Internet Engineering Task Force (IETF)
Request for Comments: 5812
Category: Standards Track
ISSN: 2070-1721

J. Halpern
Self
J. Hadi Salim
Znyx Networks

March 2010

Forwarding and Control Element Separation (ForCES) Forwarding Element Model

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 RFC 3654.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc5812.

Table of Contents

1. Introduction

RFC 3746 [RFC3746] specifies a framework by which control elements (CEs) can configure and manage one or more separate forwarding elements (FEs) within a network 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 [RFC3654] mandates that the capabilities, states and configuration information be expressed in the form of an FE model.

RFC 3444 [RFC3444] 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 (Logical Functional Block) 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.

1.1. Requirements on the FE Model

RFC 3654 [RFC3654] 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 data path (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 (e.g., components) of each type of LFB; and

- Metadata that may be exchanged between LFBs.

1.2. The FE Model in Relation to FE Implementations

The FE model proposed here is based on an abstraction using 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 is designed, and any defined LFB classes should be designed, such that different implementations of the forwarding data path 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 data path 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.

1.3. The FE Model in Relation to the ForCES Protocol

The ForCES base protocol [RFC5810] 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 intra-FE topology. The details of a particular data path 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 components 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 components and the defined operations.

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

1.4. 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 that 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 were 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 is 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 an XML schema to define the structure of the LFB library documents, as defined in [RFC3470] and [Schema1] and [Schema2]. While these LFB class definitions are not sent in the ForCES protocol, these definitions comply with the recommendations in RFC 3470 [RFC3470] on the use of XML in IETF protocols.

By using an XML schema to define the structure for the LFB library documents, we have a very clear set of syntactic restrictions to go with the desired semantic descriptions and restrictions covered in this document. As a corollary to that, if it is determined that a change in the syntax is needed, then a new schema will be required. This would be identified by a different URN to identify the namespace for such a new schema.

1.5. 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 Section 5 constitute the core of the FE model, detailing the two major aspects of the FE model: a general LFB model and a definition of the FE Object LFB, with its components, including FE capabilities and LFB topology information. Section 6 directly addresses the model requirements imposed by the ForCES requirements defined in RFC 3654 [RFC3654], while Section 7 explains how the FE model should be used in the ForCES protocol.

2. Definitions

The use of compliance terminology (MUST, SHOULD, MAY, MUST NOT) is used in accordance with RFC 2119 [RFC2119]. Such terminology is used in describing the required behavior of ForCES forwarding elements or control elements in supporting or manipulating information described in this model.

Terminology associated with the ForCES requirements is defined in RFC 3654 [RFC3654] 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 LFB modeling of individual Logical Functional Block (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 [RFC5810].

Data Path: A conceptual path taken by packets within the forwarding plane inside an FE. Note that more than one data path can exist within an FE.

LFB (Logical Functional Block) Class (or type): A template that represents 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 data path, 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 data path. 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. XML is used to provide a formal definition of the necessary structures for the modeling. 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 of information define constructs associated with and used by the class definition. These are reusable data types, supported frame (packet) formats, and metadata.

Element: Element is generally used in this document in accordance with the XML usage of the term. It refers to an XML tagged part of an XML document. For a precise definition, please see the full set of XML specifications from the W3C. This term is included in this list for completeness because the ForCES formal model uses XML.

Attribute: Attribute is used in the ForCES formal modeling in accordance with standard XML usage of the term, i.e., to provide attribute information included in an XML tag.

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.

ForCES Component: A ForCES Component is a well-defined, uniquely identifiable and addressable ForCES model building block. A component has a 32-bit ID, name, type, and an optional synopsis description. These are often referred to simply as components.

LFB Component: An LFB component is a ForCES component that defines the Operational parameters of the LFBs that must be visible to the CEs.

Structure Component: A ForCES component that is part of a complex data structure to be used in LFB data definitions. The individual parts that make up a structured set of data are referred to as structure components. These can themselves be of any valid data type, including tables and structures.

Property: ForCES components have properties associated with them, such as readability. Other examples include lengths for variable-sized components. These properties are accessed by the CE for reading (or, where appropriate, writing.) Details on the ForCES properties are found in Section 4.8.

LFB Topology: LFB topology is a representation of the logical interconnection and the placement of LFB instances along the data path 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: FE topology is a representation of how multiple FEs within a single network element (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: The LFB class library is 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.

3. ForCES Model Concepts

Some of the important ForCES concepts used throughout this document are introduced in this section. These include the capability and state abstraction, the FE and LFB model construction, and the unique addressing of the different model structures. Details of these aspects are described in Section 4 and Section 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.

The ForCES FE model includes both a capability and a state abstraction.

- The FE/LFB capability model describes the capabilities and capacities of an FE/LFB by specifying the variation in functions supported and any limitations. Capacity describes the limits of specific components (an example would be a table size limit).

- The state model describes the current state of the FE/LFB, that is, the instantaneous values or operational behavior of the FE/LFB.

Section 3.1 explains the difference between a capability model and a state model, and describes how the two can be combined in the FE model.

The ForCES model construction laid out in this document allows an FE to provide information about its structure for operation. This can be thought of as FE-level information and information about the individual instances of LFBs provided by the FE.

- The ForCES model includes the constructions for defining the class of Logical Functional Blocks (LFBs) that an FE may support. These classes are defined in this and other documents. The definition of such a class provides the information content for monitoring and controlling instances of the LFB class for ForCES purposes. Each LFB model class formally defines the operational LFB components, LFB capabilities, and LFB events. Essentially, Section 3.2 introduces the concept of LFBs as the basic functional building blocks in the ForCES model.

- The FE model also provides the construction necessary to monitor and control the FE as a whole for ForCES purposes. For consistency of operation and simplicity, this information is represented as an LFB, the FE Object LFB class and a singular LFB instance of that class, defined using the LFB model. The FE Object class defines the components to provide information at the FE level, particularly the capabilities of the FE at a coarse level, i.e., not all possible capabilities or all details about the capabilities of the FE. Part of the FE-level information is the LFB topology, which expresses the logical inter-connection between the LFB instances along the data path(s) within the FE. Section 3.3 discusses the LFB topology. The FE Object also includes information about what LFB classes the FE can support.

The ForCES model allows for unique identification of the different constructs it defines. This includes identification of the LFB classes, and of LFB instances within those classes, as well as identification of components within those instances.

The ForCES protocol [RFC5810] encapsulates target address(es) to eventually get to a fine-grained entity being referenced by the CE. The addressing hierarchy is broken into the following:

- An FE is uniquely identified by a 32-bit FEID.

- Each class of LFB is uniquely identified by a 32-bit LFB ClassID. The LFB ClassIDs are global within the network element and may be issued by IANA.

- Within an FE, there can be multiple instances of each LFB class. Each LFB class instance is identified by a 32-bit identifier that is unique within a particular LFB class on that FE.

- All the components within an LFB instance are further defined using 32-bit identifiers.

Refer to Section 3.3 for more details on addressing.

3.1. ForCES Capability Model and State Model

Capability and state modeling applies to both the FE and LFB abstraction.

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

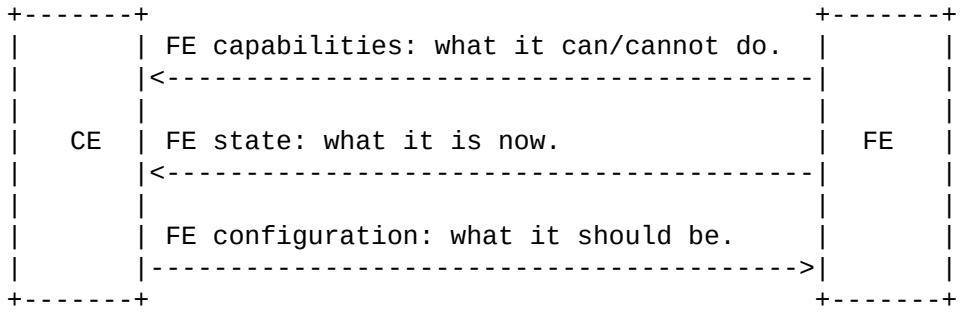


Figure 1: Illustration of FE capabilities, state, and configuration exchange in the context of CE-FE communication via ForCES.

3.1.1. FE Capability Model and State Model

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.

3.1.1.1. FE Capability Model

The FE capability model may be used to describe an FE at a coarse level. For example, an FE might be defined as follows:

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

While one could try to build an object model to fully represent the FE capabilities, other efforts found this approach 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 that 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 RFC 3317 [RFC3317] and framework PIB RFC 3318 [RFC3318].

3.1.1.2. FE State Model

The FE state model presents the snapshot view of the FE to the CE. For example, using an FE state model, an FE might 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; and

- o a specific scheduler with specific behavior and parameters will service these collected queues.

3.1.1.3. LFB Capability and State Model

Both LFB capability and state information are defined formally using the LFB modeling XML schema.

Capability information at the LFB level is an integral part of the LFB model and provides for powerful semantics. For example, when certain features of an LFB class are optional, the CE needs to be able to determine whether those optional features are supported by a given LFB instance. The schema for the definition of LFB classes provides a means for identifying such components.

State information is defined formally using LFB component constructs.

3.1.2. Relating LFB and FE Capability and State Model

Capability information at the FE level describes 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 components including capability information. Since all information is represented as LFBs, this is provided by a single instance of the FE Object LFB class. By using a single instance with a known LFB class and a known instance identification, the ForCES protocol can allow a CE to access this information whenever it needs to, including while the CE is establishing the control of the FE.

Once the FE capability is described to the CE, the FE state information can be represented at two levels. The first level is the logically separable and distinct packet processing functions, called LFBs. The second level of information describes how these individual LFBs are ordered and placed along the data path 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. This topology information is represented as components of the FE Object LFB instance, to allow the CE to fetch and manipulate this.

3.2. Logical Functional Block (LFB) 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 fine-grained LFBs will be defined in detail in one or more documents to be published separately, using the material in this model.

It is also the case that LFBs may exist in order to provide a set of components for control of FE operation by the CE (i.e., a locus of control), without tying that control to specific packets or specific parts of the data path. An example of such an LFB is the FE Object, which provides the CE with information about the FE as a whole, and allows the FE to control some aspects of the FE, such as the data path itself. Such LFBs will not have the packet-oriented properties described in this section.

In general, multiple LFBs are contained in one FE, as shown in Figure 2, and all the LFBs share the same ForCES protocol (Fp) termination point that implements the ForCES protocol logic and maintains the communication channel to and from the CE.

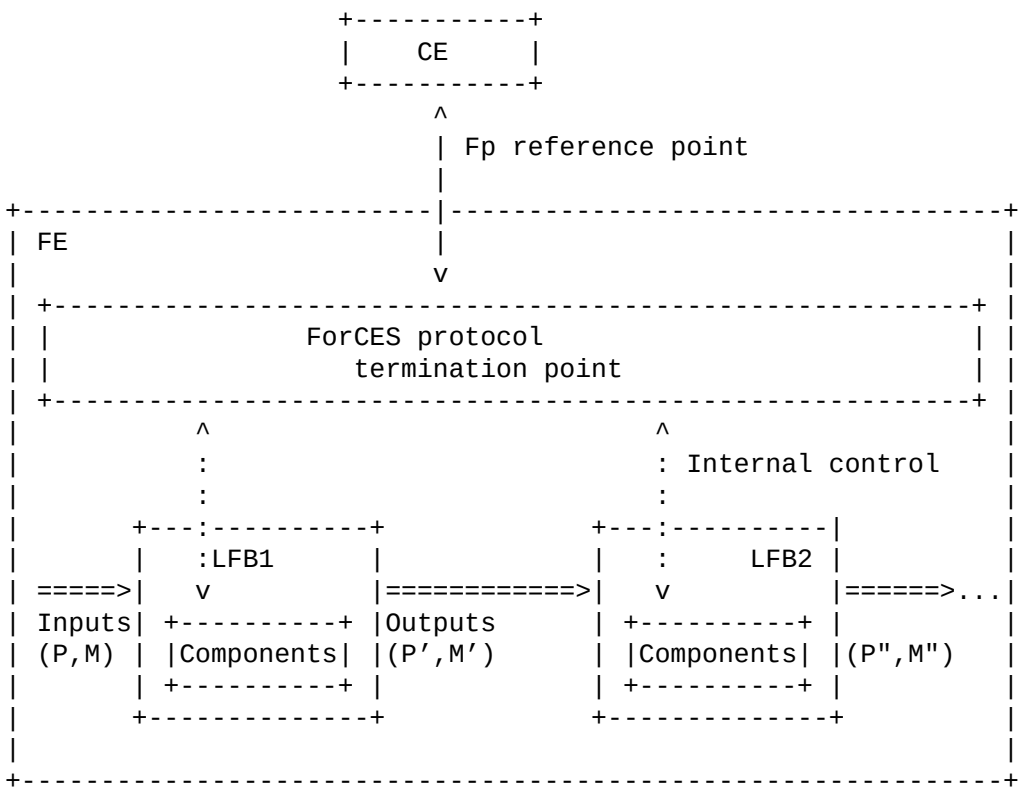


Figure 2: Generic LFB diagram.

