mrnes API

This document describes the formatting of mrnes model input files, and serves as an API for creating those files. **mrnes** is a library of data structures and methods that discrete-event simulators such as **pces** may include to represent the communication of application layer messages they simulate. The API we document here is for three files. One describes the topology of the modeled network, the other the time costs associated with performing operations such as routing and switching.

For context, as preparation for this document and an illustration of how **mrnes** can be integrated into another simulation framework, one should read MRNES-Introduction and PCES-Introduction .

An **mrnes** modeler interacts with the package through methods defined by an **mrnes** struct named 'NetworkPortal', using structs also defined by **mrnes**. The methods, types, and structs below are all given in Go, as a modeler who is building an application that uses the **mrnes** package will be defining and using these entities from an application written in Go.

The **mrnes** API has eight basic components.

- Description of input files ('topo.yaml', 'exp.yaml', 'devExec.yaml') that describe topology and architectural performance parameters.
- A set of APIs for golang methods used assemble data structures holding descriptions of network objects, and transform these into the input files needed to assemble a model.
- A method mrnes.CreateTraceManager that creates in **mrnes** memory space a data structure used by code in **mrnes** and in code that obtains access to that data structure and uses it to accumulate trace information about a simulation experiment. All the methods of the trace manager are part of the API.
- Methods mrnes.LoadTopo, mrnes.LoadDevFile, and mrnes.LoadStateParams' that are called from the main simulation program to read in the input files and initialize internal data structures in preparation for a simulation run.
- A method 'endpt.EndptState.Schedule' for an **mrnes** 'endpoint' (host, server, sensor, any device that simulates the execution of some program in the overall model) for submitting a description of computational workload whose execution requirements are defined in code that calls this method.
- A method 'mrnes.CreateBckgrndFlow' for creating a 'background flow' that consumes network resources, and by doing so impacts the performance behavior of foreground application workflows that have greater specificity. A created flow can be removed calling 'mrnes.RmBckgrndFlow'.
- A method 'mrnes.EnterNetwork' for presenting a message to the network for communication.
- Definition of the data structure returned to the program that called 'mrnes.Enternetwork' when the message is delivered.

1. Format of topo.yaml dictionaries

Before speaking to use of **mrnes** methods to build input files, we describe first what **mrnes** expects the topology input file (nominally topo.yaml) to contain. The descriptions below are of dictionaries defined in TopoCfg, using YAML-like notation. **mnres** accepts JSON descriptions as well, so the definitions are to the largest extent identifying the key-value associations one finds in both YAML and JSON. Note also that brackets [ ] denote a list, and what we provide below are labels we define in the Appendix Identity Sets. The highest level description of the contents of topo.yaml is the 'TopoCfg' dictionary below. It is simply a name (for reference purposes), then lists of descriptions of networks, routers, endpoints, and switches that make up the overall model.

An individual topology is a collection of interconnected networks, devices that are embedded in them, and interfaces that are embedded in those devices. Inside of the **mrnes** package the dictionary expressed in topo.yaml follows is a representation of the structure

```
type RtrDescSlice []RouterDesc
type EndptDescSlice []EndptDesc
type NetworkDescSlice []NetworkDesc
type SwitchDescSlice []SwitchDesc
type FlowDescSlice   []FlowDesc

type TopoCfg struct {
    Name     string           `json:"name" yaml:"name"`
    Networks NetworkDescSlice `json:"networks" yaml:"networks"`
    Routers  RtrDescSlice     `json:"routers" yaml:"routers"`
    Endpts   EndptDescSlice   `json:"endpts" yaml:"endpts"`
    Switches SwitchDescSlice  `json:"switches" yaml:"switches"`
    Flows    FlowDescSlice    `json:"flows" yaml:"flows"`
}
```

When represented in a file generated by Golang standard YAML support libraries, the key names are all lower-case (but need in the type definition to be upper-case in order to be recognized as exportable). To generate this representation it is necessary to generate a 'TopoCfgFrame' structure using

```
func CreateTopoCfgFrame(name string) TopoCfgFrame
```

to create a 'TopoCfgFrame' variable 'tcf', then create and link networks, routers, switches and endpoints to create representations (to be described below) that are then added to 'tcf', then call 'tcf.Transform' to create a variable 'topocfg' of type 'TopoCfg', and finally, save that description calling 'topocfg.WriteToFile(filename)' which puts a YAML description into the file whose path is given as the input argument.

