pces/mrnes API

Overview

This document describes the formatting of pces/mrnes model input files, and serves as an API for creating those files. It has five sections. The first, Input Files ,describes the files, their roles, and the formats of the dictionaries they represent. The second, Method Codes, describes the central role that these codes play in the flow of the simulation. The third, Identity Sets , defines sets of strings that are used in the description of the input dictionaries. The fourth, Func Classes describes pces Func classes and the structure of initialization state contained in the input files for them. The final section Processing/Routing , describes how messages are routed through pces/mrnes models.

Input Files

The interface specification for a pces simulation names 8 files that contain the model specification. From the command line theses are

- -cp Names a file that defines the structure of the model's Computational Patterns.
- -cpInit Names a file that defines the initial state of model components.
- -topo Names a file that defines computer and network topology
- -map Names a file which describes the assignment of the application's computational elements to processors.
- -exp Names a file that contains a description of assigning performance parameters to network entities
- -funcExec Names a file that contains all function execution timing information needed by the model.
- -devExec Names a file that contains device operation timing information for routers and switches
- -experiment Names a file that describes a set of experiments to run together as a group defining an 'evaluation'

The sections below describe the format of each file, using yaml notion to describe lists, dictionaries, and to some extent data types. However this is not pure yaml, as we embed in this notion model-specific information about keys and values of some of the dictionaries. When a dictionary definition includes a word that is entirely in capital letters, it means that in the actual yaml file one expects a string to be in that location, and that the capitalized word represents a set of strings from which the string must come. In section Identity Sets we define the set associated with each such word. In addition, the value of a dictionary's key may be given by a keyword that is a mixture of upper and lower case characters. This keyword will be the name of a dictionary structure as it is defined in this document. The dictionary names are types from the go representation of those dictionaries in the pces and mrnes code bodies.

1. -cp cp.yaml

The cp.yaml file defines the structure of Computational Patterns.   The file as a whole is organized as a dictionary with a dictname attribute and a dictionary indexed by a CPTYPE identifier to associate with a CompPattern data structure.

1.1 CompPatternDict

```
dictname: string
patterns:
  CPTYPE: CompPattern
```

1.2 CompPattern

A pces computational pattern is basically a graph whose nodes describe computations and whose edges describe flow of messages between the functions that perform the computation. The structure of the CompPattern dictionary is given below.

```
  cptype: CPTYPE
  name: CPNAME
  funcs: [Func]
  edges: [CmpPtnGraphEdge]
  extedges:
    CPNAME: XCPEdge
```

Here we see that the dictionary has a key `funcs` whose value is a list of (yet-to-be-defined) Func dictionaries, and likewise the dictionary has a list of CmpPtnGraphEdge dictionaries.

A model may have multiple instances where the structural pattern is the same, and a modeler developing code may wish to leverage that somehow.   Correspondingly when we create the dictionary describing a computational pattern we give it both 'cptype' and 'name' attributes, allowing (but not requiring) them to be different.

A function is permitted to send messages to functions in different computational patterns, in addition to those in its own.   We differentiate between intra- and inter- pattern messages with different lists, those found through key `edges` and those found through key `extedges`.

1.3 Func

Computational activity is simulated to occur within the context of a Func.   The details of how/when a Func's activities take place, the associated simulated execution, the output response, and the state variables the Func uses are all detailed in other places in the model description.   To describe computational pattern as a graph we need to specify the identity of the node, through a pair whose key is `label`.   Func behavior is based on its 'class' and so a dictionary pair identifying the class is included here as well. The value associated with the `class` key must be declared and supported by the pces code base as a recognized class (see Section Func Classes).

```
class: FUNCLASS
label: FUNCLABEL
```

1.4 CmpPtnGraphEdge

In describing a potential information flow between Funcs, we need to identify the Func endpoints. In addition we include pairs that identify the strings that identify message type and method code. The message type here is a label used to differentiate an edge from others with the same source. Receipt of a message delivered to a Func through one of its ingress edges triggers a (simulated) computation and response, and the choice of those is allowed to depend on the identity of the source Func, the message type, and a 'method code' which selects one of the destination Func's possible responses.

The dictionary associated with a CmpPtnGraphEdge is given by

```
srclabel: FUNCNAME
msgtype: MSGTYPE
dstlabel: FUNCNAME
methodcode: METHOD
```

Two edges are distinct (and are permitted to co-exist) if they differ in any one of these attributes.

