the interesting behaviours. Only experienced developers are expected to deal directly with the low level neosim classes; modellers will be provided with extended versions of these classes which provide a script language interface.

The most important classes of the interface are:

- **ScriptReader.** The main() for the simulation, a hook for the user to provide a sequential control program.
- **Entity.** This is the superclass for all simulation objects which change with time. An entity can send and receive events via ports to other entities.
- **Event.** The superclass for all timestamped objects which get sent between entities. An event can be something simple like a single spike, or a more complex object including a large data structure and methods.
- **Population.** A class for referring to an entire set of entities at once, and for providing rapid lookup of member entities.

- **Projection.** A class for specifying a set of connections between two populations of entities.

They are described in detail below.

# 3 The Script Reader

The *main()* for a simulation is provided by the ScriptReader interface to Neosim. This is a sequential control program which issues commands to the kernel to make entities, connect them, run and reset the simulation. The intention is for the simulation control to look like a sequential program even though the simulation may be running on a network of workstations or a parallel machine. It is also possible to control a simulation from within an entity, so another mode of operation would be for the script reader to be used as a simple bootstrap to create a "control entity" which manages the simulation.

# 4 Entities and Events

The actual work of the simulation is done by *entities* which interact with each other by sending timestamped events. Incoming events are processed by an entity's *event handler*, which can update internal state and possibly generate more events.

Figure 3 illustrates the basic classes. An entity can send events using its output ports. Events are transmitted via connections which are made to the input ports of a number of other entities. Connections have a greater than zero delay associated with them, so events arrive some amount of simulated time later.

## 4.1 Types of entities

An entity can be any object derived from class Entity. Entities are used for simulation components, such as compartmental or integrate-fire neuron models, as well as I/O entities for displaying and storing results, and interacting with the user.

## 4.2 Sending events

An entity can send an event to one of its output ports, and it will arrive some time later at all of its destinations. Any object derived from the Event class can be sent. A common type of event is a SpikeEvent, but other event types (e.g. DoubleEvent, MorphologyEvent) can also be derived and sent between entities. The event is constructed with a timestamp at the *sending time* not the arrival time. This is because the delay may be different for each destination.

```
SpikeEvent se = new SpikeEvent( sending_time );
```

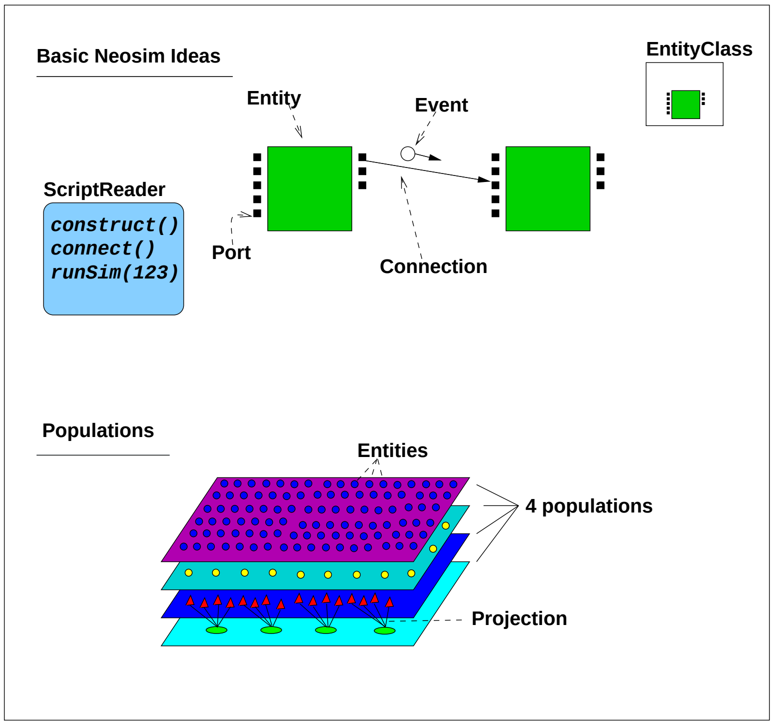


Figure 3: Terminology.

```
PortID spikegenID = 1;
sendEvent( spikegenID, se );
```

For complex events, the mechanism is the same:

```
MorphologyEvent = new MorphologyEvent( t, getMorphology() );
PortID  morphgenID = 2;
sendEvent( morphgenID, se )
```

As well as sending events via an output port to its destination entities, it is also possible to send an event directly to a given input port of an entity using:

```
SpikeEvent se = new SpikeEvent( arrival_time );
EntityID destEnt  = // look up destination entity;
PortID   destPort = // input port for event to arrive on;
sendEvent( destEnt, destPort, se )
```

### 4.3 Receiving events and advancing time

An entity receives events and advances its simulation time by providing an implementation of the `handleEvents()` method.

```
Time  handleEvents(EventList el, Time t)
```

The incoming events are passed in the sorted event list, and the method should should *advance the entity's time to at least t* and return the time actually advanced to. This may be later than t, as long as the state can be rewound back to t and no new events are generated after t.

The kernel will also call this method to advance the time of an entity even when there are now incoming events to process.

### 4.4 Minimum output delay

The entity's minimum output delay is available using `getMinOutputDelay()`; this will be the minimum delay on any of its output ports, or if there are no output connections, a positive value determined by the kernel. This output delay is imposed on every kernel request the entity makes (i.e. responses to queries, requests to construct and link other entities). If this is not acceptable to the model, the output delay can be zeroed using `zeroOutputDelay()`, and then unzeroed using `resetOutputDelay()`. For example, to perform a sequence of queries/responses/new queries at a specific instant of time while a simulation is running, zero the output delay of the population concerned, perform the updates/queries, then reset the output delay. See section 10 for more details of timing issues.

## 4.5 Querying an entity's class

Where a large number of entities share a common structure, it makes sense to store the class specific information once, and let the entities just hold the variables (such as compartment voltages).

An entity's own class can be obtained using:

```
EntityClass ec = getClass();
```

Or any named entity's class is available using a class lookup:

```
EntityClass ec = lookupEntityClass("classname");
```

Particular classes of entities can then provide more information, e.g. an entity class which is specified by a Genesis P-file:-

```
class GenesisPfileClass extends EntityClass {
  GenesisPfileClass(String pfilename);
  Entity makeEntity( Population p, int i ) { // read in entity from pfile }
  String getPFilename();
}
```

This can be accessed using:-

```
EntityClass ec = lookupEntityClass("GenesisPfileClass");
GenesisPfileClass gpf = (GenesisPfileClass)ec;
System.out.println(".p file name is "+ gpf.getPFilename());
```

Additional methods could be added to store class-specific data structures, initialisation values, etc.

## 4.6 Access to the detailed internal structure of an entity

The basic `Entity` class includes minimal state, but user types of entity may include complex state information. The query interface allows this information to be accessed from another entity. For example: what is the structure of an entity, then what's the calcium concentration of a particular dendrite?

```
--- ( in script reader ) ---
  class GetStructure extends EntityQuery {
    Object query(NeosimEntity e) {
       return ((CompartmentalEntity)e).getStructure();
    }
  }
  class GetCaConc extends EntityQuery {
     GetCaConc( int CompartmentID );
     Object query(NeosimEntity e) {
       return ((CompartmentalEntity)e).getCaConc( CompartmentID );
     }
  }
```

```
// e = lookup entity to query
Structure struct = (Structure)queryEntity( e, new GetStructure() );
int compartmentID = struct.lookupCompartment(...);
GetCaConc getca = new GetCaConc( compartmentID );
Object caconc = queryEntity( e, getca );
```

Section 6 has more details on this approach.