We next describe some of the intermediate methods in more detail.

`func CreateTopoCfgFrame(name string) TopoCfgFrame` creates a structure into which the topology components we create are placed.

```
type TopoCfgFrame struct {
    Name     string
    Endpts   []*EndptFrame
    Networks []*NetworkFrame
    Routers  []*RouterFrame
    Switches []*SwitchFrame
    Flows    []*FlowFrame
}
```

The rest of the assembly is focused on the creation of network objects and on embedding device objects into networks. We have seen already function 'CreateNetwork' which creates a network frame variable, then used as a receiver structure for network device descriptions. Once a network's assembly is completed we call 'AddNetwork' :

- `func (tf *TopoCfgFrame) AddNetwork(net *NetworkFrame)` appends network 'net' to 'tf.Networks', provided that the network is not already present (or that any network with the same name is present).

Once all the topology objects have been created, linked, and networks added to the topology configuration frame, we create a variable 'topocfg' of type 'TopoCfg' by calling 'Transform':

- `func (tf *TopoCfgFrame) Transform() TopoCfg` transforms the TopoCfgFrame into a text-based serializable form, 'TopoCfg', which we have seen above.

Finally, we save that description calling `topocfg.WriteToFile(filename)` which puts a YAML description into the file whose path is given as the input argument. We have nominally called that file topo.yaml, but of course the file name and its path are left to the user to specify.

In reading the function descriptions below it is helpful to know a technique used by this library, which is to create data structures representing topology objects (whose type names normally end in the substring 'Frame') with fields that may be pointers to other topology objects, and then when completed transform the data structure into serializable form, with a type name that normally ends in 'Desc'.

Before getting into the method definitions one should know that all the 'Frame' structures of all the devices in the network satisfy the 'NetDevice' interface, and that some arguments to the methods we document below include reference to objects satisfying that interface.

```
type NetDevice interface {
  DevName() string  // returns the .Name field of the struct
  DevID() string    // returns a unique (string) identifier for the struct
  DevType() string  // returns the type ("Switch","Router","Endpt","Network")
  DevInterfaces() []*IntrfcFrame  // list of interfaces attached to the NetDevice, if any
  DevAddIntrfc(*IntrfcFrame) error // function to add another interface to the netDevice
}
```

NetworkFrame

The definition of a NetworkFrame is

```
type NetworkFrame struct {
    Name string
    Groups []string
    NetScale string
    MediaType string
    Routers []*RouterFrame
    Endpts []*EndptFrame
    Switches []*SwitchFrame
}
```

This looks very much like a 'NetworkDesc' structure seen before, save that the lists of routers, endpoints, and switches are pointers to data structures rather than strings. This reflects our experience that it is more straightforward to build the topology description using pointers, but then convert that representation to a strictly string-based form for the purposes of writing the description out to file.

