January 2020

AUTOMATIC LINECARD (LC) CAPABILITY DETECTION

Stefano Binetti

Richard Moses S

Davide Sirtori

AUTOMATIC LINECARD (LC) CAPABILITY DETECTION

AUTHORS:

Stefano Binetti
Richard Moses S
Davide Sirtori

ABSTRACT

Presented herein are techniques to reduce software development code and allow software to automatically detect and adapt the code to the capabilities of a linecard (LC).

DETAILED DESCRIPTION

In the previous generations of software implementation for each new linecard (LC) capability/functionality introduced, the associated RP PD code EA/MA had to be rewritten/adapted based on the new capability of the LC. For example, if an LC has a 4x100GE port or 40x10GE, the complete software layering needs to be rewritten and adapted to the new LC Muxponder functionality.

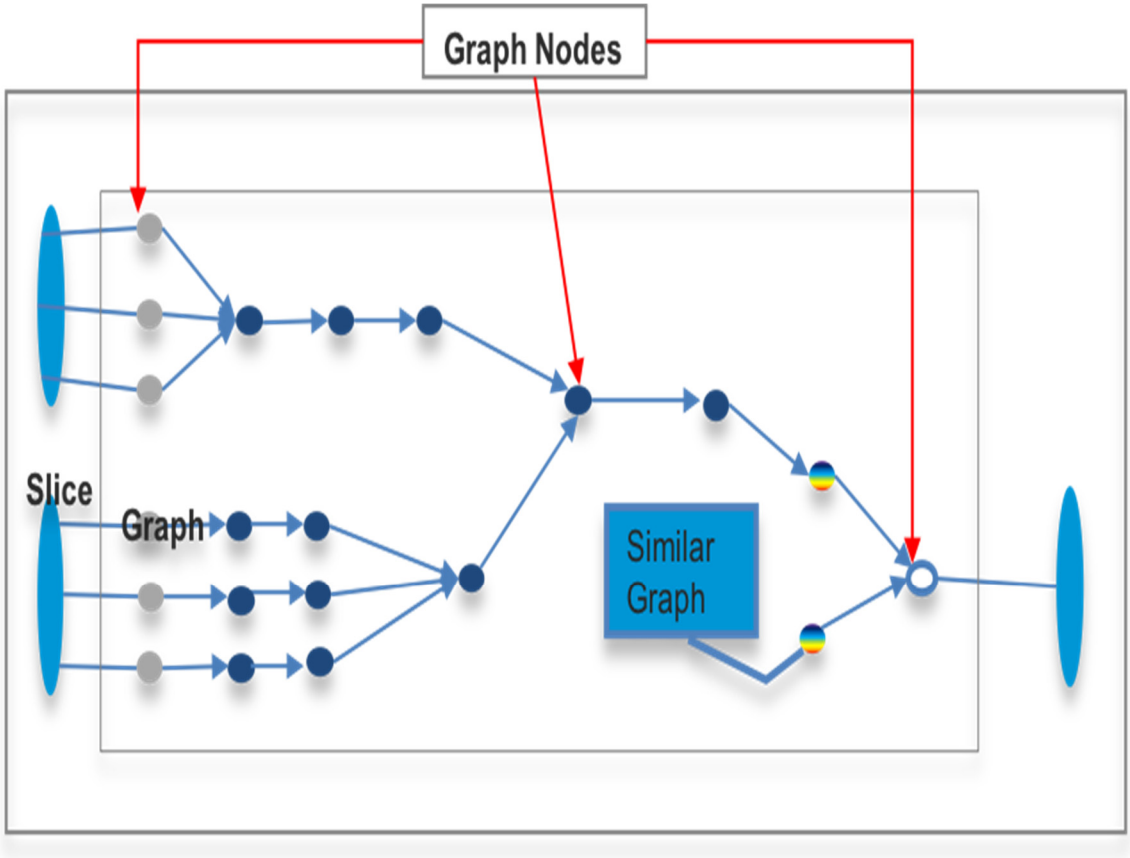
Given the limitations of conventional arrangements, there is a need for an operating system (OS) agnostic abstract representation (abstract model) of Cross Ponder / Mux Ponder / Transponder functional modules, as well as a need to support different methods of configuration, such as “Static Muxponder,” “Mixed Service Muxponder,” “L1 Cross Connection,” etc. There is also a need for the abstract model to provide a capability learning method that embeds the modules port capability, layer capabilities, cross connect restrictions, bandwidth restrictions, etc., as well as for the abstract model to define the Static Structure of the hardware that enables simplification of recovery of the device programming in warm reload conditions. The abstract model should also allow percolation of high/upper level software entity mapping, such as interface name or handles, to the driver layer, to enable single index to stitch debug data from different layers. Such capabilities would require less effort to support new modules of the same functionality.

Accordingly, presented herein is a new software architecture based on the concept of "Graph and Node." More specifically, the traffic scheme of Muxponder / Cross-Ponder / Transponder modules is chosen to be represented as a graph. In addition, the Static Muxponder scheme is represented as Static graphs defining the traffic flow from client port to trunk port, where a unique graph is defined for each of the traffic modes. The Mixed Service Muxponder scheme is represented as graphs with different branches from client port to trunk port, where a unique graph is defined for each trunk type. The correct branch connecting a client to the trunk can be chosen based on the client mode.