The string mapped to by the `srclabel` key must be referenced as the target of the `label` key in some Func in the `funcs` list of the CompPattern within which the CmpPtnGraphEdge is defined. The same is true of the string mapped to bey the `dstlabel` key.

We have constraints also on the string which is the value of the `methodcode` key. A fuller explanation of method codes and these constraints is given below in Method Codes .

1.5 XCPEdge

pces permits messages to flow from one CompPattern to another. As the description of a CmpPtnGraphEdge assumes the endpoint functions are co-resident in the same CompPattern, we use a different dictionary to describe external edges. It simply adds attributes naming the computational patterns associated with the source and destination functions, separately.

```
  srccp      CPNAME
  dstcp      CPNAME
  srclabel   FUNCNAME
  dstlabel   FUNCNAME
  msgtype    MSGLABEL
  methodcode METHOD
```

The string mapped to by the `srclabel` key must be referenced as the target of the `label` key in some Func in the `funcs` list of the CompPattern whose `name` is the target here of the `srccp` key. An entirely similar requirement exists on the string mapped to by the `dstlabel` key.

2.   -cpInit cpInit.yaml

2.1 CPInitListDict

The cpInit.yaml file holds data structures used by pces to initialize the run-time reprepentation of the model. Like cp.yaml, the overall file structure holds a dictionary CPInitListDict, this one being indexed by strings in CPNAME (rather than in CPTYPE) to map to a CPInitList dictionary.

```
dictname: string
initlist:
  CPNAME: CPInitList
```

2.2 CPInitList

A CPInitList dictionary holds serialized configuration strings for the Funcs of a given instance of a CompPattern, accessible through a dictionary that is indexed by the Func's `label` attribute. The format of the configuration string is defined by the class of the Func, but the specifics of the configuration depend on the specific identity of the Func. Identity and then format can be recovered by analysis of the `cfg` dictionary key.

In principle we allow for the configuration string to be a serialized form of JSON, but that has not been much tested. We lean heavily towards YAML through pces/mrnes.

A CPInitList also lists descriptions of all the messages exchanged between those Funcs.

```
name: CPNAME
cptype: CPTYPE
useyaml: bool
cfg:
  FUNCNAME: |
    SERIALCFG
msgs: [CompPatternMsg]
```

2.3 CompPatternMsg

The CPInitList descriptor includes a list of descriptions of messages received and sent by the CompPattern. The description holds the message type label we've seen before, as well as a boolean indicating whether the message is associated with a packet or a flow (the latter being used with multi-resolution traffic modeling).

```
msgtype: MSGTYPE
ispckt: bool
```

3. topo.yaml

File topo.yaml holds information describing the computers and networks in the pces/mrnes model. The highest level structure is a dictionary TopoCfg with a name attribute, and then lists of dictionaries for each device type.

3.1 TopoCfg

```
name: string
networks: [NetworkDesc]
routers: [RouterDesc]
endpts: [EndptDesc]
switches: [SwitchDesc]
```

3.2 NetworkDesc

A mrnes network description lists the string-valued identifiers of all the devices (endpoints, routers, switches) it encompasses. It has an key `netscale` whose value inexactly describes its size (e.g., "LAN", "WAN", "T3"), and a key `mediatype` whose value describes the physical media (e.g., "wired", "wireless"). The methodologies for initializing parameters of networks and devices may refer to modeler-defined "groups", e.g., indicating that the base basewidths for all networks belonging to group "UIUC" is 1000 Mbps. A network may belong to any number of groups the model includes.

```
name: NETWORKNAME
groups: [GROUPNAME]
netscale: NETSCALE
mediatype: MEDIA
endpts: [ENDPTNAME]
routers: [ROUTERNAME]
switches: [SWITCHNAME]
```

3.3 EndptDesc

In mrnes we use a general description of an "endpoint" to refer to any device that is performing computation. It might be a low-powered host or even lower-powered sensor, or may be an enterprise server. From the mrnes point of view the important thing is that it is a platform for executing computations and that it connects to at least one network. The processing simulated at an endpoint depends on the number of cores that are assumed to be in use (as the model supports parallel execution of computational tasks) and so the description names the number of cores. The endpoint has at least one network interface, possibly more, and so the EndptDesc includes a list of interface descriptions. Finally, the model of the CPU used by the Endpt is given, as this is needed when looking up the execution times of computations performed on that Endpt.

```
name: ENDPTNAME
groups: [GROUPNAME]
cpumodel: CPUMODEL
cores: int
interfaces: [IntrfcDesc]
```