## 4.7 Adding probes to entities

A probe is used to monitor some internal property of an entity, such as a voltage or ion concentration. The technique is to create an extra output port for each probe, with an event generator to convert the internal state into a stream of events.

```
--- ( in script reader ) ---
   class AddProbe extends EntityQuery { // a query, as it returns PortID
     AddProbe( CompartmentID cid, EventClass ec );
     Object query(NeosimEntity e) {
       return e.addOutputPort( ec );
     }
   }
   // e = lookup entity to query
   // cid = lookup compartment to probe
   EventClass ec = lookupEventClass("doubleEvent");
   PortID probePort = queryEntity( e, new AddProbe(cid,ec) );

   // now connect probe port to port 1 of a monitor entity
   connect( new SingleConnection(e, probePort, monitor, 1, 10.0) );
```

Another way to query the state of an entity on a regular basis would be to create another entity which sends repeated queries to the entity. The advantage of this is that there is no need to make an extra connection, and no need for the probed entity to be capable of generating the extra events; the disadvantage is that this technique is slightly less efficient.

## 4.8 Querying an event class

Events arrive at an entity in a call to the entity's `Time handleEvents(EventList el, Time t)` method. The event list is a vector of events in increasing time order, and may include events later than the time t to advance to. Example code to traverse this list is:-

```
Time handleEvents(EventList el, Time t) {
  for (int i=0; i < el.size(); i++) {
    Event ev = el.elementAt(i);
```

```
      Time evtime     = ev.getTime();
      ClassID evclass = ev.getClassID();
      Connection  c   = ev.getConnection();
    }
  }
```

The basic event includes a timestamp, a class identifier and the connection it arrived on. The connection includes which input port, source entity handle and output port, and also a connection id which is unique for this entity.

```
Connection c       = ev.getConnection();
EntityID deste     = c.getDstEntityID();
PortID destport    = c.getDstPortID();
ConnectionID destc = c.getConnectionID();
EntityID srce      = c.getSrcEntityID();
PortID srcport     = c.getSrcPortID();
Time delay         = c.getDelay();
```

Some events (e.g. spikes) are binary affairs - it is just necessary to know the time that an event occurred. The code to handle one could look like:

```
Event ev = el.elementAt(i);
PortID p = ev.getConnection().getDstPortID();
Time evtime = ev.getTime();
// deal with an event on p at evtime
```

Other events may include complex information, e.g. a Morphology. The integer class identifier of the event is available by calling the getClassID() method, e.g. an entity which receives morphologies and spikes:

```
ClassID morphologyEventID = NeosimClasses.getClassFromName("MorphologyEvent");
ClassID spikeEventID = NeosimClasses.getClassFromName("SpikeEvent");
ClassID evclass = ev.getClassID();
switch (evclass) {
   case morphologyEventID : // got a morphology event
   case spikeEventID : // got a spike event
   // ...
}
```

## 4.9 Extracting data with derived events

The data in derived events is available by using a dynamic cast:

```
Event ev = el.elementAt(i);
ClassID evclass = ev.getClassID();
if (evclass == morphologyEventID) {
  MorphologyEvent mev = (MorphologyEvent) ev;
  // can now use methods of morphology event, e.g. mev.getStructure()
}
```

## 5 Populations

Large scale network models often include *populations* of similar entities. Neosim allows models to be defined at this high level, in terms of populations of cells and projections between populations. Having a high level model description also simplifies efficient implementation on parallel machines.

Example populations could be a layer of one type of neuron in a cortex, or a nucleus of similar cells. “Views” of populations can also be created, and treated as new populations; an example could select all cells over a certain size, or in a restricted region. Populations are treated as a special type of *shared entity*, which allows rapid lookup of elements of the population. Another example population could be a “GeometryView” which provides rapid lookup of all entities in a given region of space, an essential optimisation for setting up distance based connections in a large model.

In order to define a population, a modeller provides a name, an *entity class*, the number of elements, and a function to initialise each entity of the population.

Every entity is member of a *Population*, and there is a hierarchical naming structure for populations. There are methods to lookup an entity or population given a fully qualified name (e.g. "Cortex/layer3/pyramidal12345"):

```
EntityHandle lookupEntity( Time t, String name )
Population   lookupPopulation( Time t, String name )
```

A *time* argument is specified, as populations can change over time, with entities added and deleted. For the script reader, this argument will be the simulation time; for entities it will be the entity's local simulation time.

The population tree structure is available:

```
PopulationTree getPopulationTree( Time t )
```

It has methods for adding and removing population nodes from the tree, and for navigating the branches.

A basic Population has a few methods:

```
String         getName()
PopulationNode getParent()
int            getId()
int            getNumEnts()
int            getIndex( EntityID eid )
boolean        isMember( Entity e )
Vector         getMemberList()      // Vector of all EntityIDs
void           construct()          // Build the population.
```

A *PopulationBuilder* extends the basic Population to include methods to instantiate a number of entities of a given class:

```
PopulationBuilder( EntityClass, int num, EntityInit ei )
```

EntityInit is a function to initialise entity i of the population.

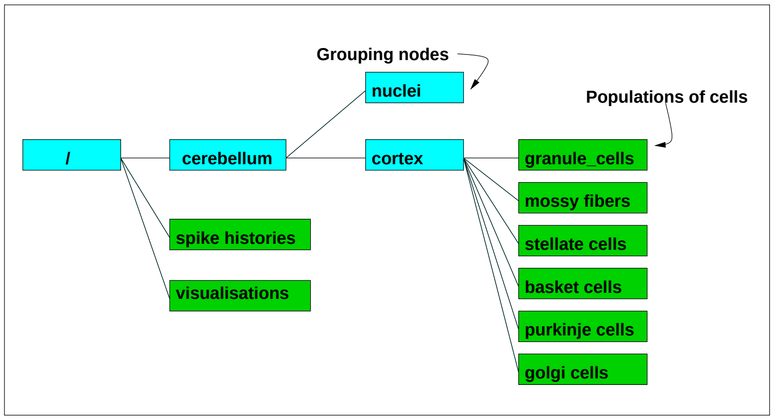


Figure 4: Example population tree.

```
interface EntityInit { void init( Entity e, Population p, int index); }
```

This can contain entity method calls to set the position, initial potentials, etc etc depending on the index in the population. Examples could include:

```
class InitMesh2D implements EntityInit {
  InitMesh2D( xsz, ysz, x1, y1, xspc, yspc );
  void init( Entity e, Population p, int index ) {
    e.setPos( index % xsz, index \ xsz );
  }
}
```

*Views* of a population can be constructed from an existing population:

```
PopulationView( Population srcpop )
final void addEntityToView( Entity e );
void buildView( Entity e )
```

by overriding the buildView() method which is called for each entity in the source population.

An example population view is a selection based on pattern matching of the name:

```
class RegexpMatch extends PopulationView {
  RegexpMatch( Population p, String regexp );
  void buildView( Entity e ) {
    if (regexp_match( regexp, e.getName() ) {
      addEntityToView( e );
    }
  }
}
```

Population views can be used to provide rapid lookup of entities, e.g. a GridDecompositionView which puts entities into grid boxes. This can speed up specification of distance-based connection patterns as connection requests can be sent to nearby entities rather than all entities.

```
class GridDecompositionView extends PopulationView {
  GridDecompositionView(double gridxsz, double gridysz, double zsz);
  Vector getMemberList( int gridx, int gridy, int gridz );
  void addEntityToView( Entity e, int x, int y, int z ) //...
  void buildView( Entity e ) {
    Pos p = e.getPos();
    int gridx = (int) p.getX() / gridxsz;
    int gridy = (int) p.getY() / gridysz;
    int gridz = (int) p.getZ() / gridzsz;
    addEntityToView( e, gridx, gridy, gridz );
  }
}
```

