Internet Draft
Expiration: April 2007
File: draft-ietf-forces-model-07.txt
Working Group: ForCES

J. Halpern
Self
E. Deleganes
Intel Corp.

October 2006

# ForCES Forwarding Element Model

draft-ietf-forces-model-07.txt

Status of this Memo



Comments are solicited and should be addressed to the working group's mailing list at forces@peach.ease.lsoft.com and/or the author(s).

Abstract

This document defines the forwarding element (FE) model used in the Forwarding and Control Element Separation (ForCES) protocol. The model represents the capabilities, state and configuration of forwarding elements within the context of the ForCES protocol, so that control elements (CEs) can control the FEs accordingly. More specifically, the model describes the logical functions that are present in an FE, what capabilities these functions support, and how these functions are or can be interconnected. This FE model is intended to satisfy the model requirements specified in the ForCES requirements draft, RFC 3564 [1].

Table of Contents

Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC-2119].

## 1. Definitions

Terminology associated with the ForCES requirements is defined in RFC 3564 [1] and is not copied here. The following list of terminology relevant to the FE model is defined in this section.

FE Model -- The FE model is designed to model the logical processing functions of an FE. The FE model proposed in this document includes three components: the modeling of individual logical functional blocks (LFB model), the logical interconnection between LFBs (LFB topology) and the FE level attributes, including FE capabilities. The FE model provides the basis to define the information elements exchanged between the CE and the FE in the ForCES protocol.

Datapath -- A conceptual path taken by packets within the forwarding plane inside an FE. Note that more than one datapath can exist within an FE.

LFB (Logical Functional Block) Class (or type) -- A template that representing a fine-grained, logically separable aspect of FE processing. Most LFBs relate to packet processing in the data path. LFB classes are the basic building blocks of the FE model.

LFB Instance -- As a packet flows through an FE along a datapath, it flows through one or multiple LFB instances, where each LFB is an instance of a specific LFB class. Multiple instances of the same LFB class can be present in an FE's datapath. Note that we often refer to LFBs without distinguishing between an LFB class and LFB instance when we believe the implied reference is obvious for the given context.

LFB Model -- The LFB model describes the content and structures in an LFB, plus the associated data definition. Four types of information are defined in the LFB model. The core part of the LFB model is the LFB class definitions; the other three types define the associated data including common data types, supported frame formats and metadata.

LFB Metadata -- Metadata is used to communicate per-packet state from one LFB to another, but is not sent across the network. The FE model defines how such metadata is identified, produced and consumed by the LFBs, but not how the per-packet state is implemented within actual hardware. Metadata is sent between the FE and the CE on redirect packets.

LFB Attribute -- Operational parameters of the LFBs that must be visible to the CEs are conceptualized in the FE model as the LFB attributes. The LFB attributes include: flags, single parameter

arguments, complex arguments, and tables that the CE can read or/and write via the ForCES protocol.

LFB Topology -- A representation of the logical interconnection and the placement of LFB instances along the datapath within one FE. Sometimes this representation is called intra-FE topology, to be distinguished from inter-FE topology. LFB topology is outside of the LFB model, but is part of the FE model.

FE Topology -- A representation of how multiple FEs within a single NE are interconnected. Sometimes this is called inter-FE topology, to be distinguished from intra-FE topology (i.e., LFB topology). An individual FE might not have the global knowledge of the full FE topology, but the local view of its connectivity with other FEs is considered to be part of the FE model. The FE topology is discovered by the ForCES base protocol or by some other means.

Inter-FE Topology -- See FE Topology.

Intra-FE Topology -- See LFB Topology.

LFB class library -- A set of LFB classes that has been identified as the most common functions found in most FEs and hence should be defined first by the ForCES Working Group.

## 2. Introduction

RFC 3746 [2] specifies a framework by which control elements (CEs) can configure and manage one or more separate forwarding elements (FEs) within a networking element (NE) using the ForCES protocol. The ForCES architecture allows Forwarding Elements of varying functionality to participate in a ForCES network element. The implication of this varying functionality is that CEs can make only minimal assumptions about the functionality provided by FEs in an NE. Before CEs can configure and control the forwarding behavior of FEs, CEs need to query and discover the capabilities and states of their FEs. RFC 3654 [1] mandates that the capabilities, states and configuration information be expressed in the form of an FE model.

RFC 3444 [11] observed that information models (IMs) and data models (DMs) are different because they serve different purposes. "The main purpose of an IM is to model managed objects at a conceptual level, independent of any specific implementations or protocols used". "DMs, conversely, are defined at a lower level of abstraction and include many details. They are intended for implementors and include protocol-specific constructs." Sometimes it is difficult to draw a clear line between the two. The FE model described in this document is primarily an information model, but also includes some aspects of a data model, such as explicit

definitions of the LFB class schema and FE schema. It is expected that this FE model will be used as the basis to define the payload for information exchange between the CE and FE in the ForCES protocol.

## 2.1. Requirements on the FE model

RFC 3654 [1] defines requirements that must be satisfied by a ForCES FE model. To summarize, an FE model must define:

- . Logically separable and distinct packet forwarding operations in an FE datapath (logical functional blocks or LFBs);
- . The possible topological relationships (and hence the sequence of packet forwarding operations) between the various LFBs;
- . The possible operational capabilities (e.g., capacity limits, constraints, optional features, granularity of configuration) of each type of LFB;
- . The possible configurable parameters (i.e., attributes) of each type of LFB;
- . Metadata that may be exchanged between LFBs.

## 2.2. The FE Model in Relation to FE Implementations

The FE model proposed here is based on an abstraction of distinct logical functional blocks (LFBs), which are interconnected in a directed graph, and receive, process, modify, and transmit packets along with metadata. The FE model should be designed such that different implementations of the forwarding datapath can be logically mapped onto the model with the functionality and sequence of operations correctly captured. However, the model is not intended to directly address how a particular implementation maps to an LFB topology. It is left to the forwarding plane vendors to define how the FE functionality is represented using the FE model. Our goal is to design the FE model such that it is flexible enough to accommodate most common implementations.

The LFB topology model for a particular datapath implementation must correctly capture the sequence of operations on the packet. Metadata generation by certain LFBs MUST always precede any use of that metadata by subsequent LFBs in the topology graph; this is required for logically consistent operation. Further, modification of packet fields that are subsequently used as inputs for further processing MUST occur in the order specified in the model for that particular implementation to ensure correctness.

## 2.3. The FE Model in Relation to the ForCES Protocol

The ForCES base protocol is used by the CEs and FEs to maintain the communication channel between the CEs and FEs. The ForCES protocol may be used to query and discover the inter-FE topology.  The details of a particular datapath implementation inside an FE, including the LFB topology, along with the operational capabilities and attributes of each individual LFB, are conveyed to the CE within information elements in the ForCES protocol. The model of an LFB class should define all of the information that needs to be exchanged between an FE and a CE for the proper configuration and management of that LFB.

Specifying the various payloads of the ForCES messages in a systematic fashion is difficult without a formal definition of the objects being configured and managed (the FE and the LFBs within). The FE Model document defines a set of classes and attributes for describing and manipulating the state of the LFBs within an FE. These class definitions themselves will generally not appear in the ForCES protocol. Rather, ForCES protocol operations will reference classes defined in this model, including relevant attributes and the defined operations.

Section 7 provides more detailed discussion on how the FE model should be used by the ForCES protocol.

### 2.4. Modeling Language for the FE Model

Even though not absolutely required, it is beneficial to use a formal data modeling language to represent the conceptual FE model described in this document. Use of a formal language can help to enforce consistency and logical compatibility among LFBs. A full specification will be written using such a data modeling language. The formal definition of the LFB classes may facilitate the eventual automation of some of the code generation process and the functional validation of arbitrary LFB topologies. These class definitions form the LFB Library. Documents which describe LFB Classes are therefore referred to as LFB Library documents.

Human readability was the most important factor considered when selecting the specification language, whereas encoding, decoding and transmission performance was not a selection factor. The encoding method for over the wire transport is not dependent on the specification language chosen and is outside the scope of this document and up to the ForCES protocol to define.

XML was chosen as the specification language in this document, because XML has the advantage of being both human and machine readable with widely available tools support. This document uses XML Schema to define the structure of the LFB Library documents, as defined in [12] and [13]. While these LFB Class definitions are not sent in the Forces protocol, these definitions comply with the recommendations in RFC 3470 [11] on the use of XML in IETF protocols.

## 2.5. Document Structure

Section 3 provides a conceptual overview of the FE model, laying the foundation for the more detailed discussion and specifications in the sections that follow. Section 4 and 5 constitute the core of the FE model, detailing the two major components in the FE model: LFB model and FE level attributes including capability and LFB topology. Section 6 directly addresses the model requirements imposed by the ForCES requirement draft [1] while Section 7 explains how the FE model should be used in the ForCES protocol.

# 3. FE Model Concepts

Some of the important concepts used throughout this document are introduced in this section. Section 3.1 explains the difference between a state model and a capability model, and describes how the two can be combined in the FE model. Section 3.2 introduces the concept of LFBs (Logical Functional Blocks) as the basic functional building blocks in the FE model. Section 3.3 discusses the logical inter-connection and ordering between LFB instances within an FE, that is, the LFB topology.

The FE model proposed in this document is comprised of two major components: the LFB model and FE level attributes, including FE capabilities and LFB topology. The LFB model provides the content and data structures to define each individual LFB class. FE attributes provide information at the FE level, particularly the capabilities of the FE at a coarse level. Part of the FE level information is the LFB topology, which expresses the logical inter-connection between the LFB instances along the datapath(s) within the FE. Details of these components are described in Section 4 and 5. The intent of this section is to discuss these concepts at the high level and lay the foundation for the detailed description in the following sections.

## 3.1. FE Capability Model and State Model

The ForCES FE model includes both a capability and a state model. The FE capability model describes the capabilities and capacities of an FE by specifying the variation in functions supported and any limitations. The FE state model describes the current state of the FE, that is, the instantaneous values or operational behavior of the FE.

Conceptually, the FE capability model tells the CE which states are allowed on an FE, with capacity information indicating certain quantitative limits or constraints. Thus, the CE has general knowledge about configurations that are applicable to a particular

FE. For example, an FE capability model may describe the FE at a coarse level such as:

- . this FE can handle IPv4 and IPv6 forwarding;
- . this FE can perform classification on the following fields: source IP address, destination IP address, source port number, destination port number, etc;
- . this FE can perform metering;
- . this FE can handle up to N queues (capacity);
- . this FE can add and remove encapsulating headers of types including IPSec, GRE, L2TP.

While one could try and build an object model to fully represent the FE capabilities, other efforts found this to be a significant undertaking. The main difficulty arises in describing detailed limits, such as the maximum number of classifiers, queues, buffer pools, and meters the FE can provide. We believe that a good balance between simplicity and flexibility can be achieved for the FE model by combining coarse level capability reporting with an error reporting mechanism. That is, if the CE attempts to instruct the FE to set up some specific behavior it cannot support, the FE will return an error indicating the problem. Examples of similar approaches include DiffServ PIB [4] and Framework PIB [5].

There is one common and shared aspect of capability that will be handled in a separate fashion. For all elements of information, certain property information is needed. All elements need information as to whether they are supported and if so whether the element is readable or writeable. Based on their type, many elements have additional common properties (for example, arrays have their current size.) There is a specific model and protocol mechanism for referencing this form of property information about elements of the model.

The FE state model presents the snapshot view of the FE to the CE. For example, using an FE state model, an FE may be described to its corresponding CE as the following:

- . on a given port, the packets are classified using a given classification filter;
- . the given classifier results in packets being metered in a certain way, and then marked in a certain way;
- . the packets coming from specific markers are delivered into a shared queue for handling, while other packets are delivered to a different queue;
- . a specific scheduler with specific behavior and parameters will service these collected queues.

Figure 1 shows the concepts of FE state, capabilities and configuration in the context of CE-FE communication via the ForCES protocol.

```
     +-------+                                          +-------+
     |       | FE capabilities: what it can/cannot do.  |       |
     |       |<-----------------------------------------|       |
     |       |                                          |       |
     |   CE  | FE state: what it is now.                |  FE   |
     |       |<-----------------------------------------|       |
     |       |                                          |       |
     |       | FE configuration: what it should be.     |       |
     |       |----------------------------------------->|       |
     +-------+                                          +-------+
```

Figure 1. Illustration of FE state, capabilities and configuration exchange in the context of CE-FE communication via ForCES.

The concepts relating to LFBs, particularly capability at the LFB level and LFB topology will be discussed in the rest of this section.

Capability information at the LFB level is an integral part of the LFB model, and is modeled the same way as the other operational parameters inside an LFB. For example, when certain features of an LFB class are optional, the CE MUST be able to determine whether those optional features are supported by a given LFB instance. Such capability information can be modeled as a read-only attribute in the LFB instance, see Section 4.7.5 for details.

Capability information at the FE level may describe the LFB classes that the FE can instantiate; the number of instances of each that can be created; the topological (linkage) limitations between these LFB instances, etc. Section 5 defines the FE level attributes including capability information.

Once the FE capability is described to the CE, the FE state information can be represented by two levels. The first level is the logically separable and distinct packet processing functions, called Logical Functional Blocks (LFBs). The second level of information describes how these individual LFBs are ordered and placed along the datapath to deliver a complete forwarding plane service. The interconnection and ordering of the LFBs is called LFB Topology. Section 3.2 discusses high level concepts around LFBs, whereas Section 3.3 discusses LFB topology issues.

### 3.2. LFB (Logical Functional Block) Modeling

Each LFB performs a well-defined action or computation on the packets passing through it. Upon completion of its prescribed function, either the packets are modified in certain ways (e.g., decapsulator, marker), or some results are generated and stored, often in the form of metadata (e.g., classifier). Each LFB typically performs a single action. Classifiers, shapers and meters are all examples of such LFBs. Modeling LFBs at such a fine granularity allows us to use a small number of LFBs to express the higher-order FE functions (such as an IPv4 forwarder) precisely, which in turn can describe more complex networking functions and vendor implementations of software and hardware. These LFBs will be defined in detail in one or more documents.

An LFB has one or more inputs, each of which takes a packet P, and optionally metadata M; and produces one or more outputs, each of which carries a packet P', and optionally metadata M'. Metadata is data associated with the packet in the network processing device (router, switch, etc.) and is passed from one LFB to the next, but is not sent across the network. In general, multiple LFBs are contained in one FE, as shown in Figure 2, and all the LFBs share the same ForCES protocol termination point that implements the ForCES protocol logic and maintains the communication channel to and from the CE.

```
                          +-----------+
                          |    CE     |
                          +-----------+
                               ^
                               | Fp reference point
                               |
     +--------------------------|----------------------------------+
     | FE                       |                                  |
     |                          v                                  |
     | +----------------------------------------------------------+ |
     | |                ForCES protocol                           | |
     | |                   termination point                      | |
     | +----------------------------------------------------------+ |
     |           ^                             ^                    |
     |           :                             : Internal control   |
     |           :                             :                    |
     |       +---:----------+              +---:----------|         |
     |       |   :LFB1      |              |   :     LFB2 |         |
     | =====>|   v          |============>|   v          |======>...|
     | Inputs| +----------+ |Outputs       | +----------+ |          |
     | (P,M) | |Attributes| |(P',M')       | |Attributes| |(P",M")   |
     |       | +----------+ |              | +----------+ |          |
     |       +--------------+              +--------------+          |
     |                                                              |
     +--------------------------------------------------------------+
```

Figure 2. Generic LFB Diagram

An LFB, as shown in Figure 2, has inputs, outputs and attributes that can be queried and manipulated by the CE indirectly via an Fp reference point (defined in RFC 3746 [2]) and the ForCES protocol termination point. The horizontal axis is in the forwarding plane for connecting the inputs and outputs of LFBs within the same FE. The vertical axis between the CE and the FE denotes the Fp reference point where bidirectional communication between the CE and FE occurs: the CE to FE communication is for configuration, control and packet injection while FE to CE communication is used for packet re-direction to the control plane, monitoring and accounting information, errors, etc. Note that the interaction between the CE and the LFB is only abstract and indirect. The result of such an interaction is for the CE to indirectly manipulate the attributes of the LFB instances.

A namespace is used to associate a unique name or ID with each LFB class. The namespace MUST be extensible so that a new LFB class can be added later to accommodate future innovation in the forwarding plane.

LFB operation is specified in the model to allow the CE to understand the behavior of the forwarding datapath. For instance, the CE must understand at what point in the datapath the IPv4 header TTL is decremented. That is, the CE needs to know if a control packet could be delivered to it either before or after this point in the datapath. In addition, the CE MUST understand where and what type of header modifications (e.g., tunnel header append or strip) are performed by the FEs. Further, the CE MUST verify that the various LFBs along a datapath within an FE are compatible to link together.

There is value to vendors if the operation of LFB classes can be expressed in sufficient detail so that physical devices implementing different LFB functions can be integrated easily into an FE design. Therefore, a semi-formal specification is needed; that is, a text description of the LFB operation (human readable), but sufficiently specific and unambiguous to allow conformance testing and efficient design, so that interoperability between different CEs and FEs can be achieved.

The LFB class model specifies information such as:

. number of inputs and outputs (and whether they are configurable)
. metadata read/consumed from inputs;
. metadata produced at the outputs;
. packet type(s) accepted at the inputs and emitted at the outputs;
. packet content modifications (including encapsulation or decapsulation);
. packet routing criteria (when multiple outputs on an LFB are present);
. packet timing modifications;
. packet flow ordering modifications;
. LFB capability information;
. Events that can be detected by the LFB, with notification to the CE;
. LFB operational attributes, etc.

Section 4 of this document provides a detailed discussion of the LFB model with a formal specification of LFB class schema. The rest of Section 3.2 only intends to provide a conceptual overview of some important issues in LFB modeling, without covering all the specific details.

### 3.2.1. LFB Outputs

An LFB output is a conceptual port on an LFB that can send information to another LFB.  The information is typically a packet and its associated metadata, although in some cases it might consist of only metadata, i.e., with no packet data.

A single LFB output can be connected to only one LFB input. This is required to make the packet flow through the LFB topology unambiguously.

Some LFBs will have a single output, as depicted in Figure 3.a.

```
   +---------------+            +-----------------+
   |               |            |                 |
   |               |            |             OUT +-->
  ...          OUT +-->        ...                |
   |               |            |    EXCEPTIONOUT +-->
   |               |            |                 |
   +---------------+            +-----------------+

     a. One output              b. Two distinct outputs

   +---------------+            +-----------------+
   |               |            |    EXCEPTIONOUT +-->
   |         OUT:1 +-->         |                 |
  ...        OUT:2 +-->        ...          OUT:1 +-->
   |         ...   +...         |           OUT:2 +-->
   |         OUT:n +-->         |           ...   +...
   +---------------+            |           OUT:n +-->
                                +-----------------+

  c. One output group       d. One output and one output group
```

Figure 3. Examples of LFBs with various output combinations.

To accommodate a non-trivial LFB topology, multiple LFB outputs are needed so that an LFB class can fork the datapath. Two mechanisms are provided for forking: multiple singleton outputs and output groups, which can be combined in the same LFB class.

Multiple separate singleton outputs are defined in an LFB class to model a pre-determined number of semantically different outputs. That is, the LFB class definition MUST include the number of outputs, implying the number of outputs is known when the LFB class is defined. Additional singleton outputs cannot be created at LFB instantiation time, nor can they be created on the fly after the LFB is instantiated.

For example, an IPv4 LPM (Longest-Prefix-Matching) LFB may have one output(OUT) to send those packets for which the LPM look-up was successful, passing a META_ROUTEID as metadata; and have another output (EXCEPTIONOUT) for sending exception packets when the LPM

look-up failed. This example is depicted in Figure 3.b. Packets emitted by these two outputs not only require different downstream treatment, but they are a result of two different conditions in the LFB and each output carries different metadata. This concept assumes the number of distinct outputs is known when the LFB class is defined. For each singleton output, the LFB class definition defines the types of frames and metadata the output emits.

An output group, on the other hand, is used to model the case where a flow of similar packets with an identical set of metadata needs to be split into multiple paths. In this case, the number of such paths is not known when the LFB class is defined because it is not an inherent property of the LFB class. An output group consists of a number of outputs, called the output instances of the group, where all output instances share the same frame and metadata emission definitions (see Figure 3.c). Each output instance can connect to a different downstream LFB, just as if they were separate singleton outputs, but the number of output instances can differ between LFB instances of the same LFB class. The class definition may include a lower and/or an upper limit on the number of outputs. In addition, for configurable FEs, the FE capability information may define further limits on the number of instances in specific output groups for certain LFBs. The actual number of output instances in a group is an attribute of the LFB instance, which is read-only for static topologies, and read-write for dynamic topologies. The output instances in a group are numbered sequentially, from 0 to N-1, and are addressable from within the LFB. The LFB has a built-in mechanism to select one specific output instance for each packet. This mechanism is described in the textual definition of the class and is typically configurable via some attributes of the LFB.

For example, consider a re-director LFB, whose sole purpose is to direct packets to one of N downstream paths based on one of the metadata associated with each arriving packet. Such an LFB is fairly versatile and can be used in many different places in a topology. For example, a redirector can be used to divide the data path into an IPv4 and an IPv6 path based on a FRAMETYPE metadata (N=2), or to fork into color specific paths after metering using the COLOR metadata (red, yellow, green; N=3), etc.

Using an output group in the above LFB class provides the desired flexibility to adapt each instance of this class to the required operation. The metadata to be used as a selector for the output instance is a property of the LFB. For each packet, the value of the specified metadata may be used as a direct index to the output instance. Alternatively, the LFB may have a configurable selector table that maps a metadata value to output instance.

Note that other LFBs may also use the output group concept to build in similar adaptive forking capability. For example, a classifier LFB with one input and N outputs can be defined easily by using the output group concept. Alternatively, a classifier LFB with one singleton output in combination with an explicit N-output re-director LFB models the same processing behavior. The decision of whether to use the output group model for a certain LFB class is left to the LFB class designers.

The model allows the output group to be combined with other singleton output(s) in the same class, as demonstrated in Figure 3.d. The LFB here has two types of outputs, OUT, for normal packet output, and EXCEPTIONOUT for packets that triggered some exception. The normal OUT has multiple instances, thus, it is an output group.

In summary, the LFB class may define one output, multiple singleton outputs, one or more output groups, or a combination thereof. Multiple singleton outputs should be used when the LFB must provide for forking the datapath, and at least one of the following conditions hold:

- the number of downstream directions are inherent from the definition of the class and hence fixed;
- the frame type and set of metadata emitted on any of the outputs are substantially different from what is emitted on the other outputs (i.e., they cannot share frame-type and metadata definitions);

An output group is appropriate when the LFB must provide for forking the datapath, and at least one of the following conditions hold:

- the number of downstream directions is not known when the LFB class is defined;
- the frame type and set of metadata emitted on these outputs are sufficiently similar or ideally identical, such they can share the same output definition.

### 3.2.2. LFB Inputs

An LFB input is a conceptual port on an LFB where the LFB can receive information from other LFBs. The information is typically a packet and associated metadata, although in some cases it might consist of only metadata, without any packet data.