3.4 RouterDesc

Like a NetworkDesc, a RouterDesc has a unique name identifier, a list of groups the modeler declares the router to belong to, and a descriptor of the router's model.   It also includes a list of descriptions of the network interfaces attached to the device.

```
name: ROUTERNAME
groups: [GROUPNAME]
model: ROUTERMODEL
interfaces: [IntrfcDesc]
```

3.5 SwitchDesc

The attributes of a SwitchDesc are identical to those of a RouterDesc.

```
name: SWITCHNAME
groups: [GROUPNAME]
model: SWITCHMODEL
interfaces: [IntrfcDesc]
```

3.6 IntrfcDesc

The interface description dictionary includes keys whose values identify the device type and instance to which it is attached, the name and type of media of the network to which it is attached. Like other components of a network it carries a list of group names the model may refer to when configuring parameters for the interface in the run-time representation.

```
name: string
groups: [GROUPNAME]
devtype: DEVTYPE
device: DEVNAME
cable: INTRFCNAME
carry: [INTRFCNAME]
wireless: [INTRFCNAME]
faces: NETNAME
```

Exactly one of the values of keys `cable`, `carry`, and `wireless` attributes is non-empty.   In the case of the interface being connected to a wireless network the `wireless` list includes the names of all the interfaces also on that network that a transmission from the interface may directly reach. If instead the interface connects to a wired network, the mrnes view is that the interface may be directly connected through a cable to some other interface, and if that is the case the `carry` attribute names that interface.   mrnes allows also the declaration that the media is wired, but that the level of detail in the model does not include declaration of cables and the interfaces they connect, but does allow one to name another interface on that same network to which transmissions are forwarded. This interface is named by the value of the `carry` key.

Finally, the name of the network the interface touches is expressed in the value of the `faces` key.

4. map.yaml

Every Func of every instance of a CompPattern in a pces model is mapped to some Endpt in the supporting mrnes network; this mapping is described in map.yaml. The file contains a dictionary with a pair assigning a name to the dictionary, and a dictionary 'map' that is indexed by name of computational pattern. The value so referenced is another data structure that includes a dictionary 'funcmap' that is indexed by the value associated with the Func's `label`, with a value being the name of the mrnes endpoint responsible for simulating the execution of the Func.

```
dictname: string
map:
  CPNAME:
    patternname: CPNAME
    funcmap:
      FUNCLABEL: ENDPTNAME
```

Every CompPattern listed in the `patterns` dictionary of cp.yaml's CompPatternDict dictionary must have a key in this dictionary's `map` dictionary. Given such a key, say "cpn", the value of `patternname` is of course "cpn", and its `funcmap` dictionary must have as keys exactly and only the strings referenced by the `label` key in all Funcs declared in cpn's CompPattern `funcs` list.

5. exp.yaml

When a mrnes model is loaded to run, the file exp.yaml is read to find performance parameters to assign to network devices, e.g., the speed of a network interface. mrnes uses a methodology to try and be efficient in the specification, rather than directly specify every parameter for every device. For example, we can declare wildcards, e.g., with one statement that every wireless interface in the model has a bandwidth of 100 Mbps.

The methodology is for each device type to read a list of parameter assignments. A given assignment specifies a constraint, a parameter, and a value to assign to that parameter. Only devices that satisfy the constraint are eligible to have that parameter assigned. Possible parameters vary with the device type. A table of assignable parameters as function of device type is given below.

Device Type	**Possible Parameters**
Network	latency, bandwidth, capacity, trace
Interface	latency, bandwidth, MTU, trace
Endpt	trace, model
Switch	buffer, trace
Router	buffer, trace

For a Network, the 'latency' parameter is the default latency for a message when transitioning between interfaces that face the given network. A value assigned is in units of seconds. 'bandwidth' quantifies modeled point to point bandwidth across the network between two specfied interfaces, 'capacity' quantifies the aggregate bandwidth the network can carry, and 'trace' is a flag set when detailed trace information is desired for traffic crossing the network.

For an Interface the 'latency' parameter is the time for the leading bit of a message to traverse the interface (in units of seconds), 'bandwidth' is the speed of the interface (in units of Mbps), MTU is the usual maximum transmission unit enforced by interfaces.

For an Endpoint the 'model' parameter refers to the endpoint `model` attribute in EndptDesc.

For Switch and Router the 'buffer' parameter places an upper bound on the total number of bytes of traffic that one of these devices can buffer in the presence of congestion.

The constraint associcated with an assignment is that a device match specifications in each of the named attributes of the device. The attributes that can be tested are listed below, as a function of the device type.