In addition, the L1 cross connection scheme is represented as graphs with open links, where a unique graph is defined based on trunk type. The correct branch can again be chosen based on the client mode and the client to trunk connections can be chosen based on bandwidth and scope restrictions published in the graph node.

In addition, the graph contains a list of graph nodes that carry a unique static unsigned 32bit identifier. Since the identifier is static, it remains unchanged over warm reload. A graph node maps to a device and block within a device. The graph and graph nodes defines the construct to carry the traffic capabilities, port mapping, layer capabilities, resource restrictions. The initialization flow from high level software to the device driver layer carries the needed high level software identifiers via a "Name/handle" mapped to static identifiers of the device driver layer.

FIG. 1, below, illustrates an example Muxponder Abstraction Model (abstract model) in accordance with the techniques presented herein.



In FIG. 1, a "Node" is a configurable or/and monitorable entity in the data path. The node can be of different types. All the layers are represented by nodes (e.g., optical, ODU, OTU, Ethernet, DWDM-carrier, etc.). A node can have the property to operate on one of given modes. The "Graph" is the relationship of Nodes that represents the data path. In context of a Muxponder or transponder application, a graph defines the traffic flow from the client port to the trunk port. The "Client Port" represents the entry point of the Muxponder module. The Client Port can start with a Face Plate Optics Port or can be an electrical interface (CAUI4 or CAUI10, etc.) in an IPoDWDM solution. If the Client Port is optical, usually referred as a Grey Optics port, then there are some lower CWDM optics that are also used on the client side. The "Trunk Port" represents the Exit Port of the muxponder module. The output is one or more DWDM carriers. Finally, the "Slice" is an autonomous muxponding entity, which can be a group of clients and trunk ports.

FIG. 2, below, illustrates a 2x100GE to 200G trunk overlaying a module.

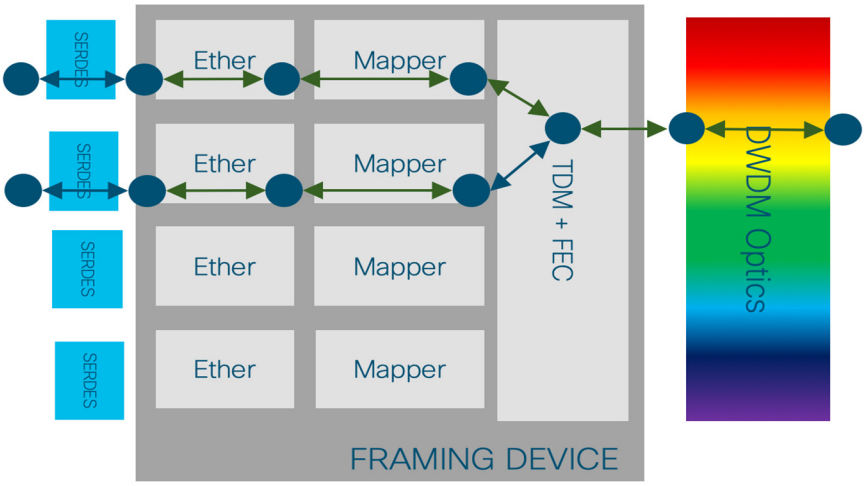


FIG. 3, below, illustrates a 4x100GE to 400G trunk overlaying a module.

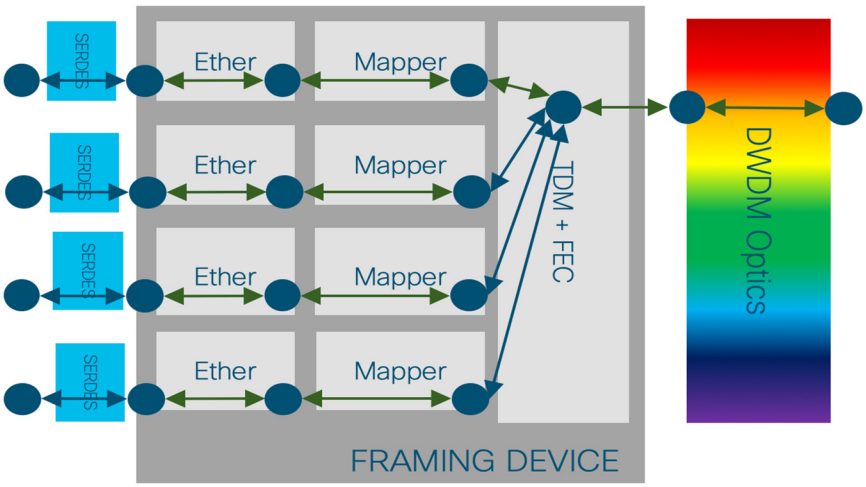


FIG. 4, below, illustrates an example 6x100 client to 2x300 trunk graph.