An LFB, as shown in Figure 2, may have inputs, outputs, and components that can be queried and manipulated by the CE via an Fp reference point (defined in RFC 3746 [RFC3746]) 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. P (with marks to indicate modification) indicates a data packet, while M (with marks to indicate modification) indicates the metadata associated with a packet. 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 the FE-to-CE communication is used for packet redirection to the control plane, reporting of monitoring and accounting information, reporting of 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 manipulate the components of the LFB instances.

An LFB can have one or more inputs. Each input takes a pair of a packet and its associated metadata. Depending upon the LFB input port definition, the packet or the metadata may be allowed to be

empty (or equivalently to not be provided). When input arrives at an LFB, either the packet or its associated metadata must be non-empty or there is effectively no input. (LFB operation generally may be triggered by input arrival, by timers, or by other system state. It is only in the case where the goal is to have input drive operation that the input must be non-empty.)

The LFB processes the input, and produces one or more outputs, each of which is a pair of a packet and its associated metadata. Again, depending upon the LFB output port definition, either the packet or the metadata may be allowed to be empty (or equivalently to be absent). Metadata attached to packets on output may be metadata that was received, or may be information about the packet processing that may be used by later LFBs in the FEs packet processing.

A namespace is used to associate a unique name and 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 data path. For instance, the CE needs to understand at what point in the data path the IPv4 header TTL is decremented by the FE. That is, the CE needs to know if a control packet could be delivered to it either before or after this point in the data path. In addition, the CE needs to understand where and what type of header modifications (e.g., tunnel header append or strip) are performed by the FEs. Further, the CE works to verify that the various LFBs along a data path within an FE are compatible to link together. Connecting incompatible LFB instances will produce a non-working data path. So the model is designed to provide sufficient information for the CE to make this determination.

Selecting the right granularity for describing the functions of the LFBs is an important aspect of this model. 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. However, the model, and the associated library of LFBs, must not be so detailed and so specific as to significantly constrain implementations. 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 the following, among other information:

- number of inputs and outputs (and whether they are configurable)
- metadata read/consumed from inputs
- metadata produced at the outputs
- packet types 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 components
- events that can be detected by the LFB, with notification to the CE
- LFB operational components

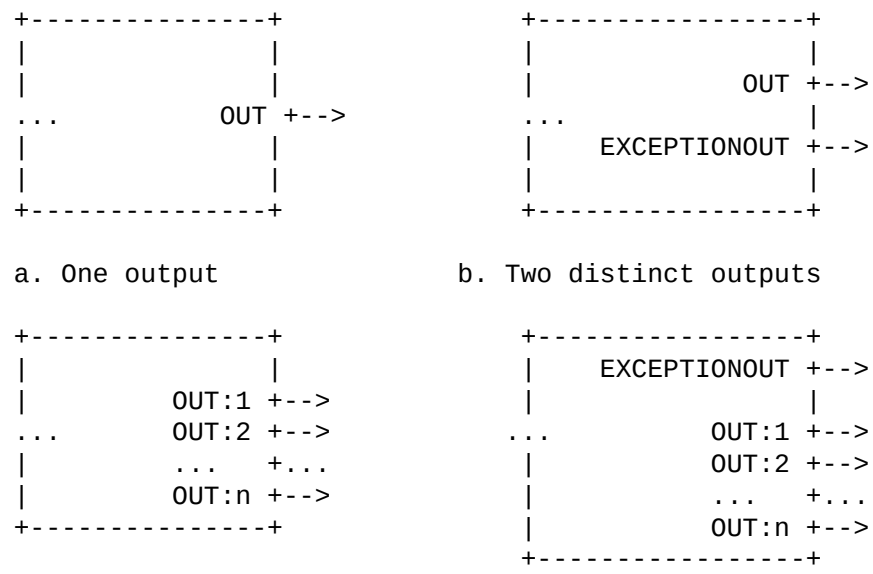
Section 4 of this document provides a detailed discussion of the LFB model with a formal specification of LFB class schema. The rest of Section 3.2 only intends to provide a conceptual overview of some important issues in LFB modeling, without covering all the specific details.

3.2.1. LFB Outputs

An LFB output is a conceptual port on an LFB that can send information to another LFB. The information sent on that port is a pair of a packet and associated metadata, one of which may be empty. (If both were empty, there would be no output.)

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

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



a. One output

b. Two distinct outputs

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 data path. 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 predetermined 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 that 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 (packets) 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 permitted 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 (packet) 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 a component 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. To use Output Port groups, the LFB has to have 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 redirector 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, given LFBs that record the type of packet in a FRAMETYPE metadatum, or a packet rate class in a COLOR metadatum, one may uses these metadata for branching. A redirector can be used to divide the data path into an IPv4 and an IPv6 path based on a FRAMETYPE metadatum (N=2), or to fork into rate-specific paths after metering using the COLOR metadatum (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 metadatum 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 data path and at least one of the following conditions hold:

- the number of downstream directions is inherent from the definition of the class and hence fixed

- the frame type and set of permitted metadata emitted on any of the outputs are different from what is emitted on the other outputs (i.e., they cannot share their frametype and permitted metadata definitions)

An output group is appropriate when the LFB must provide for forking the data path 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 on which the LFB can receive information from other LFBs. The information is typically a pair of a packet and its associated metadata. Either the packet or the metadata may for some LFBs and some situations be empty. They cannot both be empty, as then there is no input.

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 all be combined in the same LFB:

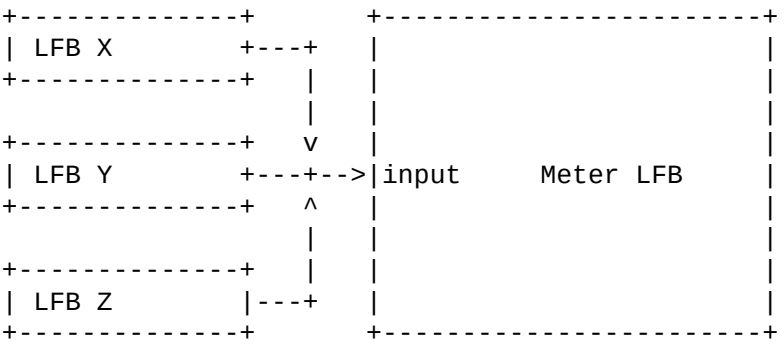
- Implicit multiplexing via a single input

o Explicit multiplexing via multiple singleton inputs

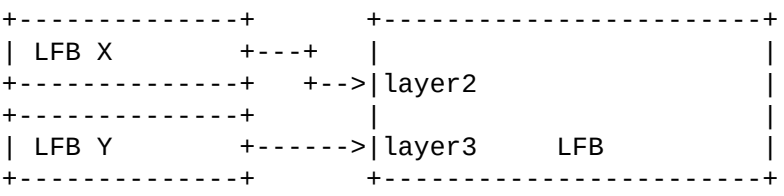
o Explicit multiplexing via a group of inputs (input group)

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 frametype 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.

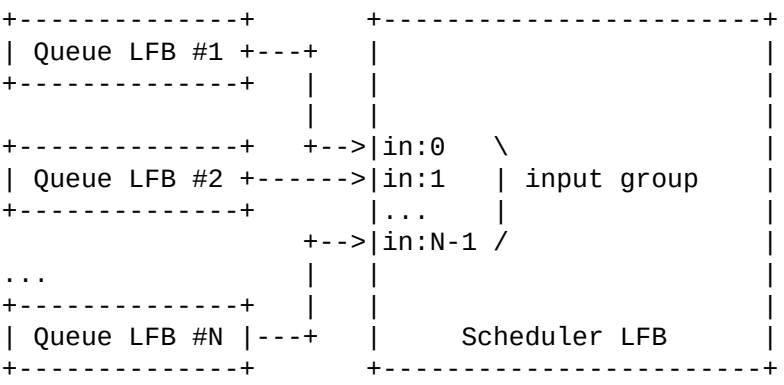
The second method to model fan-in uses individually defined singleton inputs (Figure 4.b). This model is meant for situations where the LFB needs to handle distinct types of packet streams, requiring input-specific handling inside the LFB, and where the number of such distinct cases is known when the LFB class is defined. For example, an LFB that can perform both Layer 2 decapsulation (to Layer 3) and Layer 3 encapsulation (to Layer 2) 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 versus 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.



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



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



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

Figure 4: Examples of LFBs with various input combinations.

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 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 a component defined in 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 4.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 is 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) MUST be specified. These are the types of packets that a given LFB input is capable of receiving and processing, or that a given LFB output is capable of producing. This model requires that 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 or not such processing is happening is opaque to the CE.

3.2.4. Metadata

Metadata is state that is passed from one LFB to another alongside a packet. The metadata passed with the packet assists subsequent LFBs to process that packet.

The ForCES model defines metadata as precise atomic definitions in the form of label, value pairs.

The ForCES model provides to the authors of LFB classes a way to formally define how to achieve metadata creation, modification, reading, as well as consumption (deletion).

Inter-FE metadata, i.e., metadata crossing FEs, while it is likely to be semantically similar to this metadata, is out of scope for this document.

Section 4 has informal details on metadata.

3.2.4.1. Metadata Lifecycle within the ForCES Model

Each metadatum is 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 label here is shown as a textual label, but for protocol processing it is associated with a unique numeric value (identifier).

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 specify neither 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.

3.2.4.2. Metadata Production and Consumption

For a given metadatum on a given packet path, there MUST be at least one producer LFB that creates that metadatum and SHOULD be at least one consumer LFB that needs that metadatum.

In the ForCES model, the producer and consumer LFBs of a metadatum are not required to be adjacent. In addition, there may be multiple producers and consumers for the same metadatum. When a packet path involves multiple producers of the same metadatum, then subsequent producers overwrite that metadatum 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 "unconditional" metadata, whereas the latter is "conditional" metadata. For example, deep packet inspection LFB might produce several pieces of metadata about the packet. The first metadatum might be the IP protocol (TCP, UDP, SCTP, ...) being carried, and two additional metadata items might be the source and destination port number. These additional metadata items are conditional on the value of the first metadatum (IP carried protocol) as they are only produced for protocols that

use port numbers. In the case of conditional metadata, it should be possible to determine from the definition of the LFB when "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 metadatum, or may treat it as "optional" information. In the latter case, the LFB class definition MUST explicitly define what happens if any optional metadata is not provided. One approach is to specify a default value for each optional metadatum, and assume that the default value is used for any metadata that is not provided with the packet.

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).

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 metadatum.

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

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

In the ForCES model, a new metadatum is generated by an LFB when the LFB applies a WRITE operation to a metadatum 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 whether or not the given metadatum existed. If it existed, the metadatum gets over-written; if it did not exist, the metadatum 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 metadatum associated with the packet.

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 description of how such messages are sent, and their format, is part of the Forwarding and Control Element Separation (ForCES) protocol [RFC5810] document. Indicating how such conditions are understood is part of the job of this model.

Events are declared in the LFB class definition. The LFB event declaration constitutes:

- a unique 32-bit identifier.
- An LFB component that is used to trigger the event. This entity is known as the event target.
- A condition that will happen to the event target that will result in a generation of an event to the CE. Examples of a condition include something getting created or deleted, a config change, etc.
- What should be reported to the CE by the FE if the declared condition is met.

The declaration of an event within an LFB class essentially defines what part of the LFB component(s) need to be monitored for events, what condition on the LFB monitored LFB component an FE should detect to trigger such an event, and what to report to the CE when the event is triggered.

While events may be declared by the LFB class definition, runtime activity is controlled using built-in event properties using LFB component properties (discussed in Section 3.2.6). A CE subscribes to the events on an LFB class instance by setting an event property for subscription. Each event has a subscription property that is by default off. A CE wishing to receive a specific event needs to turn on the subscription property at runtime.

Event properties also provide semantics for runtime event filtering. A CE may set an event property to further suppress events to which it has already subscribed. The LFB model defines such filters to include threshold values, hysteresis, time intervals, number of events, etc.

The contents of 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 that the CE already has, large volumes of information, or information related in complex fashions.

From a conceptual point of view, at runtime, event processing is split into:

1. Detection of something happening to the (declared during LFB class definition) event target. Processing the next step happens if the CE subscribed (at runtime) to the event.