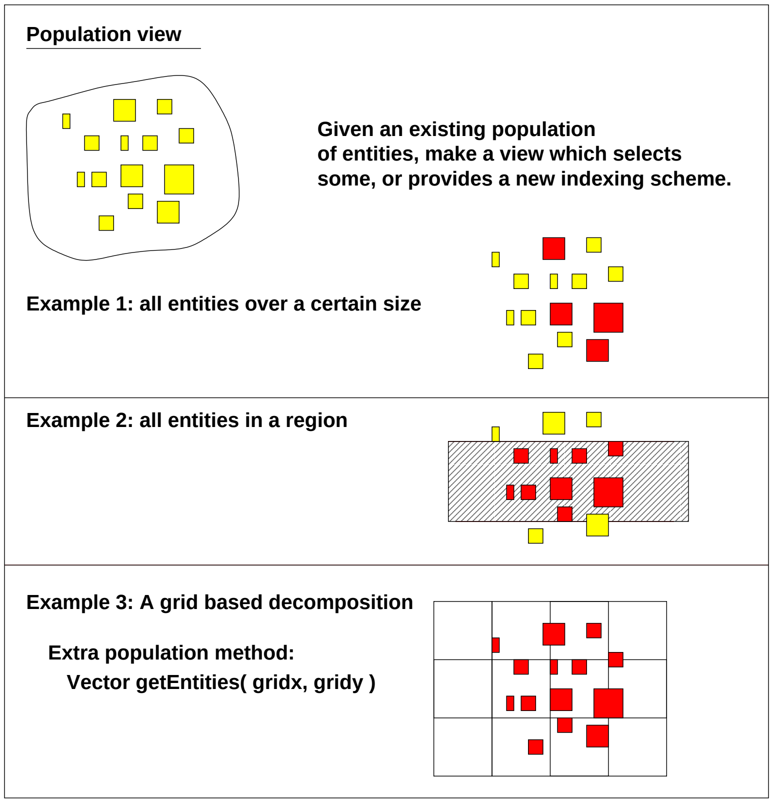


Figure 5: Population view.

### 5.1 An entity can construct a population

An entity can create a population of entities at a given simulation time.

```
EntityClass ec = lookupEntityClass( "entclass" );
Population p = new PopulationBuilder( "newentities", ec, 1, null );
construct( getLocalTime() + getMinOutputDelay(), p, NullCallback );
```

The difference between constructing populations from an entity and from the script reader, is that the *time* and *callback* must be specified. The time is the local time;. The callback is called when the population has been constructed.

# 6 Interacting with entities - the query/response technique

Entities can be physically distributed on different computers, and their simulation times may also be staggered. Because of this, it is not possible for an entity to directly access the internal state of another entity; instead entities refer to other entities using an opaque *EntityID* (an integer which uniquely identifies another entity for the course of a simulation). Entities can interact with other entities by sending timestamped *events* containing data and possibly methods.

Low level event based programming can become complex, so support is built in for the common operations, such as updating and querying the state of other entities. This comes in the form of two interfaces; the EntityUpdate and the EntityQuery interface. These let the user provide methods `update()` and `query()` which are called by the remote entity when the event arrives, and have access to the entity internals.

```
interface EntityUpdate { void  update( Entity e ); }
interface EntityQuery  { Object query( Entity e ); }
```

The simplest way to interact with entities uses the ScriptReader interface, e.g.:

```
EntityID e = lookupEntity("layer1/granule123");
class GetPos implements EntityQuery {
  Object query(Entity e) { return e.getPos(); }
}
Object pos = queryEntity( e, GetPos );
```

The queryEntity() call blocks until the results return. The underlying implementation works as follows: The script reader despatches an event containing the GetPos object to the entity being queried. This event is handled by the entity's default event handler, which sends a response event back to the script reader. This response event wakes up the script reader thread, returning the result of the query to the queryEntity call.

Updates can also be performed with a similar interface:

```
class SetPos extends EntityUpdate {
  SetPos(Pos p) {this.p=p;}
  void update(Entity e) { e.setPos(p); }
}
updateEntity( e, new SetPos( new Pos(1,2,3) ) );
```

The event handler of an entity is not permitted to block, so if an entity needs to query another entity it has to specify a callback:

```
interface Callback { void callback( Entity e, Object o ); }
```

The result of the query is passed in the `Object o` argument for the callback() function, which has access to the internals of the entity. An example which queries another entity's position and prints it is:

```
class PrintIt implements Callback {
   void callback( Entity e, Object o ) {
      System.out.println( e.getName() +" got back "+(Pos)o);
   }
}
e = // handle of entity to query
t = // >= local time + min output delay
queryEntity( t, e, GetPos, PrintIt ) );
```

## 6.1 Querying and updating populations of entities at once

Although it is possible for the script to query a population of entities one by one, waiting for the results of the query to come back each time, it is more efficient to despatch the query to all entities of a population at once and return a vector of answers.

This works in much the same way as querying individual entities.

```
Population p = lookupPopulation("layer1");
class GetPos extends EntityQuery {
  Object query(Entity e) { return e.getPos(); }
}
Vector pos = queryPopulation( p, GetPos );
```

The vector which is returned is indexed by the entity's index in the population.

Updates of all entities in a population can also be done:

```
Population p = lookupPopulation("layer1");
updatePopulation( p, new SetPos(1,2,3) );
```

The example above sets the position of all entities in the population to (1,2,3), which isn't very useful. The update() method needs access to the population and index concerned to allow code like:

```
class SetPos2Dmesh extends EntityUpdate {
  SetPos2Dmesh( Population p, int xsz, ysz, double x1, y1, xinc, yinc );
  void update(Entity e) {
    // get index in this population
    int index = p.getIndex(e.getID());
    e.setPos(x1 + xinc * (index % xsz), y1 + yinc * (index \ ysz), 0.0);
  }
}
updatePopulation( p, new SetPos2Dmesh(p, 10, 10, 0.0, 0.0, 2.0, 2.0) );
```

#### 6.1.1 Querying/updating a population from an entity's event handler

An entity sends a query to a number of other entities, and wants to change its state depending on the results. This needs a mechanism to gather a number of responses to a query and to fire off a callback when all the results come in.

e.g.

```
// in an entity's handleEvents() method:
Time t = // local time + min output delay
Population p = // population to query.
EntityQuery q = // query to run
Callback c = // function to run when the results come in
queryPopulation( t, p, q, c );
```

## 7 Projections

A *projection* is a set of connections from the output ports of entities of a source population to the input ports of entities of a destination population. Figure 6 illustrates the basic idea.

The canonical way to specify a set of connections is to provide a set of tuples: (source entity, source port, dest entity, dest port, delay). But the set of connections to make will often depend on the relative distance between entities, or other parts of their state.

The general way to define a projection is to provide a *source method* to say which entities of the destination population each member of the source population may want to connect to, and a *destination method* to say which of the connection requests should be fulfilled at the far end. Figure 7 provides two examples of this style of connection. This technique can be used to provide GENESIS's `volumeconnect` style distance/probability based connections between populations of entities. If the source and destination methods can be written in the script language, more general connection patterns than volumeconnect are possible.

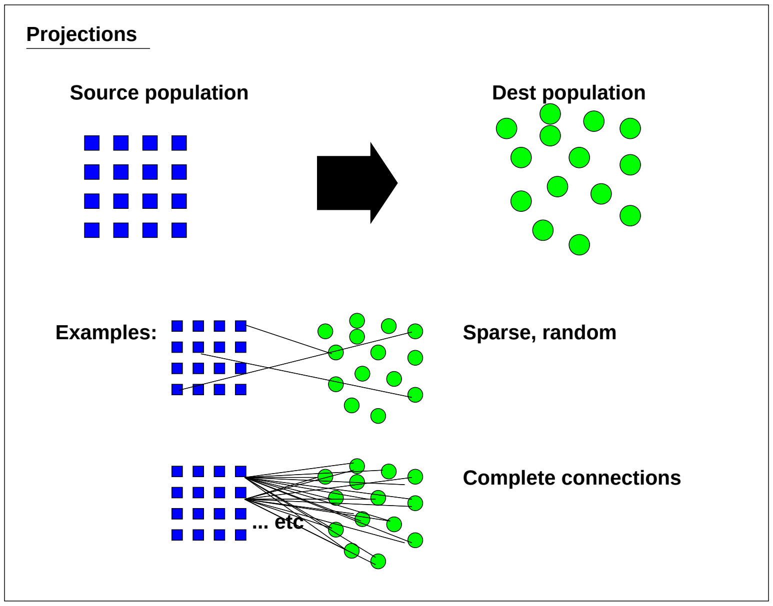


Figure 6: Projections.