Methods associated with a 'NetworkFrame' are

- `func CreateNetwork(name, NetScale string, MediaType string) *NetworkFrame` returns a pointer to a 'NetworkFrame' structure that holds a network description. 'name' is any string that uniquely identifies the network across the entire model, and 'NetScale' is a string from the set {"LAN", "WAN", "T3", "T2", "T1"}, and 'MediaType' is (currently) a string from the set {"wired", "wireless"}. Methods associated with a 'NetworkFrame' are
    - `func (nf *NetworkFrame) FacedBy(dev NetDevice) bool` . This method tells the caller whether the network represented by the receiver type argument 'nf' is faced by the NetDevice object given as an argument. In the specification only network interfaces (of type 'IntrfcFrame') have a 'Faces' attribute, and 'FacedBy' determines whether any network interface bound to 'dev' has a value for 'Faces' that is identical to 'nf' 's name.
    - `func (nf *NetworkFrame) AddGroup(groupName string)`. All network objects carry a list of strings in a slice called 'Groups', which are user-defined tags to create associations. This method includes another such tag, if it is not already associated with the NetworkFrame.
    - `func (nf *NetworkFrame) IncludeDev(dev NetDevice, intrfcType string, chkIntrfc bool) error` . NetworkFrames accumulate pointers to network devices that are associated with the network, such as endpoints, switches, routers. 'IncludeDev' adds the network object 'dev' to the appropriate list. When the 'chkIntrfc' flag is set an 'IntrfcFrame' object is created if none of 'dev''s existing interfaces face 'nf', and is bound to network device 'dev'. Input parameter 'intrfcType' is an argument passed to the method 'mrnes.CreateIntrfc' when creating that interface. An error is returned if 'chkIntrfc' is false and 'dev' has no interface that faces 'nf'.
    - `func (nf *NetworkFrame) AddRouter(rtrf *RouterFrame) error`. This method adds a router to the NetworkFrame 'nf' if not already present, returning without error if it is. If the router is not present the caller is asserting that it already has an interface that faces 'nf', and returns an error if that assumption is not satisfied.

- `func (nf *NetworkFrame) Transform() NetworkDesc`. This method creates a pointer-free representation of the network, in type 'NetworkDesc', by stepping through the fields of the 'nf' NetworkFrame and transforming pointers to network objects into strings that hold the name of the pointed to object.

IntrfcFrame

Network interfaces are the glue that hold them together, and so naturally are a core element of **mrnes** network models. As we are building a network we create structures of type 'IntrfcFrame', described below.

```
type IntrfcFrame struct {
    Name string
    Groups []string
    // type of device that is home to this interface, i.e., "Endpt", "Switch", "Router"
    DevType string
    MediaType string

    // name of endpt, switch, or router on which this interface is resident
    Device string
    // pointer to interface (on a different device) to which this interface
    // is directly (and singularly) connected. This interface and the one pointed to
    // need to have media type "wired"
    Cable *IntrfcFrame

    // slice of pointers to interface (on a different device) to which this interface
    // is directly connected, but not through a Cable.
    // Each interface and the one pointed to need to have media type
    // "wired", and have "Cable" be empty
    Carry []*IntrfcFrame

    // A wireless interface may connect to may devices,
    // this slice points to those that can be reached
    Wireless []*IntrfcFrame

    // name of the network the interface connects to.
    Faces string
}
```

'Name' is a string that uniquely identifies the interface, across all network devices expressed in the model. 'Groups' is as it is in networks, a slice of tags that help characterize the interface. 'DevType' speaks to the type of device the interface is bound to, 'Endpt', 'Switch', or 'Router', and 'Device' is the unique name of that device. The connection between two wired interfaces is either 'Cable' or 'Carry', the former being declared when we know the latency and bandwidth of the wired connection, and 'Carry' indicating that there is *some* wired connection between them, not necessarily a single cable, and that attributes of that connection depend on the network both interfaces must be facing. An interface may have declared multiple 'Carry' style connections and so these are included in a slice. Likewise, an interface to a wireless network will be able to communicate with a number of other wireless interfaces, and so here we use a slice 'Wireless' to hold pointers to them all.

Finally, every interface connects to some network, and the 'Faces' string holds the unique name of that network.

Methods associated with an 'IntrfcFrame' include

- `func CreateIntrfc(device, name, devType, mediaType, faces string) *IntrfcFrame` is a constructor whose 'device' argument is the unique name of the device to which the interface is bound, 'name' is the name given to the interface (and if empty, a default name is created), 'devType' names the device type where the interface is bound, 'Endpt', 'Switch', or 'Router', 'mediaType' is 'wired' or 'wireless', and 'faces' is the unique name of the network within which the interface makes its connections.
- `func DefaultIntrfcName(device string) string` creates a unique name for an interface, of the form 'intrfc@name[.n]' where 'name' is the unique name of the network device the interface is bound to, and 'n' is a positive integer.
- `func (ifcf *IntrfcFrame) AddGroup(groupName string)` adds a tag to the interface's 'Groups' slice, if the named 'groupName' is not already present. The method returns silently without doing anything if it turns out that the 'groupName' is already present.
- `func (ifcf *IntrfcFrame) Transform() IntrfcDesc` transforms the referenced 'IntrfcFrame' into a form suitable for storage in a text file. It copies the attributes of 'ifcf' that are already strings, and substitutes object names for the pointers to those objects in 'Cable', 'Carry', and 'Wireless' attributes.*RouterFrame*