2. Checking of the (declared during LFB class definition) condition on the LFB event target. If the condition is met, proceed with the next step.

3. Checking (runtime set) event filters if they exist to see if the event should be reported or suppressed. If the event is to be reported, proceed to the next step.

4. Submitting of the declared report to the CE.

Section 4.7.6 discusses events in more details.

3.2.6. Component Properties

LFBs and structures are made up of components, containing the information that the CE needs to see and/or change about the functioning of the LFB. These components, as described in detail in Section 4.7, may be basic values, complex structures (containing multiple components themselves, each of which can be values, structures, or tables), or tables (which contain values, structures, or tables). Components may be defined such that their appearance in LFB instances is optional. Components may be readable or writable at the discretion of the FE implementation. The CE needs to know these properties. Additionally, certain kinds of components (arrays / tables, aliases, and events) 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.

Section 4.8 describes properties in more details.

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 3.2.8) has special rules. If an FE data path 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 backward-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 backward compatible. It may also be used to remove components that 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 that 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. ForCES Model Addressing

Figure 5 demonstrates the abstraction of the different ForCES model entities. The ForCES protocol provides the mechanism to uniquely identify any of the LFB class instance components.

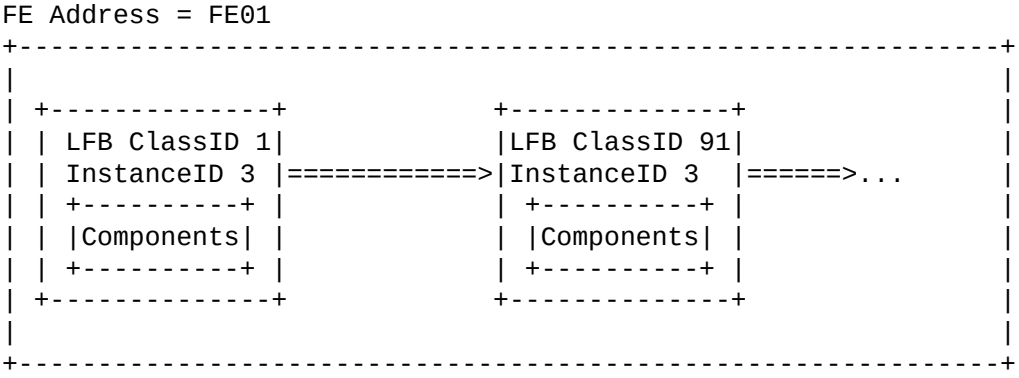


Figure 5: FE entity hierarchy.

At the top of the addressing hierarchy is the FE identifier. In the example above, the 32-bit FE identifier is illustrated with the mnemonic FE01. The next 32-bit entity selector is the LFB ClassID. In the illustration above, two LFB classes with identifiers 1 and 91 are demonstrated. The example above further illustrates one instance of each of the two classes. The scope of the 32-bit LFB class instance identifier is valid only within the LFB class. To emphasize that point, each of class 1 and 91 has an instance of 3.

Using the described addressing scheme, a message could be sent to address FE01, LFB ClassID 1, LFB InstanceID 3, utilizing the ForCES protocol. However, to be effective, such a message would have to target entities within an LFB. These entities could be carrying state, capability, etc. These are further illustrated in Figure 6 below.

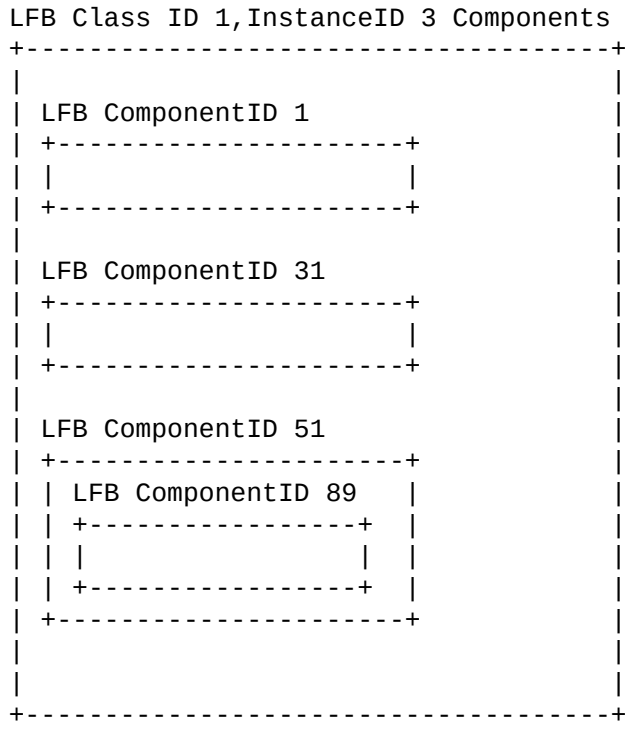


Figure 6: LFB hierarchy.

Figure 6 zooms into the components carried by LFB Class ID 1, LFB InstanceID 3 from Figure 5.

The example shows three components with 32-bit component identifiers 1, 31, and 51. LFB ComponentID 51 is a complex structure encapsulating within it an entity with LFB ComponentID 89. LFB ComponentID 89 could be a complex structure itself, but is restricted in the example for the sake of clarity.

3.3.1. Addressing LFB Components: Paths and Keys

As mentioned above, LFB components could be complex structures, such as a table, or even more complex structures such as a table whose cells are further tables, etc. The ForCES model XML schema (Section 4) allows for uniquely identifying anything with such complexity, utilizing the concept of dot-annotated static paths and content addressing of paths as derived from keys. As an example, if LFB ComponentID 51 were a structure, then the path to LFB ComponentID 89 above will be 51.89.

LFB ComponentID 51 might represent a table (an array). In that case, to select the LFB component with ID 89 from within the 7th entry of the table, one would use the path 51.7.89. In addition to supporting explicit table element selection by including an index in the dotted path, the model supports identifying table elements by their contents. This is referred to as using keys, or key indexing. So, as a further example, if ComponentID 51 was a table that was key index-able, then a key describing content could also be passed by the CE, along with path 51 to select the table, and followed by the path 89 to select the table structure element, which upon computation by the FE would resolve to the LFB ComponentID 89 within the specified table entry.

3.4. FE Data Path Modeling

Packets coming into the FE from ingress ports generally flow through one or more 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), 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 data paths 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 data path.

LFB topology is a directed graph representation of the logical data paths 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 3.4.1 discusses how the FE data paths can be modeled as LFB topology, while Section 3.4.2 focuses on issues related to LFB topology reconfiguration.

3.4.1. Alternative Approaches for Modeling FE Data Paths

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

o 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.

o Encoded State Approach

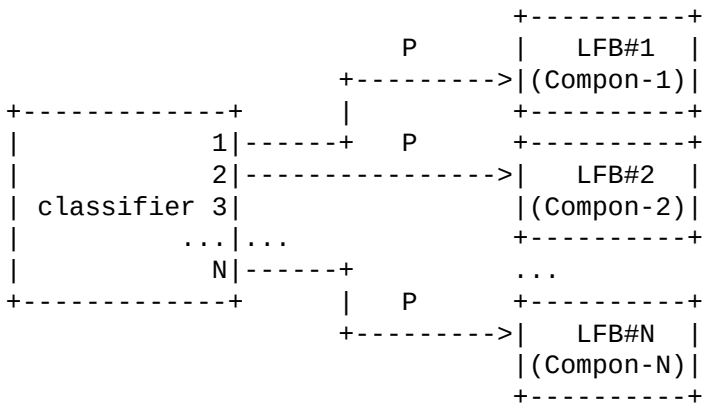
An alternate way of expressing differential treatment is by using metadata. The result of the operation of an LFB can be encoded in a metadatum, which is passed along with the packet to downstream LFBs. A downstream LFB, in turn, can use the metadata and its value (e.g., as an index into some table)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

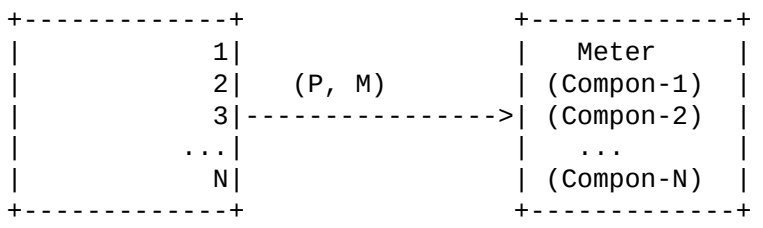
Using a simple example of a classifier with N classification outputs followed by other LFBs, Figure 7.a shows what the LFB topology looks like when using the pure topological approach. Each output from the classifier goes to one of the N LFBs where no metadata is needed. The topological approach is simple, 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 components, the encoded state approach gives a much simpler topology representation, as shown in Figure 7.b. The encoded state approach requires that a table of N rows of meter components be provided in the Meter node itself, with each row representing the attributes for one meter instance. A metadatum 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.

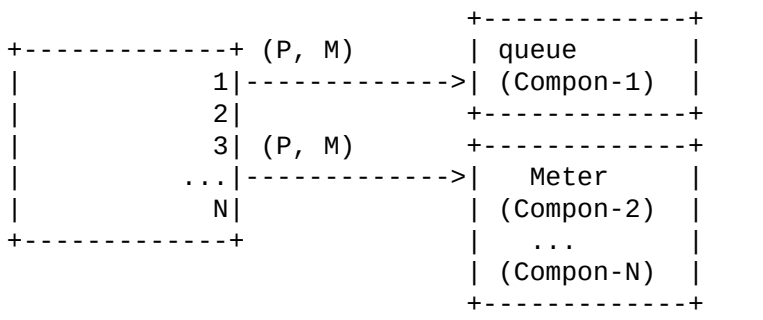
What if those N nodes (LFB#1, LFB#2, ..., LFB#N) are not of the same type? For example, if LFB#1 is a queue while the rest are all meters, what is the best way to represent such data paths? While it is still possible to use either the pure topological approach or the pure encoded state approach, the natural combination of the two appears to be the best option. Figure 7.c depicts two different functional data paths using the topological approach while leaving the N-1 meter instances distinguished by metadata only, as shown in Figure 7.c.



(a) Using pure topological approach



(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).



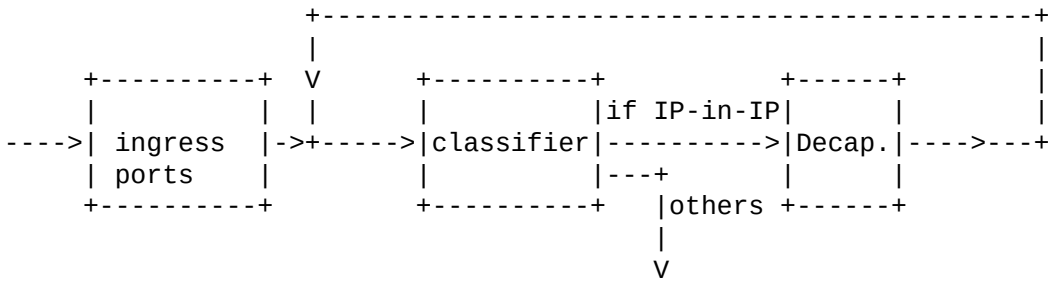
(c) Using a combination of the two, if LFB#1, LFB#2, ..., and LFB#N are of different types (e.g., queue and meter).

Figure 7: An example of how to model FE data paths.

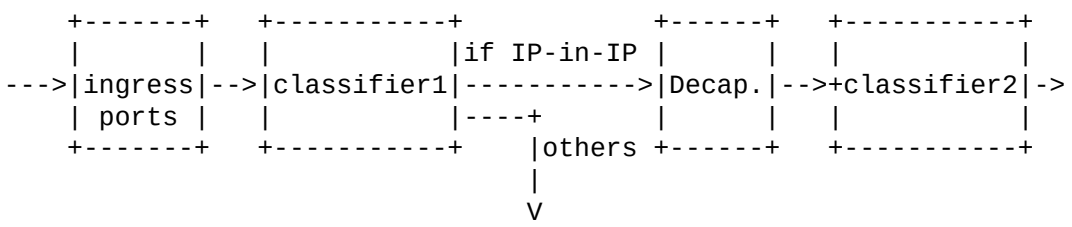
From this example, we demonstrate that each approach has a distinct advantage depending on the situation. Using the encoded state approach, fewer connections are typically needed between a fan-out node and its next LFB instances of the same type because each packet carries metadata the following nodes 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 data paths.

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 data path 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).



(a) The LFB topology with a logical loop



(b) The LFB topology without the loop utilizing two independent classifier instances.