Device Type	Testable Attributes
Network	name, group, media, scale, *
Interface	name, group, device, media, *
Endpt	name, group, model, *
Switch	name, group, model, *
Router	name, group, model, *

To limit assignment of a parameter to one specific device, the constraint would indicate that a device's 'name' be compared to the value given in the constraint. At most one device will have the specified name and have the parameter assigned.   At the other end of the spectrum selecting wildcard * as the testable attribute means the match is satisfied.   One can make an assignment to all devices with the same model type by selecting 'model' as the attribute and the model identify of interest as the matching value. Every one of the network devices can be selected based on presence of a particular group name in its `groups` list (selecting group as the testable attribute and the group name of interest as the value).   Furthermore, the constraint may specify a number of testable attribute / value pairs, with the understanding the the constraint is met only if each individual matching test passes.

The modeler may place the parameter assignment statements in any order in the file. However, before application the lists are ordered, in a 'most general first' ordering.   Specifically, for any given constrained assignment there are a set of devices that satisfy its constraints.   We order the assignments in such a way that for a given parameter P and device type, if the constraints of assignment A1 match a super-set of those which match assignment A2, then A1 appears before A2 in the ordering. What this means is that if we then step linearly through the post-order listing of assignments and apply each, then a more refined assignment (i.e. one that applies to fewer devices) is applied later, overwriting any earlier assignment. In this way we can efficiently describe large swathes of assignments, applying parameters by wildcard or broad-brush first, and then refine that assignment more locally where called for.

The dictionaries that describe these assignments are found in file exp.yaml, in an encompassing dictionary ExpCfgDict.

5.1 ExpCfgDict

5.1 ExpCfgDict

The exp.yaml file contains a dictionary comprised of a name, and a dictionary of experimental parameter dictionaries, indexed by the name associated with the parameter.

```
dictname: string
cfgs:
  EXPCFGNAME: ExpCfg
```

5.2 ExpCfg

The ExpCfg dictionary has a name, and a list of dictionaries that describe parameters to be applied

```
name: string
parameters: [ExpParameter]
```

5.3 ExpParameter

An ExpParameter dictionary identifies the type of network object the parameter applies to, a list of attribute constraints that need to be satisifed for an object in that class to receive the parameter, the identity of the parameter to be set, and the value to be set to that parameter on all network objects of the named type that satisfy the attribute constraint.

```
paramobj: NETWORKOBJ
attributes: [AttrbStruct]
param: ATTRIB
value: string
```

Note again that one AttrbStruct dictionary specifies one constraint, the list of AttrbStruct dictionaries associated with dictionary key 'attributes' means that all those constraints must match for an object's `param` to be set by `value`.

5.4 AttrbStruct

A single AttrbStruct record identifies an attribute and a value that attribute must have for a match.

```
attrbname: ATTRIB
attrbvalue: string
```

6. funcExec.yaml

funcExec.yaml holds descriptions of function timings, dependent on the type of computer on which the execution occurs, and the length of the data packet being processed. A given Func in a CompPattern may perform different computations, depending on the source Func and message type of its input. We therefore call the simulated computations 'operations' and the timings file describes timings of different operations. The FuncExecList dictionary that funcExec.yaml holds contains a dictionary indexed by a string naming the timing,

mapping to a description of the timing

mapping to a description of the timing.

6.1 FuncExecList

```
listname: string
times:
  TIMINGCODE: FuncExecDesc
```

6.2 FuncExecDesc

Description of a timing includes the operation code name (which is referenced in the executing model code), a modeler-included 'param' attribute which may be used by model code to refine its operations, identity of the model of CPU on which the measurement was taken, the length of the data packet driving the computation, and the measured execution time (in seconds)

```
identifier: string
param: string
cpumodel: CPUMODEL
pcktlen: int
exectime: float
```

7. devExec.yaml

devExec.yaml holds a dictionary DevExecList of timings of routers and switches as they route, and switch. That library has a name, and a library 'times' which is indexed by a device operation code, like 'route', and 'switch'.

7.1 DevExecList

```
listname: string
times:
  OPCODE: [DevExecDesc]
```

The value of the times dictionary entry corresponding to the operation code is a list of device timing descriptions.

7.2 DevExecDesc

```
devop: string
model: DEVMODEL
exectime: float
```

The DevExecDesc description identifies the operation code (same as the key leading to this list), a string uniquely identifying the device on which the timing is taken, and the timing itself (in units of seconds).

