Online supplementary material for "Run-Time Interoperability between Neuronal Network Simulators based on the MUSIC Framework"

February 3, 2010

A Network Representation in NEST

The network in NEST is stored as an adjacency list based on a weighted directed graph. Each node in the graph represents either a neuron, or a device for observing and manipulating neurons. All nodes are derived from the base class `Node`, which provides a common interface for the communication with the simulation kernel. The functions which are relevant for the implementation of the NEST-MUSIC interface are shown in Table 1.

Each connection is represented by a tuple containing the ids of sender and receiver, weight, delay and a receptor port. The receptor port is used by the receiver to differentiate between different incoming connections. In connection with the MUSIC interface it identifies the channel on a MUSIC output port to which events are forwarded to.

A detailed description of the network representation and scheduling algorithm of NEST is contained in Plesser et al. (2007).

B Using PyNEST in a distributed scenario

The MPI standard does not mandate that command line arguments are given to the child processes directly, but allows to "inject" them into the `argv` array during the call to `MPI_Init`(). This leads to problems when the child process does not support MPI directly, like e.g. in the case of Python. To circumvent these problems, we can use a small launcher script instead of using Python. A launcher script to run the PyNEST script *simulation.py* is shown in the following listing:

```
#!/bin/bash
exec python simulation.py
```

Function	**Parameters**	**Description**
`get_status`()	Dictionary	The supplied dictionary is filled with the internal parameters of the node and returned to the caller.
`set_status`()	Dictionary	The parameters from the supplied dictionary are set as internal parameters for the node.
`calibrate`()		Called by the `Scheduler` before the simulation starts. Used to re-calculate internal parameters based on those of the simulation kernel.
`connect_sender`()	EventT integer	Called on the *receiver* when a connection for events of type EventT is to be established. To support a specific type of event, a node has to overload this function for the respective type. The integer argument is used to select the receptor type on the receiver side.
`handle`()	EventT	Called by the scheduler to deliver events of type EventT to the node.

Table 1: Interface functions of class `nest::Node` that are used by the NEST-MUSIC interface. EventT denotes one of NEST's built-in event types.

C A complete example with NEST

To illustrate the interplay of NEST and MUSIC, we now walk through an example, in which a MUSIC eventgenerator creates a random spike train that is fed to NEST. The incoming spikes are recorded by a spike detector in NEST and forwarded to a MUSIC output port, which is connected to a MUSIC eventlogger.

The following listings show the PyNEST script (`events_in_out.py`) to set up the simulation: First, we import the PyNEST module and tell NEST that it should overwrite data files.

```
from nest import *
SetKernelStatus({”overwrite_files”: True})
```

Second, we create a `music_in_proxy` and set the port and channel it listens to to `spikes_in` and 0, respectively.

```
mip = Create(”music_in_proxy”)
SetStatus(mip, {”port_name”: ”spikes_in”,
                ”music_channel”: 0})
```

Third, we set the acceptable latency of the MUSIC port `spikes_in` to 1.0 miliseconds.

```
SetAcceptableLatency(”spikes_in”, 1.0)
```

Fourth, we create a spike detector device and set its status, so that the spikes it detects are not stored in memory, but printed to the screen directly.

```
sd = Create("spike_detector")
SetStatus(sd, {"to_memory": False, "to_screen": True})
```

Fifth, we create a `music_out_proxy`, `mop`, which listens to the MUSIC port named `spikes_out`

```
mop = Create("music_out_proxy")
SetStatus(mop, {"port_name": "spikes_out"})
```

Sixth, we connect the MUSIC input proxy `mip` to the spike detector `sd` and to channel 0 of the MUSIC output proxy `mop`.

```
Connect(mip, sd)
Connect(mip, mop, {"music_channel": 0})
```

Finally, we simulate the network for 10.0 miliseconds.

```
Simulate(10.0)
```

To be able to use PyNEST with MUSIC in a distributed way, we use a small launcher script called `nestlauncher.sh`. The content of the script is shown in the following listing:

```
#!/bin/bash
exec python events_in_out.py
```

The MUSIC configuration file `events_in_out.music` is shown in the following listing: We first define the stoptime of the simulation:

```
stoptime=0.01
```

The first application is an `eventgenerator`, which comes with the MUSIC library.

```
[generator]
  binary=eventgenerator
  np=1
  args=--timestep 0.001 --frequency 500.0 1
```