For LFB instances that receive packets from more than one other LFB instance (fan-in). There are three ways to model fan-in, all supported by the LFB model and can be combined in the same LFB:

- Implicit multiplexing via a single input

. Explicit multiplexing via multiple singleton inputs
. Explicit multiplexing via a group of inputs (input group)

The simplest form of multiplexing uses a singleton input (Figure 4.a). Most LFBs will have only one singleton input. Multiplexing into a single input is possible because the model allows more than one LFB output to connect to the same LFB input. This property applies to any LFB input without any special provisions in the LFB class. Multiplexing into a single input is applicable when the packets from the upstream LFBs are similar in frame-type and accompanying metadata, and require similar processing. Note that this model does not address how potential contention is handled when multiple packets arrive simultaneously. If contention handling needs to be explicitly modeled, one of the other two modeling solutions must be used.

The second method to model fan-in uses individually defined singleton inputs (Figure 4.b). This model is meant for situations where the LFB needs to handle distinct types of packet streams, requiring input-specific handling inside the LFB, and where the number of such distinct cases is known when the LFB class is defined. For example, a Layer 2 Decapsulation/Encapsulation LFB may have two inputs, one for receiving Layer 2 frames for decapsulation, and one for receiving Layer 3 frames for encapsulation. This LFB type expects different frames (L2 vs. L3) at its inputs, each with different sets of metadata, and would thus apply different processing on frames arriving at these inputs. This model is capable of explicitly addressing packet contention by defining how the LFB class handles the contending packets.

```
+---------------+       +-------------------------+
| LFB X         +---+   |                         |
+---------------+   |   |                         |
                    |   |                         |
+---------------+   v   |                         |
| LFB Y         +---+-->|input     Meter LFB      |
+---------------+   ^   |                         |
                    |   |                         |
+---------------+   |   |                         |
| LFB Z         |---+   |                         |
+---------------+       +-------------------------+
```

(a) An LFB connects with multiple upstream LFBs via a single input.

```
                +--------------+       +------------------------+
                | LFB X        +---+   |                        |
                +--------------+   +-->|layer2                  |
                +--------------+       |                        |
                | LFB Y        +------>|layer3     LFB          |
                +--------------+       +------------------------+
```

(b) An LFB connects with multiple upstream LFBs via two separate singleton inputs.

```
                +--------------+       +------------------------+
                | Queue LFB #1 +---+   |                        |
                +--------------+   |   |                        |
                                   |   |                        |
                +--------------+   +-->|in:0   \                |
                | Queue LFB #2 +------>|in:1   | input group    |
                +--------------+       |...    |                |
                                   +-->|in:N-1 /                |
                ...                |   |                        |
                +--------------+   |   |                        |
                | Queue LFB #N |---+   |     Scheduler LFB      |
                +--------------+       +------------------------+
```

(c) A Scheduler LFB uses an input group to differentiate which queue LFB packets are coming from.

Figure 3. Input modeling concepts (examples).

The third method to model fan-in uses the concept of an input group. The concept is similar to the output group introduced in the previous section, and is depicted in Figure 4.c. An input group consists of a number of input instances, all sharing the properties (same frame and metadata expectations). The input instances are numbered from 0 to N-1. From the outside, these inputs appear as normal inputs, i.e., any compatible upstream LFB can connect its output to one of these inputs. When a packet is presented to the LFB at a particular input instance, the index of the input where the packet arrived is known to the LFB and this information may be used in the internal processing. For example, the input index can be used as a table selector, or as an explicit precedence selector to resolve contention. As with output groups, the number of input instances in an input group is not defined in the LFB class. However, the class definition may include restrictions on the range of possible values. In addition, if an FE supports configurable topologies, it may impose further limitations on the number of instances for a particular port group(s) of a particular LFB class. Within these limitations, different instances of the same class may have a different number of input instances. The number of actual input instances in the group is an attribute of the LFB class, which is read-only for static topologies, and is read-write for configurable topologies.

As an example for the input group, consider the Scheduler LFB depicted in Figure 3.c.  Such an LFB receives packets from a number of Queue LFBs via a number of input instances, and uses the input index information to control contention resolution and scheduling.

In summary, the LFB class may define one input, multiple singleton inputs, one or more input groups, or a combination thereof.  Any input allows for implicit multiplexing of similar packet streams via connecting multiple outputs to the same input.  Explicit multiple singleton inputs are useful when either the contention handling must be handled explicitly, or when the LFB class must receive and process a known number of distinct types of packet streams.  An input group is suitable when contention handling must be modeled explicitly, but the number of inputs are not inherent from the class (and hence is not known when the class is defined), or when it is critical for LFB operation to know exactly on which input the packet was received.

### 3.2.3. Packet Type

When LFB classes are defined, the input and output packet formats (e.g., IPv4, IPv6, Ethernet, etc.) MUST be specified.  These are the types of packets a given LFB input is capable of receiving and processing, or a given LFB output is capable of producing.  This requires distinct packet types be uniquely labeled with a symbolic name and/or ID.

Note that each LFB has a set of packet types that it operates on, but does not care whether the underlying implementation is passing a greater portion of the packets.  For example, an IPv4 LFB might only operate on IPv4 packets, but the underlying implementation may or may not be stripping the L2 header before handing it over -- whether that is happening or not is opaque to the CE.

### 3.2.4. Metadata

Metadata is the per-packet state that is passed from one LFB to another. The metadata is passed with the packet to assist subsequent LFBs to process that packet.  The ForCES model captures how the per-packet state information is propagated from one LFB to other LFBs. Practically, such metadata propagation can happen within one FE, or cross the FE boundary between two interconnected FEs.  We believe that the same metadata model can be used for either situation; however, our focus here is for intra-FE metadata.

#### 3.2.4.1. Metadata Vocabulary

Metadata has historically been understood to mean "data about data". While this definition is a start, it is inadequate to describe the multiple forms of metadata, which may appear within a complex network element. The discussion here categorizes forms of metadata by two orthogonal axes.

The first axis is "internal" versus "external", which describes where the metadata exists in the network model or implementation. For example, a particular vendor implementation of an IPv4 forwarder may make decisions inside of a chip that are not visible externally. Those decisions are metadata for the packet that is "internal" to the chip. When a packet is forwarded out of the chip, it may be marked with a traffic management header. That header, which is metadata for the packet, is visible outside of the chip, and is therefore called "external" metadata.

The second axis is "implicit" versus "expressed", which specifies whether or not the metadata has a visible physical representation. For example, the traffic management header described in the previous paragraph may be represented as a series of bits in some format, and that header is associated with the packet. Those bits have physical representation, and are therefore "expressed" metadata. If the metadata does not have a physical representation, it is called "implicit" metadata. This situation occurs, for example, when a particular path through a network device is intended to be traversed only by particular kinds of packets, such as an IPv4 router. An implementation may not mark every packet along this path as being of type "IPv4", but the intention of the designers is that every packet is of that type. This understanding can be thought of as metadata about the packet, which is implicitly attached to the packet through the intent of the designers.

In the ForCES model, we do not discuss or represent metadata "internal" to vendor implementations of LFBs. Our focus is solely on metadata "external" to the LFBs, and therefore visible in the ForCES model. The metadata discussed within this model may, or may not be visible outside of the particular FE implementing the LFB model. In this regard, the scope of the metadata within ForCES is very narrowly defined.

Note also that while we define metadata within this model, it is only a model. There is no requirement that vendor implementations of ForCES use the exact metadata representations described in this document. The only implementation requirement is that vendors implement the ForCES protocol, not the model.

#### 3.2.4.2. Metadata lifecycle within the ForCES model

Each metadata can be conveniently modeled as a <label, value> pair, where the label identifies the type of information, (e.g., "color"), and its value holds the actual information (e.g., "red"). The tag here is shown as a textual label, but it can be replaced or associated with a unique numeric value (identifier).

The metadata life-cycle is defined in this model using three types of events: "write", "read" and "consume". The first "write" implicitly creates and initializes the value of the metadata, and hence starts the life-cycle. The explicit "consume" event terminates the life-cycle. Within the life-cycle, that is, after a "write" event, but before the next "consume" event, there can be an arbitrary number of "write" and "read" events. These "read" and "write" events can be mixed in an arbitrary order within the life-cycle. Outside of the life-cycle of the metadata, that is, before the first "write" event, or between a "consume" event and the next "write" event, the metadata should be regarded non-existent or non-initialized. Thus, reading a metadata outside of its life-cycle is considered an error.

To ensure inter-operability between LFBs, the LFB class specification must define what metadata the LFB class "reads" or "consumes" on its input(s) and what metadata it "produces" on its output(s). For maximum extensibility, this definition should neither specify which LFBs the metadata is expected to come from for a consumer LFB, nor which LFBs are expected to consume metadata for a given producer LFB.

While it is important to define the metadata types passing between LFBs, it is not appropriate to define the exact encoding mechanism used by LFBs for that metadata. Different implementations are allowed to use different encoding mechanisms for metadata. For example, one implementation may store metadata in registers or shared memory, while another implementation may encode metadata in-band as a preamble in the packets. In order to allow the CE to understand and control the meta-data related operations, the model represents each metadata tag as a 32-bit integer. Each LFB definition indicates in its metadata declarations the 32-bit value associated with a given metadata tag. Ensuring consistency of usage of tags is important, and outside the scope of the model.

At any link between two LFBs, the packet is marked with a finite set of active metadata, where active means the metadata is within its life-cycle. There are two corollaries of this model:

1. No un-initialized metadata exists in the model.

2. No more than one occurrence of each metadata tag can be associated with a packet at any given time.

#### 3.2.4.3. LFB Operations on Metadata

When the packet is processed by an LFB (i.e., between the time it is received and forwarded by the LFB), the LFB may perform read, write and/or consume operations on any active metadata associated with the packet. If the LFB is considered to be a black box, one of the following operations is performed on each active metadata.

```
  . IGNORE:           ignores and forwards the metadata
  . READ:             reads and forwards the metadata
  . READ/RE-WRITE:    reads, over-writes and forwards the metadata
  . WRITE:            writes and forwards the metadata
                       (can also be used to create new metadata)
  . READ-AND-CONSUME: reads and consumes the metadata
  . CONSUME           consumes metadata without reading
```

The last two operations terminate the life-cycle of the metadata, meaning that the metadata is not forwarded with the packet when the packet is sent to the next LFB.

In our model, a new metadata is generated by an LFB when the LFB applies a WRITE operation to a metadata type that was not present when the packet was received by the LFB. Such implicit creation may be unintentional by the LFB, that is, the LFB may apply the WRITE operation without knowing or caring if the given metadata existed or not. If it existed, the metadata gets over-written; if it did not exist, the metadata is created.

For LFBs that insert packets into the model, WRITE is the only meaningful metadata operation.

For LFBs that remove the packet from the model, they may either READ-AND-CONSUME (read) or CONSUME (ignore) each active metadata associated with the packet.

#### 3.2.4.4. Metadata Production and Consumption

For a given metadata on a given packet path, there MUST be at least one producer LFB that creates that metadata and SHOULD be at least one consumer LFB that needs that metadata. In this model, the producer and consumer LFBs of a metadata are not required to be adjacent. In addition, there may be multiple producers and consumers for the same metadata. When a packet path involves multiple producers of the same metadata, then subsequent producers overwrite that metadata value.

The metadata that is produced by an LFB is specified by the LFB class definition on a per output port group basis. A producer may always generate the metadata on the port group, or may generate it only under certain conditions. We call the former an "unconditional" metadata, whereas the latter is a "conditional" metadata. In the case of conditional metadata, it should be possible to determine from the definition of the LFB when a "conditional" metadata is produced.
The consumer behavior of an LFB, that is, the metadata that the LFB needs for its operation, is defined in the LFB class definition on a per input port group basis. An input port group may "require" a given metadata, or may treat it as "optional" information. In the latter case, the LFB class definition MUST explicitly define what happens if an optional metadata is not provided. One approach is to specify a default value for each optional metadata, and assume that the default value is used if the metadata is not provided with the packet.

When a consumer LFB requires a given metadata, it has dependencies on its up-stream LFBs. That is, the consumer LFB can only function if there is at least one producer of that metadata and no intermediate LFB consumes the metadata.

The model should expose these inter-dependencies. Furthermore, it should be possible to take inter-dependencies into consideration when constructing LFB topologies, and also that the dependencies can be verified when validating topologies.

For extensibility reasons, the LFB specification SHOULD define what metadata the LFB requires without specifying which LFB(s) it expects a certain metadata to come from. Similarly, LFBs SHOULD specify what metadata they produce without specifying which LFBs the metadata is meant for.

When specifying the metadata tags, some harmonization effort must be made so that the producer LFB class uses the same tag as its intended consumer(s), or vice versa.

#### 3.2.4.5. Fixed, Variable and Configurable Tag

When the produced metadata is defined for a given LFB class, most metadata will be specified with a fixed tag. For example, a Rate Meter LFB will always produce the "Color" metadata.

A small subset of LFBs need the capability to produce one or more of their metadata with tags that are not fixed in the LFB class definition, but instead can be selected per LFB instance. An example of such an LFB class is a Generic Classifier LFB. We call this capability "variable tag metadata production". If an LFB produces metadata with a variable tag, the corresponding LFB attribute, called the tag selector, specifies the tag for each such metadata. This mechanism improves the versatility of certain multi-purpose LFB classes, since it allows the same LFB class to be used in different topologies, producing the right metadata tags according to the needs of the topology. This selection of tags is variable in that the produced output may have any number of different tags. The meaning of the various tags is still defined by the metadata declaration associated with the LFB class definition. This also allows the CE to correctly set the tag values in the table to match the declared meanings of the metadata tag values.

Depending on the capability of the FE, the tag selector can be either a read-only or a read-write attribute. If the selector is read-only, the tag cannot be modified by the CE. If the selector is read-write, the tag can be configured by the CE, hence we call this "configurable tag metadata production." Note that using this definition, configurable tag metadata production is a subset of variable tag metadata production.

Similar concepts can be introduced for the consumer LFBs to satisfy different metadata needs. Most LFB classes will specify their metadata needs using fixed metadata tags. For example, a Next Hop LFB may always require a "NextHopId" metadata; but the Redirector LFB may need a "ClassID" metadata in one instance, and a "ProtocolType" metadata in another instance as a basis for selecting the right output port. In this case, an LFB attribute is used to provide the required metadata tag at run-time. This metadata tag selector attribute may be read-only or read-write, depending on the capabilities of the LFB instance and the FE.

#### 3.2.4.6. Metadata Usage Categories

Depending on the role and usage of a metadata, various amounts of encoding information MUST be provided when the metadata is defined, where some cases offer less flexibility in the value selection than others.

There are three types of metadata related to metadata usage:

. Relational (or binding) metadata
. Enumerated metadata
. Explicit/external value metadata

The purpose of the relational metadata is to refer in one LFB instance (producer LFB) to a "thing" in another downstream LFB instance (consumer LFB), where the "thing" is typically an entry in a table attribute of the consumer LFB.

For example, the Prefix Lookup LFB executes an LPM search using its prefix table and resolves to a next-hop reference. This reference needs to be passed as metadata by the Prefix Lookup LFB (producer) to the Next Hop LFB (consumer), and must refer to a specific entry in the next-hop table within the consumer.

Expressing and propagating such a binding relationship is probably the most common usage of metadata. One or more objects in the producer LFB are bound to a specific object in the consumer LFB. Such a relationship is established by the CE explicitly by properly configuring the attributes in both LFBs. Available methods include the following:

The binding may be expressed by tagging the involved objects in both LFBs with the same unique, but otherwise arbitrary, identifier. The value of the tag is explicitly configured by the CE by writing the value into both LFBs, and this value is also carried by the metadata between the LFBs.

Another way of setting up binding relations is to use a naturally occurring unique identifier of the consumer's object as a reference and as a value of the metadata (e.g., the array index of a table entry). In this case, the index is either read or inferred by the CE by communicating with the consumer LFB. Once the CE obtains the index, it needs to write it into the producer LFB to establish the binding.

Important characteristics of the binding usage of metadata are:

- The value of the metadata shows up in the CE-FE communication for both the consumer and the producer. That is, the metadata value MUST be carried over the ForCES protocol. Using the tagging technique, the value is written to both LFBs. Using the other technique, the value is written to only the producer LFB and may be READ from the consumer LFB.

- The metadata value is irrelevant to the CE, the binding is simply expressed by using the same value at the consumer and producer LFBs.

- Hence the metadata definition is not required to include value assignments. The only exception is when some special value(s) of the metadata must be reserved to convey special events. Even though these special cases must be defined with the metadata specification, their encoded values can be selected arbitrarily. For example, for the Prefix Lookup LFB example, a special value may be reserved to signal the NO-MATCH case, and the value of zero may be assigned for this purpose.

The second class of metadata is the enumerated type. An example is the "Color" metadata that is produced by a Meter LFB. As the name suggests, enumerated metadata has a relatively small number of possible values, each with a specific meaning. All possible cases must be enumerated when defining this class of metadata. Although a value encoding must be included in the specification, the actual values can be selected arbitrarily (e.g., <Red=0, Yellow=1, Green=2> and <Red=3, Yellow=2, Green 1> would be both valid encodings, what is important is that an encoding is specified).

The value of the enumerated metadata may or may not be conveyed via the ForCES protocol between the CE and FE.

The third class of metadata is the explicit type. This refers to cases where the metadata value is explicitly used by the consumer LFB to change some packet header fields. In other words, the value has a direct and explicit impact on some field and will be visible externally when the packet leaves the NE. Examples are: TTL increment given to a Header Modifier LFB, and DSCP value for a Remarker LFB. For explicit metadata, the value encoding MUST be explicitly provided in the metadata definition. The values cannot be selected arbitrarily and should conform to what is commonly expected. For example, a TTL increment metadata should be encoded as zero for the no increment case, one for the single increment case, etc. A DSCP metadata should use 0 to encode DSCP=0, 1 to encode DSCP=1, etc.

### 3.2.5. LFB Events

During operation, various conditions may occur that can be detected by LFBs. Examples range from link failure or restart to timer expiration in special purpose LFBs. The CE may wish to be notified of the occurrence of such events. The PL protocol provides for such notifications. The LFB definition includes the necessary declarations of events. The declarations include identifiers necessary for subscribing to events (so that the CE can indicate to the FE which events it wishes to receive) and to indicate in event notification messages which event is being reported.

The declaration of an event defines a condition that an FE can detect, and may report. From a conceptual point of view, event processing is split into triggering (the detection of the condition) and reporting (the generation of the notification of the event.) In between these two conceptual points there is event filtering. Properties associated with the event in the LFB instance can define filtering conditions to suppress the reporting of that event. The model thus describes event processing as if events always occur, and filtering may suppress reporting. Implementations may function in this manner, or may have more complex logic that eliminates some event processing if the reporting would be suppressed. Any implementation producing an effect equivalent to the model description is valid.

### 3.2.6. LFB Element Properties

LFBs are made up of elements, containing the information that the CE needs to see and / or change about the functioning of the LFB. These elements, as described in detail elsewhere, may be basic values, complex structures, or tables (containing values, structures, or tables.) Some of these elements are optional. Some elements may be readable or writeable at the discretion of the FE implementation. The CE needs to know these properties. Additionally, certain kinds of elements (arrays, aliases, and events as of this writing) have additional property information that the CE may need to read or write. This model defines the structure of the property information for all defined data types.

The reports with events are designed to allow for the common, closely related information that the CE can be strongly expected to need to react to the event. It is not intended to carry information the CE already has, nor large volumes of information, nor information related in complex fashions.

### 3.2.7. LFB Versioning

LFB class versioning is a method to enable incremental evolution of LFB classes. In general, an FE is not allowed to contain an LFB instance for more than one version of a particular class. Inheritance (discussed next in Section 3.2.6) has special rules. If an FE datapath model containing an LFB instance of a particular class C also simultaneously contains an LFB instance of a class C' inherited from class C; C could have a different version than C'.

LFB class versioning is supported by requiring a version string in the class definition. CEs may support multiple versions of a particular LFB class to provide backward compatibility, but FEs MUST NOT support more than one version of a particular class.

Versioning is not restricted to making backwards compatible changes. It is specifically expected to be used to make changes that cannot be represented by inheritance. Often this will be to correct errors, and hence may not be backwards compatible. It may also be used to remove elements which are not considered useful (particularly if they were previously mandatory, and hence were an implementation impediment.)

### 3.2.8. LFB Inheritance

LFB class inheritance is supported in the FE model as a method to define new LFB classes. This also allows FE vendors to add vendor-specific extensions to standardized LFBs. An LFB class specification MUST specify the base class and version number it inherits from (the default is the base LFB class). Multiple-inheritance is not allowed, however, to avoid unnecessary complexity.

Inheritance should be used only when there is significant reuse of the base LFB class definition. A separate LFB class should be defined if little or no reuse is possible between the derived and the base LFB class.

An interesting issue related to class inheritance is backward compatibility between a descendant and an ancestor class. Consider the following hypothetical scenario where a standardized LFB class "L1" exists. Vendor A builds an FE that implements LFB "L1" and vendor B builds a CE that can recognize and operate on LFB "L1". Suppose that a new LFB class, "L2", is defined based on the existing "L1" class by extending its capabilities incrementally. Let us examine the FE backward compatibility issue by considering what would happen if vendor B upgrades its FE from "L1" to "L2" and vendor C's CE is not changed. The old L1-based CE can interoperate with the new L2-based FE if the derived LFB class "L2" is indeed backward compatible with the base class "L1".

The reverse scenario is a much less problematic case, i.e., when CE vendor B upgrades to the new LFB class "L2", but the FE is not upgraded. Note that as long as the CE is capable of working with older LFB classes, this problem does not affect the model; hence we will use the term "backward compatibility" to refer to the first scenario concerning FE backward compatibility.

Backward compatibility can be designed into the inheritance model by constraining LFB inheritance to require the derived class be a functional superset of the base class (i.e. the derived class can only add functions to the base class, but not remove functions). Additionally, the following mechanisms are required to support FE backward compatibility:

1. When detecting an LFB instance of an LFB type that is unknown to the CE, the CE MUST be able to query the base class of such an LFB from the FE.
2. The LFB instance on the FE SHOULD support a backward compatibility mode (meaning the LFB instance reverts itself back to the base class instance), and the CE SHOULD be able to

configure the LFB to run in such a mode.

### 3.3. FE Datapath Modeling

Packets coming into the FE from ingress ports generally flow through multiple LFBs before leaving out of the egress ports. How an FE treats a packet depends on many factors, such as type of the packet (e.g., IPv4, IPv6 or MPLS), actual header values, time of arrival, etc. The result of LFB processing may have an impact on how the packet is to be treated in downstream LFBs. This differentiation of packet treatment downstream can be conceptualized as having alternative datapaths in the FE. For example, the result of a 6-tuple classification performed by a classifier LFB could control which rate meter is applied to the packet by a rate meter LFB in a later stage in the datapath.