Identity Sets

The specifications above frequently used a word all in upper case (an "Identity Set") in positions where in

The specifications above frequently used a word all in upper case (an "Identity Set") in positions where in conformant yaml one would find the type 'string'.   Each identity set represents a set of strings, and the application of the word representing the set is meant to convey the constraint that in actual expression of the dictionary, one of the strings in the identity set is chosen.   This does NOT necessarily mean that *any* string in that set might appear there, this technique for expression is not sophisticated enough to convey some dependencies and limitations that exist, but does help to express how declarations in one portion of the model are used in others.

Below we describe each identity set and define the set of strings it represents.

Identity Set	**Composition**
CPNAME	All values mapped to by 'name' key in CompPattern dictionaries (1.2)
CPTYPE	All values mapped to by 'cptype' key in CompPattern dictionaries (1.2)
MSGTYPE	All values mapped to by 'msgtype' key in CompPatternMsg dictionaries (1.4)
FUNCLASS	All values mapped to by 'class' key in Func dictionaries (1.3)
FUNCLABEL	All values mapped to by 'label' key in Func dictionaries (1.3)
METHOD	All values mapped to by 'methodcode' key in CompPatternMsg and XCPMsg dictionaries (1.4 and 1.5)
SERIALCFG	Result of serializing a json dictionary or yaml struct of a function's configuration
NETWORKNAME	All values mapped to by 'name' key in NetworkDesc dictionaries (3.2)
ENDPTNAME	All values mapped to by 'name' key in EndptDesc dictionaries (3.3)
CPUMODEL	All values mapped to by 'model' key in EndptDesc dictionaries (3.3)
ROUTERNAME	All values mapped to by 'name' key in RouterDesc dictionaries (3.4)
ROUTERMODEL	All values mapped to by 'model' key in RouterDesc dictionaries (3.4)
SWITCHNAME	All values mapped to by 'name' key in SwitchDesc dictionaries (3.5)
SWITCHMODEL	All values mapped to by 'model' key in SwitchDesc dictionaries (3.5)
INTRFCNAME	All values mapped to by 'name' key in IntrfcDesc dictionaries (3.6)
MEDIA	{"wired", "wireless"}
NETSCALE	{"LAN", "WAN", "T3", "T2", "T1"}
DEVTYPE	{"Endpt", "Router", "Switch"}
DEVNAME	Union of sets ENDPTNAME, ROUTERNAME, SWITCHNAME
GROUPNAME	All values in lists mapped to by key 'groups' in NetworkDesc, EndptDesc, RouterDesc

GROUPNAME	All values in lists mapped to by key 'groups' in NetworkDesc, EndptDesc, RouterDesc, SwitchDesc, and IntrfcDest dictionaries (3.2, 3.3, 3.4, 3.5, 3.6)
NETWORKOBJ	{"Network", "Endpt", "Router", "Switch", "Interface"
ATTRIB	{"name", "group", "device", "scale", "model", "media", "*"}
TIMING	All values used as keys in the 'times' dictionary of the FuncExecDesc dictionary (6.1)
OPCODE	{"route", "switch"}
DEVMODEL	Union of SWITCHMODEL and ROUTERMODEL
METHOD	Every Func class defines identifiers for the different computational activities its members may engage in, depending on the specifics of the message arrival that triggered them. METHOD is the set of these, although in practice a reference to a methodcode on an edge requirements that the class of the Func which is the destination of the edge must define that methodcode.
PROBDIST	Probability distribution, currently {"const", "exp"}

Func Classes

At a high level of abstraction every pces function execution looks the same.   A message is received from a Func through an edge that the source Func directs to the destination Func.   That edge has a `method code'. The receiving Func has tables indexed by method codes that identify which specific event handling routines to schedule in response to a message associated with that method code.   Those routines have the responsibility of advancing simulation time and of generating the output messages resulting from processing the message.

A given Func may admit multiple method codes, each implementing a different response. For example we will describe Funcs that generate messages and insert them into the network, but then which also note the receipt of those messages when they return. These two actions have different method codes.

We present here descriptions of Func classes used to implement the pces beta release example. They all are relatively simple, basically accepting a message, doing something with it, and passing the result in a single message to the next Func in a chain.

pces is capable of supporting more complex models, e.g., ones where the modeled processing time is a function of data carried along in a message, and ones where the edges selected for output messages are neither static nor limited to a single edge. A modeler who creates new Func class implementations can through these event handlers simulate whatever action and response is desired.