### 7.1 Connections inside an entity

An entity may need to store extra information about its connections in addition to the raw "connectionID" and delay which neosim stores. It can store this extra information in a table indexed by connectionID.

### 7.2 NEURON style connections

The Neuron `netcon` object defines a synaptic connection between a source membrane potential/PointProcess and a target PointProcess with a NET_RECEIVE procedure. It includes a source threshold, weight and delay. So how does this fit in with Neosim connections?

The default Neosim connection just stores src/dest entity and port, along with a delay and a connectionID. Additional information needed by the Netcon object should be stored in the Neuron entity, including the threshold at the source, and the weight at the destination end.

The next question is how do these netcon objects get constructed? The mechanism in OC is:

```
section netcon = new NetCon(source, target, threshold, delay, weight);
```

The source and targets are PointProcesses within different entities. The implementation of this in Neosim becomes:

```
class NetConSrc extends SourceMethod {
```

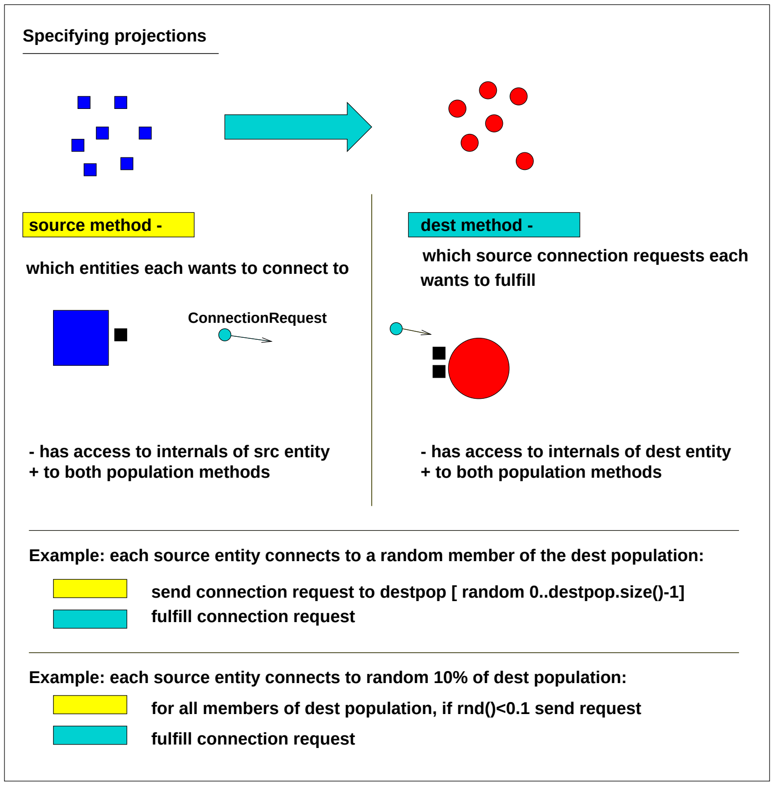


Figure 7: Specifying projections in a general way.

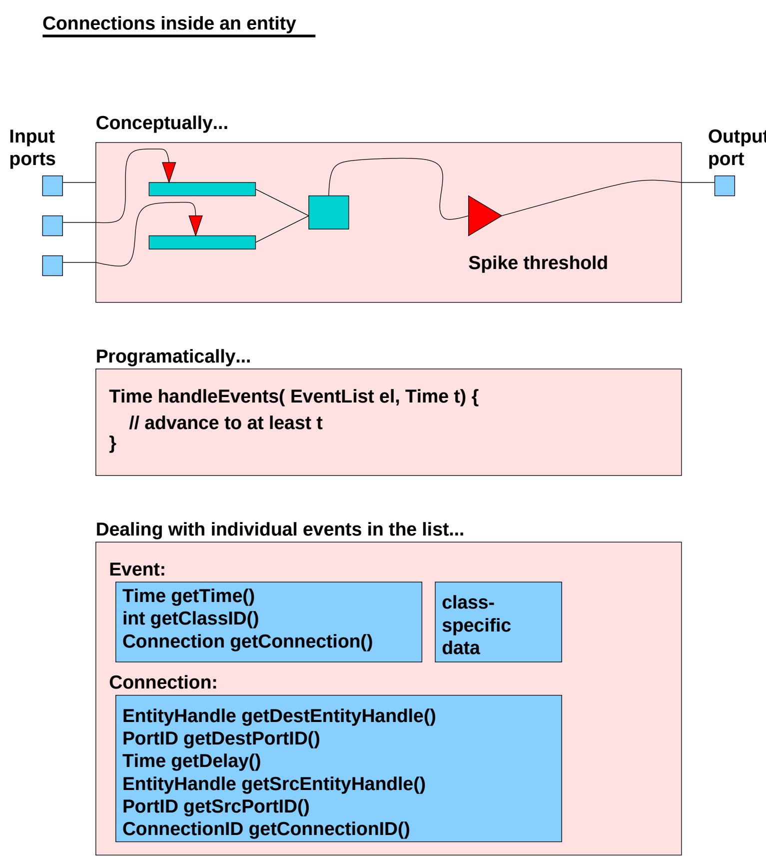


Figure 8: Connections inside an entity.

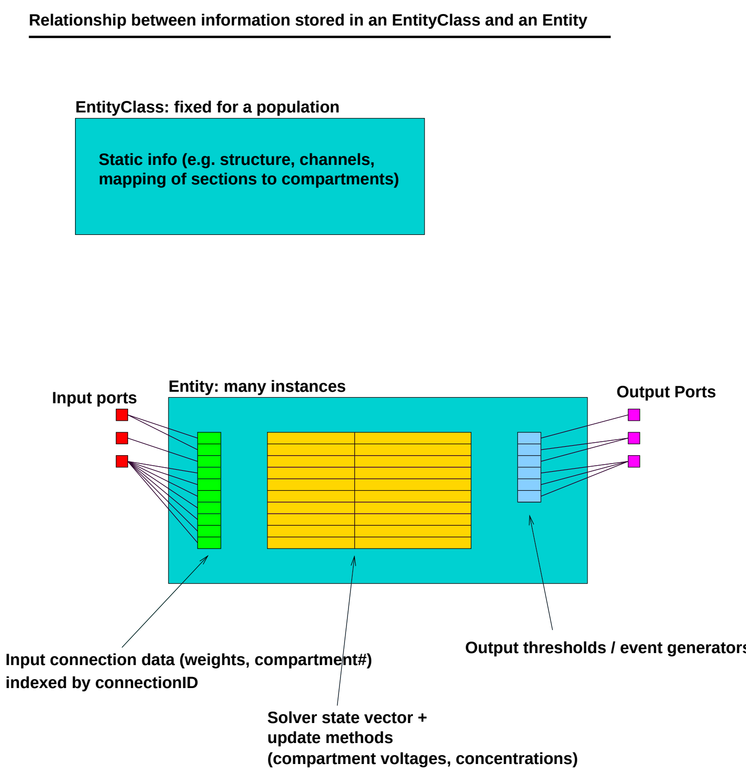


Figure 9: Distinction between EntityClass and Entity.