LFB topology is a directed graph representation of the logical datapaths within an FE, with the nodes representing the LFB instances and the directed link depicting the packet flow direction from one LFB to the next. Section 3.3.1 discusses how the FE datapaths can be modeled as LFB topology; while Section 3.3.2 focuses on issues related to LFB topology reconfiguration.

#### 3.3.1. Alternative Approaches for Modeling FE Datapaths

There are two basic ways to express the differentiation in packet treatment within an FE, one represents the datapath directly and graphically (topological approach) and the other utilizes metadata (the encoded state approach).

. Topological Approach

Using this approach, differential packet treatment is expressed by splitting the LFB topology into alternative paths. In other words, if the result of an LFB operation controls how the packet is further processed, then such an LFB will have separate output ports, one for each alternative treatment, connected to separate sub-graphs, each expressing the respective treatment downstream.

. Encoded State Approach

An alternate way of expressing differential treatment is by using metadata. The result of the operation of an LFB can be encoded in a metadata, which is passed along with the packet to downstream LFBs. A downstream LFB, in turn, can use the metadata and its value (e.g., as an index into some table) to determine how to treat the packet.

Theoretically, either approach could substitute for the other, so one could consider using a single pure approach to describe all

datapaths in an FE. However, neither model by itself results in the best representation for all practically relevant cases. For a given FE with certain logical datapaths, applying the two different modeling approaches will result in very different looking LFB topology graphs. A model using only the topological approach may require a very large graph with many links or paths, and nodes (i.e., LFB instances) to express all alternative datapaths. On the other hand, a model using only the encoded state model would be restricted to a string of LFBs, which is not an intuitive way to describe different datapaths (such as MPLS and IPv4). Therefore, a mix of these two approaches will likely be used for a practical model. In fact, as we illustrate below, the two approaches can be mixed even within the same LFB.

Using a simple example of a classifier with N classification outputs followed by other LFBs, Figure 5(a) shows what the LFB topology looks like when using the pure topological approach. Each output from the classifier goes to one of the N LFBs where no metadata is needed. The topological approach is simple, straightforward and graphically intuitive. However, if N is large and the N nodes following the classifier (LFB#1, LFB#2, ..., LFB#N) all belong to the same LFB type (e.g., meter), but each has its own independent attributes, the encoded state approach gives a much simpler topology representation, as shown in Figure 5(b). The encoded state approach requires that a table of N rows of meter attributes is provided in the Meter node itself, with each row representing the attributes for one meter instance. A metadata M is also needed to pass along with the packet P from the classifier to the meter, so that the meter can use M as a look-up key (index) to find the corresponding row of the attributes that should be used for any particular packet P.

What if those N nodes (LFB#1, LFB#2, ..., LFB#N) are not of the same type? For example, if LFB#1 is a queue while the rest are all meters, what is the best way to represent such datapaths? While it is still possible to use either the pure topological approach or the pure encoded state approach, the natural combination of the two appears to be the best option. Figure 5(c) depicts two different functional datapaths using the topological approach while leaving the N-1 meter instances distinguished by metadata only, as shown in Figure 5(c).

```
                                +----------+
                         P      |   LFB#1  |
                     +--------->|(Attrib-1)|
+-------------+      |          +----------+
|            1|------+   P      +----------+
|            2|---------------->|   LFB#2  |
| classifier 3|                 |(Attrib-2)|
|          ...|...              +----------+
|            N|------+          ...
+-------------+      |   P      +----------+
                     +--------->|   LFB#N  |
                                |(Attrib-N)|
                                +----------+
```

5(a) Using pure topological approach

```
+-------------+                 +-------------+
|            1|                 |   Meter     |
|            2|   (P, M)        | (Attrib-1)  |
|            3|---------------->| (Attrib-2)  |
|          ...|                 |   ...       |
|            N|                 | (Attrib-N)  |
+-------------+                 +-------------+
```

5(b) Using pure encoded state approach to represent the LFB
topology in 5(a), if LFB#1, LFB#2, ..., and LFB#N are of the
same type (e.g., meter).

```
                             +-------------+
+-------------+ (P, M)       | queue       |
|            1|------------->| (Attrib-1)  |
|            2|              +-------------+
|            3| (P, M)       +-------------+
|          ...|------------->|   Meter     |
|            N|              | (Attrib-2)  |
+-------------+              |   ...       |
                             | (Attrib-N)  |
                             +-------------+
```

5(c) Using a combination of the two, if LFB#1, LFB#2, ..., and
LFB#N are of different types (e.g., queue and meter).

Figure 5. An example of how to model FE datapaths

From this example, we demonstrate that each approach has a distinct advantage depending on the situation.  Using the encoded state approach, fewer connections are typically needed between a fan-out node and its next LFB instances of the same type because each packet carries metadata the following nodes can interpret and hence invoke a different packet treatment. For those cases, a pure topological approach forces one to build elaborate graphs with many more connections and often results in an unwieldy graph. On the other hand, a topological approach is the most intuitive for representing functionally different datapaths.

For complex topologies, a combination of the two is the most flexible. A general design guideline is provided to indicate which approach is best used for a particular situation. The topological approach should primarily be used when the packet datapath forks to distinct LFB classes (not just distinct parameterizations of the same LFB class), and when the fan-outs do not require changes, such as adding/removing LFB outputs, or require only very infrequent changes. Configuration information that needs to change frequently should be expressed by using the internal attributes of one or more LFBs (and hence using the encoded state approach).

```
                   +---------------------------------------------+
                   |                                             |
     +----------+  V      +----------+           +------+        |
     |          |  |      |          |if IP-in-IP|      |        |
---->| ingress  |->+----->|classifier|---------->|Decap.|---->---+
     | ports    |         |          |----+      |      |
     +----------+         +----------+    |others+------+
                                          |
                                          V
     (a)  The LFB topology with a logical loop

    +-------+   +-----------+            +------+   +-----------+
    |       |   |           |if IP-in-IP |      |   |           |
--->|ingress|-->|classifier1|----------->|Decap.|-->+classifier2|->
    | ports |   |           |----+       |      |   |           |
    +-------+   +-----------+    |others +------+   +-----------+
                                 |
                                 V
     The LFB topology without the loop utilizing two independent
     classifier instances.
```

Figure 6. An LFB topology example.

It is important to point out that the LFB topology described here is the logical topology, not the physical topology of how the FE hardware is actually laid out. Nevertheless, the actual implementation may still influence how the functionality is mapped to the LFB topology. Figure 6 shows one simple FE example. In this example, an IP-in-IP packet from an IPSec application like VPN may go to the classifier first and have the classification done based on the outer IP header; upon being classified as an IP-in-IP packet, the packet is then sent to a decapsulator to strip off the outer IP header, followed by a classifier again to perform classification on the inner IP header. If the same classifier hardware or software is used for both outer and inner IP header classification with the same set of filtering rules, a logical loop is naturally present in the LFB topology, as shown in Figure 6(a). However, if the classification is implemented by two different pieces of hardware or software with different filters (i.e., one set of filters for the outer IP header and another set for the inner IP header), then it is more natural to model them as two different instances of classifier LFB, as shown in Figure 6(b).

To distinguish between multiple instances of the same LFB class, each LFB instance has its own LFB instance ID. One way to encode the LFB instance ID is to encode it as x.y where x is the LFB class ID and y is the instance ID within each LFB class.

### 3.3.2. Configuring the LFB Topology

While there is little doubt that an individual LFB must be configurable, the configurability question is more complicated for LFB topology. Since the LFB topology is really the graphic representation of the datapaths within an FE, configuring the LFB topology means dynamically changing the datapaths, including changing the LFBs along the datapaths on an FE (e.g., creating, instantiating or deleting LFBs) and setting up or deleting interconnections between outputs of upstream LFBs to inputs of downstream LFBs.

Why would the datapaths on an FE ever change dynamically? The datapaths on an FE are set up by the CE to provide certain data plane services (e.g., DiffServ, VPN, etc.) to the Network Element's (NE) customers. The purpose of reconfiguring the datapaths is to enable the CE to customize the services the NE is delivering at run time. The CE needs to change the datapaths when the service requirements change, such as adding a new customer or when an existing customer changes their service. However, note that not all datapath changes result in changes in the LFB topology graph. Changes in the graph are dependent on the approach used to map the datapaths into LFB topology. As discussed in 3.3.1, the topological approach and encoded state approach can result in very different looking LFB topologies for the same datapaths. In general, an LFB topology based on a pure topological approach is likely to experience more frequent topology reconfiguration than one based on an encoded state approach. However, even an LFB topology based entirely on an encoded state approach may have to change the topology at times, for example, to bypass some LFBs or insert new LFBs. Since a mix of these two approaches is used to model the datapaths, LFB topology reconfiguration is considered an important aspect of the FE model.

We want to point out that allowing a configurable LFB topology in the FE model does not mandate that all FEs are required to have this capability. Even if an FE supports configurable LFB topology, the FE may impose limitations on what can actually be configured. Performance-optimized hardware implementations may have zero or very limited configurability, while FE implementations running on network processors may provide more flexibility and configurability. It is entirely up to the FE designers to decide whether or not the FE actually implements reconfiguration and if so, how much. Whether a simple runtime switch is used to enable or disable (i.e., bypass) certain LFBs, or more flexible software reconfiguration is used, is implementation detail internal to the FE and outside of the scope of FE model. In either case, the CE(s) MUST be able to learn the FE's configuration capabilities. Therefore, the FE model MUST provide a mechanism for describing the LFB topology configuration capabilities of an FE. These capabilities may include (see Section 5 for full details):

- Which LFB classes the FE can instantiate
- Maximum number of instances of the same LFB class that can be created
- Any topological limitations, For example:
  - The maximum number of instances of the same class or any class that can be created on any given branch of the graph
  - Ordering restrictions on LFBs (e.g., any instance of LFB class A must be always downstream of any instance of LFB class B).

Note that even when the CE is allowed to configure LFB topology for the FE, the CE is not expected to be able to interpret an arbitrary LFB topology and determine which specific service or application (e.g. VPN, DiffServ, etc.) is supported by the FE. However, once the CE understands the coarse capability of an FE, the CE MUST configure the LFB topology to implement the network service the NE is supposed to provide. Thus, the mapping the CE has to understand is from the high level NE service to a specific LFB topology, not the other way around. The CE is not expected to have the ultimate intelligence to translate any high level service policy into the configuration data for the FEs. However, it is conceivable that within a given network service domain, a certain amount of intelligence can be programmed into the CE to give the CE a general understanding of the LFBs involved to allow the translation from a high level service policy to the low level FE configuration to be done automatically. Note that this is considered an implementation issue internal to the control plane and outside the scope of the FE model. Therefore, it is not discussed any further in this draft.

```
        +----------+     +-----------+
   ---->| Ingress  |---->|classifier |--------------+
        |          |     |chip       |              |
        +----------+     +-----------+              |
                                                    v
                        +--------------------------------------------+
          +--------+    |   Network Processor                         |
     <----| Egress |    |   +------+    +------+   +-------+          |
          +--------+    |   |Meter |    |Marker|   |Dropper|          |
                ^       |   +------+    +------+   +-------+          |
                |       |                                             |
     +----------+-------+                                             |
     |          |                                                     |
     |    +---------+       +---------+   +------+    +---------+     |
     |    |Forwarder|<------|Scheduler|<--|Queue |    |Counter  |     |
     |    +---------+       +---------+   +------+    +---------+     |
     |--------------------------------------------------------------+
```

(a)  The Capability of the FE, reported to the CE

```
          +-----+    +-------+                      +---+
          |    A|--->|Queue1 |--------------------->|   |
   ------>|     |    +-------+                      |   |  +---+
          |     |                                   |   |  |   |
          |     |    +-------+      +-------+       |   |  |   |
          |    B|--->|Meter1 |----->|Queue2 |------>|   |->|   |
          |     |    |       |      +-------+       |   |  |   |
          |     |    |       |--+                   |   |  |   |
          +-----+    +-------+  |   +-------+       |   |  +---+
        classifier              +-->|Dropper|       |   |  IPv4
                                    +-------+       +---+  Fwd.
                                                 Scheduler
```

(b)  One LFB topology as configured by the CE and
     accepted by the FE

```
                                                 Queue1
                      +---+                    +--+
                      |  A|------------------->|  |--+
                   +->|   |                    |  |  |
                   |  |  B|--+  +--+   +--+    +--+  |
                   |  +---+  |  |  |   |  |          |
                   | Meter1  +->|  |-->|  |          |
                   |            |  |   |  |          |
                   |            +--+   +--+          |          Ipv4
                   |         Counter1 Dropper1 Queue2|    +--+  Fwd.
           +---+   |                           +--+  +--->|A |  +-+
           |  A|---+                           |  |------>|B |  | |
    ------>|  B|------------------------------>|  |  +--->|C |->| |->
           |  C|---+                           +--+  | +->|D |  | |
           |  D|-+ |                                 | |  +--+  +-+
           +---+ | |    +---+                  Queue3| | Scheduler
       Classifier1 | |  |  A|------------>       +--+  | |
                   | +->|   |                    |  |--+ |
                   |    |  B|--+  +--+ +-------->|  |    |
                   |    +---+  |  |  | |         +--+    |
                   |  Meter2   +->|  |-+                 |
                   |              |  |                   |
                   |              +--+           Queue4  |
                   |            Marker1          +--+    |
                   +---------------------------->|  |----+
                                                 |  |
                                                 +--+
```

                (c)  Another LFB topology as configured by the CE and
                     accepted by the FE

        Figure 7. An example of configuring LFB topology.

Figure 7 shows an example where a QoS-enabled router has several line cards that have a few ingress ports and egress ports, a specialized classification chip, a network processor containing codes for FE blocks like meter, marker, dropper, counter, queue, scheduler and Ipv4 forwarder.  Some of the LFB topology is already fixed and has to remain static due to the physical layout of the line cards.  For example, all of the ingress ports might be hard-wired into the classification chip so all packets flow from the ingress port into the classification engine.  On the other hand, the LFBs on the network processor and their execution order are programmable. However, certain capacity limits and linkage constraints could exist between these LFBs. Examples of the capacity limits might be: 8 meters; 16 queues in one FE; the scheduler can handle at most up to 16 queues; etc.  The linkage constraints might dictate that the classification engine may be followed by a meter, marker, dropper, counter, queue or IPv4 forwarder, but not a scheduler; queues can only be followed by a scheduler; a scheduler must be followed by the IPv4 forwarder; the last LFB in the datapath before going into the egress ports must be the IPv4 forwarder, etc.

Once the FE reports these capabilities and capacity limits to the CE, it is now up to the CE to translate the QoS policy into a desirable configuration for the FE. Figure 7(a) depicts the FE capability while 7(b) and 7(c) depict two different topologies that the CE may request the FE to configure. Note that both the ingress and egress are omitted in (b) and (c) to simplify the representation. The topology in 7(c) is considerably more complex than 7(b) but both are feasible within the FE capabilities, and so the FE should accept either configuration request from the CE.

## 4. Model and Schema for LFB Classes

The main goal of the FE model is to provide an abstract, generic, modular, implementation-independent representation of the FEs. This is facilitated using the concept of LFBs, which are instantiated from LFB classes. LFB classes and associated definitions will be provided in a collection of XML documents. The collection of these XML documents is called a LFB class library, and each document is called an LFB class library document (or library document, for short). Each of the library documents will conform to the schema presented in this section. The root element of the library document is the <LFBLibrary> element.

It is not expected that library documents will be exchanged between FEs and CEs "over-the-wire". But the model will serve as an important reference for the design and development of the CEs (software) and FEs (mostly the software part). It will also serve as a design input when specifying the ForCES protocol elements for CE-FE communication.

### 4.1. Namespace

A namespace is needed to uniquely identify the LFB type in the LFB class library. The reference to the namespace definition is contained in Section 9, IANA Considerations.

### 4.2. <LFBLibrary> Element

The <LFBLibrary> element serves as a root element of all library documents. It contains one or more of the following main blocks:

- <frameTypeDefs> for the frame declarations;
- <dataTypeDefs> for defining common data types;
- <metadataDefs> for defining metadata, and
- <LFBClassDefs> for defining LFB classes.

Each block is optional, that is, one library document may contain only metadata definitions, another may contain only LFB class definitions, yet another may contain all of the above.

In addition to the above main blocks, a library document can import other library documents if it needs to refer to definitions contained in the included document. This concept is similar to the "#include" directive in C. Importing is expressed by the <load> elements, which must precede all the above elements in the document. For unique referencing, each LFBLibrary instance document has a unique label defined in the "provide" attribute of the LFBLibrary element.

The <LFBLibrary> element also includes an optional <description> element, which can be used to provide textual description about the library document.

The following is a skeleton of a library document:

```
<?xml version="1.0" encoding="UTF-8"?>
<LFBLibrary xmlns="http://ietf.org/forces/1.0/lfbmodel"
  provides="this_library">

  <description>
    ...
  </description>

  <!-- Loading external libraries (optional) -->
  <load library="another_library"/>
  ...

  <!-- FRAME TYPE DEFINITIONS (optional) -->
  <frameTypeDefs>
    ...
  </frameTypeDefs>

  <!-- DATA TYPE DEFINITIONS (optional) -->
  <dataTypeDefs>
    ...
  </dataTypeDefs>

  <!-- METADATA DEFINITIONS (optional) -->
  <metadataDefs>
    ...
  </metadataDefs>
```

```
  <!--
    -
    -
     LFB CLASS DEFINITIONS (optional) -->
  <LFBCLassDefs>
    ...
  </LFBCLassDefs>
</LFBLibrary>
```

### 4.3. <load> Element

This element is used to refer to another LFB library document. Similar to the "#include" directive in C, this makes the objects (metadata types, data types, etc.) defined in the referred library document available for referencing in the current document.

The load element MUST contain the label of the library document to be included and may contain a URL to specify where the library can be retrieved. The load element can be repeated unlimited times. Three examples for the <load> elements:

```
<load library="a_library"/>
<load library="another_library" location="another_lib.xml"/>
<load library="yetanother_library"
location="http://www.petrimeat.com/forces/1.0/lfbmodel/lpm.xml"/>
```

### 4.4. <frameDefs> Element for Frame Type Declarations

Frame names are used in the LFB definition to define the types of frames the LFB expects at its input port(s) and emits at its output port(s). The <frameDefs> optional element in the library document contains one or more <frameDef> elements, each declaring one frame type.

Each frame definition MUST contain a unique name (NMTOKEN) and a brief synopsis. In addition, an optional detailed description may be provided.

Uniqueness of frame types MUST be ensured among frame types defined in the same library document and in all directly or indirectly included library documents.

The following example defines two frame types:

```
<frameDefs>
  <frameDef>
    <name>ipv4</name>
    <synopsis>IPv4 packet</synopsis>
```

<description>

```
        This frame type refers to an IPv4 packet.
      </description>
    </frameDef>
      <frameDef>
      <name>ipv6</name>
      <synopsis>IPv6 packet</synopsis>
      <description>
        This frame type refers to an IPv6 packet.
      </description>
    </frameDef>
    ...
  </frameDefs>
```

## 4.5. <dataTypeDefs> Element for Data Type Definitions

The (optional) <dataTypeDefs> element can be used to define commonly used data types. It contains one or more <dataTypeDef> elements, each defining a data type with a unique name. Such data types can be used in several places in the library documents, including:

- Defining other data types
- Defining attributes of LFB classes

This is similar to the concept of having a common header file for shared data types.

Each <dataTypeDef> element MUST contain a unique name (NMTOKEN), a brief synopsis, an optional longer description, and a type definition element. The name MUST be unique among all data types defined in the same library document and in any directly or indirectly included library documents. For example:

```
<dataTypeDefs>
  <dataTypeDef>
    <name>ieeemacaddr</name>
    <synopsis>48-bit IEEE MAC address</synopsis>
    ... type definition ...
  </dataTypeDef>
  <dataTypeDef>
    <name>ipv4addr</name>
    <synopsis>IPv4 address</synopsis>
    ... type definition ...
  </dataTypeDef>
  ...
</dataTypeDefs>
```

There are two kinds of data types: atomic and compound. Atomic data types are appropriate for single-value variables (e.g. integer, string, byte array).

The following built-in atomic data types are provided, but additional atomic data types can be defined with the <typeRef> and <atomic> elements:

| <name> | Meaning |
| --- | --- |
| char | 8-bit signed integer |
| uchar | 8-bit unsigned integer |
| int16 | 16-bit signed integer |
| uint16 | 16-bit unsigned integer |
| int32 | 32-bit signed integer |
| uint32 | 32-bit unsigned integer |
| int64 | 64-bit signed integer |
| uint64 | 64-bit unsigned integer |
| boolean | A true / false value where 0 = false, 1 = true |
| string[N] | A UTF-8 string represented in at most N Octets. |
| string | A UTF-8 string without a configured storage length limit. |
| byte[N] | A byte array of N bytes |
| octetstring[N] | A buffer of N octets, which may contain fewer than N octets. Hence the encoded value will always have a length. |
| float16 | 16-bit floating point number |
| float32 | 32-bit IEEE floating point number |
| float64 | 64-bit IEEE floating point number |

These built-in data types can be readily used to define metadata or LFB attributes, but can also be used as building blocks when defining new data types. The boolean data type is defined here because it is so common, even though it can be built by sub-ranging the uchar data type.

Compound data types can build on atomic data types and other compound data types. Compound data types can be defined in one of four ways. They may be defined as an array of elements of some compound or atomic data type. They may be a structure of named elements of compound or atomic data types (ala C structures). They may be a union of named elements of compound or atomic data types (ala C unions). They may also be defined as augmentations (explained below in 4.5.6) of existing compound data types.

Given that the FORCES protocol will be getting and setting attribute values, all atomic data types used here must be able to be conveyed in the FORCES protocol. Further, the FORCES protocol will need a mechanism to convey compound data types. However, the details of such representations are for the protocol document to define, not the model document. Strings and octetstrings must be conveyed with their length, as they are not delimited, and are variable length.

With regard to strings, this model defines a small set of restrictions and definitions on how they are structured. String and octetstring length limits can be specified in the LFB Class definitions. The element properties for string and octetstring elements also contain actual lengths and length limits. This duplication of limits is to allow for implementations with smaller limits than the maximum limits specified in the LFB Class definition. In all cases, these lengths are specified in octets, not in characters. In terms of protocol operation, as long as the specified length is within the FE s supported capabilities, the FE stores the contents of a string exactly as provided by the CE, and returns those contents when requested. No canonicalization, transformations, or equivalences are performed by the FE. Elements of type string (or string[n]) may be used to hold identifiers for correlation with elements in other LFBs. In such cases, an exact octet for octet match is used. No equivalences are used by the FE or CE in performing that matching. The ForCES protocol does not perform or require validation of the content of UTF-8 strings. However, UTF-8 strings SHOULD be encoded in the shortest form to avoid potential security issues described in [15]. Any entity displaying such strings is expected to perform its own validation (for example for correct multi-byte characters, and for ensuring that the string does not end in the middle of a multi-byte sequence.) Specific LFB class definitions may restrict the valid contents of a string as suited to the particular usage (for example, an element that holds a DNS name would be restricted to hold only octets valid in such a name.) FEs should validate the contents of SET requests for such restricted elements at the time the set is performed, just as range checks for range limited elements are performed. The ForCES protocol behavior defines the normative processing for requests using that protocol.