Below we describe each of the classes that exist already in pces , what they do, and the structure of their configuration dictionaries, which when serialized are included in the cpInit.yaml and srdCfg.yaml dictionaries.

connSrc

A Func of this class generates messages and pushes them further into the computational pattern, and can

A Func of this class generates messages and pushes them further into the computational pattern, and can receive notification that a message it generated has completed a round-trip and returned.   This means a connSrc Func has one behavior when it responds to internal initiation to generate a message, and a different behavior when it receives a returning message.

The model for message generation connSrc uses is the standard arrival process model using in queuing theory. Requests to initiate a message arrive (self initiated) at the Func, with inter-arrival delays between successive requests.   The delay is drawn from a probability distribution.   That distribution may be constant, e.g., generated a message every 1 second of simulation time, or have stochastic variability.   At the time of this writing pces allows either constant, or exponentially distributed inter-arrival times.

Every CompPatternMsg has to have a type, a packet length, and a message length. A connSrc Func looks up from its internal state description what those attributes are when it generates the message.

A connSrc Func can be configured to generate a fixed number of messages and then stop. Part of its configuration specifies a limit (if any) on the number of messages it will generated in the course of a single simulation run.

For every Func of type connSrc in every CompPattern that defines one, the cpInit.yaml file is expected to hold a string serialized from an instantiation of the connSrcCfg dictionary below, that specify the parameters just mentioned.

```
interarrivaldist: string
interarrivalmean: float
initmsgtype: string
initmsglen: int
initpcktlen: int
initiationlimit: int
trace: bool
route:
  METHOD: MSGTYPE
timingcode:
  METHOD: TIMING
```

The value associated with key `interarrivaldist` must presently be either "const" or "exp".   The value of `interarrivalmean` specifies the mean value of the random samples drawn, so in the case of a "const" distribution it gives the deterministic time between successive message initiations. The value of `initmsgtype` defines the message type of an initiation message.   That type needs to have been included in a CompPatternMsg dictionary (see 2.3) which is part of the `msgs` list in the CPInitList (2.1) for the ComputationPattern holding this Func. The values of keys `initmsglen` and `initpcktlen` are embedded in an initial message. The value of `initiationlimit` specified the total number of message initiations the Func will perform (if the simulation does not end before that, e.g. by having the simulation time advance beyond the declared termination time).

The mapping of method code to event processing routines that perform the simulation is embedded in the

The mapping of method code to event processing routines that perform the simulation is embedded in the pces simulator. However, the modeler can specify for a given Func which of the Func's outbound edges should carry the processed message, and can specify the execution cost by specifying a computation label found in the funcExec.yaml table. The dictionary which is the value of key `route` maps method codes to lists of message types. The processed message is passed out through the (presumably unique) outbound edge that carries as message type the value `route` maps to the method code. The dictionary which is the value of key `timingcode` maps a message codes to a timing label, i.e., the label that in funcExec.yaml represents a computation.

Run-time errors are raised if (a) the method code presented with a message for processing is not represented in the recipient Func's class, (b) there is not exactly one outbound edge with a message type label matching the one selected through the `route` dictionary, (c) there is no computational label listed as a key in the funcExecList `times` dictionary matching the one obtained through the `timingcode` dictionary, or (d) there is no FuncExecDesc dictionary that gives as CPUModel the same as the Endpt the Func is mapped to.

The boolean value of the`trace` key indicates whether detailed information about simulation activity at this Func should be gathered at run-time and written into a trace file.

The current pces implementation defines two method codes for connSrc, 'generateOp' and 'completeOp'. The former is associated with message generation, and the latter with receipt of returned messages. Processing associated with generateOp takes the following steps:

1. Sample a delay d from the inter-packet distribution, and schedule an event E to begin generation of a burst's messages d units of time in the future.
2. Schedule an event C to occur d units in the future (but after the one above) which checks to see if another message generation message should be delivered to the Func (on an edge where the Func is both source and destination).
3. The handler for event E executes, in which the message generation execution time e is looked up, and the service requirement for x units of execution time are passed to the Endpt's scheduler. That request is serviced on a FCFS basis among all concurrent requests, assuming concurrent processing made possible the presence of multiple cores in the CPU.
4. The event handler for C executes exactly d units of time after being scheduled. The handler compares the number of messages that have been initiated with the configuration threshold associated with the`iniitationlimit`key. If another message is to be generated, the event executing step 1 above is scheduled to execute immediately.
5. The event to handle the completed message generation executes y >= x units of time after the request is made in step 3, inequality reflecting queueing delay in service the request. That event handler looks up the edge out of Func that the `route` configuration indicates should be taken. If the destination Func is mapped to the same Endpt, the event handler implementing step 1 is scheduled to occur at that Func, immediately. If the destination Func is off-processor the message is handled off to the mrnes network simulator which will impose additional delays as the message makes its way across the network.

