Semi-declarative model specification in Python

James A. Bednar

The University of Edinburgh

jbednar@inf.ed.ac.uk

http://homepages.inf.ed.ac.uk/jbednar

Declarative model specifications

NeuroML, NineML, and similar initiatives allow (XML-based) declarative specifications for models:

> "there exists an X of type Y with parameters P1, P2, ... Pn" (Goddard et al. 2001)

The obvious converse of a declarative specification is imperative computer code:

```
do A; do B; do C
```

(either domain-specific (hoc, SLI, etc.) or domain-general (Matlab, C++))

Declarative == good

Declarative specifications are extremely important for:

Sharing: Same model running on different simulators

Comparing: How do these models differ?

Maintaining: Extend/fix/improve code, same spec

Evolving: Simple declarative specs can cover a wide range of different models using same codebase

Communicating: What is this model, in terms of building blocks we already know about?

"there exists an X of type Y with parameters P1, P2, ... Pn"

So, everyone's models are always declarative, right?

Well... Why not?

- Models are actually made of code; new models (usually? often?) need new code.
- Very often you want to mess with the existing code a bit, try out some new ideas, etc.
- Fully declarative specifications can be extremely verbose – no problem for an XML parser, but no good for people.
- Sometimes (**only sometimes!**) code is just clearer.

Alternative approach

One approach to meeting these requirements would be to use any common object-oriented programming language (interpreted or compiled) augmented by an extensive set of object classes which provide the high-level constructs. These classes would thereby extend the base language to turn it into a particular set of language elements. The clarity requirement suggests that a simple interpreted language (e.g. Python) would be more appropriate than a compiled language such as C++ or Java. But...

— Goddard et al. (2001)

Python code as (semi-) declarative specification

text: *there exists an X of type Y with parameters P1, P2, ... Pn*

XML: `<Y name="X" P1="2.6" P2="true" P3="Z">`

Python: `Y(name="X", P1=2.6, P2=True, P3="Z")`

I.e., Python can be used as a declarative specification by instantiating an object of class Y with attributes P1, P2, ...

OK, but...

code == bad, right?

Right:

- Code can freely mix imperative and declarative bits
- Imperative bits get hard to understand, compare, etc.
- Cannot generate XML version for arbitrary code
- External parser needs to be very elaborate to parse all of Python

Can we ensure that code is (mostly) declarative, getting some or all of the advantages without the limitations?

Use Parameters

Python attributes are poor substitutes for declarative parameters in XML, with no: • types, • descriptions, • range checking, • units, • dynamic values • inheritance • automatic extraction . . .

. . . unless you use Python Parameters (topographica.org) or Traits (enthought.org).

Python+Parameters allows fully declarative specifications (in principle with output to XML), imperative specifications, and anything in between – allows smooth transition to declarative spec as model matures, with benefits from every step.

Code without full Parameters

```
class Connector(object):

  def __init__(self, weights=0.0, delays=None,
               space=Space(), safe=True,
               verbose=False):
    self.weights = weights
    self.space   = space
    self.safe    = safe
    self.verbose = verbose
    ... self.delays ...
```

From the 15 Mar 2012 version of PyNN

https://neuralensemble.org/svn/PyNN/trunk/src/connectors.py

```
class AllToAllConnector(Connector):
  parameter_names = ('allow_self_connections',)

  def __init__(self,allow_self_connections=True,
               weights=0.0, delays=None, space=Space(),
               safe=True, verbose=False):
    """
    Create a new connector.

    `allow_self_connections` -- if the connector is ...
    `weights` -- may either be a float, a
                 RandomDistribution object, a list/...
    `delays`  -- as `weights`. If `None`, all synaptic
                 delays will be set to the global...
    `space` -- a `Space` object, needed if you wish to
               specify distance-dependent...
    """
    Connector.__init__(self, weights, delays, space,
                       safe, verbose)
    assert isinstance(allow_self_connections, bool)
    self.allow_self_connections = allow_self_connections
```

```
class FixedProbabilityConnector(Connector):
  parameter_names = ('allow_self_connections','p_connect'

  def __init__(self,p_connect,allow_self_connections=True
               weights=0.0, delays=None, space=Space(),
               safe=True, verbose=False):
    """
    Create a new connector.
    `p_connect` -- a float between zero and one. Each ...
    `allow_self_connections` -- if the connector is ...
    `weights` -- may either be a float, a
                 RandomDistribution object, a list/...
    `delays`  -- as `weights`. If `None`, all synaptic
                 delays will be set to the global...
    `space` -- a `Space` object, needed if you wish to
               specify distance-dependent...
    """
    Connector.__init__(self, weights, delays, space,
                       safe, verbose)
    assert isinstance(allow_self_connections, bool)
    self.allow_self_connections = allow_self_connections
    self.p_connect = float(p_connect)
    assert 0 <= self.p_connect
```

Code with Parameters

```
class Connector(param.Parameterized):
    weights = param.Parameter(0.0,doc="""
        May either be a float, a RandomDistribution...
    delays = param.Parameter(None,doc="""
        May either be a float, a RandomDistribution...
    space = param.ObjectSelector(Space(),class_=Space,
        doc="""Allows you to specify distance-...
    safe = param.Boolean(True)
    verbose = param.Boolean(False)
    def __init__(self,**params): ... self.delays ...

class ProbabilisticConnector(Connector):
    allow_self_connections = param.Boolean(True,doc="""
        If the connector is used to connect...

class AllToAllConnector(ProbabilisticConnector):...

class FixedProbabilityConnector(ProbabilisticConnector):
     p_connect = param.Number(0.5,bounds=(0,1),doc="""
         Probability with which each potential...
```

Parameters: Declarative *code*

- Ranges, types, etc. declared, not imperatively checked
- Much less duplication – most param specs inherited
- Much less code: no checking code (assertions), usually no `__init__` needed
- Makes assumptions explicit
- Type, default value, doc, range specified together – always match
- Rest of code never needs to check any of these
- Automatic help, generated documentation
- Yet Parameters are just Python attributes – rest of your code can stay the same

Provides a clear path to declarative model

- Model specification file can be purely declarative
- Or semi-declarative (add `for` loops and variables for repetitive structure)
- Or imperative (full Python code all over the place)

But at least in the first two cases it is easy to iterate over a bunch of nested Parameterized objects after instantiation and spit out matching purely declarative XML, Python code, JSON, neuroTools ParameterSets – whatever you like.

Summary

- Declarative specification is such a good idea it should be applied to code too
- Declarative model spec then almost comes for free
- Parameters is a completely general (not even science specific) pure-Python module for declarative code, with no external dependencies.
- Documentation:
  http://topographica.org/Reference_Manual/param-module.html
- Download (until we build a separate package):

svn co https://topographica.svn.sf.net/svnroot/topographica/trunk/topographica/param

- Traits package almost identical except for GUI packages supported

References

Goddard, N. H., Hucka, M., Howell, F., Cornelis, H., Shankar, K., & Beeman, D. (2001). Towards NeuroML: Model description methods for collaborative modelling in neuroscience. *Philosophical Transactions: Biological Sciences*, *356* (1412), 1209–1228.