The second application runs our simulation in NEST using the small helper script `nestlauncher.sh`. We connect the port `out` of the `generator` with the port `spikes_in` of NEST.

```
[nest]
  binary=nestlauncher.sh
  np=1
  generator.out -> nest.spikes_in [1]
```

The third application is an `eventlogger`, which is also part of MUSIC and just prints the events it receives to the screen. We connect the port `spikes_out` of NEST with the `logger`.

```
[logger]
  binary=eventlogger
  np=1
  args=—timestep 0.001
  nest.spikes_out -> logger.in [1]
```

Finally, the multi-simulation is run by calling `mpirun −np 3 music events_in_out.music`. The output in the listing below shows that both NEST's `spikedetector` (upper block, time in miliseconds) and the MUSIC `eventlogger` (lower block, time in seconds) see the spikes at the same time:

```
1 3.534
1 4.11
1 8.789
1 9.066

Rank 0: Event (0, 0.003534) detected at 0
Rank 0: Event (0, 0.00411) detected at 0
Rank 0: Event (0, 0.008789) detected at 0
Rank 0: Event (0, 0.009066) detected at 0
```

D Sequence diagrams for NEST

See Figures 1 and 2.

E A complete example with MOOSE

Connecting a MOOSE model with MUSIC is straightforward. We will here walk through a sample MOOSE script where a model is set up to receive action potentials from MUSIC. See Appendix F for sequence diagrams that explain how MOOSE and MUSIC objects interact.

In this example, note that MOOSE arranges objects in a tree structure resembling the Unix filesystem. In this scheme, the string `/parentObj/childObj` acts as an identifier for an object named `childObj` which is contained in an object named `parentObj`, which in turn is situated in the top-level container object simply called / (or the `root` object).

First, a MUSIC port is declared. This is done by calling the `addPort` function on the `/music` singleton object. The arguments passed to this function are:

Argument	Value
Direction	"in"
Port type	"event"
Port name	"myPort"

This command creates an instance of the `InputEventPort` class. The name of the corresponding MUSIC port is "`myPort`". If we were interested in sending