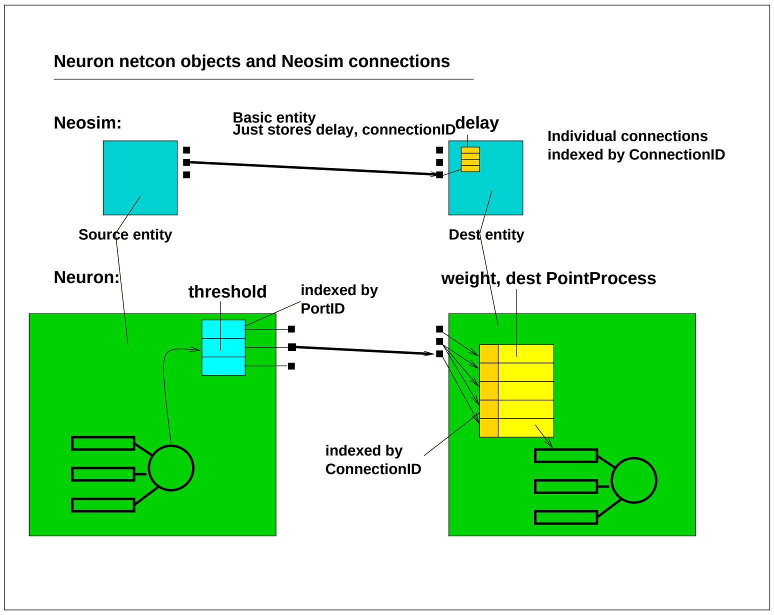


Figure 10: Relationship between Neosim and Neuron connections.

```
  NetConSrc( PortID srcp, Double thresh);
  void sendRequests(Entity srce, Population destPop, DestMethod dm) {
    ((NeuronEntity) srce).setThreshold( srcp, thresh );
    bcastRequest( destPop, new ConnectionRequest( srce, srcp, dm ) );
  }
}
class NetConDest extends DestMethod {
  NetConDest( PortID destp, double delay, double weight );
  boolean considerRequest( Entity deste, ConnectionRequest cr ) {
    ConnectionID cid = makeConnection( new Connection( cr.srce, cr.srcPort, deste, destp, delay ) );
    return true;
  }
}
EntityID srce   = // source entity
PortID srcport  = // ID of source point process
EntityID deste  = // dest entity
PortID destport = // ID of dest point process
connect( new GeneralProjection( srce, deste,
              new NetConSrc(srcport, threshold),
              new NetConDest(dstPort,delay,weight) ) ) );
```

## 8 Example

This minimal example defines two new entity classes, a producer and a consumer, connects them together, and runs a simulation for 1 second.

The first step is to define the new classes of entity, first the producer which outputs a sine wave on its single output port:

```
class ProducerClass extends EntityClass {
  public double tstep;
  ProducerClass( String name, double tstep ) { super(name); this.tstep = tstep; }
  Entity makeEntity( Population p, int index ) {
    ProducerEntity e = new ProducerEntity(p, this);
    e.init( p, index );
  }
}

class ProducerEntity extends Entity {
  ProducerClass pc;
  PortID outp;
  ProducerEntity( Population p, ProducerClass pc ) {
    super(p, pc); this.pc = pc;
    outp = addOutPort( DoubleEventID );
  }
  Time handleEvents( EventList el, Time t ) {
    for (int i=0; i < el.size(); i++) {
      defaultEventHandler( el.elementAt(i) );
    }
    for (double tt = localTime(); tt < t; tt += pc.tstep) {
      sendEvent( outp, new DoubleEvent( tt, sine(tt) ) );
    }
    return t;
  }
}
```

The consumer can be defined similarly:

```
class ConsumerClass extends EntityClass {
  ConsumerClass( String name ) { super(name); }
  Entity makeEntity( Population p, int index ) {
    ProducerEntity e = new ProducerEntity(p, this);
  }
}

class ConsumerEntity extends Entity {
  PortID inp;
  ConsumerEntity( Population p, ConsumerClass cc ) {
    super(p, cc);
```

```
    inp = addInPort( DoubleEventID );
  }
  Time handleEvents( EventList el, Time t ) {
    for (int i=0; i < el.size(); i++) {
      Event ev = el.elementAt(i);
      if (ev.getClassID() == DoubleEventID) {
        DoubleEvent dev = (DoubleEvent)ev;
        System.out.println("Received "+dev.getDouble()+" at "+dev.getTime());
      } else {
        defaultEventHandler( ev );
      }
    }
    return t;
  }
}
```

This simulation should be set up from a script reader:

```
class SimpleSimulation extends ScriptReader {
  void bootstrap(Object args[]) {
    Time tstep = ms(10);
    ProducerClass pc = new ProducerClass( "prod10ms", tstep );
    Population p = new PopulationBuilder( "producers", pc, 1, null );
    construct( p );

    ConsumerClass cc = new ConsumerClass( "cons" );
    Population c = new PopulationBuilder( "consumers", cc, 1, null );
    construct( c );

    Time delay = ms(10);
    EntityID prod = lookupEntity("producers0");
    EntityID cons = lookupEntity("consumers0");
    connect( new Projection( new Connection( prod, 1, cons, 1, delay ) ) );

    Time runtime = seconds(1);
    runSim( runtime );
  }
}
```

# 9 Appendix: Listing of classes and methods

This appendix lists neosim classes and methods visible to those implementing new entities, events and script readers. Each class description is of the format:

- Methods which a derived class can override.

- Methods which can be called from within class (equivalent to “final method” in Java)
- Comments

More detail is available from the `javadoc` documentation (section 12).
Figure 11 shows the basic classes of Neosim.

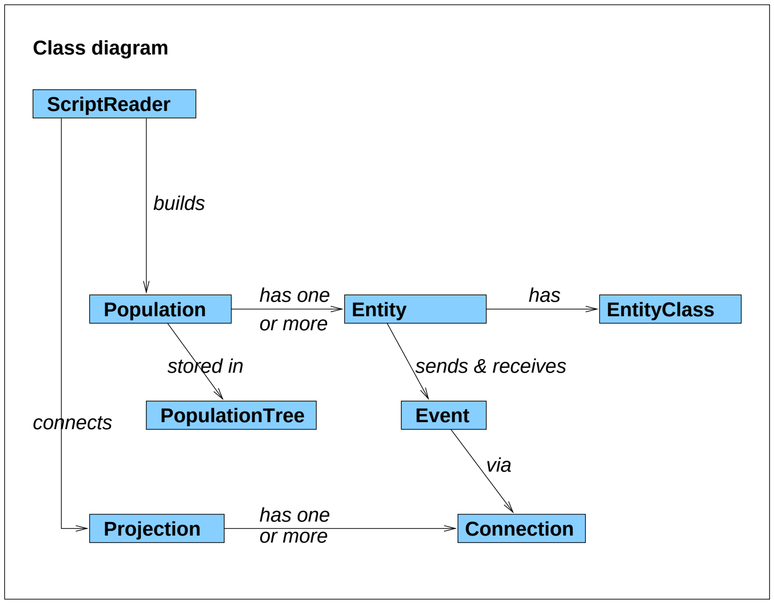


Figure 11: Class diagram.

The basic classes do very little in themselves; they are intended to be extended to provide facilities for specific types of entities and script readers. Figure 12 illustrates two extensions of the ScriptReader class for handling different script languages.

Figure 13 shows two built in types of populations; the PopulationBuilder, used for creating populations, and the PopulationView, used for creating indexes and restricted views of existing populations. Routines for building particular sizes and shapes of populations can extend these classes.

## 9.1 Entity

An instance of this is a runnable entity. Input ports correspond to input receptors, Output ports correspond to spike generators, voltage probes etc. Many connections can be made to/from any port. Input ConnectionIDs are unique for an entity, so can be used to look up extra connection information stored inside the entity.

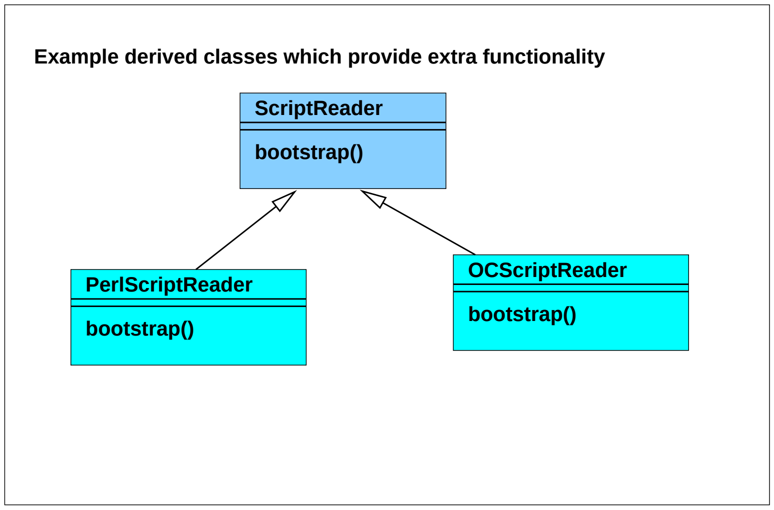


Figure 12: Script readers.