For the definition of the actual type in the <dataTypeDef> element, the following elements are available: <typeRef>, <atomic>, <array>, <struct>, and <union>.

The predefined type alias is somewhere between the atomic and compound data types.  It behaves like a structure, one element of which has special behavior. Given that the special behavior is tied to the other parts of the structure, the compound result is treated as a predefined construct.

#### 4.5.1. <typeRef> Element for Aliasing Existing Data Types

The <typeRef> element refers to an existing data type by its name. The referred data type MUST be defined either in the same library document, or in one of the included library documents. If the referred data type is an atomic data type, the newly defined type will also be regarded as atomic. If the referred data type is a compound type, the new type will also be compound. Some usage examples follow:

```
<dataTypeDef>
  <name>short</name>
  <synopsis>Alias to int16</synopsis>
  <typeRef>int16</typeRef>
</dataTypeDef>
<dataTypeDef>
  <name>ieeemacaddr</name>
  <synopsis>48-bit IEEE MAC address</synopsis>
  <typeRef>byte[6]</typeRef>
</dataTypeDef>
```

#### 4.5.2. <atomic> Element for Deriving New Atomic Types

The <atomic> element allows the definition of a new atomic type from an existing atomic type, applying range restrictions and/or providing special enumerated values. Note that the <atomic> element can only use atomic types as base types, and its result MUST be another atomic type.

For example, the following snippet defines a new "dscp" data type:

```
<dataTypeDef>
  <name>dscp</name>
  <synopsis>Diffserv code point.</synopsis>
  <atomic>
    <baseType>uchar</baseType>
    <rangeRestriction>
      <allowedRange min="0" max="63"/>
    </rangeRestriction>
    <specialValues>
      <specialValue value="0">
        <name>DSCP-BE</name>
        <synopsis>Best Effort</synopsis>
      </specialValue>
```

...

```
      </specialValues>
    </atomic>
  </dataTypeDef>
```

### 4.5.3. <array> Element to Define Arrays

The <array> element can be used to create a new compound data type as an array of a compound or an atomic data type. The type of the array entry can be specified either by referring to an existing type (using the <typeRef> element) or defining an unnamed type inside the <array> element using any of the <atomic>, <array>, <struct>, or <union> elements.

The array can be "fixed-size" or "variable-size", which is specified by the "type" attribute of the <array> element. The default is "variable-size". For variable size arrays, an optional "max-length" attribute specifies the maximum allowed length. This attribute should be used to encode semantic limitations, not implementation limitations. The latter should be handled by capability attributes of LFB classes, and should never be included in data type definitions. If the "max-length" attribute is not provided, the array is regarded as of unlimited-size.

For fixed-size arrays, a "length" attribute MUST be provided that specifies the constant size of the array.

The result of this construct MUST always be a compound type, even if the array has a fixed size of 1.

Arrays MUST only be subscripted by integers, and will be presumed to start with index 0.

In addition to their subscripts, arrays may be declared to have content keys. Such a declaration has several effects:

- Any declared key can be used in the ForCES protocol to select an element for operations (for details, see the protocol).

- In any instance of the array, each declared key must be unique within that instance. No two elements of an array may have the same values on all the fields which make up a key.

Each key is declared with a keyID for use in the protocol, where the unique key is formed by combining one or more specified key fields. To support the case where an array of an atomic type with unique values can be referenced by those values, the key field identifier may be "*" (i.e., the array entry is the key). If the value type of the array is a structure or an array, then the key is one or more fields, each identified by name. Since the field may be an element

of the structure, the element of an element of a structure, or further nested, the field name is actually a concatenated sequence of part identifiers, separated by decimal points ("."). The syntax for key field identification is given following the array examples.

The following example shows the definition of a fixed size array with a pre-defined data type as the array elements:

```
<dataTypeDef>
  <name>dscp-mapping-table</name>
  <synopsis>
    A table of 64 DSCP values, used to re-map code space.
  </synopsis>
  <array type="fixed-size" length="64">
      <typeRef>dscp</typeRef>
  </array>
</dataTypeDef>
```

The following example defines a variable size array with an upper limit on its size:

```
<dataTypeDef>
  <name>mac-alias-table</name>
  <synopsis>A table with up to 8 IEEE MAC addresses</synopsis>
  <array type="variable-size" max-length="8">
      <typeRef>ieeemacaddr</typeRef>
  </array>
</dataTypeDef>
```

The following example shows the definition of an array with a local (unnamed) type definition:

```
<dataTypeDef>
  <name>classification-table</name>
  <synopsis>
    A table of classification rules and result opcodes.
  </synopsis>
  <array type="variable-size">
    <struct>
      <element elementID="1">
        <name>rule</name>
        <synopsis>The rule to match</synopsis>
        <typeRef>classrule</typeRef>
      </element>
      <element elementID="2">
        <name>opcode</name>
        <synopsis>The result code</synopsis>
        <typeRef>opcode</typeRef>
```

```
      </element>
```

```
    </struct>
  </array>
</dataTypeDef>
```

In the above example, each entry of the array is a <struct> of two fields ("rule" and "opcode").

The following example shows a table of IP Prefix information that can be accessed by a multi-field content key on the IP Address and prefix length. This means that in any instance of this table, no two entries can have the same IP address and prefix length.

```
<dataTypeDef>
  <name>ipPrefixInfo_table</name>
  <synopsis>
    A table of information about known prefixes
  </synopsis>
  <array type="variable-size">
    <struct>
      <element elementID="1">
        <name>address-prefix</name>
        <synopsis>the prefix being described</synopsis>
        <typeRef>ipv4Prefix</typeRef>
      </element>
      <element elementID="2">
        <name>source</name>
        <synopsis>
            the protocol or process providing this information
        </synopsis>
        <typeRef>uint16</typeRef>
      </element>
      <element elementID="3">
        <name>prefInfo</name>
        <synopsis>the information we care about</synopsis>
        <typeRef>hypothetical-info-type</typeRef>
      </element>
    </struct>
    <key keyID="1">
      <keyField> address-prefix.ipv4addr </keyField>
      <keyField> address-prefix.prefixlen </keyField>
      <keyField> source </keyField>
    </key>
  </array>
</dataTypeDef>
```

Note that the keyField elements could also have been simply address-prefix and source, since all of the fields of address-prefix are being used.

#### 4.5.3.1 Key Field References

In order to use key declarations, one must refer to fields that are potentially nested inside other fields in the array. If there are nested arrays, one might even use an array element as a key (but great care would be needed to ensure uniqueness.)

The key is the combination of the values of each field declared in a keyField element.

Therefore, the value of a keyField element MUST be a concatenated Sequence of field identifiers, separated by a "." (period) character. Whitespace is permitted and ignored.

A valid string for a single field identifier within a keyField depends upon the current context. Initially, in an array key declaration, the context is the type of the array. Progressively, the context is whatever type is selected by the field identifiers processed so far in the current key field declaration.

When the current context is an array, (e.g., when declaring a key for an array whose content is an array) then the only valid value for the field identifier is an explicit number.

When the current context is a structure, the valid values for the field identifiers are the names of the elements of the structure. In the special case of declaring a key for an array containing an atomic type, where that content is unique and is to be used as a key, the value "*" can be used as the single key field identifier.

### 4.5.4. <struct> Element to Define Structures

A structure is comprised of a collection of data elements. Each data element has a data type (either an atomic type or an existing compound type) and is assigned a name unique within the scope of the compound data type being defined. These serve the same function as "struct" in C, etc.

The actual type of the field can be defined by referring to an existing type (using the <typeDef> element), or can be a locally defined (unnamed) type created by any of the <atomic>, <array>, <struct>, or <union> elements.

A structure definition is a series of element declarations. Each element carries an elementID for use by the ForCES protocol. In addition, the element contains the name, a synopsis, an optional description, an optional declaration that the element itself is optional, and the typeRef declaration that specifies the element type.

For a dataTypeDef of a struct, the structure definition may be inherited from, and augment, a previously defined structured type. This is indicated by including the derivedFrom attribute on the struct declaration.

The result of this construct MUST be a compound type, even when the <struct> contains only one field.

An example:

```
<dataTypeDef>
  <name>ipv4prefix</name>
  <synopsis>
    IPv4 prefix defined by an address and a prefix length
  </synopsis>
  <struct>
    <element elementID="1">
      <name>address</name>
      <synopsis>Address part</synopsis>
      <typeRef>ipv4addr</typeRef>
    </element>
    <element elementID="2">
      <name>prefixlen</name>
      <synopsis>Prefix length part</synopsis>
      <atomic>
        <baseType>uchar</baseType>
        <rangeRestriction>
          <allowedRange min="0" max="32"/>
        </rangeRestriction>
      </atomic>
    </element>
  </struct>
</dataTypeDef>
```

#### 4.5.5. <union> Element to Define Union Types

Similar to the union declaration in C, this construct allows the definition of overlay types. Its format is identical to the <struct> element.

The result of this construct MUST be a compound type, even when the union contains only one element.

#### 4.5.6 <alias> Element

It is sometimes necessary to have an element in an LFB or structure refer to information in other LFBs. The <alias> declaration creates the constructs for this. The content of an <alias> element MUST be a

named type. It can be a base type of a derived type. The actual value referenced by an alias is known as its target. When a GET or SET operation references the alias element, the value of the target is returned or replaced. Write access to an alias element is permitted if write access to both the alias and the target are permitted.

The target of an <alias> element is determined by its properties. Like all elements, the properties MUST include the support / read / write permission for the alias. In addition, there are several fields in the properties which define the target of the alias. These fields are the ID of the LFB class of the target, the ID of the LFB instance of the target, and a sequence of integers representing the path within the target LFB instance to the target element. The type of the target element must match the declared type of the alias. Details of the alias property structure in the section of this document on properties.

Note that the read / write property of the alias refers to the value. The CE can only determine if it can write the target selection properties of the alias by attempting such a write operation. (Property elements do not themselves have properties.)

#### 4.5.6. Augmentations

Compound types can also be defined as augmentations of existing compound types. If the existing compound type is a structure, augmentation may add new elements to the type. The type of an existing element can only be replaced with an augmentation derived from the current type, an existing element cannot be deleted. If the existing compound type is an array, augmentation means augmentation of the array element type.

One consequence of this is that augmentations are compatible with the compound type from which they are derived. As such, augmentations are useful in defining attributes for LFB subclasses with backward compatibility. In addition to adding new attributes to a class, the data type of an existing attribute may be replaced by an augmentation of that attribute, and still meet the compatibility rules for subclasses.

For example, consider a simple base LFB class A that has only one attribute (attr1) of type X. One way to derive class A1 from A can be by simply adding a second attribute (of any type). Another way to derive a class A2 from A can be by replacing the original attribute (attr1) in A of type X with one of type Y, where Y is an augmentation of X. Both classes A1 and A2 are backward compatible with class A.

The syntax for augmentations is to include a derivedFrom element in a structure definition, indicating what structure type is being augmented. Element names and element IDs within the augmentation must not be the same as those in the structure type being augmented.

### 4.6. <metadataDefs> Element for Metadata Definitions

The (optional) <metadataDefs> element in the library document contains one or more <metadataDef> elements. Each <metadataDef> element defines a metadata.

Each <metadataDef> element MUST contain a unique name (NMTOKEN). Uniqueness is defined to be over all metadata defined in this library document and in all directly or indirectly included library documents. The <metadataDef> element MUST also contain a brief synopsis, the mandatory tag value to be used for this metadata, an optional detailed description, and a mandatory type definition information. Only atomic data types can be used as value types for metadata.

Two forms of type definitions are allowed. The first form uses the <typeRef> element to refer to an existing atomic data type defined in the <dataTypeDefs> element of the same library document or in one of the included library documents. The usage of the <typeRef> element is identical to how it is used in the <dataTypeDef> elements, except here it can only refer to atomic types.
The latter restriction is not yet enforced by the XML schema.

The second form is an explicit type definition using the <atomic> element. This element is used here in the same way as in the <dataTypeDef> elements.

The following example shows both usages:

```
<metadataDefs>
  <metadataDef>
    <name>NEXTHOPID</name>
    <synopsis>Refers to a Next Hop entry in NH LFB</synopsis>
    <metadataID>17</metaDataID>
    <typeRef>int32</typeRef>
  </metadataDef>
  <metadataDef>
    <name>CLASSID</name>
    <synopsis>
      Result of classification (0 means no match).
    </synopsis>
    <metadataID>21</metadataID>
    <atomic>
```

```
          <baseType>int32</baseType>
```

```
        <specialValues>
          <specialValue value="0">
            <name>NOMATCH</name>
            <synopsis>
              Classification didn t result in match.
            </synopsis>
          </specialValue>
        </specialValues>
      </atomic>
    </metadataDef>
  </metadataDefs>
```

### 4.7. <LFBClassDefs> Element for LFB Class Definitions

The (optional) <LFBClassDefs> element can be used to define one or more LFB classes using <LFBClassDef> elements. Each <LFBClassDef> element MUST define an LFB class and include the following elements:

- <name> provides the symbolic name of the LFB class. Example: "ipv4lpm"
- <synopsis> provides a short synopsis of the LFB class. Example: "IPv4 Longest Prefix Match Lookup LFB"
- <version> is the version indicator
- <derivedFrom> is the inheritance indicator
- <inputPorts> lists the input ports and their specifications
- <outputPorts> lists the output ports and their specifications
- <attributes> defines the operational attributes of the LFB
- <capabilities> defines the capability attributes of the LFB
- <description> contains the operational specification of the LFB
- The LFBClassID attribute of the LFBClassDef element defines the ID for this class. These must be globally unique.
- <events> defines the events that can be generated by instances of this LFB.

LFB Class Names must be unique, in order to enable other documents to reference the classes by name, and to enable human readers to understand references to class names. While a complex naming structure could be created, simplicity is preferred. As given in the IANA considerations section of this document, the IANA will maintain a registry of LFB Class names and Class identifiers, along with a reference to the document defining the class.

Here is a skeleton of an example LFB class definition:

```
<LFBClassDefs>
  <LFBClassDef LFBClassID="12345">
    <name>ipv4lpm</name>
    <synopsis>IPv4 Longest Prefix Match Lookup LFB</synopsis>
```

<version>1.0</version>

```
      <derivedFrom>baseclass</derivedFrom>

      <inputPorts>
        ...
      </inputPorts>

      <outputPorts>
        ...
      </outputPorts>

      <attributes>
        ...
      </attributes>

      <capabilities>
        ...
      </capabilities>

      <description>
        This LFB represents the IPv4 longest prefix match lookup
        operation.
        The modeled behavior is as follows:
           Blah-blah-blah.
      </description>

    </LFBClassDef>
    ...
  </LFBClassDefs>
```

The individual attributes and capabilities will have elementIDs for use by the ForCES protocol. These parallel the elementIDs used in structs, and are used the same way. Attribute and capability elementIDs must be unique within the LFB class definition.

Note that the <name>, <synopsis>, and <version> elements are required, all other elements are optional in <LFBClassDef>. However, when they are present, they must occur in the above order.

### 4.7.1. <derivedFrom> Element to Express LFB Inheritance

The optional <derivedFrom> element can be used to indicate that this class is a derivative of some other class. The content of this element MUST be the unique name (<name>) of another LFB class. The referred LFB class MUST be defined in the same library document or in one of the included library documents.

It is assumed that the derived class is backwards compatible with the base class.

### 4.7.2. <inputPorts> Element to Define LFB Inputs

The optional <inputPorts> element is used to define input ports. An LFB class may have zero, one, or more inputs. If the LFB class has no input ports, the <inputPorts> element MUST be omitted. The <inputPorts> element can contain one or more <inputPort> elements, one for each port or port-group. We assume that most LFBs will have exactly one input. Multiple inputs with the same input type are modeled as one input group. Input groups are defined the same way as input ports by the <inputPort> element, differentiated only by an optional "group" attribute.

Multiple inputs with different input types should be avoided if possible (see discussion in Section 3.2.1). Some special LFBs will have no inputs at all. For example, a packet generator LFB does not need an input.

Single input ports and input port groups are both defined by the <inputPort> element; they are differentiated by only an optional "group" attribute.

The <inputPort> element MUST contain the following elements:

. <name> provides the symbolic name of the input. Example: "in". Note that this symbolic name must be unique only within the scope of the LFB class.
. <synopsis> contains a brief description of the input. Example: "Normal packet input".
. <expectation> lists all allowed frame formats. Example: {"ipv4" and "ipv6"}. Note that this list should refer to names specified in the <frameDefs> element of the same library document or in any included library documents. The <expectation> element can also provide a list of required metadata. Example: {"classid", "vifid"}. This list should refer to names of metadata defined in the <metadataDefs> element in the same library document or in any included library documents. For each metadata, it must be specified whether the metadata is required or optional. For each optional metadata, a default value must be specified, which is used by the LFB if the metadata is not provided with a packet.

In addition, the optional "group" attribute of the <inputPort> element can specify if the port can behave as a port group, i.e., it is allowed to be instantiated. This is indicated by a "yes" value (the default value is "no").

An example <inputPorts> element, defining two input ports, the second one being an input port group:

```
<inputPorts>
  <inputPort>
    <name>in</name>
    <synopsis>Normal input</synopsis>
    <expectation>
      <frameExpected>
        <ref>ipv4</ref>
        <ref>ipv6</ref>
      </frameExpected>
      <metadataExpected>
        <ref>classid</ref>
        <ref>vifid</ref>
        <ref dependency="optional" defaultValue="0">vrfid</ref>
      </metadataExpected>
    </expectation>
  </inputPort>
  <inputPort group="yes">
    ... another input port ...
  </inputPort>
</inputPorts>
```

For each <inputPort>, the frame type expectations are defined by the <frameExpected> element using one or more <ref> elements (see example above). When multiple frame types are listed, it means that "one of these" frame types is expected. A packet of any other frame type is regarded as incompatible with this input port of the LFB class. The above example list two frames as expected frame types: "ipv4" and "ipv6".

Metadata expectations are specified by the <metadataExpected> element. In its simplest form, this element can contain a list of <ref> elements, each referring to a metadata. When multiple instances of metadata are listed by <ref> elements, it means that "all of these" metadata must be received with each packet (except metadata that are marked as "optional" by the "dependency" attribute of the corresponding <ref> element). For a metadata that is specified "optional", a default value MUST be provided using the "defaultValue" attribute. The above example lists three metadata as expected metadata, two of which are mandatory ("classid" and "vifid"), and one being optional ("vrfid").

The schema also allows for more complex definitions of metadata expectations. For example, using the <one-of> element, a list of metadata can be specified to express that at least one of the specified metadata must be present with any packet. For example:

```
<metadataExpected>
  <one-of>
    <ref>prefixmask</ref>
    <ref>prefixlen</ref>
  </one-of>
</metadataExpected>
```

The above example specifies that either the "prefixmask" or the "prefixlen" metadata must be provided with any packet.

The two forms can also be combined, as it is shown in the following example:

```
<metadataExpected>
  <ref>classid</ref>
  <ref>vifid</ref>
  <ref dependency="optional" defaultValue="0">vrfid</ref>
  <one-of>
    <ref>prefixmask</ref>
    <ref>prefixlen</ref>
  </one-of>
</metadataExpected>
```

Although the schema is constructed to allow even more complex definitions of metadata expectations, we do not discuss those here.

### 4.7.3. <outputPorts> Element to Define LFB Outputs

The optional <outputPorts> element is used to define output ports. An LFB class may have zero, one, or more outputs. If the LFB class has no output ports, the <outputPorts> element MUST be omitted. The <outputPorts> element can contain one or more <outputPort> elements, one for each port or port-group. If there are multiple outputs with the same output type, we model them as an output port group. Some special LFBs may have no outputs at all (e.g., Dropper).

Single output ports and output port groups are both defined by the <outputPort> element; they are differentiated by only an optional "group" attribute.

The <outputPort> element MUST contain the following elements:

. <name> provides the symbolic name of the output. Example: "out". Note that the symbolic name must be unique only within the scope of the LFB class.
. <synopsis> contains a brief description of the output port. Example: "Normal packet output".
. <product> lists the allowed frame formats. Example: {"ipv4", "ipv6"}. Note that this list should refer to symbols specified in

the <frameDefs> element in the same library document or in any included library documents. The <product> element may also contain the list of emitted (generated) metadata. Example: {"classid", "color"}. This list should refer to names of metadata specified in the <metadataDefs> element in the same library document or in any included library documents. For each generated metadata, it should be specified whether the metadata is always generated or generated only in certain conditions. This information is important when assessing compatibility between LFBs.

In addition, the optional "group" attribute of the <outputPort> element can specify if the port can behave as a port group, i.e., it is allowed to be instantiated. This is indicated by a "yes" value (the default value is "no").

The following example specifies two output ports, the second being an output port group:

```
<outputPorts>
  <outputPort>
    <name>out</name>
    <synopsis>Normal output</synopsis>
    <product>
      <frameProduced>
        <ref>ipv4</ref>
        <ref>ipv4bis</ref>
      </frameProduced>
      <metadataProduced>
        <ref>nhid</ref>
        <ref>nhtabid</ref>
      </metadataProduced>
    </product>
  </outputPort>
  <outputPort group="yes">
    <name>exc</name>
    <synopsis>Exception output port group</synopsis>
    <product>
      <frameProduced>
        <ref>ipv4</ref>
        <ref>ipv4bis</ref>
      </frameProduced>
      <metadataProduced>
        <ref availability="conditional">errorid</ref>
      </metadataProduced>
    </product>
  </outputPort>
</outputPorts>
```

The types of frames and metadata the port produces are defined inside the <product> element in each <outputPort>. Within the <product> element, the list of frame types the port produces is listed in the <frameProduced> element. When more than one frame is listed, it means that "one of" these frames will be produced.

The list of metadata that is produced with each packet is listed in the optional <metadataProduced> element of the <product>. In its simplest form, this element can contain a list of <ref> elements, each referring to a metadata type. The meaning of such a list is that "all of" these metadata are provided with each packet, except those that are listed with the optional "availability" attribute set to "conditional". Similar to the <metadataExpected> element of the <inputPort>, the <metadataProduced> element supports more complex forms, which we do not discuss here further.

### 4.7.4. <attributes> Element to Define LFB Operational Attributes

Operational parameters of the LFBs that must be visible to the CEs are conceptualized in the model as the LFB attributes. These include, for example, flags, single parameter arguments, complex arguments, and tables. Note that the attributes here refer to only those operational parameters of the LFBs that must be visible to the CEs. Other variables that are internal to LFB implementation are not regarded as LFB attributes and hence are not covered.

Some examples for LFB attributes are:

- Configurable flags and switches selecting between operational modes of the LFB
- Number of inputs or outputs in a port group
- Metadata CONSUME vs.PROPAGATE mode selector
- Various configurable lookup tables, including interface tables, prefix tables, classification tables, DSCP mapping tables, MAC address tables, etc.
- Packet and byte counters
- Various event counters
- Number of current inputs or outputs for each input or output group

There may be various access permission restrictions on what the CE can do with an LFB attribute. The following categories may be supported:

- No-access attributes. This is useful when multiple access modes may be defined for a given attribute to allow some flexibility for different implementations.
- Read-only attributes.