Figure 8: 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 8 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 8.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 8.b.

3.4.2. 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 data paths within an FE, configuring the LFB topology means dynamically changing the data paths, including changing the LFBs along the data paths on an FE (e.g., creating/instantiating, updating, or deleting LFBs) and setting up or deleting interconnections between outputs of upstream LFBs to inputs of downstream LFBs.

Why would the data paths on an FE ever change dynamically? The data paths on an FE are set up by the CE to provide certain data plane services (e.g., Diffserv, VPN) to the network element's (NE) customers. The purpose of reconfiguring the data paths is to enable the CE to customize the services the NE is delivering at run time. The CE needs to change the data paths when the service requirements change, such as adding a new customer or when an existing customer changes their service. However, note that not all data path changes result in changes in the LFB topology graph. Changes in the graph are dependent on the approach used to map the data paths into LFB topology. As discussed in Section 3.4.1, the topological approach and encoded state approach can result in very different looking LFB topologies for the same data paths. 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 data paths, 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 an implementation detail internal to the FE and outside the scope of the 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
- The 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)

The CE needs some programming help in order to cope with the range of complexity. In other words, 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) 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 document.

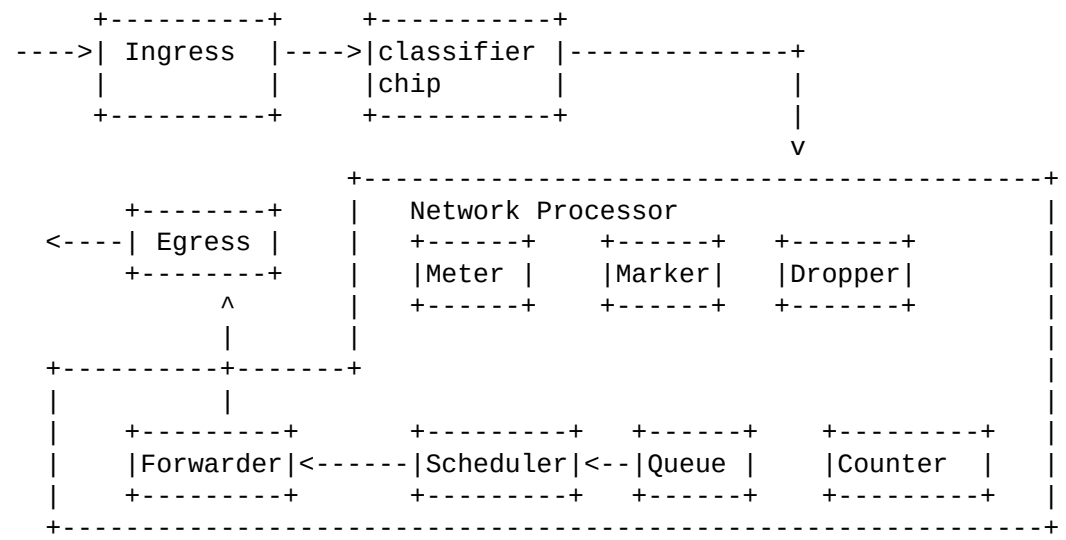


Figure 9: The capability of an FE as reported to the CE.

Figure 9 shows an example where a QoS-enabled (quality-of-service) router has several line cards that have a few ingress ports and egress ports, a specialized classification chip, and 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 hardwired 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
- 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 data path before going into the egress ports must be the IPv4 forwarder

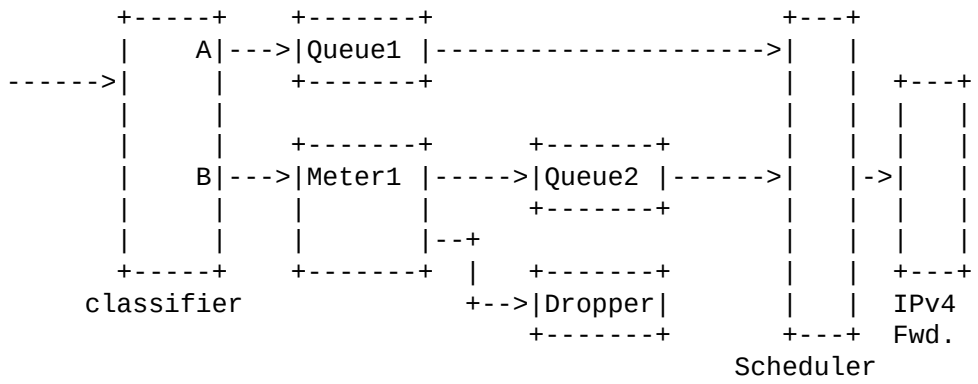


Figure 10: An LFB topology as configured by the CE and accepted by the FE.

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 9 depicts the FE capability, while Figure 10 and Figure 11 depict two different topologies that the CE may request the FE to configure. Note that Figure 11 is not fully drawn, as inter-LFB links are included to suggest potential complexity, without drawing in the endpoints of all such links.

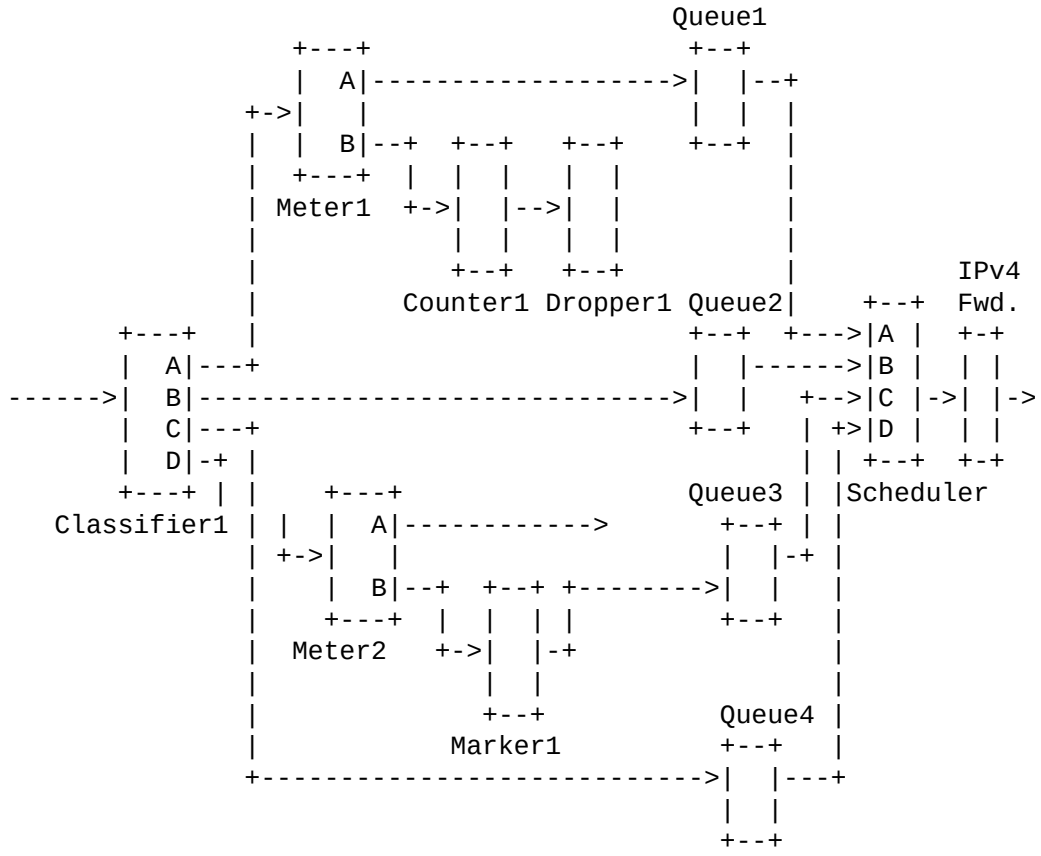


Figure 11: Another LFB topology as configured
by the CE and accepted by the FE.

Note that both the ingress and egress are omitted in Figure 10 and Figure 11 to simplify the representation. The topology in Figure 11 is considerably more complex than Figure 10, 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 an LFB class library, and each document is called an LFB class library document (or library document, for short). Each of the library documents MUST conform to the schema presented in this section. The schema here and the rules for conforming to the schema are those defined by the W3C in the definitions of XML schema in XML schema [Schema1] and XML schema DataTypes [Schema2]. 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.

The following sections describe the portions of an LFBLibrary XML document. The descriptions primarily provide the necessary semantic information to understand the meaning and uses of the XML elements. The XML schema below provides the final definition on what elements are permitted, and their base syntax. Unfortunately, due to the limitations of English and XML, there are constraints described in the semantic sections that are not fully captured in the XML schema, so both sets of information need to be used to build a compliant library document.

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. A library document contains a sequence of top-level elements. The following is a list of all the elements that can occur directly in the <LFBLibrary> element. If they occur, they must occur in the order listed.

- <description> providing a text description of the purpose of the library document,
- <load> for loading information from other library documents,
- <frameDefs> for the frame declarations,

- <dataTypeDefs> for defining common data types,

- <metadataDefs> for defining metadata, and

- <LFBClassDefs> for defining LFB classes.

Each element is optional. One library document may contain only metadata definitions, another may contain only LFB class definitions, and yet another may contain all of the above.

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 the C programming language. Importing is expressed by the use of <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. Note that what this performs is a ForCES inclusion, not an XML inclusion. The semantic content of the library referenced by the <load> element is included, not the xml content. Also, in terms of the conceptual processing of <load> elements, the total set of documents loaded is considered to form a single document for processing. A given document is included in this set only once, even if it is referenced by <load> elements several times, even from several different files. As the processing of LFBLibrary information is not order dependent, the order for processing loaded elements is up to the implementor, as long as the total effect is as if all of the information from all the files were available for referencing when needed. Note that such computer processing of ForCES model library documents may be helpful for various implementations, but is not required to define the libraries, or for the actual operation of the protocol itself.

The following is a skeleton of a library document:

```
<?xml version="1.0" encoding="UTF-8"?>
<LFBLibrary xmlns="urn:ietf:params:xml:ns:forces:lfbmodel:1.0"
  provides="this_library">

  <description>

  </description>

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

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

  <!-- 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. Below are 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.example.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 components 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, 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. The <dataTypeDef> element MAY also include an optional longer description, 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.
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, as defined under atomic types (Section 4.5.2).

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 components of some compound or atomic data type. They may be a structure of named components of compound or atomic data types (cf. C structures). They may be a union of named components of compound or atomic data types (cf. C unions). They may also be defined as augmentations (explained in Section 4.5.7) of existing compound data types.

Given that the ForCES protocol will be getting and setting component 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 ForCES protocol [RFC5810] document to define, not the model document. Strings and octetstrings must be conveyed by the protocol with their length, as they are not delimited, the value does not itself include the length, and these items 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 component properties for string and octetstring components 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. Components of type string (or string[n]) MAY be used to hold identifiers for correlation with components 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 [RFC5810] 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 [UNICODE]. 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, a component 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 components at the time the set is performed, just as range checks for range-limited components 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. Alias is used to allow a component inside an LFB to be an indirect reference to another component inside the same or a different LFB class or instance. The alias component behaves like a structure, one component 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 Renaming 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. Depending upon context, this document and others refer to such arrays as tables or arrays interchangeably, without semantic or syntactic implication. 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 "maxlength" attribute specifies the maximum allowed length. This attribute should be used to encode semantic limitations, not implementation limitations. The latter (support for implementation constraints) should be handled by capability components of LFB classes, and should never be included as the maxlength in a data type array that is regarded as being 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 is always 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 a component for operations (for details, see the ForCES protocol [RFC5810]).

- In any instance of the array, each declared key MUST be unique within that instance. That is, no two components of an array may have the same values on all the fields that make up a key.

Each key is declared with a keyID for use in the ForCES protocol [RFC5810], 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 components of the value type, each identified by

name. Since the field MAY be a component of the contained structure, a component of a component of a structure, or further nested, the field name is actually a concatenated sequence of component 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 predefined data type as the array content type:

```
  <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" maxlength="8">
            <typeRef>ieeemacaddr</typeRef>
        </array>
      </dataTypeDef>
```

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

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

In the above example, each entry of the array is a <struct> of two components ("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, prefix length, and information source. This means that in any instance of this table, no two entries can have the same IP address, prefix length, and information source.