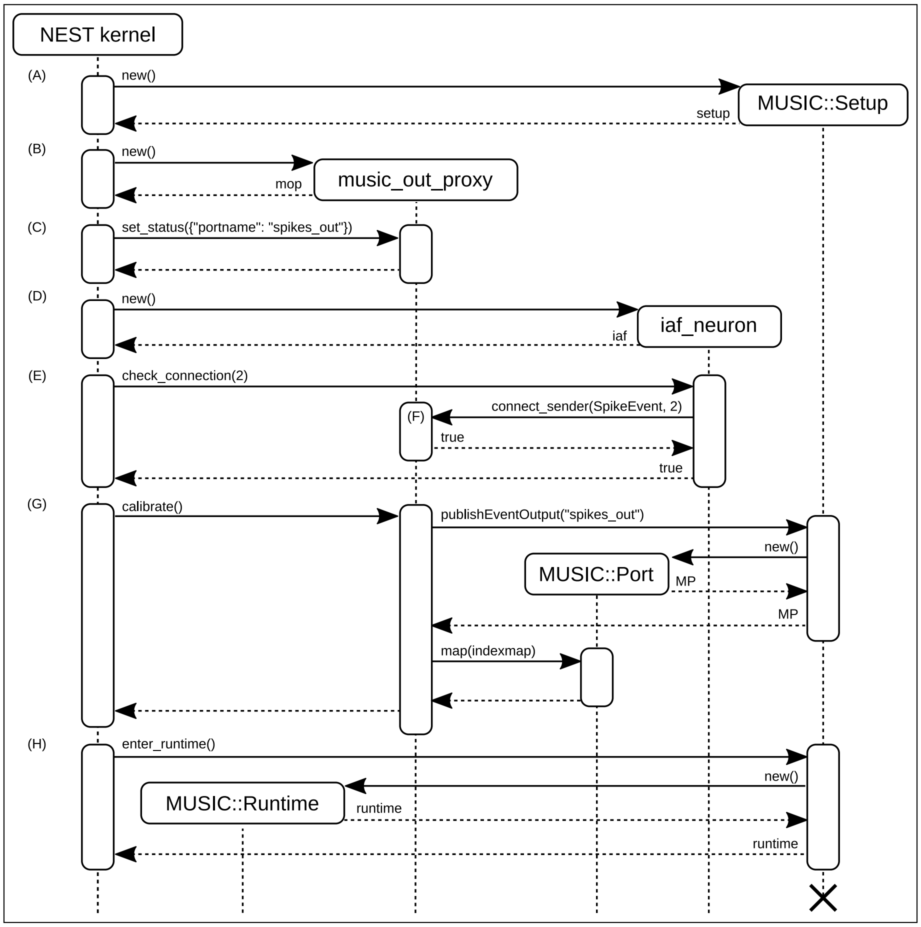

Figure 1: Sequence diagram showing the events from setup phase to runtime phase of a network containing a `music_out_proxy`. (A) On startup, NEST creates a MUSIC setup object. (B) The proxy is created by the command `mop` = `Create`(”`music_out_proxy`”). (C) The portname of the proxy is set using `SetStatus`(`mop`, ”`portname`”: ”`spikes_out`”). (D) A neuron is created by calling `iaf` = `Create`(”`iaf_neuron`”). (E) The neuron is connected to the MUSIC output channel 2 of the port the proxy represents using `Connect`(`n`, `mop`, ”`receptor_type`”: 2). (F) The proxy stores the connection in its `indexmap`, which contains the indices of all local channels of the port. (G) Calling `Simulate`() first calls `calibrate`() on the proxy, where it registers itself with MUSIC using the `indexmap`, built previously. (H) NEST then enters the MUSIC runtime phase by calling `enter_runtime`() on the setup object, which returns a runtime object and destroys itself thereafter.