RouterFrame

The type description of a RouterFrame is

```
type RouterFrame struct {
    Name       string
    Groups     []string
    Model      string
    Interfaces []*IntrfcFrame
}
```

This is very closely structured as the 'RouterDesc' type seen earlier. The methods associated with it are

- `func CreateRouter(name, model string) *RouterFrame` is a constructor whose arguments are the globally unique name for the router, and a string that identifies the model of the router for the purposes of looking up the route operation time cost for that model.
- `func (rf *RouterFrame) DevName() string` is a method required for the 'NetDev' interface, and returns the value 'rf.Name'.
- `func (rf *RouterFrame) DevType() string` is a method required for the 'NetDev' interface, and returns the value "Router".

- `func (rf *RouterFrame) DevID() string` is a method required for the 'NetDev' interface, and returns the value 'rf.Name'.
- `func (rf *RouterFrame) DevModel() string` is a method required for the 'NetDev' interface, and returns the value 'rf.Model'.
- `func (rf *RouterFrame) DevInterfaces() []*IntrfcFrame` is a method required for the 'NetDev' interface, and returns the slice 'rf.Interfaces'.
- `func DefaultRouterName() string` creates a unique string identifier for a router, of the form "rtr[n]" where n is some positive integer. If when 'CreateRouter' is called the 'name' argument is empty, this method is called to create the name.
- `func (rf *RouterFrame) AddIntrfc(intrfc *IntrfcFrame) error` is called to append the interface argument to 'rf''s slice of InterfaceFrames provided that there is not already an InterfaceFrame reference in that list with a name that is identical to the name of 'intrfc'. In case of a match an error is returned.
- `func (rf *RouterFrame) DevAddIntrfc(iff *IntrfcFrame) error` is a method required for the 'NetDev' interface and simply returns 'rf.AddIntrfc(iff)'.
- `func (rf *RouterFrame) AddGroup(groupName string)` appends the string argument 'groupName' to the slice 'rf.Groups', if not already present. If already present the request is simply ignored.
- `func (rf *RouterFrame) Transform() RouterDesc` is called to create the representation of the router that can be written to file. It copies 'rf.Name', 'rf.Groups', and 'rf.Model' directly as these are all string based, and replaces every pointer to an InterfaceFrame with the result of call the InterfaceFrame method 'Transform', the result of which is a string-based 'InterfaceDesc' structure.
- `func (rf *RouterFrame) WirelessConnectTo(dev NetDevice, faces string)`. A wireless network in **mrnes** requires the presence of a router that serves as a wireless hub. Endpoints may connect with the hub, either via a cable or through the wireless medium. To create a wireless network a user of this model-building library must create a router which will serve as the hub, and then connect other devices to it. 'WirelessConnectTo' provides the logic for making that connection. The receiver struct pointer 'rf' is the hub, the argument 'dev' identifies the network device to be wirelessly connected, and the 'faces' argument names the network involved in the connection. 'WirelessConnectTo' looks for interfaces at both the router and the device that face the named network with a wireless mediaType. If either does not exist one is created and bound to the device that requires it.

SwitchFrame

For every method associated with 'RouterFrame' listed above, there is a corresponding method associated with 'SwitchFrame', with exactly the same names (substituting 'Switch' for 'Router' where appropriate) and same input arguments. The sole different is

- `func DefaultSwitchName(label string) string` which produces an output of the form "switch(label).%n" where n is some positive integer.