```
      <dataTypeDef>
        <name>ipPrefixInfo_table</name>
        <synopsis>
          A table of information about known prefixes
        </synopsis>
        <array type="variable-size">
          <struct>
            <component componentID="1">
              <name>address-prefix</name>
              <synopsis>the prefix being described</synopsis>
              <typeRef>ipv4Prefix</typeRef>
            </component>
            <component componentID="2">
              <name>source</name>
              <synopsis>
                  the protocol or process providing this information
              </synopsis>
              <typeRef>uint16</typeRef>
            </component>
            <component componentID="3">
              <name>prefInfo</name>
              <synopsis>the information we care about</synopsis>
              <typeRef>hypothetical-info-type</typeRef>
            </component>
          </struct>
          <contentKey contentKeyID="1">
            <contentKeyField> address-prefix.ipv4addr</contentKeyField>
            <contentKeyField> address-prefix.prefixlen</contentKeyField>
            <contentKeyField> source</contentKeyField>
          </contentKey>
        </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 components that are potentially nested inside other components 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 components 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 "*" MUST be used as the single key field identifier.

In reference array or structure elements, it is possible to construct keyFields that do not exist. keyField references SHOULD never reference optional structure components. For references to array elements, care must be taken to ensure that the necessary array elements exist when creating or modifying the overall array element. Failure to do so will result in FEs returning errors on the creation attempt.

4.5.4. <struct> Element to Define Structures

A structure is composed of a collection of data components. Each data component 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. These components are defined using <component> elements. A <struct> element MAY contain an optional derivation indication, a <derivedFrom> element. The structure definition MUST contain a sequence of one or more <component> elements.

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

The <component> element MUST include a componentID attribute. This provides the numeric ID for this component, for use by the protocol. The <component> MUST contain a component name and a synopsis. It MAY contain a <description> element giving a textual description of the component. The definition MAY also include an <optional> element, which indicates that the component being defined is optional. The definition MUST contain elements to define the data type of the component, as described above.

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 optional derivedFrom attribute in the struct declaration before the definition of the augmenting or replacing components. Section 4.5.7 describes how this is done in more detail.

The componentID attribute for different components in a structure (or in an LFB) MUST be distinct. They do not need to be in order, nor do they need to be sequential. For clarity of human readability, and ease of maintenance, it is usual to define at least sequential sets of values. But this is for human ease, not a model or protocol requirement.

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

An example is the following:

```
<dataTypeDef>
 <name>ipv4prefix</name>
 <synopsis>
  IPv4 prefix defined by an address and a prefix length
 </synopsis>
 <struct>
  <component componentID="1">
   <name>address</name>
   <synopsis>Address part</synopsis>
   <typeRef>ipv4addr</typeRef>
  </component>
  <component componentID="2">
   <name>prefixlen</name>
   <synopsis>Prefix length part</synopsis>
   <atomic>
    <baseType>uchar</baseType>
    <rangeRestriction>
     <allowedRange min="0" max="32"/>
    </rangeRestriction>
   </atomic>
  </component>
 </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 is always a compound type, even when the union contains only one element.

4.5.6. <alias> Element

It is sometimes necessary to have a component in an LFB or structure refer to information (a component) in other LFBs. This can, for example, allow an ARP LFB to share the IP->MAC Address table with the local transmission LFB, without duplicating information. Similarly, it could allow a traffic measurement LFB to share information with a traffic enforcement LFB. The <alias> declaration creates the constructs for this. This construct tells the CE and FE that any manipulation of the defined data is actually manipulation of data

defined to exist in some specified part of some other LFB instance. The content of an <alias> element MUST be a named type. Whatever component the alias references (which is determined by the alias component properties, as described below), that component must be of the same type as that declared for the alias. Thus, when the CE or FE dereferences the alias component, the type of the information returned is known. The type can be a base type or 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 is permitted.

The target of a component declared by an <alias> element is determined by the information in the component's properties. Like all components, the properties include the support / read / write permission for the alias. In addition, there are several fields (components) in the alias properties that define the target of the alias. These components 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 component. The type of the target element must match the declared type of the alias. Details of the alias property structure are described in Section 4.8 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 components do not themselves have properties.)

4.5.7. 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 component MAY be replaced in the definition of an augmenting structure, but MAY only be replaced with an augmentation derived from the current type of the existing component. An existing component cannot be deleted. If the existing compound type is an array, augmentation means augmentation of the array element type.

Augmentation MUST NOT be applied to unions.

One consequence of this is that augmentations are backward compatible with the compound type from which they are derived. As such, augmentations are useful in defining components for LFB subclasses with backward compatibility. In addition to adding new components to a class, the data type of an existing component MAY be replaced by an

augmentation of that component, and still meet the compatibility rules for subclasses. This compatibility constraint is why augmentations cannot be applied to unions.

For example, consider a simple base LFB class A that has only one component (comp1) of type X. One way to derive class A1 from A can be by simply adding a second component (of any type). Another way to derive a class A2 from A can be by replacing the original component (comp1) 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. Component names and component IDs for new components within the augmentation MUST NOT be the same as those in the structure type being augmented. For those components where the data type of an existing component is being replaced with a suitable augmenting data type, the existing component name and component ID MUST be used in the augmentation. Other than the constraint on existing elements, there is no requirement that the new component IDs be sequential with, greater than, or in any other specific relationship to the existing component IDs except different. It is expected that using values sequential within an augmentation, and distinct from the previously used values, will be a common method to enhance human readability.

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 metadatum.

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 tag value to be used for this metadata, and value type definition information. Only atomic data types can be used as value types for metadata. The <metadataDef> element MAY contain a detailed description element.

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 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.
- <components> defines the operational components of the LFB.
- <capabilities> defines the capability components 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 maintains a registry of LFB class names and class identifiers, along with a reference to the document defining the class.

Below is a skeleton of an example LFB class definition. Note that in order to keep from complicating the XML schema, the order of elements in the class definition is fixed. Elements, if they appear, must appear in the order shown.

```
<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>

  <components>
   ...
  </components>

  <capabilities>
   ...
  </capabilities>

  <events>
   ...
  </events>

  <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 components and capabilities will have componentIDs for use by the ForCES protocol. These parallel the componentIDs used in structs, and are used the same way. Component and capability componentIDs 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.

The componentID attribute for different items in an LFB class definition (or components in a struct) MUST be distinct. They do not need to be in order, nor do they need to be sequential. For clarity of human readability, and ease of maintenance, it is usual to define at least sequential sets of values. But this is for human ease, not a model or protocol requirement.

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. In the absence of a <derivedFrom>, the class is conceptually derived from the common, empty, base class.

It is assumed that a derived class is backward compatible with its base class. A derived class MAY add components to a parent class, but cannot delete components. This also applies to input and output ports, events, and capabilities.

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 4.7.3). 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 only by 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", "vpnid"}. 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 metadatum, it must be specified whether the metadatum is required or optional. For each optional metadatum, a default value must be specified, which is used by the LFB if the metadatum 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 "true" value (the default value is "false").

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

```
<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="true">
  ... 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 lists 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 metadatum. 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 metadatum 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. An example is the following:

```
<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 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 MUST 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 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 only by 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 metadatum, it should be specified whether the metadatum 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 "true" value (the default value is "false").

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="true">
  <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 metadatum 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. <components> Element to Define LFB Operational Components

Operational parameters of the LFBs that must be visible to the CEs are conceptualized in the model as the LFB components. These include, for example, flags, single parameter arguments, complex arguments, and tables. Note that the components 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 components and hence are not covered.

Some examples for LFB components are:

- Configurable flags and switches selecting between operational modes of the LFB
- Number of inputs or outputs in a port group
- 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

The ForCES model supports the definition of access permission restrictions on what the CE can do with an LFB component. The following categories are supported by the model:

- No-access components. This is useful for completeness, and to allow for defining objects that are used by other things, but not directly referencable by the CE. It is also useful for an FE that is reporting that certain defined, and typically accessible, components are not supported for CE access by a reporting FE.
- Read-only components.
- Read-write components.
- Write-only components. This could be any configurable data for which read capability is not provided to the CEs (e.g., the security key information).
- Read-reset components. The CE can read and reset this resource, but cannot set it to an arbitrary value. Example: Counters.

- Firing-only components. A write attempt to this resource will trigger some specific actions in the LFB, but the actual value written is ignored.

The LFB class MUST define only one possible access mode for a given component.

The components of the LFB class are listed in the <components> element. Each component is defined by an <component> element. A <component> element contains some or all of the following elements, some of which are mandatory:

- <name> MUST occur, and defines the name of the component. This name must be unique among the components of the LFB class. Example: "version".

- <synopsis> is also mandatory, and provides a brief description of the purpose of the component.

- <optional/> is an optional element, and if present indicates that this component is optional.

- The data type of the component can be defined either via a reference to a predefined data type or by 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>, or <union>. Their usage is identical to how they are used inside <dataTypeDef> elements (see Section 4.5). Some form of data type definition MUST be included in the component definition.

- The <defaultValue> element is optional, and if present is used to specify a default value for a component. If a default value is specified, the FE must ensure that the component has that value when the LFB is initialized or reset. If a default value is not specified for a component, the CE MUST make no assumptions as to what the value of the component will be upon initialization. The CE must either read the value or set the value, if it needs to know what it is.

- The <description> element MAY also appear. If included, it provides a longer description of the meaning or usage of the particular component being defined.

The <component> element also MUST have a componentID attribute, which is a numeric value used by the ForCES protocol.

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

Whether optional components are supported, and whether components 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 component. An access control setting of "trigger-only" means that this component is included only for use in event detection.

The following example defines two components for an LFB:

```
<components>
 <component access="read-only" componentID="1">
  <name>foo</name>
  <synopsis>number of things</synopsis>
  <typeRef>uint32</typeRef>
 </component>
 <component access="read-write" componentID="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>
 </component>
</components>
```

The first component ("foo") is a read-only 32-bit unsigned integer, defined by referring to the built-in "uint32" atomic type. The second component ("bar") is also an integer, but uses the <atomic> element to provide additional range restrictions. This component has access mode of read-write allowing it to be both read and written. A default value of 10 is provided for bar. Although the access for bar is read-write, some implementations MAY offer only more restrictive access, and this would be reported in the component properties.

Note that not all components 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 components anyway. The ForCES protocol mechanisms should include appropriate error indicators for this case.

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

4.7.5. <capabilities> Element to Define LFB Capability Components

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. Some of these limitations are captured by the property information of the LFB components. The model allows for the notion of additional capability information.

Such capability-related information is expressed by the capability components of the LFB class. The capability components 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 component. The format of the <capability> element is almost the same as the <component> 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 components follow:

- The version of the LFB class with which this LFB instance complies
- Supported optional features of the LFB class
- Maximum number of configurable outputs for an output group
- Metadata pass-through limitations of the LFB
- Additional range restriction on operational components

The following example lists two capability attributes:

```
<capabilities>
 <capability componentID="3">
  <name>version</name>
  <synopsis>
   LFB class version this instance is compliant with.
  </synopsis>
  <typeRef>version</typeRef>
 </capability>
 <capability componentID="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. High-level view on the declaration and operation of LFB events is described in Section 3.2.5.

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 (per LFB class) ID of the event. The element will include:

- <eventTarget> element indicating which LFB field (component) 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.

The example below demonstrates the different constructs.

The <events> element has a baseID attribute value, which is normally <events baseID="number">. The value of the baseID is the starting componentID for the path that identifies events. It must not be the same as the componentID of any top-level components (including capabilities) 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 in the derived LFB <events> element. Instead, the baseID value from the parent LFB class is used. In the example shown, the baseID is 7.