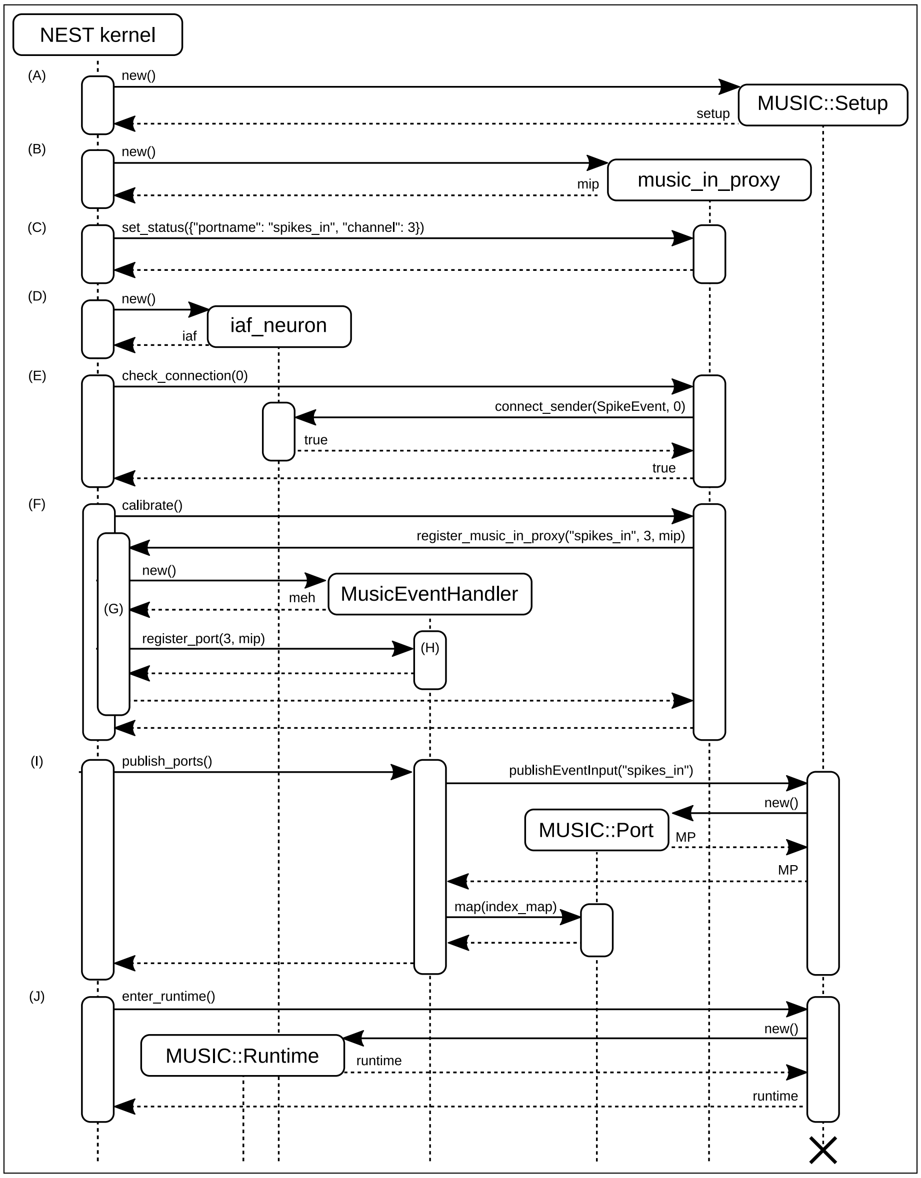

Figure 2: Sequence diagram showing the events from setup phase to runtime phase of a network containing a `music_out_proxy`. (A) On startup, NEST creates a MUSIC setup object. (B) The proxy is created by the command `mip` = `Create`(”`music_in_proxy`”). (C) The portname and and the channel of the proxy are set using `SetStatus`(`mip`, ”`portname`”: ”`spikes_in`”, ”`channel`”: 3). (D) A neuron is created by calling `iaf` = `Create`(”`iaf_neuron`”). (E) The proxy is connected to the neuron using `Connect`(`mip`, `n`). (F) Calling `Simulate`() first calls `calibrate`() on the proxy, where it registers itself with NEST kernel using the function `register_music_in_proxy`() with the portname, the channel, and itself as arguments. (G) the NEST kernel stores the new `MusicEventHandler` in its `music_in_portmap` under the name `spikes_in`. (H) The `MusicEventHandler` registers `mip` as the proxy for channel 3. (I) The function `publish_ports`() creates and maps a MUSIC output port object for the ports that are known to NEST. (J) NEST then enters the MUSIC runtime phase by calling `enter_runtime`() on the setup object, which returns a runtime object and destroys itself thereafter.

spike-time information to MUSIC, we would simply set the `Direction` argument to "`out`".

```
call /music addPort "in" "event" "myPort"
```

Next, the rate of calling the MUSIC `tick`() function is specified. This is done by creating a "clock" which will request the `/music` object to do its calculations at regular intervals. Here, clock #4 will invoke `/music`'s calculations every 1e-3 seconds of simulation time.

```
setclock 4 1e-3
useclock /music 4
```

When the input port, "`myPort`", was declared above, a few new objects were created automatically:

- `/music/myPort`
- `/music/myPort/channel`[0], . . . , [`width` − 1]

Here the width of "`myPort`" was used to create an array of channels: `channel`[0] . . . `channel`[`width` − 1]. The width was requested from the MUSIC API. Each of the objects in the `channel`[\#] array is an instance of the `InputEventChannel` class, which emits spike-times as received from MUSIC.

It is now simple to send spike-times to synapses in the model. Here the 1st event-generating channel in "`myPort`" is connected to the " `AMPA`" synaptic channel situated in "`myCompartment`".

```
addmsg \
    /music/myPort/channel[ 0 ]/event \
    /myCompartment/AMPA/synapse
```

Finally, the simulation is run for 1.0 seconds (of simulation time).

```
reset
step 1.0 -time
```

Here, the `reset` command will invoke the initialization routines in the `Music`, `OutputEventPort` and `InputEventPort` instances. This involves setting up the MUSIC `Runtime` object, and mapping local indices to global indices. The step command starts the main simulation loop, which invokes the calculation routine of the `Music` instance, which in turn calls MUSIC's `tick`() function.

F Sequence diagrams for MOOSE

See Figures 3 and 4.