. Read-write attributes.

. Write-only attributes. This could be any configurable data for which read capability is not provided to the CEs. (e.g., the security key information)
. Read-reset attributes. The CE can read and reset this resource, but cannot set it to an arbitrary value. Example: Counters.
. Firing-only attributes. A write attempt to this resource will trigger some specific actions in the LFB, but the actual value written is ignored.

The LFB class may define more than one possible access mode for a given attribute (for example, "write-only" and "read-write"), in which case it is left to the actual implementation to pick one of the modes. In such cases, a corresponding capability attribute must inform the CE about the access mode the actual LFB instance supports (see next subsection on capability attributes).

The attributes of the LFB class are listed in the <attributes> element. Each attribute is defined by an <attribute> element. An <attribute> element MUST contain the following elements:

. <name> defines the name of the attribute. This name must be unique among the attributes of the LFB class. Example: "version".
. <synopsis> should provide a brief description of the purpose of the attribute.
. <optional/> indicates that this attribute is optional.
. The data type of the attribute can be defined either via a reference to a predefined data type or providing a local definition of the type. The former is provided by using the <typeRef> element, which must refer to the unique name of an existing data type defined in the <dataTypeDefs> element in the same library document or in any of the included library documents. When the data type is defined locally (unnamed type), one of the following elements can be used: <atomic>, <array>, <struct>, and <union>. Their usage is identical to how they are used inside <dataTypeDef> elements (see Section 4.5).
. The optional <defaultValue> element can specify a default value for the attribute, which is applied when the LFB is initialized or reset.

The attribute element also MUST have an elementID attribute, which is a numeric value used by the ForCES protocol.

In addition to the above elements, the <attribute> element includes an optional "access" attribute, which can take any of the following values or even a list of these values: "read-only", "read-write", "write-only", "read-reset", and "trigger-only". The default access mode is "read-write".

Whether optional elements are supported, and whether elements defined as read-write can actually be written can be determined for a given LFB instance by the CE by reading the property information of that element.

The following example defines two attributes for an LFB:

```
<attributes>
  <attribute access="read-only" elementID=  1  >
    <name>foo</name>
    <synopsis>number of things</synopsis>
    <typeRef>uint32</typeRef>
  </attribute>
  <attribute access="read-write write-only" elementID=  2  >
    <name>bar</name>
    <synopsis>number of this other thing</synopsis>
    <atomic>
      <baseType>uint32</baseType>
      <rangeRestriction>
        <allowedRange min="10" max="2000"/>
      </rangeRestriction>
    </atomic>
    <defaultValue>10</defaultValue>
  </attribute>
</attributes>
```

The first attribute ("foo") is a read-only 32-bit unsigned integer, defined by referring to the built-in "uint32" atomic type. The second attribute ("bar") is also an integer, but uses the <atomic> element to provide additional range restrictions. This attribute has two possible access modes, "read-write" or "write-only". A default value of 10 is provided.

Note that not all attributes are likely to exist at all times in a particular implementation. While the capabilities will frequently indicate this non-existence, CEs may attempt to reference non-existent or non-permitted attributes anyway. The FORCES protocol mechanisms should include appropriate error indicators for this case.

The mechanism defined above for non-supported attributes can also apply to attempts to reference non-existent array elements or to set read-only elements.

#### 4.7.5. <capabilities> Element to Define LFB Capability Attributes

The LFB class specification provides some flexibility for the FE implementation regarding how the LFB class is implemented. For

example, the instance may have some limitations that are not inherent from the class definition, but rather the result of some implementation limitations. For example, an array attribute may be defined in the class definition as "unlimited" size, but the physical implementation may impose a hard limit on the size of the array.

Such capability related information is expressed by the capability attributes of the LFB class. The capability attributes are always read-only attributes, and they are listed in a separate <capabilities> element in the <LFBClassDef>. The <capabilities> element contains one or more <capability> elements, each defining one capability attribute. The format of the <capability> element is almost the same as the <attribute> element, it differs in two aspects: it lacks the access mode attribute (because it is always read-only), and it lacks the <defaultValue> element (because default value is not applicable to read-only attributes).

Some examples of capability attributes follow:

- The version of the LFB class that this LFB instance complies with;
- Supported optional features of the LFB class;
- Maximum number of configurable outputs for an output group;
- Metadata pass-through limitations of the LFB;
- Maximum size of configurable attribute tables;
- Additional range restriction on operational attributes;
- Supported access modes of certain attributes (if the access mode of an operational attribute is specified as a list of two or mode modes).

The following example lists two capability attributes:

```
<capabilities>
  <capability elementID="3">
    <name>version</name>
    <synopsis>
      LFB class version this instance is compliant with.
    </synopsis>
    <typeRef>version</typeRef>
  </capability>
  <capability elementID="4">
    <name>limitBar</name>
    <synopsis>
      Maximum value of the "bar" attribute.
    </synopsis>
    <typeRef>uint16</typeRef>
  </capability>
```

```
</capabilities>
```

#### 4.7.6. <events> Element for LFB Notification Generation

The <events> element contains the information about the occurrences for which instances of this LFB class can generate notifications to the CE.

The <events> definition needs a baseID attributevalue, which is normally <events baseID= number >. The value of the baseID is the starting element for the path which identifies events. It must not be the same as the elementID of any top level attribute or capability of the LFB class. In derived LFBs (i.e. ones with a <derivedFrom> element) where the parent LFB class has an events declaration, the baseID must not be present. Instead, the value from the parent class is used.

The <events> element contains 0 or more <event> elements, each of which declares a single event. The <event> element has an eventID attribute giving the unique ID of the event. The element will include:

- <eventTarget> element indicating which LFB field is tested to generate the event;
- condition element indicating what condition on the field will generate the event from a list of defined conditions;
- <eventReports> element indicating what values are to be reported in the notification of the event.

##### 4.7.6.1 <eventTarget> Element

The <eventTarget> element contains information identifying a field in the LFB. Specifically, the <target> element contains one or more <eventField> or <eventSubscript> elements. These elements represent the textual equivalent of a path select component of the LFB. The <eventField> element contains the name of an element in the LFB or struct. The first element in a <target> MUST be an <eventField> element and MUST name a field in the LFB. The following element MUST identify a valid field within the containing context. If an <eventField> identifies an array, and is not the last element in the target, then the next element MUST be an <eventSubscript>. <eventSubscript> elements MUST occur only after <eventField> names that identifies an array. An <eventSubscript> may contain a numeric value to indicate that this event applies to a specific element of the array. More commonly, the event is being defined across all elements of the array. In that case, <eventSubscript> will contain a name. The name in an <eventSubscript> element is not a field name. It is a variable name for use in the <report> elements of this LFB definition. This name MUST be distinct from any field name that can validly occur in the <eventReport> clause.  Hence it SHOULD be distinct from any field name used in the LFB or in structures used within the LFB.

The <eventTarget> provides additional components for the path used to reference the event. The path will be the baseID for events, followed by the ID for the specific event, followed by a value for each <eventSubscript> element in the <eventTarget>. This will identify a specific occurrence of the event. So, for example, it will appear in the event notification LFB. It is also used for the SET-PROPERTY operation to subscribe to a specific event. A SET-PROPERTY of the subscription property (but not of any other writeable properties) may be sent by the CE with any prefix of the path of the event. So, for an event defined on a table, a SET-PROPERTY with a path of the baseID and the eventID will subscribe the CE to all occurrences of that event on any element of the table. This is particularly useful for the <eventCreated/> and <eventDestroyed/> conditions. Events using those conditions will generally be defined with a field / subscript sequence that identifies an array and ends with an <eventSubscript> element. Thus, the event notification will indicate which array entry has been created or destroyed. A typical subscriber will subscribe for the array, as opposed to a specific element in an array, so it will use a shorter path.

Thus, if there is an LFB with an event baseID of 7, and a specific event with an event ID of 8, then one can subscribe to the event by referencing the properties of the LFB element with path 7.8. If the event target has no subscripts (for example, it is a simple attribute of the LFB) then one can also reference the event threshold and filtering properties via the properties on element 7.8. If the event target is defined as an element of an array, then the target definition will include an <eventSubscript> element. In that case, one can subscribe to the event for the entire array by referencing the properties of 7.8. One can also subscribe to a specific element, x, of the array by referencing the subscription property of 7.8.x and also access the threshold and filtering properties of 7.8.x. If the event is targeting an element of an array within an array, then there will be two (or conceivably more) <eventSubscript> elements in the target. If so, for the case of two elements, one would reference the properties of 7.8.x.y to get to the threshold and filtering properties of an individual event.

Threshold and filtering conditions can only be applied to individual events. For events defined on elements of an array, this specification does not allow for defining a threshold or filtering condition on an event for all elements of an array.

### 4.7.6.2 <events> Element Conditions

The condition element represents a condition that triggers a notification. The list of conditions is:

- <eventCreated/> the target must be an array, ending with a subscript indication. The event is generated when an entry in the array is created. This occurs even if the entry is created by CE direction.
- <eventDeleted/> the target must be an array, ending with a subscript indication. The event is generated when an entry in the array is destroyed. This occurs even if the entry is destroyed by CE direction.
- <eventChanged/> the event is generated whenever the target element changes in any way. For binary attributes such as up/down, this reflects a change in state. It can also be used with numeric attributes, in which case any change in value results in a detected trigger.
- <eventGreaterThan/> the event is generated whenever the target element becomes greater than the threshold. The threshold is an event property.
- <eventLessThan/> the event is generated whenever the target element becomes less than the threshold. The threshold is an event property.

As described in the Event Properties section, event items have properties associated with them. These properties include the subscription information (indicating whether the CE wishes the FE to generate event reports for the event at all, thresholds for events related to level crossing, and filtering conditions that may reduce the set of event notifications generated by the FE. Details of the filtering conditions that can be applied are given in that section. The filtering conditions allow the FE to suppress floods of events that could result from oscillation around a condition value. For FEs that do not wish to support filtering, the filter properties can either be read only or not supported.

### 4.7.6.3 <eventReports> Element

The <eventReports> element of an <event> describes the information to be delivered by the FE along with the notification of the occurrence of the event. The <reports> element contains one or more <eventReport> elements. Each <report> element identifies a piece of data from the LFB to be reported. The notification carries that data as if the collection of <eventReport> elements had been defined in a structure. Each <eventReport> element thus MUST identify a field in the LFB. The syntax is exactly the same as used in the <eventTarget> element, using <eventField> and <eventSubscript>

elements.  <eventSubcripts> may contain integers.  If they contain names, they MUST be names from <eventSubscript> elements of the <eventTarget>. The selection for the report will use the value for the subscript that identifies that specific element triggering the event. This can be used to reference the structure / field causing the event, or to reference related information in parallel tables. This event reporting structure is designed to allow the LFB designer to specify information that is likely not known a priori by the CE and is likely needed by the CE to process the event. While the structure allows for pointing at large blocks of information (full arrays or complex structures) this is not recommended. Also, the variable reference / subscripting in reporting only captures a small portion of the kinds of related information. Chaining through index fields stored in a table, for example, is not supported. In general, the <eventReports> mechanism is an optimization for cases that have been found to be common, saving the CE from having to query for information it needs to understand the event. It does not represent all possible information needs.

If any elements referenced by the eventReport are optional, then the report MUST support optional elements. Any components which do not exist are not reported.

### 4.7.7. <description> Element for LFB Operational Specification

The <description> element of the <LFBClass> provides unstructured text (in XML sense) to verbally describe what the LFB does.

4.8.Properties

Elements of LFBs have properties which are important to the CE. The most important property is the existence / readability / writeability of the element. Depending up the type of the element, other information may be of importance.

The model provides the definition of the structure of property information. There is a base class of property information. For the array, alias, and event elements there are subclasses of property information providing additional fields. This information is accessed by the CE (and updated where applicable) via the PL protocol. While some property information is writeable, there is no mechanism currently provided for checking the properties of a property element. Writeability can only be checked by attempting to modify the value.

### 4.8.1. Basic Properties

The basic property definition, along with the scalar for accessibility is below. Note that this access permission information is generally read-only.

```
<dataTypeDef>
  <name>accessPermissionValues</name>
  <synopsis>
    The possible values of attribute access permission
  </synopsis>
  <atomic>
    <baseType>uchar</baseType>
    <specialValues>
      <specialValue value="0">
        <name>None</name>
        <synopsis>Access is prohibited</synopsis>
      </specialValue>
       <specialValue value="1">
        <name> Read-Only </name>
        <synopsis>Access is read only</synopsis>
      </specialValue>
      <specialValue value="2">
        <name>Write-Only</name>
        <synopsis>
          The attribute may be written, but not read
        </synopsis>
      </specialValue>
      <specialValue value="3">
        <name>Read-Write</name>
        <synopsis>
          The attribute may be read or written
        </synopsis>
      </specialValue>
    </specialValues>
  </atomic>
</dataTypeDef>

<dataTypeDef>
  <name>baseElementProperties</name>
  <synopsis>basic properties, accessibility</synopsis>
  <struct>
    <element elementID="1">
      <name>accessibility</name>
      <synopsis>
          does the element exist, and
          can it be read or written
      </synopsis>
      <typeRef>accessPermissionValues</typeRef>
    </element>
  </struct>
</dataTypeDef>
```

### 4.8.2. Array Properties

The properties for an array add a number of important pieces of information. These properties are also read-only.

```
<dataTypeDef>
  <name>arrayElementProperties</name>
  <struct>
    <derivedFrom>baseElementProperties</derivedFrom>
    <element elementID=  2  >
      <name>entryCount</name>
      <synopsis>the number of entries in the array</synopsis>
      <typeRef>uint32</typeRef>
    </element>
    <element elementID=  3  >
      <name>highestUsedSubscript</name>
      <synopsis>the last used subscript in the array</synopsis>
      <typeRef>uint32</typeRef>
    </element>
    <element elementID=  4  >
      <name>firstUnusedSubscript</name>
      <synopsis>
        The subscript of the first unused array element
      </synopsis>
      <typeRef>uint32</typeRef>
    </element>
  </struct>
</dataTypeDef>
```

### 4.8.3. String Properties

The properties of a string specify the actual octet length and the maximum octet length for the element. The maximum length is included because an FE implementation may limit a string to be shorter than the limit in the LFB Class definition.

```
<dataTypeDef>
  <name>stringElementProperties</name>
  <struct>
    <derivedFrom>baseElementProperties</derivedFrom>
    <element elementID=  2  >
      <name>stringLength</name>
      <synopsis>the number of octets in the string</synopsis>
      <typeRef>uint32</typeRef>
    </element>
    <element elementID=  3  >
      <name>maxStringLength</name>
      <synopsis>
```

the maximum number of octets in the string

```
            </synopsis>
            <typeRef>uint32</typeRef>
          </element>
        </struct>
      </dataTypeDef>
```

### 4.8.4. Octetstring Properties

The properties of an octetstring specify the actual length and the maximum length, since the FE implementation may limit an octetstring to be shorter than the LFB Class definition.

```
      <dataTypeDef>
        <name>octetstringElementProperties</name>
        <struct>
          <derivedFrom>baseElementProperties</derivedFrom>
          <element elementID=  2  >
            <name>octetstringLength</name>
            <synopsis>
              the number of octets in the octetstring
            </synopsis>
            <typeRef>uint32</typeRef>
          </element>
          <element elementID=  3  >
            <name>maxOctetstringLength</name>
            <synopsis>
              the maximum number of octets in the octetstring
            </synopsis>
            <typeRef>uint32</typeRef>
          </element>
        </struct>
      </dataTypeDef>
```

### 4.8.5. Event Properties

The properties for an event add three (usually) writeable fields. One is the subscription field.  0 means no notification is generated.  Any non-zero value (typically 1 is used) means that a notification is generated.  The hysteresis field is used to suppress generation of notifications for oscillations around a condition value, and is described in the text for events.  The threshold field is used for the <eventGreaterThan/> and <eventLessThan/> conditions. It indicates the value to compare the event target against.  Using the properties allows the CE to set the level of interest.  FEs which do not supporting setting the threshold for events will make this field read-only.

```
        <dataTypeDef>
          <name>eventElementProperties</name>
          <struct>
            <derivedFrom>baseElementProperties</derivedFrom>
            <element elementID=  2  >
              <name>registration</name>
              <synopsis>
                has the CE registered to be notified of this event
              </synopsis>
              <typeRef>uint32</typeRef>
            </element>
            <element elementID=  3  >
              <name>threshold</name>
              <synopsis> comparison value for level crossing events
              </synopsis>
              </optional
              <typeRef>uint32</typeRef>
            </element>
            <element elementID=  4  >
              <name>eventHysteresis</name>
              <synopsis> region to suppress event recurrence notices
              </synopsis>
              </optional>
              <typeRef>uint32</typeRef>
            </element>
            <element elementID=  5  >
              <name>eventCount</name>
              <synopsis> number of occurrences to suppress
              </synopsis>
              </optional>
              <typeRef>uint32</typeRef>
            </element>
            <element elementID=  6  >
              <name>eventHysteresis</name>
              <synopsis> time interval in ms between notifications
              </synopsis>
              </optional>
              <typeRef>uint32</typeRef>
            </element>
          </struct>
        <dataTypeDef>
```

#### 4.8.5.1. Common Event Filtering

The event properties have values for controlling several filter conditions.  Support of these conditions is optional, but all conditions SHOULD be supported.  Events which are reliably known not to be subject to rapid occurrence or other concerns may not support all filter conditions.

Currently, three different filter condition variables are defined. These are eventCount, eventInterval, and eventHysteresis. Setting the condition variables to 0 (their default value) means that the condition is not checked.

Conceptually, when an event is triggered, all configured conditions are checked. If no filter conditions are triggered, or if any trigger conditions are met, the event notification is generated. If there are filter conditions, and no condition is met, then no event notification is generated. Event filter conditions have reset behavior when an event notification is generated. If any condition is passed, and the notification is generated, the notification reset behavior is performed on all conditions, even those which had not passed. This provides a clean definition of the interaction of the various event conditions.

An example of the interaction of conditions is an event with an eventCount property set to 5 and an eventInterval property set to 500 milliseconds. Suppose that a burst of occurrences of this event is detected by the FE. The first occurrence will cause a notification to be sent to the CE. Then, if four more occurrences are detected rapidly (less than 0.5 seconds) they will not result in notifications. If two more occurrences are detected, then the second of those will result in a notification. Alternatively, if more than 500 milliseconds has passed since the notification and an occurrence is detected, that will result in a notification. In either case, the count and time interval suppression is reset no matter which condition actually caused the notification.

#### 4.8.5.2. Event Hysteresis Filtering

Events with numeric conditions can have hysteresis filters applied to them. The hysteresis level is defined by a property of the event. This allows the FE to notify the CE of the hysteresis applied, and if it chooses, the FE can allow the CE to modify the hysteresis. This applies to <eventChanged/> for a numeric field, and to <eventGreaterThan/> and <eventLessThan/>. The content of a <variance> element is a numeric value. When supporting hysteresis, the FE MUST track the value of the element and make sure that the condition has become untrue by at least the hysteresis from the event property. To be specific, if the hysteresis is V, then

. For a <eventChanged/> condition, if the last notification was for value X, then the <changed/> notification MUST NOT be generated until the value reaches X +/- V.
. For a <eventGreaterThan/> condition with threshold T, once the event has been generated at least once it MUST NOT be generated

again until the field first becomes less than or equal to T - -
V, and then exceeds T.

. For a <eventLessThan/> condition with threshold T, once the event has been generate at least once it MUST NOT be generated again until the field first becomes greater than or equal to T + V, and then becomes less than T.

#### 4.8.5.3. Event Count Filtering

Events may have a count filtering condition. This property, if set to a non-zero value, indicates the number of occurrences of the event that should be considered redundant and not result in a notification. Thus, if this property is set to 1, and no other conditions apply, then every other detected occurrence of the event will result in a notification. This particular meaning is chosen so that the value 1 has a distinct meaning from the value 0.

A conceptual implementation (not required) for this might be an internal suppression counter. Whenever an event is triggered, the counter is checked. If the counter is 0, a notification is generated. Whether a notification is generated or not, the counter is incremented. If the counter exceeds the configured value, it is reset to 0. In this conceptual implementation the reset behavior when a notification is generated can be thought of as setting the counter to 1.

#### 4.8.5.4. Event Time Filtering

Events may have a time filtering condition. This property represents the minimum time interval (in the absence of some other filtering condition being passed) between generating notifications of detected events. This condition MUST only be passed if the time since the last notification of the event is longer than the configured interval in milliseconds.

Conceptually, this can be thought of as a stored timestamp which is compared with the detection time, or as a timer that is running that resets a suppression flag. In either case, if a notification is generated due to passing any condition then the time interval detection MUST be restarted.