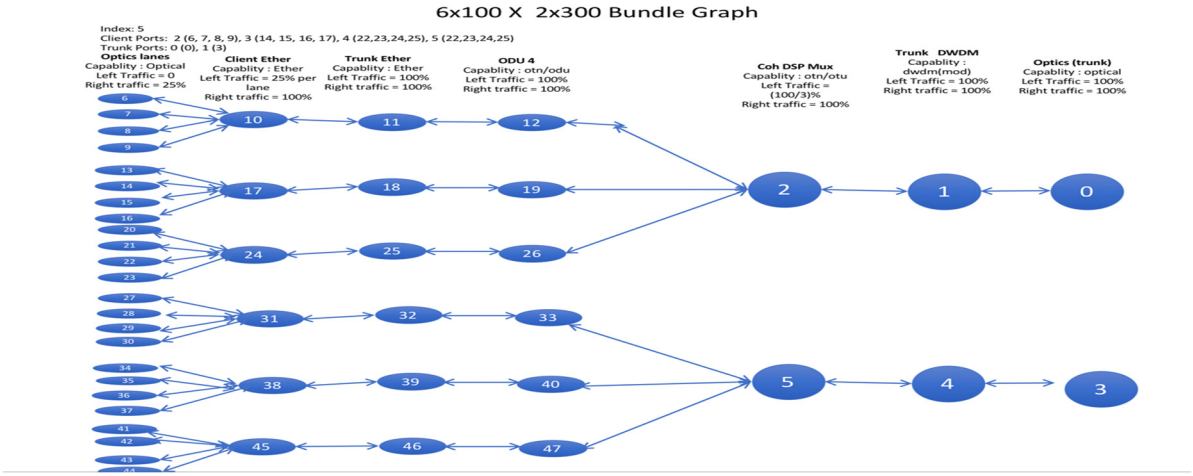
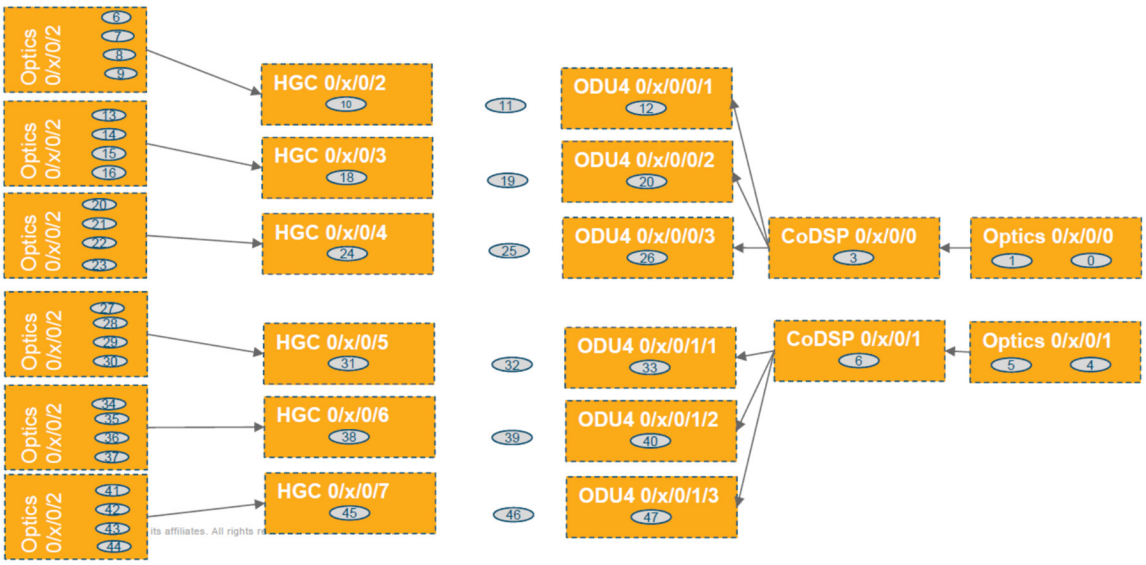


FIG. 5, below, illustrates an example IOS XR: CMA Graph to XR interface mapping.



As noted, presented herein is a new software architecture that is based on the concept of Graph and Node. Each Muxponder and Transponder instance is always represented by a Graph that represent the client to trunk relationship and a node that represent the technology specific capability such as Ethernet, OTN, Fiber Channel, etc. In addition, the adaptation function is represented as a Node to Node relationship. For example 100GE mapped over an ODU4 via GMP mapping is represented as 100GE -->

ODU4, where 100GE and ODU4 are 2 distinct node and the mapping is a graph relationship. This methodology is extremely flexible and allows the software representation of any Muxponder/Transponder functionality, while also allowing easy integration of the XR upper layer software entity. The upper layer software using this Graph/Node modeling approach can always retain the same increase in the speed at which a new hardware functionality will be supported by the platform. In essence, the techniques presented herein provide the capability for the lower layer software, which is closest to the hardware and has better knowledge of its capability, to the upper layer software in a programmatic and efficient way to minimize the upper layer software development.