```
<events baseID="7">
 <event eventID="7">
   <name>Foochanged</name>
   <synopsis>
       An example event for a scalar
   </synopsis>
   <eventTarget>
     <eventField>foo</eventField>
   </eventTarget>
   <eventChanged/>
   <eventReports>
     <!-- report the new state -->
     <eventReport>
       <eventField>foo</eventField>
     </eventReport>
   </eventReports>
 </event>

 <event eventID="8">
   <name>Goof1changed</name>
   <synopsis>
       An example event for a complex structure
   </synopsis>
   <eventTarget>
     <!-- target is goo.f1 -->
     <eventField>goo</eventField>
     <eventField>f1</eventField>
   </eventTarget>
   <eventChanged/>
   <eventReports>
     <!-- report the new state of goo.f1 -->
     <eventReport>
     <eventField>goo</eventField>
     <eventField>f1</eventField>
     </eventReport>
   </eventReports>
 </event>
```

```
    <event eventID="9">
      <name>NewbarEntry</name>
      <synopsis>
          Event for a new entry created on table bar
      </synopsis>
      <eventTarget>
        <eventField>bar</eventField>
        <eventSubscript>_barIndex_</eventSubscript>
      </eventTarget>
      <eventCreated/>
      <eventReports>
        <eventReport>
         <eventField>bar</eventField>
         <eventSubscript>_barIndex_</eventSubscript>
       </eventReport>
       <eventReport>
        <eventField>foo</eventField>
       </eventReport>
      </eventReports>
    </event>

    <event eventID="10">
      <name>Gah11changed</name>
      <synopsis>
          Event for table gah, entry index 11 changing
      </synopsis>
      <eventTarget>
        <eventField>gah</eventField>
        <eventSubscript>11</eventSubscript>
      </eventTarget>
      <eventChanged/>
      <eventReports>
        <eventReport>
         <eventField>gah</eventField>
         <eventSubscript>11</eventSubscript>
       </eventReport>
      </eventReports>
    </event>
```

```
    <event eventID="11">
      <name>Gah10field1</name>
      <synopsis>
          Event for table gah, entry index 10, column field1 changing
      </synopsis>
      <eventTarget>
        <eventField>gah</eventField>
        <eventSubscript>10</eventSubscript>
        <eventField>field1</eventField>
      </eventTarget>
      <eventChanged/>
      <eventReports>
        <eventReport>
         <eventField>gah</eventField>
         <eventSubscript>10</eventSubscript>
        </eventReport>
      </eventReports>
    </event>
  </events>
```

4.7.6.1. <eventTarget> Element

The <eventTarget> element contains information identifying a field in the LFB that is to be monitored for events.

The <eventTarget> element contains one or more <eventField>s each of which MAY be followed by one or more <eventSubscript> elements. Each of these two elements represents the textual equivalent of a path select component of the LFB.

The <eventField> element contains the name of a component in the LFB or a component nested in an array or structure within the LFB. The name used in <eventField> MUST identify a valid component within the containing LFB context. The first element in an <eventTarget> MUST be an <eventField> element. In the example shown, four LFB components foo, goo, bar, and gah are used as <eventField>s.

In the simple case, an <eventField> identifies an atomic component. This is the case illustrated in the event named Foochanged. <eventField> is also used to address complex components such as arrays or structures.

The first defined event, Foochanged, demonstrates how a scalar LFB component, foo, could be monitored to trigger an event.

The second event, Goof1changed, demonstrates how a member of the complex structure goo could be monitored to trigger an event.

The events named NewbarEntry, Gah11changed, and Gah10field1 represent monitoring of arrays bar and gah in differing details.

If an <eventField> identifies a complex component, then a further <eventField> MAY be used to refine the path to the target element. Defined event Goof1changed demonstrates how a second <eventField> is used to point to member f1 of the structure goo.

If an <eventField> identifies an array, then the following rules apply:

- <eventSubscript> elements MUST be present as the next XML element after an <eventField> that identifies an array component. <eventSubscript> MUST NOT occur other than after an array reference, as it is only meaningful in that context.

- An <eventSubscript> contains either:

  * A numeric value to indicate that the event applies to a specific entry (by index) of the array. As an example, event Gah11changed shows how table gah's index 11 is being targeted for monitoring.

Or

  * It is expected that the more common usage is to have the event being defined across all elements of the array (i.e., a wildcard for all indices). In that case, the value of the <eventSubscript> MUST be a name rather than a numeric value. That same name can then be used as the value of <eventSubscript> in <eventReport> elements as described below. An example of a wild card table index is shown in event NewBarentry where the <eventSubscript> value is named _barIndex_

- An <eventField> MAY follow an <eventSubscript> to further refine the path to the target element. (Note: this is in the same spirit as the case where <eventField> is used to further refine <eventField> in the earlier example of a complex structure example of Goof1changed.) The example event Gah10field1 illustrates how the column field1 of table gah is monitored for changes.

It should be emphasized that the name in an <eventSubscript> element in defined event NewbarEntry is not a component name. It is a variable name for use in the <eventReport> elements (described in Section 4.7.6.3) of the given LFB definition. This name MUST be distinct from any component name that can validly occur in the <eventReport> clause.

4.7.6.2. <eventCondition> Element

The event 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. The event example NewbarEntry demonstrates the <eventCreated/> condition.

<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 component changes in any way. For binary components 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. Event examples Foochanged, Gah11changed, and Gah10field1 illustrate the <eventChanged/> condition.

<eventGreaterThan/>: The event is generated whenever the target component becomes greater than the threshold. The threshold is an event property.

<eventLessThan/>: The event is generated whenever the target component becomes less than the threshold. The threshold is an event property.

4.7.6.3. <eventReports> Element

The <eventReports> element of an <event> declares the information to be delivered by the FE along with the notification of the occurrence of the event.

The <eventReports> element contains one or more <eventReport> elements. Each <eventReport> element identifies a piece of data from the LFB class to be reported. The notification carries that data as if the collection of <eventReport> elements had been defined in a structure. The syntax is exactly the same as used in the <eventTarget> element, using <eventField> and <eventSubscript> elements, and so the same rules apply. Each <eventReport> element thus MUST identify a component in the LFB class. <eventSubcript> MAY

contain integers. If they contain names, they MUST be names from <eventSubscript> elements of the <eventTarget> in the event. 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 component causing the event, or to reference related information in parallel tables.

In the example shown, in the case of the event Foochanged, the report will carry the value of foo. In the case of the defined event NewbarEntry acting on LFB component bar, which is an array, there are two items that are reported as indicated by the two <eventReport> declarations:

- The first <eventReport> details what new entry was added in the table bar. Recall that _barIndex_ is declared as the event's <eventTarget> <eventSubcript> and that by virtue of using a name instead of a numeric value, the <eventSubcript> is implied to be a wildcard and will carry whatever index of the new entry.

- The second <eventReport> includes the value of LFB component foo at the time the new entry was created in bar. Reporting foo in this case is provided to demonstrate the flexibility of event reporting.

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 components referenced by the eventReport are optional, then the report MUST use a protocol format that supports optional elements and allows for the non-existence of such elements. Any components that do not exist are not reported.

4.7.6.4. Runtime Control of Events

The high-level view of the declaration and operation of LFB events is described in Section 3.2.5.

The <eventTarget> provides additional components used in the path to reference the event. The path constitutes the baseID for events, followed by the ID for the specific event, followed by a value for each <eventSubscript> element if it exists in the <eventTarget>.

The event path will uniquely identify a specific occurrence of the event in the event notification to the CE. In the example provided above, at the end of Section 4.7.6, a notification with path of 7.7 uniquely identifies the event to be that caused by the change of foo; an event with path 7.9.100 uniquely identifies the event to be that caused by a creation of table bar entry with index/subscript 100.

As described in Section 4.8.5, event elements 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 be either read-only or not supported.

In addition to identifying the event sources, the CE also uses the event path to activate runtime control of the event via the event properties (defined in Section 4.8.5) utilizing SET-PROP as defined in the ForCES protocol [RFC5810] operation.

To activate event generation on the FE, a SET-PROP message referencing the event and registration property of the event is issued to the FE by the CE with any prefix of the path of the event. So, for an event defined on the example table bar, a SET-PROP with a path of 7.9 will subscribe the CE to all occurrences of that event on any entry of the table. This is particularly useful for the <eventCreated/> and <eventDestroyed/> conditions on tables. 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 entry in an array, so it will use a shorter path.

In the example provided, subscribing to 7.8 implies receiving all declared events from table bar. Subscribing to 7.8.100 implies receiving an event when subscript/index 100 table entry is created.

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.

4.7.7. <description> Element for LFB Operational Specification

The <description> element of the <LFBClass> provides unstructured text (in XML sense) to explain what the LFB does to a human user.

4.8. Properties

Components of LFBs have properties that are important to the CE. The most important property is the existence / readability / writeability of the element. Depending on the type of the component, 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 components, there are subclasses of property information providing additional fields. This information is accessed by the CE (and updated where applicable) via the ForCES 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 dataTypeDef for accessibility, is below. Note that this access permission information is generally read-only.

```
<dataTypeDef>
  <name>accessPermissionValues</name>
  <synopsis>
    The possible values of component 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 to the component is read only
        </synopsis>
      </specialValue>
      <specialValue value="2">
        <name>Write-Only</name>
        <synopsis>
          The component MAY be written, but not read
        </synopsis>
      </specialValue>
      <specialValue value="3">
        <name>Read-Write</name>
        <synopsis>
          The component MAY be read or written
        </synopsis>
      </specialValue>
    </specialValues>
  </atomic>
</dataTypeDef>
<dataTypeDef>
  <name>baseElementProperties</name>
  <synopsis>basic properties, accessibility</synopsis>
  <struct>
    <component componentID="1">
      <name>accessibility</name>
      <synopsis>
          does the component exist, and
          can it be read or written
      </synopsis>
      <typeRef>accessPermissionValues</typeRef>
    </component>
  </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>
  <synopsis>Array Element Properties definition</synopsis>
  <struct>
    <derivedFrom>baseElementProperties</derivedFrom>
    <component componentID="2">
      <name>entryCount</name>
      <synopsis>the number of entries in the array</synopsis>
      <typeRef>uint32</typeRef>
    </component>
    <component componentID="3">
      <name>highestUsedSubscript</name>
      <synopsis>the last used subscript in the array</synopsis>
      <typeRef>uint32</typeRef>
    </component>
    <component componentID="4">
      <name>firstUnusedSubscript</name>
      <synopsis>
        The subscript of the first unused array element
      </synopsis>
      <typeRef>uint32</typeRef>
    </component>
  </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>
            <synopsis>string Element Properties definition </synopsis>
            <struct>
              <derivedFrom>baseElementProperties</derivedFrom>
              <component componentID="2">
                <name>stringLength</name>
                <synopsis>the number of octets in the string</synopsis>
                <typeRef>uint32</typeRef>
              </component>
              <component componentID="3">
                <name>maxStringLength</name>
                <synopsis>
                  the maximum number of octets in the string
                  </synopsis>
                <typeRef>uint32</typeRef>
              </component>
            </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>
             <synopsis>octetstring Element Properties definition
             </synopsis>
             <struct>
               <derivedFrom>baseElementProperties</derivedFrom>
               <component componentID="2">
                 <name>octetstringLength</name>
                 <synopsis>
                   the number of octets in the octetstring
                 </synopsis>
                 <typeRef>uint32</typeRef>
               </component>
               <component componentID="3">
                 <name>maxOctetstringLength</name>
                 <synopsis>
                   the maximum number of octets in the octetstring
                 </synopsis>
                 <typeRef>uint32</typeRef>
               </component>
             </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 below (Section 4.8.5.2). 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 that do not support setting the threshold for events will make this field read-only.

```
<dataTypeDef>
  <name>eventElementProperties</name>
  <synopsis>event Element Properties definition</synopsis>
  <struct>
    <derivedFrom>baseElementProperties</derivedFrom>
    <component componentID="2">
      <name>registration</name>
      <synopsis>
        has the CE registered to be notified of this event
      </synopsis>
      <typeRef>uint32</typeRef>
    </component>
    <component componentID="3">
      <name>threshold</name>
      <synopsis> comparison value for level crossing events
      </synopsis>
      <optional/>
      <typeRef>uint32</typeRef>
    </component>
    <component componentID="4">
      <name>eventHysteresis</name>
      <synopsis> region to suppress event recurrence notices
      </synopsis>
      <optional/>
      <typeRef>uint32</typeRef>
    </component>
    <component componentID="5">
      <name>eventCount</name>
      <synopsis> number of occurrences to suppress
      </synopsis>
      <optional/>
      <typeRef>uint32</typeRef>
    </component>
    <component componentID="6">
      <name>eventInterval</name>
      <synopsis> time interval in ms between notifications
      </synopsis>
      <optional/>
      <typeRef>uint32</typeRef>
    </component>
  </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 that 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 that 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 an <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 an <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 an <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 or not a notification is generated, the counter is incremented. If the counter exceeds the configured value, it is set to 0.

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 that 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 component to which the subject alias refers.