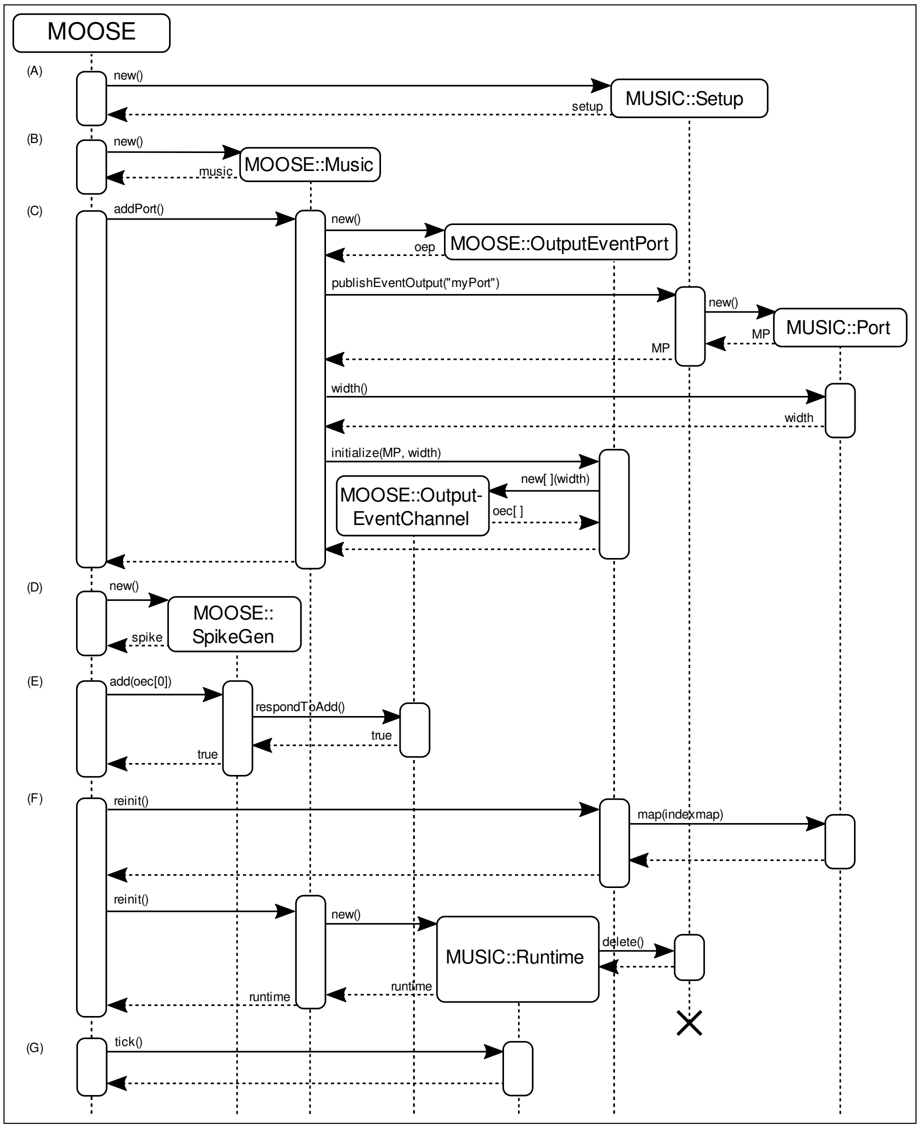

Figure 3:
Sequence diagram showing the events of the setup and runtime phases of a network containing a `MOOSE::OutputEventPort`. (A) On startup, MOOSE creates a MUSIC setup object. (B) An instance of `MOOSE::Music` is created automatically at the location `/music`. (C) An output event port is represented in MOOSE by the command `call /music addPort` ”`out`” ”`event`” ”`myPort`”. This involves creating an instance of OutputEventPort, publishing the port in MUSIC, finding out its width, and creating a corresponding number of `MOOSE::OutputEventChannel` instances. (D) A `SpikeGen` object is created by calling `create SpikeGen /myCompartment/spike`. (E) This spike-generating object is then connected to the MUSIC output channel 0 of the port using `addmsg /myCompartment/spike/event /music/myPort/channel`[0]`/synapse`. (F) The user calls `reset`, which invokes initialization routines on all MOOSE objects. Instances of `MOOSE::OutputEventPort` map global indices to local indices through a MUSIC API call. The MUSIC runtime phase begins as the `MOOSE::Music` object requests for the MUSIC runtime object, which also leads to the MUSIC setup object being destroyed. (G) The MOOSE `step` call leads to MUSIC `tick`() being called at regular intervals.