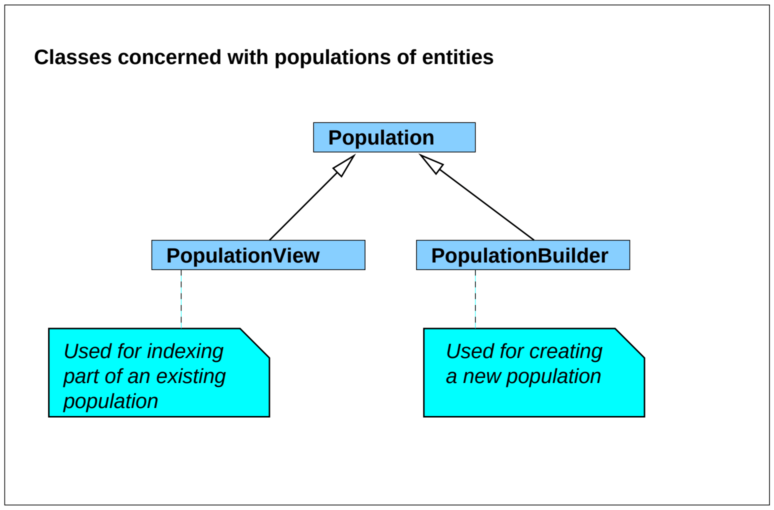


Figure 13: Population classes.

- Entity( Population p, EntityClass e )

```
void         init(Population p, int index)
Time         handleEvents(EventList el, Time t)
void         reset()
void         destroy()

void         sendEvent( PortID c, Event e )
void         sendEvent( EntityID deste, PortID destp, Event e )
void         defaultEventHandler( Event e )
Population   getPopulation()
EntityClass  getClass()
EntityID     getID()

PopulationTree getPopulationTree(Time t)
EntityClasses  getEntityClasses()
EventClasses   getEventClasses()
EntityClass    lookupEntityClass( String name );
EntityID       lookupEntity( Time t, String name );
Population     lookupPopulation( Time t, String name );

void         construct ( Time t, PopulationBuilder p, Callback c)
void         destroy ( Time t, Population p, Callback c )
void         connect ( Time t, Projection p, Callback c )
void         disconnect ( Time t, Projection p, Callback c )
void         updateEntity( Time t, EntityID e, EntityUpdate eu, Callback c )
void         queryEntity( Time t, EntityID e, EntityQuery eq, Callback c )
void         updatePopulation( Time t, Population p, EntityUpdate eu, Callback c )
void         queryPopulation( Time t, Population p, EntityQuery eq, Callback c )
void         queryOutputConnections( Time t, PortID p, ConnectionQuery cq, Callback cb );

Vector       getInPortIDs()
Vector       getOutPortIDs()
PortID       addOutPort(EventClass ec)
PortID       addInPort (EventClass ec)
Vector       getClassesOfInPort(PortID pid)
Vector       getClassesOfOutPort(PortID pid)
int          getNumInConnections(PortID inport)
int          getNumOutConnections(PortID outport)
Vector       getInConnections(PortID inport)
void         addOutConnection(PortID outport, Time delay,
               EntityID dest, PortID destinport, ConnectionID destconn)
void         deleteOutConnection(PortID outport, EntityID dest,
               PortID destinport, ConnectionID destconn)
ConnectionID addInConnection(Connection c)
void         deleteInConnection(ConnectionID cid)
void         deleteInPort(PortID)
void         deleteOutPort(PortID)

void         zeroOutputDelay()
void         resetOutputDelay()
```

```
Time        getMinOutputDelay()
Time        getLocalTime()
```

- The crucial method for an entity to implement is `handleEvents()` which should advance the entity's time to at least t and return the time actually advanced to (which may be later than t, as long as the state can be rewound back to t and no new events are generated after t). `reset()` and `destroy` inform an entity that it should reset its simulation time, or remove any allocated storage.

  The entity's minimum output delay is available using `getMinOutputDelay()`; this will be the minimum delay on any of its output ports, or if there are no output connections, a positive value determined by the kernel. This output delay is imposed on every kernel request the entity makes (i.e. responses to queries, requests to construct and link other entities). If this is not acceptable to the model, the output delay can be zeroed using `zeroOutputDelay()`, and then unzeroed using `resetOutputDelay()`. For example, to perform a sequence of queries/responses/new queries at a specific instant of time while a simulation is running, zero the output delay of the population concerned, perform the updates/queries, then reset the output delay.

## 9.2 ScriptReader

The script reader interface provides the central control point, responsible for building and running the simulation.

- `void bootstrap(String args[])`

- ```
  void   initSim()
  void   runSim(double t)
  void   reset()
  void   construct ( PopulationBuilder p)
  void   destroy ( Population p )
  void   connect ( Projection p )
  void   disconnect ( Projection p )
  void   updateEntity( EntityHandle e, EntityUpdate eu )
  Object queryEntity(  EntityHandle e, EntityQuery eq )
  void   updatePopulation( Population p, EntityUpdate eu )
  Vector queryPopulation(  Population p, EntityQuery eq )
  PopulationTree getPopulationTree()
  NeosimClasses  getNeosimClasses()
  EntityClass    lookupEntityClass( String name )
  EntityHandle   lookupEntity( String name )
  Population     lookupPopulation( String name )
  ```

- The script reader issues commands to build and run the simulation.
  ```

## 9.3 EntityClass

An instance of this is a specification of how to make a particular kind of entity. Each entity stores a reference to its class, so the EntityClass can be used to keep any static information which doesn't vary between entity instances, e.g. morphology (if static), details of channels, etc. The basic EntityClass just has a name.

- ```
  EntityClass( String name )
  Entity makeEntity(Population p, int index)
  ```

- A derived EntityClass provides a `makeEntity` method to create an instance of the entity.

## 9.4 Event

A basic event just has a time stamp and a class id. Derived types of event can include extra data and methods.

- ```
  Event( Time t )
  Time         getTime()
  ClassID      getClassID()
  NeosimClass  getClass()
  EntityID     getSrcEntityID()
  PortID       getSrcEntityPortID()
  Connection   getConnection()
  ```

## 9.5 Population extends PopulationNode

A population has a name, a position in the population tree, and a number of entities. Derived versions include a PopulationBuilder for constructing a number of similar entities (maybe in a 2D layer, or randomly distributed in some volume) and a PopulationView which provides a view of an existing population based on some entity characteristic (e.g. all the entities in a given region, or a grid based decomposition of a volume of entities).

- ```
  Population( String name, PopulationNode parent )
  int           getId()
  int           getNumEnts()
  int           getIndex( EntityID eid )
  boolean       isMember( Entity e )
  Vector        getMemberList()
  ```

### 9.5.1 PopulationView extends Population

A population view provides a way of referring to a collection of existing entities. It is specified by providing a population and a method to decide whether to add each entity to the view. A typical view would be a grid based decomposition which provides a method to look up all entities in a given grid square.
  ```

- ```
  PopulationView( String name, Population srcPop )
  void buildView( Entity e )

  void      addEntityToView( Entity e )
  ```
- Derived population views should override the buildView() method which is called for each entity of the source population and considers whether to add the given entity to the view (by calling addEntityToView()).

#### 9.5.2 PopulationBuilder extends Population

A population builder builds a number of entities of a given class and initialises each one according to its index in the population.

- ```
  PopulationBuilder( String name, EntityClass ec, int size, EntityInit ei  )
  ```
- The population initialisation method EntityInit is called after each entity has been instantiated.