```
<dataTypeDef>
  <name>aliasElementProperties</name>
  <synopsis>alias Element Properties definition</synopsis>
  <struct>
    <derivedFrom>baseElementProperties</derivedFrom>
    <component componentID="2">
      <name>targetLFBClass</name>
      <synopsis>the class ID of the alias target</synopsis>
      <typeRef>uint32</typeRef>
    </component>
    <component componentID="3">
      <name>targetLFBInstance</name>
      <synopsis>the instance ID of the alias target</synopsis>
      <typeRef>uint32</typeRef>
    </component>
    <component componentID="4">
      <name>targetComponentPath</name>
      <synopsis>
        the path to the component target
        each 4 octets is read as one path element,
        using the path construction in the ForCES protocol,
        [2].
      </synopsis>
      <typeRef>octetstring[128]</typeRef>
    </component>
  </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="urn:ietf:params:xml:ns:forces:lfbmodel:1.0"
 xmlns:lfb="urn:ietf:params:xml:ns:forces:lfbmodel:1.0"
 targetNamespace="urn:ietf:params:xml:ns:forces:lfbmodel:1.0"
 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],
           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="component" 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="componentID" use="required"
                            type="xsd:unsignedInt"/>
           </xsd:complexType>
           <!-- key declaration to make componentIDs unique in a struct
```

```
            -->
            <xsd:key name="structComponentID">
              <xsd:selector xpath="lfb:component"/>
              <xsd:field xpath="@componentID"/>
            </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="components" type="LFBComponentsType"
                             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:unsignedInt"/>
          </xsd:complexType>
          <!-- Key constraint to ensure unique attribute names within
               a class:
          -->
          <xsd:key name="components">
            <xsd:selector xpath="lfb:components/lfb:component"/>
            <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>
          <xsd:key name="componentIDs">
            <xsd:selector xpath="lfb:components/lfb:component"/>
            <xsd:field xpath="@componentID"/>
          </xsd:key>
          <xsd:key name="capabilityIDs">
            <xsd:selector xpath="lfb:capabilities/lfb:capability"/>
            <xsd:field xpath="@componentID"/>
          </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="xsd:boolean" use="optional"
                   default="0"/>
  </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="xsd:boolean" use="optional"
                     default="0"/>
    </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="LFBComponentsType">
      <xsd:sequence>
        <xsd:element name="component" 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="componentID" use="required"
                           type="xsd:unsignedInt"/>
          </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="componentID" 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 Components 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 information that can be used in understanding and controlling the forwarding operations. Some of this information will be read-only, while others parts may also be writeable.

In order to make the FE information easily accessible, the information is represented in an LFB. This LFB has a class, FEObject. The LFBClassID for this class is 1. Only one instance of this class will ever be present in an FE, and the instance ID of that instance in the protocol is 1. Thus, by referencing the components of class:1, instance:1 a CE can get the general information about the FE. The FEObject 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 [RFC5810] document.

5.1. XML for FEObject Class Definition

```
<?xml version="1.0" encoding="UTF-8"?>
<LFBLibrary xmlns="urn:ietf:params:xml:ns:forces:lfbmodel:1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  provides="FEObject">
  <dataTypeDefs>
    <dataTypeDef>
      <name>LFBAdjacencyLimitType</name>
      <synopsis>Describing the Adjacent LFB</synopsis>
      <struct>
        <component componentID="1">
          <name>NeighborLFB</name>
          <synopsis>ID for that LFB class</synopsis>
          <typeRef>uint32</typeRef>
        </component>
        <component componentID="2">
          <name>ViaPorts</name>
```

```
            <synopsis>
              the ports on which we can connect
            </synopsis>
            <array type="variable-size">
              <typeRef>string</typeRef>
            </array>
          </component>
        </struct>
      </dataTypeDef>
      <dataTypeDef>
        <name>PortGroupLimitType</name>
        <synopsis>
          Limits on the number of ports in a given group
        </synopsis>
        <struct>
          <component componentID="1">
            <name>PortGroupName</name>
            <synopsis>Group Name</synopsis>
            <typeRef>string</typeRef>
          </component>
          <component componentID="2">
            <name>MinPortCount</name>
            <synopsis>Minimum Port Count</synopsis>
            <optional/>
            <typeRef>uint32</typeRef>
          </component>
          <component componentID="3">
            <name>MaxPortCount</name>
            <synopsis>Max Port Count</synopsis>
            <optional/>
            <typeRef>uint32</typeRef>
          </component>
        </struct>
      </dataTypeDef>
      <dataTypeDef>
        <name>SupportedLFBType</name>
        <synopsis>table entry for supported LFB</synopsis>
        <struct>
          <component componentID="1">
            <name>LFBName</name>
            <synopsis>
              The name of a supported LFB class
            </synopsis>
            <typeRef>string</typeRef>
          </component>
          <component componentID="2">
            <name>LFBClassID</name>
            <synopsis>the id of a supported LFB class</synopsis>
```

```
                <typeRef>uint32</typeRef>
              </component>
              <component componentID="3">
                <name>LFBVersion</name>
                <synopsis>
                  The version of the LFB class used
                  by this FE.
                </synopsis>
                <typeRef>string</typeRef>
              </component>
              <component componentID="4">
                <name>LFBOccurrenceLimit</name>
                <synopsis>
                  the upper limit of instances of LFBs of this class
                </synopsis>
                <optional/>
                <typeRef>uint32</typeRef>
              </component>
              <!-- For each port group, how many ports can exist
              -->
              <component componentID="5">
                <name>PortGroupLimits</name>
                <synopsis>Table of Port Group Limits</synopsis>
                <optional/>
                <array type="variable-size">
                  <typeRef>PortGroupLimitType</typeRef>
                </array>
              </component>
      <!-- for the named LFB Class, the LFB Classes it may follow -->
              <component componentID="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>
              </component>
      <!-- for the named LFB Class, the LFB Classes that may follow it
        -->
              <component componentID="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>
                </component>
                <component componentID="8">
                  <name>UseableParentLFBClasses</name>
                  <synopsis>
                    List of LFB classes from which this class has
                    inherited, and which the FE is willing to allow
                    for references to instances of this class.
                  </synopsis>
                  <optional/>
                  <array type="variable-size">
                    <typeRef>uint32</typeRef>
                  </array>
                </component>
              </struct>
            </dataTypeDef>
            <dataTypeDef>
              <name>FEStateValues</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>
                <component componentID="1">
                  <name>NeighborID</name>
                  <synopsis>Neighbors FEID</synopsis>
                  <typeRef>uint32</typeRef>
```

```
        </component>
        <component componentID="2">
          <name>InterfaceToNeighbor</name>
          <synopsis>
            FE's interface that connects to this neighbor
          </synopsis>
          <optional/>
          <typeRef>string</typeRef>
        </component>
        <component componentID="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 FEs are adjacent on more than
            one interface.
          </synopsis>
          <optional/>
          <typeRef>string</typeRef>
        </component>
      </struct>
    </dataTypeDef>
    <dataTypeDef>
      <name>LFBSelectorType</name>
      <synopsis>
        Unique identification of an LFB class-instance
      </synopsis>
      <struct>
        <component componentID="1">
          <name>LFBClassID</name>
          <synopsis>LFB Class Identifier</synopsis>
          <typeRef>uint32</typeRef>
        </component>
        <component componentID="2">
          <name>LFBInstanceID</name>
          <synopsis>LFB Instance ID</synopsis>
          <typeRef>uint32</typeRef>
        </component>
      </struct>
    </dataTypeDef>
    <dataTypeDef>
      <name>LFBLinkType</name>
      <synopsis>
        Link between two LFB instances of topology
      </synopsis>
      <struct>
        <component componentID="1">
          <name>FromLFBID</name>
```

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

```
            instances
          </synopsis>
          <array type="variable-size">
            <typeRef>LFBSelectorType</typeRef>
          </array>
        </component>
        <component access="read-write" componentID="3">
          <name>FEName</name>
          <synopsis>name of this FE</synopsis>
          <typeRef>string[40]</typeRef>
        </component>
        <component access="read-write" componentID="4">
          <name>FEID</name>
          <synopsis>ID of this FE</synopsis>
          <typeRef>uint32</typeRef>
        </component>
        <component access="read-only" componentID="5">
          <name>FEVendor</name>
          <synopsis>vendor of this FE</synopsis>
          <typeRef>string[40]</typeRef>
        </component>
        <component access="read-only" componentID="6">
          <name>FEModel</name>
          <synopsis>model of this FE</synopsis>
          <typeRef>string[40]</typeRef>
        </component>
        <component access="read-only" componentID="7">
          <name>FEState</name>
          <synopsis>State of this FE</synopsis>
          <typeRef>FEStateValues</typeRef>
        </component>
        <component access="read-write" componentID="8">
          <name>FENeighbors</name>
          <synopsis>table of known neighbors</synopsis>
          <optional/>
          <array type="variable-size">
            <typeRef>FEConfiguredNeighborType</typeRef>
          </array>
        </component>
      </components>
      <capabilities>
        <capability componentID="30">
          <name>ModifiableLFBTopology</name>
          <synopsis>
            Whether Modifiable LFB is supported
          </synopsis>
          <optional/>
          <typeRef>boolean</typeRef>
```

```
                </capability>
                <capability componentID="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 components of the FEObject LFB are supported is accessed by the properties information for those components.

5.2.1. ModifiableLFBTopology

This component has a boolean value that indicates whether the LFB topology of the FE may be changed by the CE. If the component is absent, the default value is assumed to be true, and the CE presumes that 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 SupportedLFBs element 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 component, is an array that contains the information about each supported LFB class. The array structure type is defined as the SupportedLFBType dataTypeDef.

Each entry in the SupportedLFBs array 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. This capability information on the acceptable ordering and connection of LFBs MAY be omitted if the implementor concludes that the actual constraints are such that the information would be misleading for the CE.

5.2.2.1. LFBName

This component has as its value the name of the LFB class being described.

5.2.2.2. LFBClassID

LFBClassID is 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

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

5.2.2.6. CanOccurAfters and LFBAdjacencyLimitType

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

The array entries describe a permissible positioning of the described LFB class, referred to here as the SupportedLFB. Specifically, each array entry 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 component (of the LFBAdjacencyLimitType dataTypeDef) of the CanOccurAfters array entry. If this neighbor can only be connected to a specific set of input port groups, then the viaPort component is included. This component is an array, with one entry for each input port group of the SupportedLFB that can be connected to an output port of the NeighborLFB.

(For example, within a SupportedLFBs entry, each array entry of the CanOccurAfters array must have a unique NeighborLFB, and within each such array entry each viaPort must represent a distinct and valid input port group of the SupportedLFB. The LFB class definition schema does not include these uniqueness constraints.)

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 entries.

The array entries list those LFB classes that the SupportedLFB may precede in the topology. In this component, the entries in the viaPort component of the array value represent the output port groups of the SupportedLFB that may be connected to the NeighborLFB. As with CanOccurAfters, viaPort may have multiple entries if multiple output ports may legitimately connect to the given NeighborLFB class.

(And a similar set of uniqueness constraints applies to the CanOccurBefore clauses, even though an LFB may occur both in CanOccurAfter and CanOccurBefore.)

5.2.2.8. UseableParentLFBClasses

The UseableParentLFBClasses array, if present, is used to hold a list of parent LFB class IDs. All the entries in the list must be IDs of classes from which the SupportedLFB class being described has

inherited (either directly or through an intermediate parent.) (If an FE includes improper values in this list, improper manipulations by the CE are likely, and operational failures are likely.) In addition, the FE, by including a given class in the last, is indicating to the CE that a given parent class may be used to manipulate an instance of this supported LFB class.

By allowing such substitution, the FE allows for the case where an instantiated LFB may be of a class not known to the CE, but could still be manipulated. While it is hoped that such situations are rare, it is desirable for this to be supported. This can occur if an FE locally defines certain LFB instances, or if an earlier CE had configured some LFB instances. It can also occur if the FE would prefer to instantiate a more recent, more specific and suitable LFB class rather than a common parent.

In order to permit this, the FE MUST be more restrained in assigning LFB instance IDs. Normally, instance IDs are qualified by the LFB class. However, if two LFB classes share a parent, and if that parent is listed in the UseableParentLFBClasses for both specific LFB classes, then all the instances of both (or any, if multiple classes are listing the common parent) MUST use distinct instances. This permits the FE to determine which LFB instance is intended by CE manipulation operations even when a parent class is used.

5.2.2.9. 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 be 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 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. FE Components

The <components> element is included if the class definition contains the definition of the components of the FE object that are not considered "capabilities". Some of these components are writeable and some are read-only, which is determinable by examining the property information of the components.

5.3.1. FEState

This component carries the overall state of the FE. The possible values are the strings AdminDisable, OperDisable, and OperEnable. The starting state is OperDisable, and the transition to OperEnable is controlled by the FE. The CE controls the transition from OperEnable to/from AdminDisable. For details, refer to the ForCES protocol document [RFC5810].

5.3.2. LFBSelectors and LFBSelectorType

The LFBSelectors component 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 dataTypeDef.

Each entry in the array describes a single LFB instance in the FE. The array entry 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 component contains information about each inter-LFB link inside the FE, where each link is described in an LFBLinkType dataTypeDef. The LFBLinkType component contains sufficient information to identify precisely the end points of a link. The FromLFBID and ToLFBID components 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 components select the entries 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 component 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 dataTypeDef.

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. This component is optional.

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.

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 the FE Model

This section describes how the proposed FE model meets the requirements outlined in Section 5 of RFC 3654 [RFC3654]. The requirements can be separated into general requirements (Section 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 Functional 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 components. These components express the capability of the FE and support flexible implementations by allowing an FE to specify which optional features are supported. The components also indicate whether they are configurable by the CE for an LFB class. Configurable components 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 component settings. By querying the settings of the components 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 components 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 components. The FE components 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 [RFC5810]:

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 components,

6. Manipulation of LFB components, and

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 [RFC5810] document mandates a specific mechanism. The LFB class definition does include the capability for an FE to be configured with, and to provide 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 is visualized in Figure 12:

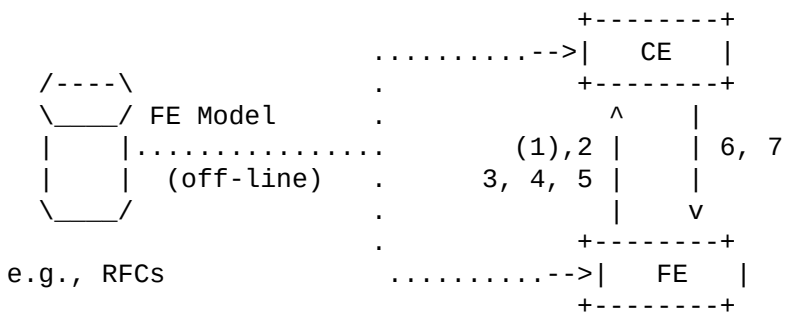


Figure 12: Relationship between the FE model and the ForCES protocol messages, where (1) is part of the ForCES base protocol, and the rest are defined by the FE model.

The actual encoding of these messages is defined by the ForCES protocol [RFC5810] document and is 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 components of the LFBs, in which case such components 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).

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 13 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 13. 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 [RFC5810] supports FE topology query from FEs, it is optional for the CE to use it, assuming that the CE has other means to gather such topology information.

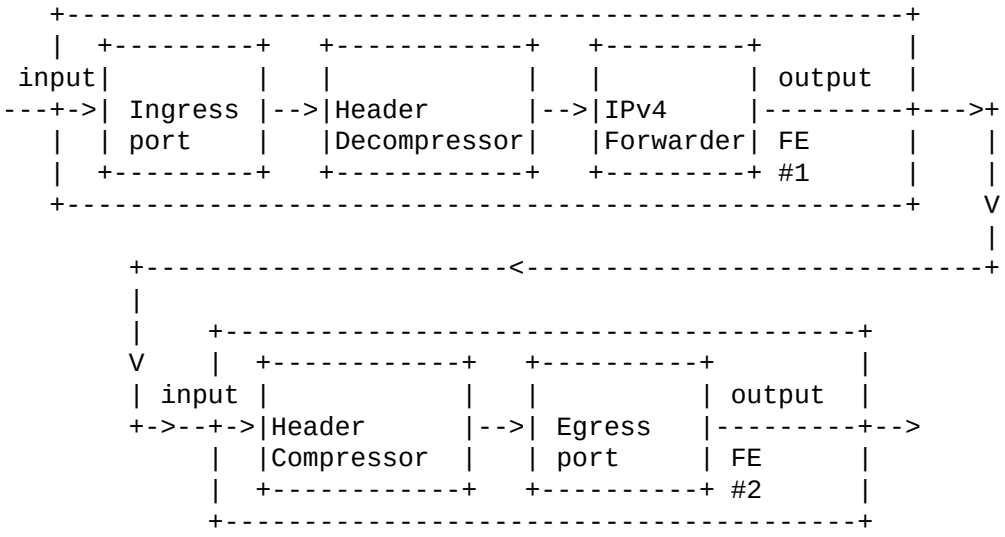


Figure 13: An example of two FEs connected together.

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 are the following:

- 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 7.2.

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 whether or not an optional feature is supported by a given LFB instance. 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 components 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 components of the LFB. The actual values of these components may, therefore, be obtained using the same component query mechanisms as used for other LFB components.

Capability components are read-only arguments. In cases where some implementations may allow CE modification of the value, the information must be represented as an operational component, not a capability component.

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

Much of this restrictive information is captured by the component property information, and so can be accessed uniformly for all information within the model.

7.5. State Query of LFB Components

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 components:

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

7.6. LFB Component Manipulation

The FE model provides for the definition of LFB classes. Each class has a globally unique identifier. Information within the class is represented as components and assigned identifiers within the scope of that class. This model also specifies that instances of LFB classes have identifiers. The combination of class identifiers, instance identifiers, and component identifiers is used by the protocol to reference the LFB information in the protocol operations.

7.7. LFB Topology Reconfiguration

Operations that will be needed to reconfigure LFB topology are as follows:

- 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 LFB Definition

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 plane LFB might be build. This example is a fictional case of an interface supporting a coarse WDM optical interface that carries frame relay traffic. The statistical information (including error statistics) is omitted.

Later portions of this example include references to protocol operations. The operations described are operations the protocol needs to support. The exact format and fields are purely informational here, as the ForCES protocol [RFC5810] document defines the precise syntax and semantics of its operations.