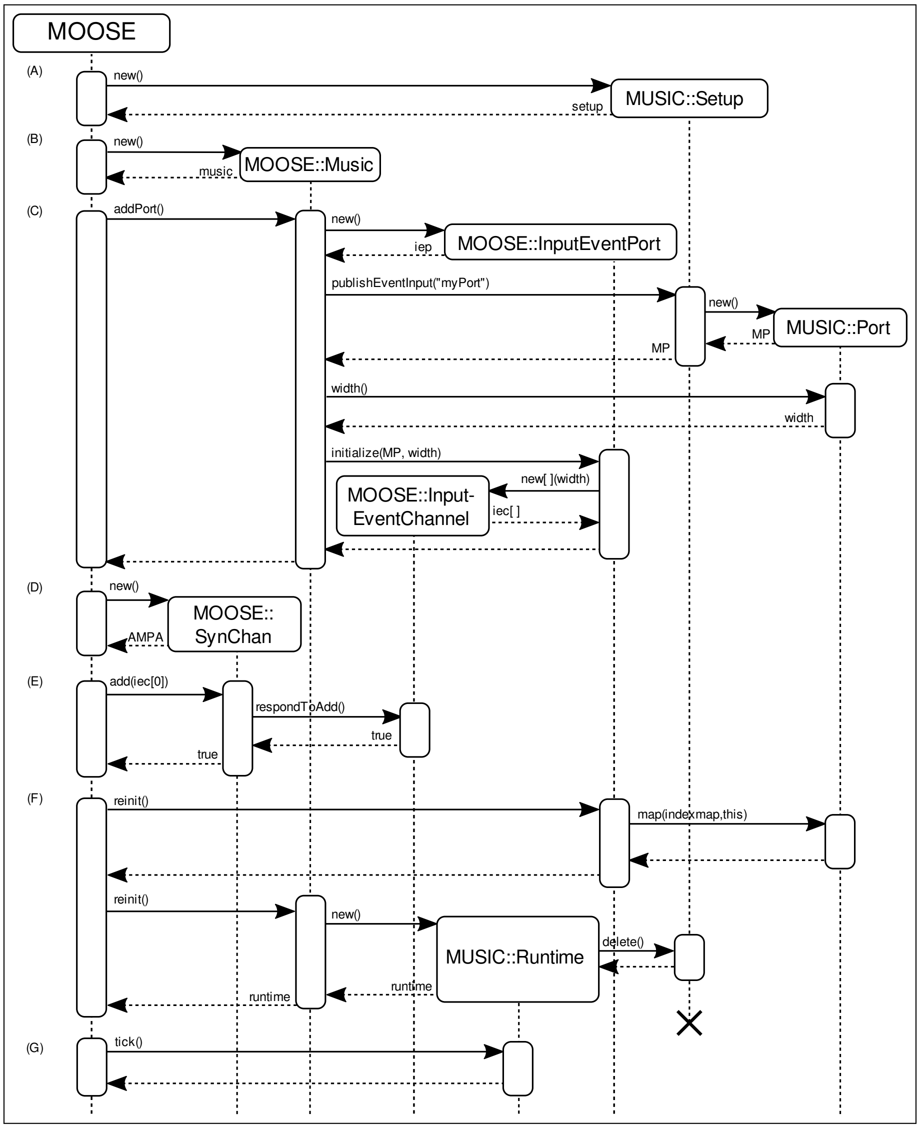

Figure 4:
Sequence diagram showing the events of the setup and runtime phases of a network containing a `MOOSE::InputEventPort`. (A) On startup, MOOSE creates a MUSIC setup object. (B) An instance of `MOOSE::Music` is created automatically at the location `/music`. (C) An input event port is represented in MOOSE by the command `call /music addPort "in" "event" "myPort"`. This involves creating an instance of InputEventPort, publishing the port in MUSIC, finding out its width, and creating a corresponding number of `MOOSE::InputEventChannel` instances. (D) A `SynChan` object is created by calling `create SynChan /myCompartment/AMPA`. (E) This synaptic channel object is then connected to the MUSIC input channel 0 of the port using `addmsg /music/myPort/channel[0]/event /myCompartment/AMPA/synapse`.
(F) The user calls `reset`, which invokes initialization routines on all MOOSE objects. Instances of `MOOSE::InputEventPort` map global indices to local indices through a MUSIC API call. Note that no MUSIC EventHandler object needs to be created since `MOOSE::InputEventPort` derives from the MUSIC EventHandler class. The MUSIC runtime phase begins as the `MOOSE::Music` object requests for the MUSIC runtime object, which also leads to the MUSIC setup object being destroyed. (G) The MOOSE `step` call leads to MUSIC `tick()` being called at regular intervals.

References

H. E. Plesser, J. M. Eppler, A. Morrison, M. Diesmann, and M.-O. Gewaltig. Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. In A.-M. Kermarrec, L. Bougé, and T. Priol, editors, *Euro-Par 2007: Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 672–681, Berlin, 2007. Springer-Verlag.