EndptFrame

The EndptFrame structure looks a great deal like SwitchFrame and RouterFrame

```
type EndptFrame struct {
    Name       string
    Groups     []string
    Model      string
    Cores      int
    Accel      map[string]string
    Interfaces []*IntrfcFrame
}
```

The sole difference is its inclusion of a 'Cores' attribute to allow a user say more about the CPU's capabilities than just the Model identifier, and inclusion of a 'Accel'dictionary to describe the presence of on-board hardware accelerators on the endpoint.

mrnes models may use the general Endpt structure for a variety of devices that are similar in that they have interfaces, and cores, but may have additional user-specified functionality that depends on refinement of their description. So we offer six constructors for Endpts that differ only in a tag they include to classify the Endpt as being a 'Host', 'Node', 'Sensor', 'Srvr', 'EUD', or have no constructor at all. Each constructor accepts as argument a name, a CPU model, and a number of cores. The default name constructed in each case differs according to the type.

- `func CreateHost(name, model string, cores int) *EndptFrame`. Puts 'Host' in the EndPt's 'Groups' slice, and has a default name of the form "Host-Endpt.(n)" where n is a positive integer.
- `func CreateNode(name, model string, cores int) *EndptFrame`. Does not put a tag in the 'Groups' slice, and has a default name of the form "Node-Endpt.(n)" where n is a positive integer.
- `func CreateSensor(name, model string, cores int) *EndptFrame`. Puts 'Sensor' in the EndPt's 'Groups' slice, and has a default name of the form "Sensor-Endpt.(n)" where n is a positive integer.
- `func CreateEUD(name, model string, cores int) *EndptFrame`. Puts 'EUD' in the EndPt's 'Groups' slice, and has a default name of the form "EUD-Endpt.(n)" where n is a positive integer.
- `func CreateEndpt(name, etype string, model string, cores int) *EndptFrame` is the method called by all the other Endpt constructors, where they pass along the values of 'name', 'model', and 'cores' they passed themselves, but then specify 'etype' to be "Host", "Sensor", "Srvr", or "EUD" according to their own tag.
- `func (epf *EndptFrame) AddIntrfc(iff *IntrfcFrame) error` behaves exactly as so the versions of 'AddIntrfc' for switches and routers, including the interface frame if it is not already present (or iff.Name matches the name on some InterfaceFrame already bound to 'epf'.) As with the switch and router versions of 'AddIntrfc' an error is returned if the offered interface is already found to be present.
- `func (epf *EndptFrame) DevName() string` is a method required by the 'NetDev' interface and returns the value 'epf.Name'.
- `func (epf *EndptFrame) DevType() string` is a method required by the 'NetDev' interface and returns the value "Endpt".
- `func (epf *EndptFrame) DevInterfaces() []*IntrfcFrame` is a method required by the 'NetDev' interface and returns the slice 'epf.Interfaces'.

- `func (epf *EndptFrame) DevAddIntrfc(iff *IntrfcFrame) error` is a method required by the 'NetDev' interface and simply returns the results of a call to 'epf.AddIntrfc(iff)'.
- `func (epf *EndptFrame) Transform() EndptDesc` is called to create the representation of the endpoint that can be written to file. It copies 'epf.Name', 'epf.Groups', 'epf.Model' and 'epf.Cores' directly as these are all string based, and replaces every pointer to an InterfaceFrame with the result of call the InterfaceFrame method 'Transform', the result of which is a string-based 'InterfaceDesc' structure.
- There are methods with obvious functionality to indicate whether an endpoint is a particular type, to. include a particular type in an endpoint's 'Groups' list, to set and get the cores and model attributes: `func (epf *EndptFrame) SetEUD()`, `func (epf *EndptFrame) IsEUD() bool` ,`func(epf *EndptFreame) AddAccel()`,`func (epf *EndptFrame) SetHost()`, `func (epf *EndptFrame) IsHost() bool`, `func (epf *EndptFrame) SetSrvr()`, `func (epf *EndptFrame) IsSrvr() bool`, `func (epf *EndptFrame) SetCores(cores int)`.