### 4.8.6. Alias Properties

The properties for an alias add three (usually) writeable fields. These combine to identify the target element the subject alias refers to.

```
        <dataTypeDef>
          <name>aliasElementProperties</name>
          <struct>
            <derivedFrom>baseElementProperties</derivedFrom>
            <element elementID=  2  >
              <name>targetLFBClass</name>
              <synopsis>the class ID of the alias target</synopsis>
              <typeRef>uint32</typeRef>
            </element>
            <element elementID=  3  >
              <name>targetLFBInstance</name>
              <synopsis>the instance ID of the alias target</synopsis>
              <typeRef>uint32</typeRef>
            </element>
            <element elementID=  4  >
              <name>targetElementPath</name>
              <synopsis>
                the path to the element target
                each 4 octets is read as one path element,
                using the path construction in the PL protocol.
              </synopsis>
              <typeRef>octetstring[128]</typeRef>
            </element>
          </struct>
        </dataTypeDef>
```

## 4.9. XML Schema for LFB Class Library Documents

```
   <?xml version="1.0" encoding="UTF-8"?>
   <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://ietf.org/forces/1.0/lfbmodel"
    xmlns:lfb="http://ietf.org/forces/1.0/lfbmodel"
    targetNamespace="http://ietf.org/forces/1.0/lfbmodel"
    attributeFormDefault="unqualified"
    elementFormDefault="qualified">
   <xsd:annotation>
     <xsd:documentation xml:lang="en">
     Schema for Defining LFB Classes and associated types (frames,
     data types for LFB attributes, and metadata).
     </xsd:documentation>
   </xsd:annotation>
   <xsd:element name="description" type="xsd:string"/>
   <xsd:element name="synopsis" type="xsd:string"/>
   <!-- Document root element: LFBLibrary -->
   <xsd:element name="LFBLibrary">
     <xsd:complexType>
       <xsd:sequence>
         <xsd:element ref="description" minOccurs="0"/>
```

```
<xsd:element name="load" type="loadType" minOccurs="0"
```

```
                      maxOccurs="unbounded"/>
         <xsd:element name="frameDefs" type="frameDefsType"
                      minOccurs="0"/>
         <xsd:element name="dataTypeDefs" type="dataTypeDefsType"
                      minOccurs="0"/>
         <xsd:element name="metadataDefs" type="metadataDefsType"
                      minOccurs="0"/>
         <xsd:element name="LFBClassDefs" type="LFBClassDefsType"
                      minOccurs="0"/>
       </xsd:sequence>
       <xsd:attribute name="provides" type="xsd:Name" use="required"/>
     </xsd:complexType>
     <!-- Uniqueness constraints -->
     <xsd:key name="frame">
       <xsd:selector xpath="lfb:frameDefs/lfb:frameDef"/>
       <xsd:field xpath="lfb:name"/>
     </xsd:key>
     <xsd:key name="dataType">
       <xsd:selector xpath="lfb:dataTypeDefs/lfb:dataTypeDef"/>
       <xsd:field xpath="lfb:name"/>
     </xsd:key>
     <xsd:key name="metadataDef">
       <xsd:selector xpath="lfb:metadataDefs/lfb:metadataDef"/>
       <xsd:field xpath="lfb:name"/>
     </xsd:key>
     <xsd:key name="LFBClassDef">
       <xsd:selector xpath="lfb:LFBClassDefs/lfb:LFBClassDef"/>
       <xsd:field xpath="lfb:name"/>
     </xsd:key>
   </xsd:element>
   <xsd:complexType name="loadType">
     <xsd:attribute name="library" type="xsd:Name" use="required"/>
     <xsd:attribute name="location" type="xsd:anyURI" use="optional"/>
   </xsd:complexType>
   <xsd:complexType name="frameDefsType">
     <xsd:sequence>
       <xsd:element name="frameDef" maxOccurs="unbounded">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element name="name" type="xsd:NMTOKEN"/>
             <xsd:element ref="synopsis"/>
             <xsd:element ref="description" minOccurs="0"/>
           </xsd:sequence>
         </xsd:complexType>
       </xsd:element>
     </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="dataTypeDefsType">
```

```
     <xsd:sequence>
```

```
      <xsd:element name="dataTypeDef" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="name" type="xsd:NMTOKEN"/>
            <xsd:element ref="synopsis"/>
            <xsd:element ref="description" minOccurs="0"/>
            <xsd:group ref="typeDeclarationGroup"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <!--
     Predefined (built-in) atomic data-types are:
         char, uchar, int16, uint16, int32, uint32, int64, uint64,
         string[N], string, byte[N], boolean, octetstring[N],
         float16, float32, float64
  -->
  <xsd:group name="typeDeclarationGroup">
    <xsd:choice>
      <xsd:element name="typeRef" type="typeRefNMTOKEN"/>
      <xsd:element name="atomic" type="atomicType"/>
      <xsd:element name="array" type="arrayType"/>
      <xsd:element name="struct" type="structType"/>
      <xsd:element name="union" type="structType"/>
      <xsd:element name="alias" type="typeRefNMTOKEN"/>
    </xsd:choice>
  </xsd:group>
  <xsd:simpleType name="typeRefNMTOKEN">
    <xsd:restriction base="xsd:token">
      <xsd:pattern value="\c+"/>
      <xsd:pattern value="string\[\d+\]"/>
      <xsd:pattern value="byte\[\d+\]"/>
      <xsd:pattern value="octetstring\[\d+\]"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="atomicType">
    <xsd:sequence>
      <xsd:element name="baseType" type="typeRefNMTOKEN"/>
      <xsd:element name="rangeRestriction"
                   type="rangeRestrictionType" minOccurs="0"/>
      <xsd:element name="specialValues" type="specialValuesType"
                   minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="rangeRestrictionType">
    <xsd:sequence>
      <xsd:element name="allowedRange" maxOccurs="unbounded">
```

```
        <xsd:complexType>
```

```
           <xsd:attribute name="min" type="xsd:integer"
   use="required"/>
           <xsd:attribute name="max" type="xsd:integer"
   use="required"/>
         </xsd:complexType>
       </xsd:element>
     </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="specialValuesType">
     <xsd:sequence>
       <xsd:element name="specialValue" maxOccurs="unbounded">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element name="name" type="xsd:NMTOKEN"/>
             <xsd:element ref="synopsis"/>
           </xsd:sequence>
           <xsd:attribute name="value" type="xsd:token"/>
         </xsd:complexType>
       </xsd:element>
     </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="arrayType">
     <xsd:sequence>
       <xsd:group ref="typeDeclarationGroup"/>
       <xsd:element name="contentKey" minOccurs="0"
                    maxOccurs="unbounded">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element name="contentKeyField" maxOccurs="unbounded"
                          type="xsd:string"/>
           </xsd:sequence>
           <xsd:attribute name="contentKeyID" use="required"
                          type="xsd:integer"/>
         </xsd:complexType>
         <!--declare keys to have unique IDs -->
         <xsd:key name="contentKeyID">
           <xsd:selector xpath="lfb:contentKey"/>
           <xsd:field xpath="@contentKeyID"/>
         </xsd:key>
       </xsd:element>
     </xsd:sequence>
     <xsd:attribute name="type" use="optional"
                    default="variable-size">
       <xsd:simpleType>
         <xsd:restriction base="xsd:string">
           <xsd:enumeration value="fixed-size"/>
           <xsd:enumeration value="variable-size"/>
         </xsd:restriction>
```

```
      </xsd:simpleType>
```

```
      </xsd:attribute>
      <xsd:attribute name="length" type="xsd:integer" use="optional"/>
      <xsd:attribute name="maxLength" type="xsd:integer"
                     use="optional"/>
    </xsd:complexType>
    <xsd:complexType name="structType">
      <xsd:sequence>
        <xsd:element name="derivedFrom" type="typeRefNMTOKEN"
                     minOccurs="0"/>
        <xsd:element name="element" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="name" type="xsd:NMTOKEN"/>
              <xsd:element ref="synopsis"/>
              <xsd:element name="optional" minOccurs="0"/>
              <xsd:group ref="typeDeclarationGroup"/>
            </xsd:sequence>
            <xsd:attribute name="elementID" use="required"
                           type="xsd:integer"/>
          </xsd:complexType>
          <!-- key declaration to make elementIDs unique in a struct
          -->
          <xsd:key name="structElementID">
            <xsd:selector xpath="lfb:element"/>
            <xsd:field xpath="@elementID"/>
          </xsd:key>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="metadataDefsType">
      <xsd:sequence>
        <xsd:element name="metadataDef" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="name" type="xsd:NMTOKEN"/>
              <xsd:element ref="synopsis"/>
              <xsd:element name="metadataID" type="xsd:integer"/>
              <xsd:element ref="description" minOccurs="0"/>
              <xsd:choice>
                <xsd:element name="typeRef" type="typeRefNMTOKEN"/>
                <xsd:element name="atomic" type="atomicType"/>
              </xsd:choice>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="LFBClassDefsType">
```

```
      <xsd:sequence>
```

```
       <xsd:element name="LFBClassDef" maxOccurs="unbounded">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element name="name" type="xsd:NMTOKEN"/>
             <xsd:element ref="synopsis"/>
             <xsd:element name="version" type="versionType"/>
             <xsd:element name="derivedFrom" type="xsd:NMTOKEN"
                          minOccurs="0"/>
             <xsd:element name="inputPorts" type="inputPortsType"
                          minOccurs="0"/>
             <xsd:element name="outputPorts" type="outputPortsType"
                          minOccurs="0"/>
             <xsd:element name="attributes" type="LFBAttributesType"
                          minOccurs="0"/>
             <xsd:element name="capabilities"
                          type="LFBCapabilitiesType" minOccurs="0"/>
             <xsd:element name="events"
                          type="eventsType" minOccurs="0"/>
             <xsd:element ref="description" minOccurs="0"/>
           </xsd:sequence>
           <xsd:attribute name="LFBClassID" use="required"
                          type="xsd:integer"/>
         </xsd:complexType>
         <!-- Key constraint to ensure unique attribute names within
              a class:
         -->
         <xsd:key name="attributes">
           <xsd:selector xpath="lfb:attributes/lfb:attribute"/>
           <xsd:field xpath="lfb:name"/>
         </xsd:key>
         <xsd:key name="capabilities">
           <xsd:selector xpath="lfb:capabilities/lfb:capability"/>
           <xsd:field xpath="lfb:name"/>
         </xsd:key>
         <!-- does the above ensure that attributes and capabilities
              have different names?
              If so, the following is the elementID constraint
         -->
         <xsd:key name="attributeIDs">
           <xsd:selector xpath="lfb:attributes/lfb:attribute"/>
           <xsd:field xpath="@elementID"/>
         </xsd:key>
         <xsd:key name="capabilityIDs">
           <xsd:selector xpath="lfb:attributes/lfb:capability"/>
           <xsd:field xpath="@elementID"/>
         </xsd:key>
       </xsd:element>
     </xsd:sequence>
```

```
   </xsd:complexType>
```

```
   <xsd:simpleType name="versionType">
     <xsd:restriction base="xsd:NMTOKEN">
       <xsd:pattern value="[1-9][0-9]*\.([1-9][0-9]*|0)"/>
     </xsd:restriction>
   </xsd:simpleType>
   <xsd:complexType name="inputPortsType">
     <xsd:sequence>
       <xsd:element name="inputPort" type="inputPortType"
                    maxOccurs="unbounded"/>
     </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="inputPortType">
     <xsd:sequence>
       <xsd:element name="name" type="xsd:NMTOKEN"/>
       <xsd:element ref="synopsis"/>
       <xsd:element name="expectation" type="portExpectationType"/>
       <xsd:element ref="description" minOccurs="0"/>
     </xsd:sequence>
     <xsd:attribute name="group" type="booleanType" use="optional"
                    default="no"/>
   </xsd:complexType>
   <xsd:complexType name="portExpectationType">
     <xsd:sequence>
       <xsd:element name="frameExpected" minOccurs="0">
         <xsd:complexType>
           <xsd:sequence>
             <!-- ref must refer to a name of a defined frame type -->
             <xsd:element name="ref" type="xsd:string"
                          maxOccurs="unbounded"/>
           </xsd:sequence>
         </xsd:complexType>
       </xsd:element>
       <xsd:element name="metadataExpected" minOccurs="0">
         <xsd:complexType>
           <xsd:choice maxOccurs="unbounded">
             <!-- ref must refer to a name of a defined metadata -->
             <xsd:element name="ref" type="metadataInputRefType"/>
             <xsd:element name="one-of"
                          type="metadataInputChoiceType"/>
           </xsd:choice>
         </xsd:complexType>
       </xsd:element>
     </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="metadataInputChoiceType">
     <xsd:choice minOccurs="2" maxOccurs="unbounded">
       <!-- ref must refer to a name of a defined metadata -->
       <xsd:element name="ref" type="xsd:NMTOKEN"/>
```

```
      <xsd:element name="one-of" type="metadataInputChoiceType"/>
```

```
        <xsd:element name="metadataSet" type="metadataInputSetType"/>
      </xsd:choice>
    </xsd:complexType>
    <xsd:complexType name="metadataInputSetType">
      <xsd:choice minOccurs="2" maxOccurs="unbounded">
        <!-- ref must refer to a name of a defined metadata -->
        <xsd:element name="ref" type="metadataInputRefType"/>
        <xsd:element name="one-of" type="metadataInputChoiceType"/>
      </xsd:choice>
    </xsd:complexType>
    <xsd:complexType name="metadataInputRefType">
      <xsd:simpleContent>
        <xsd:extension base="xsd:NMTOKEN">
          <xsd:attribute name="dependency" use="optional"
                         default="required">
            <xsd:simpleType>
              <xsd:restriction base="xsd:string">
                <xsd:enumeration value="required"/>
                <xsd:enumeration value="optional"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:attribute>
          <xsd:attribute name="defaultValue" type="xsd:token"
                         use="optional"/>
        </xsd:extension>
      </xsd:simpleContent>
    </xsd:complexType>
    <xsd:complexType name="outputPortsType">
      <xsd:sequence>
        <xsd:element name="outputPort" type="outputPortType"
                     maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="outputPortType">
      <xsd:sequence>
        <xsd:element name="name" type="xsd:NMTOKEN"/>
        <xsd:element ref="synopsis"/>
        <xsd:element name="product" type="portProductType"/>
        <xsd:element ref="description" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="group" type="booleanType" use="optional"
                     default="no"/>
    </xsd:complexType>
    <xsd:complexType name="portProductType">
      <xsd:sequence>
        <xsd:element name="frameProduced">
          <xsd:complexType>
            <xsd:sequence>
```

```
            <!-- ref must refer to a name of a defined frame type
```

```
                -->
             <xsd:element name="ref" type="xsd:NMTOKEN"
                          maxOccurs="unbounded"/>
           </xsd:sequence>
         </xsd:complexType>
       </xsd:element>
       <xsd:element name="metadataProduced" minOccurs="0">
         <xsd:complexType>
           <xsd:choice maxOccurs="unbounded">
             <!-- ref must refer to a name of a defined metadata
             -->
             <xsd:element name="ref" type="metadataOutputRefType"/>
             <xsd:element name="one-of"
                          type="metadataOutputChoiceType"/>
           </xsd:choice>
         </xsd:complexType>
       </xsd:element>
     </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="metadataOutputChoiceType">
     <xsd:choice minOccurs="2" maxOccurs="unbounded">
       <!-- ref must refer to a name of a defined metadata -->
       <xsd:element name="ref" type="xsd:NMTOKEN"/>
       <xsd:element name="one-of" type="metadataOutputChoiceType"/>
       <xsd:element name="metadataSet" type="metadataOutputSetType"/>
     </xsd:choice>
   </xsd:complexType>
   <xsd:complexType name="metadataOutputSetType">
     <xsd:choice minOccurs="2" maxOccurs="unbounded">
       <!-- ref must refer to a name of a defined metadata -->
       <xsd:element name="ref" type="metadataOutputRefType"/>
       <xsd:element name="one-of" type="metadataOutputChoiceType"/>
     </xsd:choice>
   </xsd:complexType>
   <xsd:complexType name="metadataOutputRefType">
     <xsd:simpleContent>
       <xsd:extension base="xsd:NMTOKEN">
         <xsd:attribute name="availability" use="optional"
                        default="unconditional">
           <xsd:simpleType>
             <xsd:restriction base="xsd:string">
               <xsd:enumeration value="unconditional"/>
               <xsd:enumeration value="conditional"/>
             </xsd:restriction>
           </xsd:simpleType>
         </xsd:attribute>
       </xsd:extension>
     </xsd:simpleContent>
```

```
   </xsd:complexType>
```

```
<xsd:complexType name="LFBAttributesType">
  <xsd:sequence>
    <xsd:element name="attribute" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="name" type="xsd:NMTOKEN"/>
          <xsd:element ref="synopsis"/>
          <xsd:element ref="description" minOccurs="0"/>
          <xsd:element name="optional" minOccurs="0"/>
          <xsd:group ref="typeDeclarationGroup"/>
          <xsd:element name="defaultValue" type="xsd:token"
                       minOccurs="0"/>
        </xsd:sequence>
        <xsd:attribute name="access" use="optional"
                       default="read-write">
          <xsd:simpleType>
            <xsd:list itemType="accessModeType"/>
          </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="elementID" use="required"
                       type="xsd:integer"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
<xsd:simpleType name="accessModeType">
  <xsd:restriction base="xsd:NMTOKEN">
    <xsd:enumeration value="read-only"/>
    <xsd:enumeration value="read-write"/>
    <xsd:enumeration value="write-only"/>
    <xsd:enumeration value="read-reset"/>
    <xsd:enumeration value="trigger-only"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:complexType name="LFBCapabilitiesType">
  <xsd:sequence>
    <xsd:element name="capability" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="name" type="xsd:NMTOKEN"/>
          <xsd:element ref="synopsis"/>
          <xsd:element ref="description" minOccurs="0"/>
          <xsd:element name="optional" minOccurs="0"/>
          <xsd:group ref="typeDeclarationGroup"/>
        </xsd:sequence>
        <xsd:attribute name="elementID" use="required"
                       type="xsd:integer"/>
      </xsd:complexType>
```

```
      </xsd:element>
```

```
     </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="eventsType">
     <xsd:sequence>
       <xsd:element name="event" maxOccurs="unbounded">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element name="name" type="xsd:NMTOKEN"/>
             <xsd:element ref="synopsis"/>
             <xsd:element name="eventTarget" type="eventPathType"/>
             <xsd:element ref="eventCondition"/>
             <xsd:element name="eventReports" type="eventReportsType"
                          minOccurs="0"/>
             <xsd:element ref="description" minOccurs="0"/>
           </xsd:sequence>
           <xsd:attribute name="eventID" use="required"
                          type="xsd:integer"/>
         </xsd:complexType>
       </xsd:element>
     </xsd:sequence>
     <xsd:attribute name="baseID" type="xsd:integer"
                    use="optional"/>

   </xsd:complexType>
   <!-- the substitution group for the event conditions -->
   <xsd:element name="eventCondition" abstract="true"/>
   <xsd:element name="eventCreated"
               substitutionGroup="eventCondition"/>
   <xsd:element name="eventDeleted"
               substitutionGroup="eventCondition"/>
   <xsd:element name="eventChanged"
               substitutionGroup="eventCondition"/>
   <xsd:element name="eventGreaterThan"
               substitutionGroup="eventCondition"/>
   <xsd:element name="eventLessThan"
               substitutionGroup="eventCondition"/>
   <xsd:complexType name="eventPathType">
     <xsd:sequence>
       <xsd:element ref="eventPathPart" maxOccurs="unbounded"/>
     </xsd:sequence>
   </xsd:complexType>
   <!-- the substitution group for the event path parts -->
   <xsd:element name="eventPathPart" type="xsd:string"
                abstract="true"/>
   <xsd:element name="eventField" type="xsd:string"
                substitutionGroup="eventPathPart"/>
   <xsd:element name="eventSubscript" type="xsd:string"
                substitutionGroup="eventPathPart"/>
```

```
   <xsd:complexType name="eventReportsType">
```

```
      <xsd:sequence>
        <xsd:element name="eventReport" type="eventPathType"
                     maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:simpleType name="booleanType">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="0"/>
        <xsd:enumeration value="1"/>
      </xsd:restriction>
    </xsd:simpleType>
    </xsd:schema>
```

## 5. FE Attributes and Capabilities

A ForCES forwarding element handles traffic on behalf of a ForCES control element. While the standards will describe the protocol and mechanisms for this control, different implementations and different instances will have different capabilities. The CE MUST be able to determine what each instance it is responsible for is actually capable of doing. As stated previously, this is an approximation. The CE is expected to be prepared to cope with errors in requests and variations in detail not captured by the capabilities information about an FE.

In addition to its capabilities, an FE will have attribute information that can be used in understanding and controlling the forwarding operations. Some of the attributes will be read only, while others will also be writeable.

In order to make the FE attribute information easily accessible, the information will be stored in an LFB. This LFB will have a class, FEObject. The LFBClassID for this class is 1. Only one instance of this class will ever be present, and the instance ID of that instance in the protocol is 1. Thus, by referencing the elements of class:1, instance:1 a CE can get all the information about the FE. For model completeness, this LFB Class is described in this section.

There will also be an FEProtocol LFB Class. LFBClassID 2 is reserved for that class. There will be only one instance of that class as well. Details of that class are defined in the ForCES protocol document.