```
<?xml version="1.0" encoding="UTF-8"?>
      <LFBLibrary xmlns="urn:ietf:params:xml:ns:forces:lfbmodel:1.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       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>
              <component componentID="1">
                <name>LaserFrequency</name>
                <synopsis>encoded frequency(channel)</synopsis>
                <typeRef>uint32</typeRef>
              </component>
              <component componentID="2">
```

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

```
            </component>
          </struct>
        </dataTypeDef>
        <dataTypeDef>
          <name>PortStatusValues</name>
          <synopsis>
              The possible values of status.  Used for both
              administrative and operational status
          </synopsis>
          <atomic>
            <baseType>uchar</baseType>
            <specialValues>
              <specialValue value="0">
                <name>Disabled </name>
                <synopsis>the component is disabled</synopsis>
              </specialValue>
              <specialValue value="1">
                <name>Enabled</name>
                <synopsis>FE is operatively enabled</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>
          <!-- dummy classid, but needs to be a valid value -->
        <LFBClassDef LFBClassID="255">
          <name>FrameLaserLFB</name>
          <synopsis>Fictional LFB for Demonstrations</synopsis>
          <version>1.0</version>
          <inputPorts>
            <inputPort group="true">
              <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="true">
                <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="true">
                <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>
    <components>
      <component access="read-write" componentID="1">
        <name>AdminPortState</name>
        <synopsis>is this port allowed to function</synopsis>
        <typeRef>PortStatusValues</typeRef>
      </component>
      <component access="read-write" componentID="2">
        <name>FrequencyInformation</name>
        <synopsis>
            table of information per CWDM frequency
        </synopsis>
        <array type="variable-size">
          <typeRef>frequencyInformationType</typeRef>
        </array>
      </component>
    </components>
    <capabilities>
      <capability componentID="31">
        <name>OperationalState</name>
        <synopsis>
            whether the port over all is operational
        </synopsis>
        <typeRef>PortStatusValues</typeRef>
      </capability>
      <capability componentID="32">
        <name>MaximumFrequencies</name>
        <synopsis>
            how many laser frequencies are there
        </synopsis>
        <typeRef>uint16</typeRef>
      </capability>
      <capability componentID="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. 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. The following paragraphs describe a potential operational device and how it might use this LFB definition.

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 to 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 metadata, 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 metadata 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 metadata. 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 the LFB port, in which case no metadata 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.

8.1.1. 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
  T = PATH-DATA
    Path: flags = none, length = 4, path = 2, channel, 4, entryIdx
    DataRaw: DLCI, Enabled(1), false, out-idx
```

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

The response would confirm the creation of the specified entry. 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 LFB's 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.

8.1.2. 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 component to count the error, and a component in the LFB to hold the required portion of the packet. The component 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.

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 metadata, and ship the packet to the CE.

Other aspects of error handling are discussed under events below.

8.2. LFB Components

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

The other component 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-PROP on path 2.channel.4 will give the CE the properties of that

FrameRelayCircuits 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 in which the CE may be interested. 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-PROP
  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 14 and 15, 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 hysteresis property on that event. The request would look like:

```
T = SET-PROP
  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 suppresses 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. One 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 hysteresis level of 1 will suppress this sort of condition. Many other events have oscillations that are somewhat wider, so larger hysteresis settings can be used with those.

9. IANA Considerations

The ForCES model creates the need for a unique XML namespace for ForCES library definition usage, and unique class names and numeric class identifiers.

9.1. URN Namespace Registration

IANA has registered a new XML namespace, as per the guidelines in RFC 3688 [RFC3688].

URI: The URI for this namespace is
urn:ietf:params:xml:ns:forces:lfbmodel:1.0

Registrant Contact: IESG

XML: none, this is an XML namespace

9.2. LFB Class Names and LFB Class Identifiers

In order to have well defined ForCES LFB Classes, and well defined identifiers for those classes, IANA has created a registry of LFB class names, corresponding class identifiers, and the document that defines the LFB class. The registry policy is simply first come,

first served (FCFS) with regard to LFB class names. With regard to LFB class identifiers, identifiers less than 65536 are reserved for assignment by IETF Standards-Track RFCs. Identifiers equal to or above 65536, in the 32-bit class ID space, are available for assignment on a first come, first served basis. All registry entries must be documented in a stable, publicly available form.

Since this registry provides for FCFS allocation of a portion of the class identifier space, it is necessary to define rules for naming classes that are using that space. As these can be defined by anyone, the needed rule is to keep the FCFS class names from colliding with IETF-defined class names. Therefore, all FCFS class names MUST start with the string "Ext-".

Table 1 tabulates the above information.

IANA has created a registry of ForCES LFB Class Names and the corresponding ForCES LFB Class Identifiers, with the location of the definition of the ForCES LFB Class, in accordance with the rules in the following table.

LFB Class Name	LFB Class Identifier	Place Defined	Description
Reserved	0	RFC 5812	Reserved
FE Object	1	RFC 5812	Defines ForCES Forwarding Element information
FE Protocol Object	2	[2]	Defines parameters for the ForCES protocol operation
IETF defined LFBs	3-65535	Standards Track RFCs	Reserved for IETF defined RFCs
ForCES LFB Class names beginning EXT-	>65535	Any Publicly Available Document	First Come, First Served for any use

Table 1

10. Authors Emeritus

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

Ellen Delganes, Intel Corp.
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. Particular thanks to Evangelos Haleplidis for help getting the XML right.

12. Security Considerations

The FE model describes the representation and organization of data sets and components in the FEs. The ForCES framework document [RFC3746] 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 is defined in separate documents, and thus the security issues will be addressed there.

13. References

13.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC5810] Doria, A., Ed., Hadi Salim, J., Ed., Haas, R., Ed., Khosravi, H., Ed., Wang, W., Ed., Dong, L., Gopal, R., and J. Halpern, "Forwarding and Control Element Separation (ForCES) Protocol Specification", RFC 5810, March 2010.

[RFC3688] Mealling, M., "The IETF XML Registry", BCP 81, RFC 3688, January 2004.

[Schema1] Thompson, H., Beech, D., Maloney, M., and N. Mendelsohn, "XML Schema Part 1: Structures", W3C REC-xmlschema-1, http://www.w3.org/TR/xmlshcema-1/, May 2001.

[Schema2] Biron, P. and A. Malhotra, "XML Schema Part 2: Datatypes", W3C REC-xmlschema-2, http://www.w3.org/TR/xmlschema-2/, May 2001.

13.2. Informative References

[RFC3654] Khosravi, H. and T. Anderson, "Requirements for Separation of IP Control and Forwarding", RFC 3654, November 2003.

[RFC3746] Yang, L., Dantu, R., Anderson, T., and R. Gopal, "Forwarding and Control Element Separation (ForCES) Framework", RFC 3746, April 2004.

[RFC3317] Chan, K., Sahita, R., Hahn, S., and K. McCloghrie, "Differentiated Services Quality of Service Policy Information Base", RFC 3317, March 2003.

[RFC3318] Sahita, R., Hahn, S., Chan, K., and K. McCloghrie, "Framework Policy Information Base", RFC 3318, March 2003.

[RFC3444] Pras, A. and J. Schoenwaelder, "On the Difference between Information Models and Data Models", RFC 3444, January 2003.

[RFC3470] Hollenbeck, S., Rose, M., and L. Masinter, "Guidelines for the Use of Extensible Markup Language (XML) within IETF Protocols", BCP 70, RFC 3470, January 2003.

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

Authors' Addresses

Joel Halpern
Self
P.O. Box 6049
Leesburg, VA 20178
USA

Phone: +1 703 371 3043
EMail: jmh@joelhalpern.com

Jamal Hadi Salim
Znyx Networks
Ottawa, Ontario
Canada

EMail: hadi@mojatatu.com