FlowFrame

The definition of a FlowFrame is

```
type FlowFrame struct {
    Name string
    Groups []string
    SrcDev string
    DstDev string
    ReqRate float64
    Mode string
    FrameSize int
}
```

This is exactly the 'FlowDesc' structure seen before.

Methods associated with a 'FlowFrame' are

- `func CreateFlowFrame(name,srcDev, dstDev, mode string, reqRate float64, frameSize int ) *FlowFrame` returns a pointer to a 'FlowFrame' structure that holds a flow's description. 'Name' is any string that uniquely identifies the flow across the entire model. 'Groups' is a list of tags, as with other network devices. 'SrcDev' and 'DstDev' are the Name attributes of the endpoints that initiate and receive the flow. 'ReqRate' is the bandwidth---expressed in units of Mbit/sec---requested for the flow. 'Mode' is one of 'pckt', 'elastic-flow', 'inelastic-flow', each of which describes the nature of the flow. 'pckt' means the flow is represented as a stream of individual packets. 'inelastic-flow' means that if the request to establish the flow is requested, the flow is assured to be delivered at the rate provided as 'ReqRate'. 'elastic-flow' means that the flow may be established at a rate that is smaller than that expressed in 'ReqRate', depending on bandwidth availability. In the case that 'Mode' is 'pckt', 'ReqRate' gives the aggregate rate that packet bits are inserted into the network. The per-packet inter-arrival time is chosen to yield this aggregate rate. Finally, 'FrameSize' is the number of bytes assumed to be in the packets, either when 'Mode' is 'pckt' or when the impact of the represented 'virtual' packets are considered at interfaces at devices along the flow's route.

- The single methods associated with a 'FlowFrame' of significance is

  `func (ff *FlowFrame) AddGroup(groupName string)`.

  This enables a user to augment the list of a flow's group tags with another.

Connections

The methods documented so far give us ways to create network objects. Other methods support testing for the existence of connections already established, making new connections. These include

- `func CableIntrfcFrames(intrfc1, intrfc2 *IntrfcFrame)` sets the 'Cable' attributes of the two interfaces pointing to each other.
- `func CarryIntrfcFrames(intrfc1, intrfc2 *IntrfcFrame)` adds the interfaces to each other's 'Carry' slices, when not already present.
- `func ConnectDevs(dev1, dev2 NetDevice, cable bool, faces string)` is called to ensure that the the devices 'dev1' and 'dev2' are connected, meaning that there is an interface intrfc1 bound to 'dev1' and an interface intrfc2 bound to 'dev2' such that either the 'Cable' attributes of intrfc1 and intrfc2 point to each other, or intrf1 is in intrfc2's 'Carry' slice and vice versa. If it is found that 'dev1' and 'dev2' are already so connected, the method returns immediately and silently. Otherwise 'ConnectDevs' looks for an interface on 'dev1' that can be used (meaning it faces the right network and is not already commited to a connection through non-nil 'Cable' attribute) and an interface on 'dev2' that can be used. In either case if no such interface is discovered one is created, and then finally, depending on the value of 'cable', either the interfaces' 'Cable' attributes are set to point to each other or each interface is placed in the other's 'Carry' slice.
- `func ConnectNetworks(net1, net2 *NetworkFrame, newRtr bool) (*RouterFrame, error)` ensures that two networks are connected by the existence of a Router that has one interface facing one of the networks, and another interface facing another. The networks involved are identified as input variables 'net1' and 'net2'. If Boolean input argument 'newRtr' is true a new router that satisfies this connectivity requirement is always created, even if there exists already a router that faces both networks. If it should happen that a new router is created it is returned as one of the two return arguments. The error return is non-nil if errors are passed up from the logic that adds interfaces to devices, which should not happen.

2. Format of exp.yaml dictionaries

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, interrupt delay, 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. The interrupt delay is the amount of time (in units of seconds) between when a message completely arrives and so generates an interrupt, and when processing on the associated messsage is initiated.

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 associated 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, groups, media, scale, *
Interface	name, groups, device type, device name, media, *
Endpt	name, groups, model, *
Switch	name, groups, model, *
Router	name, groups, 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 'groups' string is assumed to be a comma-separated list of group names, and a match is considered to have happened if the entity is a member of any of the groups listed.