### 5.1. XML for FEObject Class definition

```
<?xml version="1.0" encoding="UTF-8"?>
<LFBLibrary xmlns="http://ietf.org/forces/1.0/lfbmodel"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
          xsi:schemaLocation="http://ietf.org/forces/1.0/lfbmodel.xsd"
```

```
        provides="FEObject">
   <!--
     -
     -
      xmlns and schemaLocation need to be fixed -->
        <dataTypeDefs>
          <dataTypeDef>
            <name>LFBAdjacencyLimitType</name>
            <synopsis>Describing the Adjacent LFB</synopsis>
            <struct>
              <element elementID="1">
                <name>NeighborLFB</name>
                <synopsis>ID for that LFB Class</synopsis>
                <typeRef>uint32</typeRef>
              </element>
              <element elementID="2">
                <name>ViaPorts</name>
                <synopsis>
                  the ports on which we can connect
                </synopsis>
                <array type="variable-size">
                  <typeRef>string</typeRef>
                </array>
              </element>
            </struct>
          </dataTypeDef>
          <dataTypeDef>
            <name>PortGroupLimitType</name>
            <synopsis>
              Limits on the number of ports in a given group
            </synopsis>
            <struct>
              <element elementID="1">
                <name>PortGroupName</name>
                <synopsis>Group Name</synopsis>
                <typeRef>string</typeRef>
              </element>
              <element elementID="2">
                <name>MinPortCount</name>
                <synopsis>Minimum Port Count</synopsis>
                <optional/>
                <typeRef>uint32</typeRef>
              </element>
              <element elementID="3">
                <name>MaxPortCount</name>
                <synopsis>Max Port Count</synopsis>
                <optional/>
                <typeRef>uint32</typeRef>
```

```
        </element>
      </struct>
    </dataTypeDef>
    <dataTypeDef>
```

```
            <name>SupportedLFBType</name>
            <synopsis>table entry for supported LFB</synopsis>
            <struct>
              <element elementID="1">
                <name>LFBName</name>
                <synopsis>
                  The name of a supported LFB Class
                </synopsis>
                <typeRef>string</typeRef>
              </element>
              <element elementID="2">
                <name>LFBClassID</name>
                <synopsis>the id of a supported LFB Class</synopsis>
                <typeRef>uint32</typeRef>
              </element>
              <element elementID=  3  >
                <name>LFBVersion</name>
                <synopsis>
                  The version of the LFB Class used
                  by this FE.
                </synopsis>
                <typeRef>string</typeRef>
              <element elementID="4">
                <name>LFBOccurrenceLimit</name>
                <synopsis>
                  the upper limit of instances of LFBs of this class
                </synopsis>
                <optional/>
                <typeRef>uint32</typeRef>
              </element>
              <!-- For each port group, how many ports can exist
              -->
              <element elementID="5">
                <name>PortGroupLimits</name>
                <synopsis>Table of Port Group Limits</synopsis>
                <optional/>
                <array type="variable-size">
                  <typeRef>PortGroupLimitType</typeRef>
                </array>
              </element>
   <!-- for the named LFB Class, the LFB Classes it may follow -->
              <element elementID="6">
                <name>CanOccurAfters</name>
                <synopsis>
                  List of LFB Classes that this LFB class can follow
                </synopsis>
                <optional/>
                <array type="variable-size">
```

```
                  <typeRef>LFBAdjacencyLimitType</typeRef>
```

```
                </array>
              </element>
   <!-- for the named LFB Class, the LFB Classes that may follow it
     -->
              <element elementID="7">
                <name>CanOccurBefores</name>
                <synopsis>
                  List of LFB Classes that can follow this LFB class
                </synopsis>
                <optional/>
                <array type="variable-size">
                  <typeRef>LFBAdjacencyLimitType</typeRef>
                </array>
              </element>
            </struct>
          </dataTypeDef>
          <dataTypeDef>
            <name>FEStatusValues</name>
            <synopsis>The possible values of status</synopsis>
            <atomic>
              <baseType>uchar</baseType>
              <specialValues>
                <specialValue value="0">
                  <name>AdminDisable</name>
                  <synopsis>
                    FE is administratively disabled
                </synopsis>
                </specialValue>
                <specialValue value="1">
                  <name>OperDisable</name>
                  <synopsis>FE is operatively disabled</synopsis>
                </specialValue>
                <specialValue value="2">
                  <name>OperEnable</name>
                  <synopsis>FE is operating</synopsis>
                </specialValue>
              </specialValues>
            </atomic>
          </dataTypeDef>
          <dataTypeDef>
            <name>FEConfiguredNeighborType</name>
            <synopsis>Details of the FE's Neighbor</synopsis>
            <struct>
              <element elementID="1">
                <name>NeighborID</name>
                <synopsis>Neighbors FEID</synopsis>
                <typeRef>uint32</typeRef>
              </element>
```

```
              <element elementID="2">
```

```
                <name>InterfaceToNeighbor</name>
                <synopsis>
                  FE's interface that connects to this neighbor
                </synopsis>
                <optional/>
                <typeRef>string</typeRef>
              </element>
              <element elementID=  3  >
                <name>NeighborInterface</name>
                <synopsis>
                  The name of the interface on the neighbor to
                  which this FE is adjacent.  This is required
                  In case two FE s are adjacent on more than
                  one interface.
                </synopsis>
                <optional/>
                <typeRef>string</typeRef>
              </element>
            </struct>
          </dataTypeDef>
          <dataTypeDef>
            <name>LFBSelectorType</name>
            <synopsis>
              Unique identification of an LFB class-instance
            </synopsis>
            <struct>
              <element elementID="1">
                <name>LFBClassID</name>
                <synopsis>LFB Class Identifier</synopsis>
                <typeRef>uint32</typeRef>
              </element>
              <element elementID="2">
                <name>LFBInstanceID</name>
                <synopsis>LFB Instance ID</synopsis>
                <typeRef>uint32</typeRef>
              </element>
            </struct>
          </dataTypeDef>
          <dataTypeDef>
            <name>LFBLinkType</name>
            <synopsis>
              Link between two LFB instances of topology
            </synopsis>
            <struct>
              <element elementID="1">
                <name>FromLFBID</name>
                <synopsis>LFB src</synopsis>
                <typeRef>LFBSelectorType</typeRef>
```

```
        </element>
```

```
              <element elementID="2">
                <name>FromPortGroup</name>
                <synopsis>src port group</synopsis>
                <typeRef>string</typeRef>
              </element>
              <element elementID="3">
                <name>FromPortIndex</name>
                <synopsis>src port index</synopsis>
                <typeRef>uint32</typeRef>
              </element>
              <element elementID="4">
                <name>ToLFBID</name>
                <synopsis>dst LFBID</synopsis>
                <typeRef>LFBSelectorType</typeRef>
              </element>
              <element elementID="5">
                <name>ToPortGroup</name>
                <synopsis>dst port group</synopsis>
                <typeRef>string</typeRef>
              </element>
              <element elementID="6">
                <name>ToPortIndex</name>
                <synopsis>dst port index</synopsis>
                <typeRef>uint32</typeRef>
              </element>
            </struct>
          </dataTypeDef>
        </dataTypeDefs>
        <LFBClassDefs>
          <LFBClassDef LFBClassID="1">
            <name>FEObject</name>
            <synopsis>Core LFB: FE Object</synopsis>
            <version>1.0</version>
            <attributes>
              <attribute access="read-write" elementID="1">
                <name>LFBTopology</name>
                <synopsis>the table of known Topologies</synopsis>
                <array type="variable-size">
                  <typeRef>LFBLinkType</typeRef>
                </array>
              </attribute>
              <attribute access="read-write" elementID="2">
                <name>LFBSelectors</name>
                <synopsis>
                   table of known active LFB classes and
                   instances
                </synopsis>
                <array type="variable-size">
```

```
<typeRef>LFBSelectorType</typeRef>
```

```
            </array>
          </attribute>
          <attribute access="read-write" elementID="3">
            <name>FEName</name>
            <synopsis>name of this FE</synopsis>
            <typeRef>string[40]</typeRef>
          </attribute>
          <attribute access="read-write" elementID="4">
            <name>FEID</name>
            <synopsis>ID of this FE</synopsis>
            <typeRef>uint32</typeRef>
          </attribute>
          <attribute access="read-only" elementID="5">
            <name>FEVendor</name>
            <synopsis>vendor of this FE</synopsis>
            <typeRef>string[40]</typeRef>
          </attribute>
          <attribute access="read-only" elementID="6">
            <name>FEModel</name>
            <synopsis>model of this FE</synopsis>
            <typeRef>string[40]</typeRef>
          </attribute>
          <attribute access="read-only" elementID="7">
            <name>FEState</name>
            <synopsis>model of this FE</synopsis>
            <typeRef>FEStatusValues</typeRef>
          </attribute>
          <attribute access="read-write" elementID="8">
            <name>FENeighbors</name>
            <synopsis>table of known neighbors</synopsis>
            <array type="variable-size">
              <typeRef>FEConfiguredNeighborType</typeRef>
            </array>
          </attribute>
        </attributes>
        <capabilities>
          <capability elementID="30">
            <name>ModifiableLFBTopology</name>
            <synopsis>
              Whether Modifiable LFB is supported
            </synopsis>
            <optional/>
            <typeRef>boolean</typeRef>
          </capability>
          <capability elementID="31">
            <name>SupportedLFBs</name>
            <synopsis>List of all supported LFBs</synopsis>
            <optional/>
```

```
          <array type="variable-size">
```

```
                  <typeRef>SupportedLFBType</typeRef>
                </array>
              </capability>
            </capabilities>
          </LFBClassDef>
        </LFBClassDefs>
      </LFBLibrary>
```

## 5.2. FE Capabilities

The FE Capability information is contained in the capabilities element of the class definition. As described elsewhere, capability information is always considered to be read-only.

The currently defined capabilities are ModifiableLFBTopology and SupportedLFBs. Information as to which attributes of the FE LFB are supported is accessed by the properties information for those elements.

### 5.2.1. ModifiableLFBTopology

This element has a boolean value that indicates whether the LFB topology of the FE may be changed by the CE. If the element is absent, the default value is assumed to be true, and the CE presumes the LFB topology may be changed. If the value is present and set to false, the LFB topology of the FE is fixed. If the topology is fixed, the LFBs supported clause may be omitted, and the list of supported LFBs is inferred by the CE from the LFB topology information. If the list of supported LFBs is provided when ModifiableLFBTopology is false, the CanOccurBefore and CanOccurAfter information should be omitted.

### 5.2.2. SupportedLFBs and SupportedLFBType

One capability that the FE should include is the list of supported LFB classes. The SupportedLFBs element, is an array that contains the information about each supported LFB Class. The array structure type is defined as the SupportedLFBType dataTypeDef.

Each occurrence of the SupportedLFBs array element describes an LFB class that the FE supports. In addition to indicating that the FE supports the class, FEs with modifiable LFB topology should include information about how LFBs of the specified class may be connected to other LFBs. This information should describe which LFB classes the specified LFB class may succeed or precede in the LFB topology. The FE should include information as to which port groups may be connected to the given adjacent LFB class. If port group information is omitted, it is assumed that all port groups may be used.

#### 5.2.2.1. LFBName

This element has as its value the name of the LFB Class being described.

#### 5.2.2.2. LFBClassID

The numeric ID of the LFB Class being described. While conceptually redundant with the LFB Name, both are included for clarity and to allow consistency checking.

#### 5.2.2.3. LFBVersion

The version string specifying the LFB Class version supported by this FE. As described above in versioning, an FE can support only a single version of a given LFB Class.

#### 5.2.2.4. LFBOccurrenceLimit

This element, if present, indicates the largest number of instances of this LFB class the FE can support. For FEs that do not have the capability to create or destroy LFB instances, this can either be omitted or be the same as the number of LFB instances of this class contained in the LFB list attribute.

#### 5.2.2.5. PortGroupLimits and PortGroupLimitType

The PortGroupLimits element is an array of information about the port groups supported by the LFB class. The structure of the port group limit information is defined by the PortGroupLimitType dataTypeDef.

Each PortGroupLimits array element contains information describing a single port group of the LFB class. Each array element contains the name of the port group in the PortGroupName element, the fewest number of ports that can exist in the group in the MinPortCount element, and the largest number of ports that can exist in the group in the MaxPortCount element.

5.2.2.6.CanOccurAfters and LFBAdjacencyLimitType

The CanOccurAfters element is an array that contains the list of LFBs the described class can occur after. The array elements are defined in the LFBAdjacencyLimitType dataTypeDef.

The array elements describe a permissible positioning of the described LFB class, referred to here as the SupportedLFB. Specifically, each array element names an LFB that can topologically

precede that LFB class. That is, the SupportedLFB can have an input port connected to an output port of an LFB that appears in the CanOccurAfters array. The LFB class that the SupportedLFB can follow is identified by the NeighborLFB element of the LFBAdjacencyLimitType array element. If this neighbor can only be connected to a specific set of input port groups, then the viaPort element is included. This element occurs once for each input port group of the SupportedLFB that can be connected to an output port of the NeighborLFB.

[e.g., Within a SupportedLFBs element, each array element of the CanOccurAfters array must have a unique NeighborLFB, and within each array element each viaPort must represent a distinct and valid input port group of the SupportedLFB. The LFB Class definition schema does not yet support uniqueness declarations]

#### 5.2.2.7. CanOccurBefores and LFBAdjacencyLimitType

The CanOccurBefores array holds the information about which LFB classes can follow the described class. Structurally this element parallels CanOccurAfters, and uses the same type definition for the array element.

The array elements list those LFB classes that the SupportedLFB may precede in the topology. In this element, the viaPort element of the array value represents the output port group of the SupportedLFB that may be connected to the NeighborLFB. As with CanOccurAfters, viaPort may occur multiple times if multiple output ports may legitimately connect to the given NeighborLFB class.

[And a similar set of uniqueness constraints apply to the CanOccurBefore clauses, even though an LFB may occur both in CanOccurAfter and CanOccurBefore.]

#### 5.2.2.8. LFBClassCapabilities

While it would be desirable to include class capability level information, this is not included in the model. While such information belongs in the FE Object in the supported class table, the contents of that information would be class specific. The currently expected encoding structures for transferring information between the CE and FE are such that allowing completely unspecified information would be likely to induce parse errors. We could specify that the information is encoded in an octetstring, but then we would have to define the internal format of that octet string.

As there also are not currently any defined LFB Class level Capabilities that the FE needs to report, this information is not

present now, but may be added in a future version of the FE Protocol Object. (This is an example of a case where versioning, rather than inheritance, would be needed, since the FE Object must have class ID 1 and instance ID 1 so that the protocol behavior can start by finding this object.)

### 5.3. FEAttributes

The attributes element is included if the class definition contains the attributes of the FE that are not considered "capabilities". Some of these attributes are writeable, and some are read-only, which should be indicated by the capability information.

#### 5.3.1. FEStatus

This attribute carries the overall state of the FE. For now, it is restricted to the strings AdminDisable, OperDisable and OperEnable.

#### 5.3.2. LFBSelectors and LFBSelectorType

The LFBSelectors element is an array of information about the LFBs currently accessible via ForCES in the FE. The structure of the LFB information is defined by the LFBSelectorType.

Each entry in the array describes a single LFB instance in the FE. The array element contains the numeric class ID of the class of the LFB instance and the numeric instance ID for this instance.

#### 5.3.3. LFBTopology and LFBLinkType

The optional LFBTopology element contains information about each inter-LFB link inside the FE, where each link is described in an LFBLinkType element. The LFBLinkType element contains sufficient information to identify precisely the end points of a link. The FromLFBID and ToLFBID fields specify the LFB instances at each end of the link, and must reference LFBs in the LFB instance table. The FromPortGroup and ToPortGroup must identify output and input port groups defined in the LFB classes of the LFB instances identified by FromLFBID and ToLFBID. The FromPortIndex and ToPortIndex fields select the elements from the port groups that this link connects. All links are uniquely identified by the FromLFBID, FromPortGroup, and FromPortIndex fields. Multiple links may have the same ToLFBID, ToPortGroup, and ToPortIndex as this model supports fan in of inter-LFB links but not fan out.

### 5.3.4. FENeighbors and FEConfiguredNeighborType

The FENeighbors element is an array of information about manually configured adjacencies between this FE and other FEs. The content of the array is defined by the FEConfiguredNeighborType element.

This array is intended to capture information that may be configured on the FE and is needed by the CE, where one array entry corresponds to each configured neighbor. Note that this array is not intended to represent the results of any discovery protocols, as those will have their own LFBs.

While there may be many ways to configure neighbors, the FE-ID is the best way for the CE to correlate entities. And the interface identifier (name string) is the best correlator. The CE will be able to determine the IP address and media level information about the neighbor from the neighbor directly. Omitting that information from this table avoids the risk of incorrect double configuration.

Information about the intended forms of exchange with a given neighbor is not captured here, only the adjacency information is included.

5.3.4.1.NeighborID

This is the ID in some space meaningful to the CE for the neighbor. If this table remains, we probably should add an FEID from the same space as an attribute of the FE.

5.3.4.2.InterfaceToNeighbor

This identifies the interface through which the neighbor is reached.

5.3.4.3.NeighborInterface

This identifies the interface on the neighbor through which the neighbor is reached. The interface identification is needed when either only one side of the adjacency has configuration information, or the two FEs are adjacent on more than one interface.

## 6. Satisfying the Requirements on FE Model

This section describes how the proposed FE model meets the requirements outlined in Section 5 of RFC 3654 [1]. The requirements can be separated into general requirements (Sections 5, 5.1 - 5.4) and the specification of the minimal set of logical functions that the FE model must support (Section 5.5).

The general requirement on the FE model is that it be able to express the logical packet processing capability of the FE, through both a capability and a state model. In addition, the FE model is expected to allow flexible implementations and be extensible to allow defining new logical functions.

A major component of the proposed FE model is the Logical Function Block (LFB) model. Each distinct logical function in an FE is modeled as an LFB. Operational parameters of the LFB that must be visible to the CE are conceptualized as LFB attributes. These attributes express the capability of the FE and support flexible implementations by allowing an FE to specify which optional features are supported. The attributes also indicate whether they are configurable by the CE for an LFB class. Configurable attributes provide the CE some flexibility in specifying the behavior of an LFB. When multiple LFBs belonging to the same LFB class are instantiated on an FE, each of those LFBs could be configured with different attribute settings. By querying the settings of the attributes for an instantiated LFB, the CE can determine the state of that LFB.

Instantiated LFBs are interconnected in a directed graph that describes the ordering of the functions within an FE. This directed graph is described by the topology model. The combination of the attributes of the instantiated LFBs and the topology describe the packet processing functions available on the FE (current state).

Another key component of the FE model is the FE attributes. The FE attributes are used mainly to describe the capabilities of the FE, but they also convey information about the FE state.

The FE model includes only the definition of the FE Object LFB itself. Meeting the full set of working group requirements requires other LFBs. The class definitions for those LFBs will be provided in other documents.

## 7. Using the FE model in the ForCES Protocol

The actual model of the forwarding plane in a given NE is something the CE must learn and control by communicating with the FEs (or by other means). Most of this communication will happen in the post-association phase using the ForCES protocol. The following types of information must be exchanged between CEs and FEs via the ForCES protocol:

1) FE topology query;
2) FE capability declaration;
3) LFB topology (per FE) and configuration capabilities query;

4) LFB capability declaration;

5) State query of LFB attributes;
6) Manipulation of LFB attributes;
7) LFB topology reconfiguration.

Items 1) through 5) are query exchanges, where the main flow of information is from the FEs to the CEs. Items 1) through 4) are typically queried by the CE(s) in the beginning of the post-association (PA) phase, though they may be repeatedly queried at any time in the PA phase. Item 5) (state query) will be used at the beginning of the PA phase, and often frequently during the PA phase (especially for the query of statistical counters).

Items 6) and 7) are "command" types of exchanges, where the main flow of information is from the CEs to the FEs. Messages in Item 6) (the LFB re-configuration commands) are expected to be used frequently. Item 7) (LFB topology re-configuration) is needed only if dynamic LFB topologies are supported by the FEs and it is expected to be used infrequently.

The inter-FE topology (item 1 above) can be determined by the CE in many ways. Neither this document nor the Forces protocol mandates a specific mechanism. The LFB Class definition does include the capability for an FE to be configured with, and provides to the CE in response to a query, the identity of its neighbors. There may also be defined specific LFB classes and protocols for neighbor discovery. Routing protocols may be used by the CE for adjacency determination. The CE may be configured with the relevant information.

The relationship between the FE model and the seven post-association messages are visualized in Figure 9:

```
                                          +--------+
                             ..........-->|   CE   |
           /----\            .            +--------+
           \____/ FE Model   .              ^    |
           |    |................        (1),2 |    | 6, 7
           |    |  (off-line)   .      3, 4, 5 |    |
           \____/            .              |    v
                             .            +--------+
         e.g. RFCs           ..........-->|   FE   |
                                          +--------+
```

Figure 9. Relationship between the FE model and the ForCES protocol messages, where (1) is part of the ForCES base protocol, and the rest are defined by the FE model.

The actual encoding of these messages is defined by the ForCES protocol and beyond the scope of the FE model. Their discussion is nevertheless important here for the following reasons:

- These PA model components have considerable impact on the FE model. For example, some of the above information can be represented as attributes of the LFBs, in which case such attributes must be defined in the LFB classes.
- The understanding of the type of information that must be exchanged between the FEs and CEs can help to select the appropriate protocol format and the actual encoding method (such as XML, TLVs).
- Understanding the frequency of these types of messages should influence the selection of the protocol format (efficiency considerations).

An important part of the FE model is the port the FE uses for its message exchanges to and from the CE. In the case that a dedicated port is used for CE-FE communication, we propose to use a special port LFB, called the CE-FE Port LFB (a subclass of the general Port LFB in Section 6.1), to model this dedicated CE-FE port. The CE-FE Port LFB acts as both a source and sink for the traffic from and to the CE. Sometimes the CE-FE traffic does not have its own dedicated port, instead the data fabric is shared for the data plane traffic and the CE-FE traffic. A special processing LFB can be used to model the ForCES packet encapsulation and decapsulation in such cases.

The remaining sub-sections of this section address each of the seven message types.

## 7.1. FE Topology Query

An FE may contain zero, one or more external ingress ports. Similarly, an FE may contain zero, one or more external egress ports. In other words, not every FE has to contain any external ingress or egress interfaces. For example, Figure 10 shows two cascading FEs. FE #1 contains one external ingress interface but no external egress interface, while FE #2 contains one external egress interface but no ingress interface. It is possible to connect these two FEs together via their internal interfaces to achieve the complete ingress-to-egress packet processing function. This provides the flexibility to spread the functions across multiple FEs and interconnect them together later for certain applications.

While the inter-FE communication protocol is out of scope for ForCES, it is up to the CE to query and understand how multiple FEs are inter-connected to perform a complete ingress-egress packet processing function, such as the one described in Figure 10.  The inter-FE topology information may be provided by FEs, may be hard-coded into CE, or may be provided by some other entity (e.g., a bus manager) independent of the FEs.  So while the ForCES protocol supports FE topology query from FEs, it is optional for the CE to use it, assuming the CE has other means to gather such topology information.

```
   +-----------------------------------------------------+
   |  +---------+   +------------+   +---------+         |
 input|         |   |            |   |         | output  |
---+->| Ingress |-->|Header      |-->|IPv4     |---------+--->+
   |  | port    |   |Decompressor|   |Forwarder| FE      |    |
   |  +---------+   +------------+   +---------+ #1      |    |
   +-----------------------------------------------------+    V
                                                              |
        +-----------------------<-----------------------------+
        |
        |    +----------------------------------------+
        V    |  +------------+   +----------+         |
        | input |            |   |          | output  |
        +->--+->|Header      |-->| Egress   |---------+-->
             |  |Compressor  |   | port     | FE      |
             |  +------------+   +----------+ #2      |
             +----------------------------------------+
```

Figure 10. An example of two FEs connected together.

Once the inter-FE topology is discovered by the CE after this query, it is assumed that the inter-FE topology remains static.  However, it is possible that an FE may go down during the NE operation, or a board may be inserted and a new FE activated, so the inter-FE topology will be affected.  It is up to the ForCES protocol to provide a mechanism for the CE to detect such events and deal with the change in FE topology.  FE topology is outside the scope of the FE model.

### 7.2. FE Capability Declarations

FEs will have many types of limitations.  Some of the limitations must be expressed to the CEs as part of the capability model.  The CEs must be able to query these capabilities on a per-FE basis. Examples:

. Metadata passing capabilities of the FE.  Understanding these capabilities will help the CE to evaluate the feasibility of LFB topologies, and hence to determine the availability of certain services.

. Global resource query limitations (applicable to all LFBs of the FE).

. LFB supported by the FE.
. LFB class instantiation limit.
. LFB topological limitations (linkage constraint, ordering etc.)

### 7.3. LFB Topology and Topology Configurability Query

The ForCES protocol must provide the means for the CEs to discover the current set of LFB instances in an FE and the interconnections between the LFBs within the FE. In addition, sufficient information should be available to determine whether the FE supports any CE-initiated (dynamic) changes to the LFB topology, and if so, determine the allowed topologies. Topology configurability can also be considered as part of the FE capability query as described in Section 9.3.

### 7.4. LFB Capability Declarations

LFB class specifications define a generic set of capabilities. When an LFB instance is implemented (instantiated) on a vendor's FE, some additional limitations may be introduced. Note that we discuss only those limitations that are within the flexibility of the LFB class specification. That is, the LFB instance will remain compliant with the LFB class specification despite these limitations. For example, certain features of an LFB class may be optional, in which case it must be possible for the CE to determine if an optional feature is supported by a given LFB instance or not. Also, the LFB class definitions will probably contain very few quantitative limits (e.g., size of tables), since these limits are typically imposed by the implementation. Therefore, quantitative limitations should always be expressed by capability arguments.

LFB instances in the model of a particular FE implementation will possess limitations on the capabilities defined in the corresponding LFB class. The LFB class specifications must define a set of capability arguments, and the CE must be able to query the actual capabilities of the LFB instance via querying the value of such arguments. The capability query will typically happen when the LFB is first detected by the CE. Capabilities need not be re-queried in case of static limitations. In some cases, however, some capabilities may change in time (e.g., as a result of adding/removing other LFBs, or configuring certain attributes of some other LFB when the LFBs share physical resources), in which case additional mechanisms must be implemented to inform the CE about the changes.

The following two broad types of limitations will exist:

. Qualitative restrictions. For example, a standardized multi-field classifier LFB class may define a large number of

classification fields, but a given FE may support only a subset of those fields.
. Quantitative restrictions, such as the maximum size of tables, etc.

The capability parameters that can be queried on a given LFB class will be part of the LFB class specification. The capability parameters should be regarded as special attributes of the LFB. The actual values of these arguments may be, therefore, obtained using the same attribute query mechanisms as used for other LFB attributes.

Capability attributes will typically be read-only arguments, but in certain cases they may be configurable. For example, the size of a lookup table may be limited by the hardware (read-only), in other cases it may be configurable (read-write, within some hard limits).