Response to a message accompanied by the 'completeOp' method code is much simpler.

1. Look up the computation label for FuncExecList as the value of the Func's timingcode["completeOp"], and

1. Look up the computation label for FuncExecList as the value of the Func's timingcode["completeOp"], and then look up the execution requirements x. This delay might model any number of things an application might on the message's return, e.g., store a record in a database. In our model this is distilled down to a computational service requirement of x, which is requested from the Endpt's scheduler.
2. The event to handle the service completion executes some y >= x units of time later, again, inequality reflecting any queuing delay.   Note that a message m returning to the Func complete for service with message generation tasks requested sometime after m was initiated.
3. The round-trip completion is noted by the absence of a post-processing message, and so the completion of the chain of executions is recorded for post-simulation-run processing.

processPckt

An instance of a Func from processPckt class processes a received packet, and sends it along out a single egress edge. The execution time associated with processing and the egress edge selected depend on the method code of the message whose receipt triggers the processing. The processPckt class uses the same method and data structures as connSrc to select the modeled execution time, and to select the outbound edge over which the processed message is sent.

The configuration dictionary for processPckt funcs is given below.

```
route:
  METHOD: MSGTYPE
timingcode:
  METHOD: TIMING
trace: bool
```

As with connSrc, the value `trace` selects whether detailed information about these calls are included in a trace file.

This Func class is simple, but general.   We use it in the beta release example model to represent different encyption operations. It is a matter of configuration.   If the simulation is modeling the use of RC6 with a 3072 byte key, then with a Func that represents encryption we configure the value of timingcode["processOp"] to be the computation label associated with doing RC6 encryption with a 3072-byte key.   Elsewhere in the model we use a processPckt Func to model the decryption, the difference being that timingcode["processOp"] is configured to have in that instance a computation code that refers to decryption.

The sequence of actions taken when a message arrives at a processPckt Func is

1. A computation label is extracted as the value of `timingcode["processOp"]`, and an execution delay requirement x is extracted from the `times` dictionary of FuncExecList. The Func's Endpt scheduler is requested to provide x units of service.
2. At some time y >= x after the scheduling request is made, the event processing the completion of the service request executes.   Based on the Func configuration of `route`, an outbound edge from the Func is selected.   If the destination Func is co-resident with the source on the same Endpt, then the arrival of the message is scheduled to occur immediately at the destination Func. Otherwise the message is handled off

   to the mrnes network simulator to deliver (with additional delays) to the destination Func.

[illegible]

cycleDst

A Func of the cycleDst class generates messages, but according to a particular pattern asked for by one customer. The Func is configured to hold a list of N labels of CompPatterns that exist somewhere in the topology and are destinations for the messages.   Message generation goes through a number K cycles.   In each cycle, each of the N destinations is sent a burst of B messages, with the destinations being visited cyclically. So for instance if K=2, N=3 and B=4, the first four messages are sent to the first destination in the list, the second group of four messages are sent to the second destination in the list, and the third set of four messages are sent to the third destination in the list.   The cycle then repeats with the fourth set of four messages going to the first destination in the list, and so on, with the generation process ending after the second cycle through the destination list.

As with the connSrc generation process there may be a delay between successive packet generation initiations. The delay may be 0, constant, or randomly generated. As with the connSrc class the only probability distribution currently implemented is exponential.   The delays beween message in a burst have a different characterization than the delay between the last message in a burst and the first message in the next burst of the same cycle, and the delay between the initiation of the last message in one cycle and the first message of the next has its own characterization.

There is an important distinction at play between message initiation and actual message generation. The mrnes model for advancing simulation time includes computational requests queuing for service at a bank of CPU cores. The message initiation pattern described above speaks to when a request for a message to be generated is created, and is independent of the times when messages are actually generated. So for instance, if the inter-message initiation delay within a burst is zero, then at the same instant in simulation time all B requests in a burst for message generation are delivered to the CPU simulation.