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.

2.1 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
```

2.2 ExpCfg

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

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

2.3 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'.

2.4 AttrbStruct

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

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

3. Format of devExec.yaml dictionaries

mrnes allows for model-dependent times to be ascribed to the switching and routing actions performed by a switch and router. The file nominally called devExec.yaml contains a dictionary with those times. The dictionary structure is described by

```
type DevExecList struct {
    ListName string `json:"listname" yaml:"listname"`
    // key of Times map is the device operation.  Each operation has a list
    // of descriptions of the timing of that operation, as a function of device model
    Times map[string][]DevExecDesc    `json:"times" yaml:"times"`
}

type DevExecDesc struct {
    DevOp    string  `json:"devop" yaml:"devop"`
    Model    string  `json:"model" yaml:"model"`
    PcktLen  int     `json:"pcktlen" yaml:"pcktlen"`
    ExecTime float64 `json:"exectime" yaml:"exectime"`
    Bndwdth  float64 `json:"bndwdth" yaml:"bndwdth"`
}
```

Here we see that the 'DevExecDesc' structure has the information needed for a given operation on a given device model, as measured with the specified packet length, and bandwidth on the device interfaces. All of these are organized in a 'DevExecList' structure, by operation type.

To create a 'DevExecList' structure with a given name (for organizational reference only, **mrnes** logic ignores this) one calls

```
func CreateDevExecList(listname string) *DevExecList
```

To add a device operation timing to an existing dictionary one calls

```
func (del *DevExecList) AddTiming(devOp, model string, execTime float64, pcktlen int,
bndwdth float64)
```

Here of course 'devOp' is the **mrnes**-recognized operation "switch" or "route", 'model' identifies the model of switch or router being referenced, and 'execTime' is the execution time of that operation, expressed in units of seconds, measured with the given packet length and interface bandwidths. The same operation on the same device model may have multiple 'DevExecDesc' entries that differ among themselves by PcktLen and/or Bndwidth. If the timing for an operation is requested with a packet length not represented in the table, an interpolation based on existing entries is created in response.

3.1 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.

3.2 DevExecDesc

```
bndwdth: float
exectime: float
identifier: string
model: DEVMODEL
param: string
exectime: float
```

The DevExecDesc identifier is the same as the opcode indexing this list, the model of the device which the timing is taken, and the bandwidth of the interfaces through which the operation (typically 'switch' or 'route') when the measurement was made.

4. Identity Sets

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
NETWORKNAME	All values mapped to by 'name' key in NetworkDesc dictionaries
ENDPTNAME	All values mapped to by 'name' key in EndptDesc dictionaries
CPUMODEL	All values mapped to by 'model' key in EndptDesc dictionaries
ROUTERNAME	All values mapped to by 'name' key in RouterDesc dictionaries
ROUTERMODEL	All values mapped to by 'model' key in RouterDesc dictionaries
SWITCHNAME	All values mapped to by 'name' key in SwitchDesc dictionaries
SWITCHMODEL	All values mapped to by 'model' key in SwitchDesc dictionaries
INTRFCNAME	All values mapped to by 'name' key in IntrfcDesc dictionaries
MEDIA	{"wired", "wireless"}
NETSCALE	{"LAN", "WAN", "T3", "T2", "T1"}
DEVTYPE	{"Endpt", "Router", "Switch"}
DEVNAME	Union of sets ENDPTNAME, ROUTERNAME, SWITCHNAME
DEVMODEL	All values mapped to by 'model' key in RouterDesc and SwitchDesc dictionaries
GROUPNAME	All values in lists mapped to by key 'groups' in NetworkDesc, EndptDesc, RouterDesc, SwitchDesc, and IntrfcDest dictionaries
NETWORKOBJ	{"Network", "Endpt", "Router", "Switch", "Interface"
ATTRIB	{"name", "group", "device", "scale", "model", "media", "*"}
OPCODE	{"route", "switch"}
DEVMODEL	Union of SWITCHMODEL and ROUTERMODEL