Assuming that capabilities will not change frequently, the efficiency of the protocol/schema/encoding is of secondary concern.

### 7.5. State Query of LFB Attributes

This feature must be provided by all FEs. The ForCES protocol and the data schema/encoding conveyed by the protocol must together satisfy the following requirements to facilitate state query of the LFB attributes:

. Must permit FE selection. This is primarily to refer to a single FE, but referring to a group of (or all) FEs may optional be supported.
. Must permit LFB instance selection. This is primarily to refer to a single LFB instance of an FE, but optionally addressing of a group of LFBs (or all) may be supported.
. Must support addressing of individual attribute of an LFB.
. Must provide efficient encoding and decoding of the addressing info and the configured data.
. Must provide efficient data transmission of the attribute state over the wire (to minimize communication load on the CE-FE link).

### 7.6. LFB Attribute Manipulation

The FE Model provides for the definition of LFB Classes. Each class has a globally unique identifier. Elements within the class are assigned identifiers within that scope. This model also specifies that instances of LFB Classes have identifiers. The combination of class identifiers, instance identifiers, and element identifiers are used by the protocol to reference the LFB information in the protocol operations.

### 7.7. LFB Topology Re-configuration

Operations that will be needed to reconfigure LFB topology:

- Create a new instance of a given LFB class on a given FE.
- Connect a given output of LFB x to the given input of LFB y.
- Disconnect: remove a link between a given output of an LFB and a given input of another LFB.
- Delete a given LFB (automatically removing all interconnects to/from the LFB).

## 8. Example

This section contains an example LFB definition.  While some properties of LFBs are shown by the FE Object LFB, this endeavors to show how a data plain LFB might be build.  This example is a fictional case of an interface supporting a coarse WDM optical interface carry Frame Relay traffic.  The statistical information (including error statistics) is omitted.)


```
<?xml version="1.0" encoding="UTF-8"?>
<LFBLibrary xmlns="http://ietf.org/forces/1.0/lfbmodel"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ietf.org/forces/1.0/lfbmodel"
 provides="LaserFrameLFB">
  <frameDefs>
    <frameDef>
      <name>FRFrame</name>
      <synopsis>
          A frame relay frame, with DLCI without
          stuffing)
      </synopsis>
    </frameDef>
    <frameDef>
      <name>IPFrame</name>
       <synopsis>An IP Packet</synopsis>
    </frameDef>
  </frameDefs>
  <dataTypeDefs>
    <dataTypeDef>
      <name>frequencyInformationType</name>
      <synopsis>
          Information about a single CWDM frequency
      </synopsis>
      <struct>
        <element elementID="1">
          <name>LaserFrequency</name>
          <synopsis>encoded frequency(channel)</synopsis>
```

<typeRef>uint32</typeRef>

```
          </element>
          <element elementID="2">
            <name>FrequencyState</name>
            <synopsis>state of this frequency</synopsis>
            <typeRef>PortStatusValues</typeRef>
          </element>
          <element elementID="3">
            <name>LaserPower</name>
            <synopsis>current observed power</synopsis>
            <typeRef>uint32</typeRef>
          </element>
          <element elementID="4">
            <name>FrameRelayCircuits</name>
            <synopsis>
                Information about circuits on this Frequency
            </synopsis>
            <array>
              <typeRef>frameCircuitsType</typeRef>
            </array>
          </element>
        </struct>
      </dataTypeDef>
      <dataTypeDef>
        <name>frameCircuitsType</name>
        <synopsis>
            Information about a single Frame Relay circuit
        </synopsis>
        <struct>
          <element elementID="1">
            <name>DLCI</name>
            <synopsis>DLCI of the circuit</synopsis>
            <typeRef>uint32</typeRef>
          </element>
          <element elementID="2">
            <name>CircuitStatus</name>
            <synopsis>state of the circuit</synopsis>
            <typeRef>PortStatusValues</typeRef>
          </element>
          <element elementID="3">
            <name>isLMI</name>
            <synopsis>is this the LMI circuit</synopsis>
            <typeRef>boolean</typeRef>
          </element>
          <element elementID="4">
            <name>associatedPort</name>
            <synopsis>
                which input / output port is associated
                with this circuit
```

```
            </synopsis>
```

```
             <typeRef>uint32</typeRef>
           </element>
         </struct>
       </dataTypeDef>
       <dataTypeDef>
         <name>PortStatusValues</name>
         <synopsis>
             The possible values of status.  Used for both
             administrative and operation status
         </synopsis>
         <atomic>
           <baseType>uchar</baseType>
           <specialValues>
             <specialValue value="0">
               <name>Disabled </name>
               <synopsis>the component is disabled</synopsis>
             </specialValue>
             <specialValue value="1">
               <name>Enable</name>
               <synopsis>FE is operatively disabled</synopsis>
             </specialValue>
           </specialValues>
         </atomic>
       </dataTypeDef>
     </dataTypeDefs>
     <metadataDefs>
       <metadataDef>
         <name>DLCI</name>
         <synopsis>The DLCI the frame arrived on</synopsis>
         <metadataID>12</metadataID>
         <typeRef>uint32</typeRef>
       </metadataDef>
       <metadataDef>
         <name>LaserChannel</name>
         <synopsis>The index of the laser channel</synopsis>
         <metadataID>34</metadataID>
         <typeRef>uint32</typeRef>
       </metadataDef>
     </metadataDefs>
     <LFBClassDefs>
       <LFBClassDef LFBClassID="-255">
         <name>FrameLaserLFB</name>
         <synopsis>Fictional LFB for Demonstrations</synopsis>
         <version>1.0</version>
         <inputPorts>
           <inputPort group="yes">
             <name>LMIfromFE</name>
             <synopsis>
```

Ports for LMI traffic, for transmission

```
            </synopsis>
            <expectation>
              <frameExpected>
                <ref>FRFrame</ref>
              </frameExpected>
              <metadataExpected>
                <ref>DLCI</ref>
                <ref>LaserChannel</ref>
              </metadataExpected>
            </expectation>
          </inputPort>
          <inputPort>
            <name>DatafromFE</name>
            <synopsis>
                Ports for data to be sent on circuits
            </synopsis>
            <expectation>
              <frameExpected>
                <ref>IPFrame</ref>
              </frameExpected>
              <metadataExpected>
                <ref>DLCI</ref>
                <ref>LaserChannel</ref>
              </metadataExpected>
            </expectation>
          </inputPort>
        </inputPorts>
        <outputPorts>
          <outputPort group="yes">
            <name>LMItoFE</name>
            <synopsis>
                Ports for LMI traffic for processing
            </synopsis>
            <product>
              <frameProduced>
                <ref>FRFrame</ref>
              </frameProduced>
              <metadataProduced>
                <ref>DLCI</ref>
                <ref>LaserChannel</ref>
              </metadataProduced>
            </product>
          </outputPort>
          <outputPort group="yes">
            <name>DatatoFE</name>
            <synopsis>
                Ports for Data traffic for processing
            </synopsis>
```

<product>

```
               <frameProduced>
                 <ref>IPFrame</ref>
               </frameProduced>
               <metadataProduced>
                 <ref>DLCI</ref>
                 <ref>LaserChannel</ref>
               </metadataProduced>
             </product>
           </outputPort>
         </outputPorts>
         <attributes>
           <attribute access="read-write" elementID="1">
             <name>AdminPortState</name>
             <synopsis>is this port allowed to function</synopsis>
             <typeRef>PortStatusValues</typeRef>
           </attribute>
           <attribute access="read-write" elementID="2">
             <name>FrequencyInformation</name>
             <synopsis>
                 table of information per CWDM frequency
             </synopsis>
             <array type="variable-size">
               <typeRef>frequencyInformationType</typeRef>
             </array>
           </attribute>
         </attributes>
         <capabilities>
           <capability elementID="31">
             <name>OperationalState</name>
             <synopsis>
                 whether the port over all is operational
             </synopsis>
             <typeRef>PortStatusValues</typeRef>
           </capability>
           <capability elementID="32">
             <name>MaximumFrequencies</name>
             <synopsis>
                 how many laser frequencies are there
             </synopsis>
             <typeRef>uint16</typeRef>
           </capability>
           <capability elementID="33">
             <name>MaxTotalCircuits</name>
             <synopsis>
                 Total supportable Frame Relay Circuits, across
                 all laser frequencies
             </synopsis>
             <optional/>
```

```
          <typeRef>uint32</typeRef>
```

```
          </capability>
        </capabilities>
        <events baseID="61">
          <event eventID="1">
            <name>FrequencyState</name>
            <synopsis>
                The state of a frequency has changed
            </synopsis>
            <eventTarget>
              <eventField>FrequencyInformation</eventField>
              <eventSubscript>_FrequencyIndex_</eventSubscript>
              <eventField>FrequencyState</eventField>
            </eventTarget>
            <eventChanged/>
            <eventReports>
              <!-- report the new state -->
              <eventReport>
                <eventField>FrequencyInformation</eventField>
                <eventSubscript>_FrequencyIndex_</eventSubscript>
                <eventField>FrequencyState</eventField>
              </eventReport>
            </eventReports>
          </event>
          <event eventID="2">
            <name>CreatedFrequency</name>
            <synopsis>A new frequency has appeared</synopsis>
            <eventTarget>
              <eventField>FrequencyInformation></eventField>
              <eventSubscript>_FrequencyIndex_</eventSubscript>
            </eventTarget>
            <eventCreated/>
            <eventReports>
              <eventReport>
                <eventField>FrequencyInformation</eventField>
                <eventSubscript>_FrequencyIndex_</eventSubscript>
                <eventField>LaserFrequency</eventField>
              </eventReport>
            </eventReports>
          </event>
          <event eventID="3">
            <name>DeletedFrequency</name>
            <synopsis>
                A frequency Table entry has been deleted
            </synopsis>
            <eventTarget>
              <eventField>FrequencyInformation</eventField>
              <eventSubscript>_FrequencyIndex_</eventSubscript>
            </eventTarget>
```

```
        <eventDeleted/>
```

```
          </event>
          <event eventID="4">
            <name>PowerProblem</name>
            <synopsis>
                there are problems with the laser power level
            </synopsis>
            <eventTarget>
              <eventField>FrequencyInformation</eventField>
              <eventSubscript>_FrequencyIndex_</eventSubscript>
              <eventField>LaserPower</eventField>
            </eventTarget>
            <eventLessThan/>
            <eventReports>
              <eventReport>
                <eventField>FrequencyInformation</eventField>
                <eventSubscript>_FrequencyIndex_</eventSubscript>
                <eventField>LaserPower</eventField>
              </eventReport>
              <eventReport>
                <eventField>FrequencyInformation</eventField>
                <eventSubscript>_FrequencyIndex_</eventSubscript>
                <eventField>LaserFrequency</eventField>
              </eventReport>
            </eventReports>
          </event>
          <event eventID="5">
            <name>FrameCircuitChanged</name>
            <synopsis>
                the state of an Fr circuit on a frequency
                has changed
            </synopsis>
            <eventTarget>
              <eventField>FrequencyInformation</eventField>
              <eventSubscript>_FrequencyIndex_</eventSubscript>
              <eventField>FrameRelayCircuits</eventField>
              <eventSubscript>FrameCircuitIndex</eventSubscript>
              <eventField>CircuitStatus</eventField>
            </eventTarget>
            <eventChanged/>
            <eventReports>
              <eventReport>
                <eventField>FrequencyInformation</eventField>
                <eventSubscript>_FrequencyIndex_</eventSubscript>
                <eventField>FrameRelayCircuits</eventField>
                <eventSubscript>FrameCircuitIndex</eventSubscript>
                <eventField>CircuitStatus</eventField>
              </eventReport>
              <eventReport>
```

<eventField>FrequencyInformation</eventField>

```
                 <eventSubscript>_FrequencyIndex_</eventSubscript>
                 <eventField>FrameRelayCircuits</eventField>
                 <eventSubscript>FrameCircuitIndex</eventSubscript>
                 <eventField>DLCI</eventField>
               </eventReport>
             </eventReports>
           </event>
         </events>
       </LFBClassDef>
     </LFBClassDefs>
   </LFBLibrary>
```

### 8.1.1. Data Handling

This LFB is designed to handle data packets coming in from or going out to the external world. It is not a full port, and it lacks many useful statistics. But it serves to show many of the relevant behaviors.

Packets arriving without error from the physical interface come in on a Frame Relay DLCI on a laser channel. These two values are used by the LFB too look up the handling for the packet. If the handling indicates that the packet is LMI, then the output index is used to select an LFB port from the LMItoFE port group. The packet is sent as a full Frame Relay frame (without any bit or byte stuffing) on the selected port. The laser channel and DLCI are sent as meta-data, even though the DLCI is also still in the packet.

Good packets that arrive and are not LMI and have a frame relay type indicator of IP are sent as IP packets on the port in the DatatoFE port group, using the same index field from the table based on the laser channel and DLCI. The channel and DLCI are attached as meta-data for other use (classifiers, for example.)

The current definition does not specify what to do if the Frame Relay type information is not IP.

Packets arriving on input ports arrive with the Laser Channel and Frame Relay DLCI as meta-data. As such, a single input port could have been used. With the structure that is defined (which parallels the output structure), the selection of channel and DLCI could be restricted by the arriving input port group (LMI vs. data) and port index. As an alternative LFB design, the structures could require a 1-1 relationship between DLCI and LFB port, in which case no meta-data would be needed. This would however be quite complex and noisy. The intermediate level of structure here allows parallelism between input and output, without requiring excessive ports.

### 8.1.2. Setting up a DLCI

When a CE chooses to establish a DLCI on a specific laser channel, it sends a SET request directed to this LFB. The request might look like

```
T = SET-OPERATION
  T = PATH-DATA
    Path: flags = first-avail, length = 4, path = 2, channel, 4
    DataRaw: DLCI, Enable(1), false, out-idx
```

Which would establish the DLCI as enabled, with traffic going to a specific element of the output port group DatatoFE. (The CE would ensure that output port is connected to the right place before issuing this request.

The response to the operation would include the actual index assigned to this Frame Relay circuit. This table is structured to use separate internal indices and DLCIs. An alternative design could have used the DLCI as index, trading off complexities.

One could also imagine that the FE has an LMI LFB. Such an LFB would be connected to the LMItoFE and LMIfromFE port groups. It would process LMI information. It might be the LFBs job to set up the frame relay circuits. The LMI LFB would have an alias entry that points to the Frame Relay circuits table it manages, so that it can manipulate those entities.

### 8.1.3. Error Handling

The LFB will receive invalid packets over the wire. Many of these will simply result in incrementing counters. The LFB designer might also specify some error rate measures. This puts more work on the FE, but allows for more meaningful alarms.

There may be some error conditions that should cause parts of the packet to be sent to the CE. The error itself is not something that can cause an event in the LFB. There are two ways this can be handled.

One way is to define a specific field to count the error, and a field in the LFB to hold the required portion of the packet. The field could be defined to hold the portion of the packet from the most recent error. One could then define an event that occurs whenever the error count changes, and declare that reporting the event includes the LFB field with the packet portion. For rare but extremely critical errors, this is an effective solution. It ensures reliable delivery of the notification. And it allows the CE to control if it wants the notification. (Use of the event variance

property would suppress multiple notifications. It would suppress them even if they were many hours apart, so the CE is unlikely to use that.)

Another approach is for the LFB to have a port that connects to a redirect sink. The LFB would attach the laser channel, the DLCI, and the error indication as meta-data, and ship the packet to the CE.

Other aspects of error handling are discussed under events below.

### 8.2. LFB Attributes

This LFB is defined to have two top level attributes. One reflects the administrative state of the LFB. This allows the CE to disable the LFB completely.

The other attribute is the table of information about the laser channels. It is a variable sized array. Each array entry contains an identifier for what laser frequency this entry is associated with, whether that frequency is operational, the power of the laser at that frequency, and a table of information about frame relay circuits on this frequency. There is no administrative status since a CE can disable an entry simply by removing it. (Frequency and laser power of a non-operational channel are not particularly useful. Knowledge about what frequencies can be supported would be a table in the capabilities section.)

The Frame Relay circuit information contains the DLCI, the operational circuit status, whether this circuit is to be treated as carrying LMI information, and which port in the output port group of the LFB traffic is to be sent to. As mentioned above, the circuit index could, in some designs, be combined with the DLCI.

### 8.3. Capabilities

The capability information for this LFB includes whether the underlying interface is operational, how many frequencies are supported, and how many total circuits, across all channels, are permitted. The maximum number for a given laser channel can be determined from the properties of the FrameRelayCircuits table. A GET-Properties on path 2.channel.4 will give the CE the properties of the array which include the number of entries used, the first available entry, and the maximum number of entries permitted.

### 8.4. Events

This LFB is defined to be able to generate several events that the CE may be interested in. There are events to report changes in

operational state of frequencies, and the creation and deletion of frequency entries.  There is an event for changes in status of individual frame relay circuits.  So an event notification of 61.5.3.11 would indicate that there had been a circuit status change on subscript 11 of the circuit table in subscript 3 of the frequency table.  The event report would include the new status of the circuit and the DLCI of the circuit.  Arguably, the DLCI is redundant, since the CE presumably knows the DLCI based on the circuit index.  It is included here to show including two pieces of information in an event report.

As described above, the event declaration defines the event target, the event condition, and the event report content.  The event properties indicate whether the CE is subscribed to the event, the specific threshold for the event, and any filter conditions for the event.

Another event shown is a laser power problem.  This event is generated whenever the laser falls below the specified threshold. Thus, a CE can register for the event of laser power loss on all circuits.  It would do this by:

```
T = SET-Properties
  Path-TLV: flags=0, length = 2, path = 61.4
    Path-TLV: flags = property-field, length = 1, path = 2
      Content = 1 (register)
    Path-TLV: flags = property-field, length = 1, path = 3
      Content = 15 (threshold)
```

This would set the registration for the event on all entries in the table.  It would also set the threshold for the event, causing reporting if the power falls below 15.  (Presumably, the CE knows what the scale is for power, and has chosen 15 as a meaningful problem level.)

If a laser oscillates in power near the 15 mark, one could get a lot of notifications.  (If it flips back and forth between 9 and 10, each flip down will generate an event.)  Suppose that the CE decides to suppress this oscillation somewhat on laser channel 5.  It can do this by setting the variance property on that event.  The request would look like:

```
T = SET-Properties
  Path-TLV: flags=0, length = 3, path = 61.4.5
    Path-TLV: flags = property-field, length = 1, path = 4
      Content = 2 (hysteresis)
```

Setting the hysteresis to 2 suppress a lot of spurious notifications.  When the level first falls below 10, a notification is generated. If the power level increases to 10 or 11, and then falls back below 10, an event will not be generated. The power has to recover to at least 12 and fall back below 10 to generate another event. Once common cause of this form of oscillation is when the actual value is right near the border. If it is really 9.5, tiny changes might flip it back and forth between 9 and 10. A variance level of 1 will suppress this sort of condition. Many other events have oscillations that are somewhat wider, so larger variance settings can be used with those.

## 9. IANA Considerations

This model creates the need for unique class names and numeric class identifiers. To meet that goal, IANA will maintain a registry of LFB Class names, corresponding class identifiers, and the document which defines the LFB Class. The registry policy is simply first come first served with regard to LFB Class names. With regard to LFB Class identifiers, identifiers less than 65536 are reserved for assignment by RFCs. Identifiers above 65536 are available for assignment on a first come, first served basis. Registry entries must be documented in a stable, publicly available form.

The LFBLibrary element and all of its sub-elements are defined in the following namespace:

http://ietf.org/forces/1.0/lfbmodel

[Editor s Note: A registry template registry name, and other parts required for a new IANA registry are still needed here.]

## 10. Authors Emeritus

The following are the authors who were instrumental in the creation of earlier releases of this document.

Lily Yang, Intel Corp.
Ram Gopal, Nokia Research Center
Alan DeKok, Infoblox, Inc.
Zsolt Haraszti, Clovis Solutions

## 11. Acknowledgments

Many of the colleagues in our companies and participants in the ForCES mailing list have provided invaluable input into this work.

## 12. Security Considerations

The FE model describes the representation and organization of data sets and attributes in the FEs. The ForCES framework document [2] provides a comprehensive security analysis for the overall ForCES architecture. For example, the ForCES protocol entities must be authenticated per the ForCES requirements before they can access the information elements described in this document via ForCES. Access to the information contained in the FE model is accomplished via the ForCES protocol, which will be defined in separate documents, and thus the security issues will be addressed there.

## 13. Normative References

[1] Khosravi, H. et al., "Requirements for Separation of IP Control and Forwarding", RFC 3654, November 2003.

[2] Yang, L. et al., "Forwarding and Control Element Separation (ForCES) Framework", RFC 3746, April 2004.

## 14. Informative References

[3] Bernet, Y. et al., "An Informal Management Model for Diffserv Routers", RFC 3290, May 2002.

[4] Chan, K. et al., "Differentiated Services Quality of Service Policy Information Base", RFC 3317, March 2003.

[5] Sahita, R. et al., "Framework Policy Information Base", RFC 3318, March 2003.

[6] Moore, B. et al., "Information Model for Describing Network Device QoS Datapath Mechanisms", RFC 3670, January 2004.

[7] Snir, Y. et al., "Policy Framework QoS Information Model", RFC 3644, Nov 2003.

[8] Li, M. et al., "IPsec Policy Information Base", work in progress, April 2004, <draft-ietf-ipsp-ipsecpib-10.txt>.

[9] Quittek, J. et Al., "Requirements for IP Flow Information Export", RFC 3917, October 2004.

[10] Duffield, N., "A Framework for Packet Selection and Reporting", work in progress, January 2005, <draft-ietf-psamp-framework-10.txt>.

[11] Pras, A. and Schoenwaelder, J., RFC 3444 "On the Difference between Information Models and Data Models", January 2003.

[12] Hollenbeck, S. et al., "Guidelines for the Use of Extensible Markup Language (XML) within IETF Protocols", RFC 3470, January 2003.

[13] Thompson, H., Beech, D., Maloney, M. and N. Mendelsohn, "XML Schema Part 1: Structures", W3C REC-xmlschema-1, May 2001, <http://www.w3.org/TR/xmlschema-1/>.

[14] Biron, P. and A. Malhotra, "XML Schema Part 2: Datatypes", W3C REC-xmlschema-2, May 2001, <http://www.w3.org/TR/xmlschema-2/>.

[15] Davis, M. and M. Suignard, "UNICODE Security Considerations", July 2005,<http://www.unicode.org/reports/tr36/tr36-3.html>.

## 15. Authors' Addresses

Joel M. Halpern
Self
P.O. Box 6049
Leesburg, VA 20178
Phone: +1 703 371 3043
Email: jmh@joelhalpern.com

Ellen Deleganes
Intel Corp.
Mail Stop: CO5-156
15400 NW Greenbrier Parkway
Beaverton, OR 97006
Phone: +1 503 677-4996
Email: ellen.m.deleganes@intel.com

## 16. Intellectual Property Right

The authors are not aware of any intellectual property right issues pertaining to this document.

## 17. Copyright Statement