When a message is first created for initiation, it is given a header that names the Computational Pattern and Func which generated it, and a Computational Pattern and Func that is the message's intended target. That information is like an IP header, but unlike an IP header is not required for routing.   For these models where a message is sent to some Func and a return is expected, on the outbound leg the header would name the target Func, and the return message carry the original source Func as the destination.   We give the modeler control over the header by including in the processPcktCfg configuration dictionaries that when populated, change the Func destination header after the processing of the message at that Func is completed. Like other configuration options, the method code selects

The configuration dictionary for a cycleDst Func is given below.

```
pcktdist: PRBDIST
pcktmu: float
burstdist: PRBDIST
burstmu: float
cycledist: PRBDIST
cyclemu: float
cycles: int
initmsgtype: MSGTYPE
initmsglen: int
```

```
initpcktlen: int
dsts: [FUNCLABEL]
trace: bool
route:
  METHOD: MSGTYPE
timingcode:
  METHOD: TIMING
tgtcp:
  METHOD: CPNAME
tgtlbl:
  METHOD: FUNCLABEL
```

The value associated with the `pcktmu` key is the mean inter-initiation time between messages within a burst, the value associated with the `burstmu` key is the mean inter-initiation time between the last message of a burst and the first message of the next burst, and the value associated with the `cyclemu` key is the mean time between the last message of a cycle and the first message of the next cycle. `cycles` tells how many cycles should be performed. Initial message characteristics are described just as they are in the connSrc configuration, `trace`, `route`, and `timingcode` entries are likewise identical in function. The `dsts` list contains the `label` values of the destination Funcs involved in the cyclic message generation.

Dictionaries `tgtcp` and `tgtlbl` are both indexed by method code, and as described earlier will change the target coordinates of a message leaving the Func. Note however that it is legal for these dictionaries to be empty, or simply lack entries for some method codes. Their contents are applied only when specified.

Like srcConn, the 'generateOp' and 'completeOp' method codes are defined. The cycleDist processing associated with 'generateOp' is at one level of abstraction identical to that of connSrc. The key differences are that the delays between message initiations have more complexity, and that destination Func labels are chosen with each generation. The processing associated with the 'completeOp' method code is identical to that of a connSrc Func.

finish

A model should have a finish class Func be the terminus of a message's traversal. It performs internal bookkeeping that records the delay between message initiation and its arrival at the finish Func, and stops the propagation.

The only element of the finish configuration dictionary is a flag indicating whether a message arrival at the Func should be logged into a detailed trace file.

```
trace: bool
```

Processing/Routing

The logic of delivering messages from one Func to another has two layers. At the application layer the routing is governed by the information applied to Func outbound messages.   When a message is presented to a Func, pces determines what computational logic will be applied. The message delivery is associated with edge directed from one Func to the recipient, and combination of source Func identity (both its label and the computational pattern that holds it) and message type on that edge define the method code to use.   This mapping is inferred at start-up time by analysis of all the CompPatternEdge and XCPEdge dictionaries defined by the model in cp.yaml.

To handle the message receipt, pces scheduled execution of an event (EnterFunc) that starts the execution of every Func response, and provides the identity of the Func receiving the message, and the particular message being delivered. EnterFunc looks up the method code, and uses this to look up the method-code-specific routine that starts the message processing. This routine may analyze and alter the Func's internal configuration and state, and use the results of that to look up an execution time requirement.   It then requests from the hosting Endpt CPU that amount of service, and in the request records the specific method code, message, and identity of the event handler to schedule to report the service request completion.

When the service request completion handler executes it handles generation of messages that model the Func's response. It may generate any number of messages, including none. For each message it selects one of the Func's egress edges to 'carry' it to the next Func, and notes the destination Func's computation pattern and label on the message. Finally, it places all the response messages in a data structure accessible outside of the event handler, and schedules the pces event handler ExitFunc to execute immediately.

The corresponding ExitFunc execution acquires the list of responding messages. For each it determines whether the destination Func resides on the current Endpt CPU, and if so schedules the EnterFunc routine to execute immediately. Here then the model edges define application level routing within an Endpt. If it happens that the destination Func resides on a different Endpt, then ExitFunc hands the message off to the mrnes network simulator. The handoff request names the identities of the present and destination Endpts, and the global identity of the destination Func. It also names the pces routine ReEnter for the mrnes simulator to schedule when the message has traversed the network and arrived at its destination Endpt.

The network route the message takes and the accued simulation time are all governed by the mrnes model. At present mrnes finds the shortest path between source and destination Endpts, where the waypoints on the path are switches and routers, and edges in the network graph are defined by the declared connections between device interfaces (see 3.6). The cost of every edge it assumed to be 1. mrnes discovers paths on a per-demand basis, and caches the results of the calculation so (at the cost of memory) the path between any given source and destination Endpt is done at most once.