and a list of the same amount of post-synaptic neurons (or devices) and connects the corresponding elements in a one-to-one fashion. Because of the function call overhead, this function is not very efficient to use when creating large networks.

To avoid such overhead, the functions `Convergent Connect` and `DivergentConnect` can be used to create multiple connections with a single call. In addition, randomized variants for both of these functions exist to support the user in creating networks on the basis of knowledge of connectivity statistics. However, random connection parameters (e.g., weight, delay, or time constants) need to be specified by user code and supplied to NEST after the creation of connections.

4.2. TOPOLOGY MODULE

To ease the creation of complex networks with spatial structure, NEST provides the Topology Module (Plesser and Enger, 2013). It supports the user in connecting neurons and initializing synapse parameters based on their topological relationships in the network. In contrast to the CSA, the topology module is very much tailored to building structured networks consisting of layers in NEST with minimal overhead. The CSA has a wider focus in that it is simulator-independent and supports arbitrary connectivity patterns that can also include repetitive elements. We are currently investigating if future versions of the Topology Module can be based on the CSA.

4.3. SUPPORTING CONNECTIVITY-GENERATING LIBRARIES

As detailed above, NEST provides multiple methods for connecting neurons into a network. However, while the native routines scale very well (Helias et al., 2012), they are only suitable for creating simple patterns such as convergent/divergent connectivity without looping over them in user code. On the other hand, the topology module (Plesser and Enger, 2013) allows the creation of more complex structures, but requires neurons to be organized in special data structures (e.g., layers).

To support the connection generator interface in NEST and thus make more connectivity-generating libraries available to users, we created the `ConnectionGeneratorModule`. It is implemented as a plugin for NEST which extends both user interfaces, SLI and PyNEST, and builds on `libneurosim` (see section 2.2).

All neurons and devices in NEST are identified uniquely by an integer number, their global id (*GID*). As all existing connection routines in NEST work either on single GIDs or on lists of GIDs, we decided to also use this convention when a user specifies cells for a connection generator. These GIDs are internally mapped to contiguous ranges of integer indices starting at zero, for use by the connection generator. Our new interface for using connection generators in NEST consists of the following functions:

`CGConnect` takes a `ConnectionGenerator` *cg*, lists of GIDs for *pre*- and *post*-synaptic populations, and a *param_map*. It creates the connections between neurons in *pre* and *post* as prescribed by the rules in *cg*. The parameter map *param_map* maps parameter names (e.g., weight, delay) to their index for the parameter value vector created by the call to `next()` in the connection generator interface (see section 2.1). In the current implementation, only arities 0 and 2 are supported.

`CGParse` takes a serialized version of a connection generator in the string *xml* and returns the corresponding `ConnectionGenerator` object. A special use of this function exists on supercomputers, where Python is often not available on the compute nodes, or where the memory and performance penalty would not be acceptable and a pure SLI-based solution is preferable.

`CGParseFile` takes a file name *fname* and parses the serialized version of a connection generator contained therein.

`CGSelectImplementation` takes an XML tag *tag* representing the parent node of a serialized connection generator and the name of a library *library* to provide a parser for such an XML file. This information determines which library should carry out the parsing for `CGParse` and `CGParseFile`.

In order to use the new interface in NEST, the user first has to construct a `ConnectionGenerator` object. This can be done at the Python level by either using `csa` or the Python bindings of `libcsa` (see section 3.3). When PyNEST is used, this object can be directly given to `CGConnect`, which wraps the `ConnectionGenerator` object into a SLI Datum of type `connectiongeneratortype` that can be handed over to NEST's simulation kernel. It is then iterated at the C++ level in case of `libcsa`, or by calling back into Python in case of `csa`.

Another way to construct a `ConnectionGenerator` object is by parsing an XML serialization of the object. Such a serialization could be created at the Python level, created by an external tool, or written by hand. At the SLI level this serialization can then be given to one of the SLI functions `CGParse` or `CGParseFile`, which reinstantiate the original object using the functions `fromXML` or `fromXMLFile` in the connection generator API. This object (of type `connectiongeneratortype`) can be given to SLI's version of the `CGConnect` function. Note that the step of creating the serialization of the `ConnectionGenerator` can also be carried out on another machine. In this way, simulations using CSA can be run on machines where Python is not available.

Figure 3 shows the different entities in NEST involved in a user call to `CGConnect` in PyNEST. After setting the masks for the connection generator to tell it which neurons are local and which are remote (see section 2.1), the NEST kernel iteratively calls `next()`. This function returns source and target indices, and values for weight and delay if the arity of the connection generator is 2, until there are no more connections. The connections are internally established one by one calling NEST's basic `Network::connect()` function at the C++ level.

5. USING CONNECTION GENERATORS IN PyNN

PyNN (http://www.neuralensemble.org/PyNN; Davison et al., 2009) is a simulator-independent API for describing neuronal networks in Python. Given a PyNN/Python model description, the user can choose which simulator to use without needing to change the model script. This is achieved through a set of *simulator backends*. Each backend is a Python module that implements the API for a specific simulator, for example by providing a mapping from standard model names and units in PyNN to simulator-specific ones.