#### 9.5.3 PopulationNode

The PopulationNode is the superclass for every node in the population tree. It just stores a name and a parent. A root node returns null as the parent.

- ```
  PopulationNode( String name, PopulationNode parent )
  String e        getName()
  PopulationNode  getParent()
  ```

#### 9.5.4 PopulationTree extends PopulationNode

The PopulationTree stores grouping nodes and populations of all entities in the simulation.

- ```
  void            addNode(PopulationNode n)
  void            deleteNode(PopulationNode n)
  PopulationNode  getNode(String name)
  Vector          getChildren()
  EntityID    getEntityID(String entityName)
  ```

### 9.6 Connection

A connection links a source entity/port to a destination entity/port, and has a delay. It is possible to check the connection details for events arriving on a port to get the delay, and source entity/port. ConnectionIDs are assigned when the addInConnection() method of the destination entity is called.

- ```
  Connection( EntityID srce, PortID srcport, EntityID deste, PortID destport, Time delay )
  EntityID     getDstEntityID()
  PortID       getDestPort()
  Time         getDelay()
  ```
  ```

```
EntityID     getSrcEntityID()
PortID       getSrcPort()
ConnectionID getConnectionID()
```

## 9.7 Projection

A projection stores a specification of a set of connections from one population onto another, and can be specified as an array of connections to make from output ports of source entities to input ports of destination entities. If the connections to make depend on the state of the source and/or destination entities, a GeneralProjection (see below) should be used instead.

- `Projection( Connection[] connections )`
  `Connection[] getConnections()`

### 9.7.1 GeneralProjection extends Projection

A general projection can be specified by providing two methods; a source method runs on each member of a source population and decides which entities in the dest population it wants to connect to; a dest method has access to the internals of the destination entity and decides which requests to fulfill.

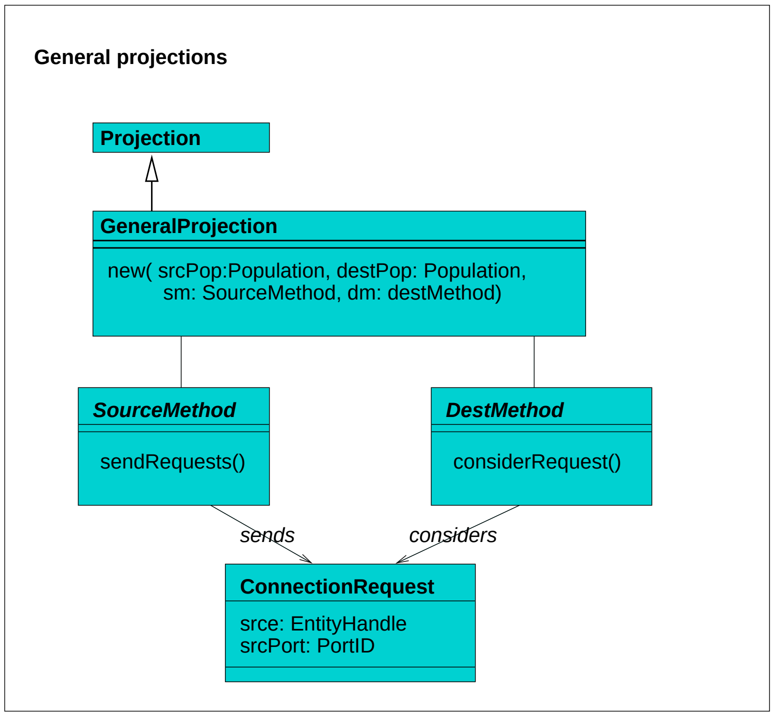


Figure 14: Classes used for a general projection.

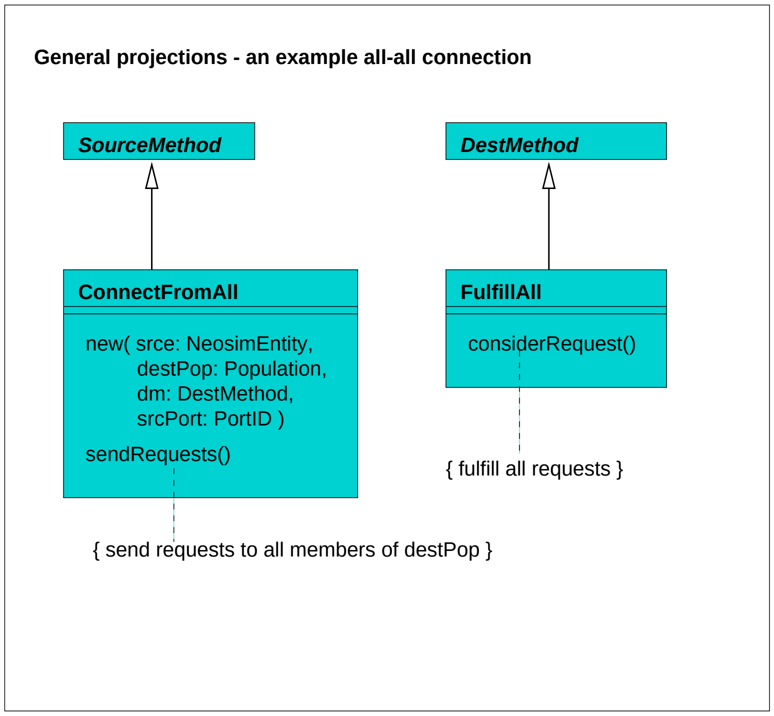


Figure 15: Example general projection - all to all.

- ```
  GeneralProjection(Population srcPop,   Population destPop,
                    SourceMethod sm,     DestMethod dm)
  ```

- The SourceMethod can be written in a script language; it decides which members of the dest population it might want to connect to (maybe looking up population methods to check for candidate entities within range of this entity, or selecting an entity at random, or possibly broadcasting a request to all members of the destination population).

### 9.7.2 ConnectionRequest

The message which a source entity sends to a dest entity it wants to connect to. By default it contains the source entity ID and port no, as well as the DestMethod.

- `ConnectionRequest( EntityID srce, PortID srcPort, DestMethod dm)`

- Derived connection requests can include extra information which the destination entity might need to consider, e.g. the source entity's position, or axon shape.

### 9.7.3 SourceMethod

The part of a general projection which runs on each source entity and sends requests to make connections to candidate destination entities.

- ```
  void sendRequests( Entity srce, Population destPop, DestMethod dm )

  void sendRequest(  EntityID deste, ConnectionRequest cr )
  void bcastRequest( Population destpop, ConnectionRequest cr )
  ```

- The sendRequests method checks the source entity's details and sends connection requests to individual destination entities (using sendRequest) or to all members of a population (using bcastRequest).

### 9.7.4 DestMethod

The part of a general projection which runs on a destination entity and decides to fulfill/deny connection requests.

- ```
  boolean considerRequest( Entity deste, ConnectionRequest cr )

  ConnectionID makeConnection( Connection c );
  ```

- The considerRequest method checks the source entity's details against the dest entity's details, and returns True or False depending on whether or not the connection will be made. The connection is made by calling `makeConnection()`.
  ```

## 9.8 Queries, updates and callbacks

### 9.8.1 EntityQuery

The EntityQuery interface is provided for extracting information from entities.

- `Object query( Entity e );`
- Derived queries can return any type of object from a (maybe derived) entity. Queries are issued from the script reader using the queryEntity() and queryPopulation() methods. Queries can also be issued from entities provided a callback is provided to deal with the response.

### 9.8.2 EntityUpdate

The EntityUpdate interface is provided for modifying entities; unlike the query interface it does not return a value.

- `void update( Entity e );`
- Updates are issued from the script reader using the updateEntity() and updatePopulation() methods. Updates can also be issued from entities.

### 9.8.3 EntityInit

The EntityInit interface is provided for initialising entities.

- `void init( Entity e, Population p, int index);`
- The init method initialises the entity to be member *index* of the given population. It can also set population specific parameters such as position, initial soma potential, etc.

### 9.8.4 Callback

The Callback interface is used for notifying entities that some query has completed.

- `void callback( Entity e, Object o);`

# 10 Appendix: Timing issues

Entities in Neosim are not kept in lockstep; a snapshot of the simulation while it is running will show entities at different simulation times. This has some implications for building and running models.

The amount of time which entities are able to lag behind "current simulation time" is determined by their *minimum output delay.* This must be greater than zero. A practical minimum is the integration timestep (e.g. 20usec), but in practice it is expected it will be of the order of 1-10 ms to account for axonal delay.

The figure below illustrates 4 entities at the start of a simulation. Each entity has a separate output delay (either 2 or 3 ms in this example), and at the start all entities are at time 0.

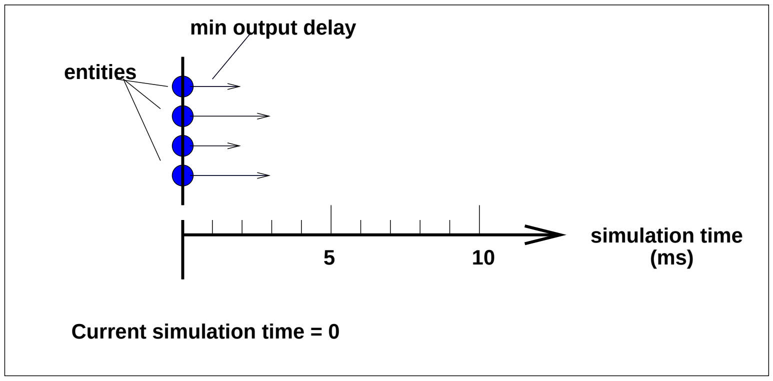


Figure 16: At the start all entities are at time 0.

When the simulation is set running, current simulation time can advance to the minimum output delay of all entities, and any entity can be advanced up to the current simulation time by calling its `handleEvents()` method. If all 4 entities were on different processors, they could all be advanced in parallel. Figures 17 and 18 show simulation time advancing. And at t=10ms.

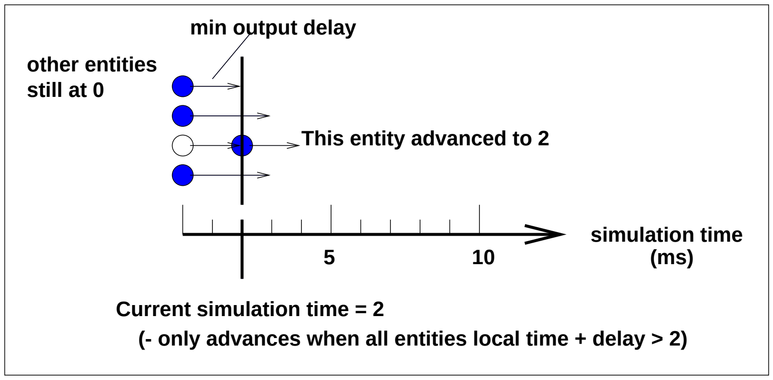


Figure 17: One entity is advanced to its right window time.

Note that at any point of time the entities' local times are staggered, and that the entities lag behind current simulation time by anything from 0 to their min output delay. The times of entities will "leapfrog" each other. This has performance advantages, as each entity can do a large number of timestep updates

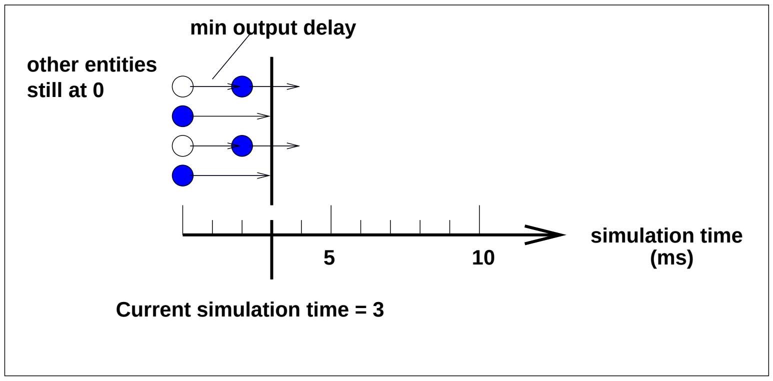


Figure 18: The current simulation time can now advance.

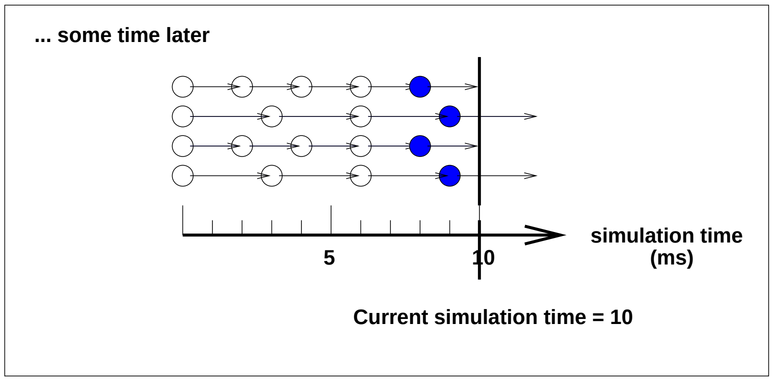


Figure 19: Some time later.

in one function call, with the state variables likely to be in cache. The other advantage is that entities can be physically distributed on different processors, and there can be a degree of slackness in the synchronisation requirements between processors. The catch, from an ease of programming point of view is that entities cannot manipulate other entities directly and instantaneously for causality reasons. An entity wishing to create more entities has to do so by posting and event after its minimum output delay.

## 11 Appendix: C++ implementation issues

The interface above is largely specified using Java notation. The important issues involved in interfacing C++ are:

- **Object serialization:** In Java, a serializable version of a class can be created just by implementing **java.io.serializable**. In C++, this is not automatic. Automatic serialization of complex classes makes programming operations on remote entities much simpler, as an object (containing parameters etc) can be sent to the remote entity, and the response object can be returned, without having to pack and unpack data manually. Java-like serialization can be implemented in C++ by deriving all objects which might be transmitted from a C++ interface class:

```
class Serializable {
  void getClassID() = 0;
  void getByteArray(byte* array, int &len) = 0;
  void readFromByteArray(byte *array, int len) = 0;
  void pack_int(    byte* &array, int v, int &len )    {
    *((int*)array) = v; len -= sizeof(int); array+=len; }
  void pack_double( byte* &array, double v, int &len ) {
    *((double*)array) = v; len -= sizeof(int); array+=len; }
  // ...
  int unpack_int( byte * &array, int &len ) {
    int i = *((int*)array); array += sizeof(int); return i; }
  // ...
}

class MySerializableClass : public Serializable {
  int x, y, z;
public:
  void getByteArray(byte* array, int &len) {
    pack_int( array, x, len );
    pack_int( array, y, len );
    pack_int( array, z, len );
  }
  void readFromByteArray(byte *array, int len) {
    x = unpack_int( array, len  );
```

```
      y = unpack_int( array, len  );
      z = unpack_int( array, len  );
    }
  }
```

- **Migration of entities** For entities to migrate between processors, they have to be seralizable; derived types of entities should be able to serialize their internal state.
- **Garbage collection** In Java, garbage collection is automatic; C++ programmers have to make sure that unused objects are deleted. For entities, this means removing any allocated space when the **destroy()** method is calles.
- **Bindings** C++ header files corresponding to the Java classes will be provided to give access to data.

# 12 javadoc interface documentation

A comprehensive listing of the interface classes and methods generated using javadoc is available at:

- `javadoc interface documentation:`
  `http://www.dcs.ed.ac.uk/home/fwh/neosim/doc/interface`

# 13 Download

A sequential java implementation of Neosim with example is available at:

- `Download:`
  `http://www.dcs.ed.ac.uk/home/fwh/neosim/dload/neosim.tar.gz`