ATLAS TDAQ/DCS DataCollection - Modelling

Description of the switch parameterized model

Document Version: 1.0
Document Date: 16.06.2002
Document Status: Draft

H. Niewodniczanski Institute of Nuclear Physics, Cracow: K. Korcyl

Table 1 Document Change Record

Title:	ATLAS TDA/DCS DataCollection - Modelling Description of the switch parameterized model		
ID:	ATLAS-TDAQ-2002-XXX		
Version	**Issue**	**Date**	**Comment**
1	1	16th June 2002	Initial version

1 Introduction

The document provides description of internal construction of the switch parameterized model. The model originally was platform independent, however after the OPNET support ceased and the Ptolemy became the only platform, a number of modifications, for performance improvement were introduced to provide better co-operation with the Ptolemy kernel.

1.1 Purpose of the document.

The document aims to provide explanation to an internal construction of the switch parameterized model. The explanation should be useful in case of new module developments needs to be added to existing framework.

1.2 Glossary, acronyms and abbreviation

1.2.1 Glossary

- Ptolemy star: - an atomic unit of computation in a Ptolemy application. Every Ptolemy simulation ultimately consists of executing the methods of the stars used to define the simulation.
- GigEMarp packet - special packet used during switch learning phase to allow switches to buld their routing tables
- Contaneir - an STL container
- VLAN tagged frames - an Ethernet frames with VLAN and priority tags

Acronyms and Abbreviations

Ptolemy – modeling platform from EECS Department of UC Berkeley

STL - Standard Template Library

VLAN - Virtual Local Area Network

MTU - Maximum Transfer Unit

OO – object oriented

1.3 References

1 Ptolemy http://ptolemy.eecs.berkeley.edu

2 Modeling large networks using parameterized switches - http://korcyl.home.cern.ch/korcyl/opnetwork2000/op2k_paper.pdf

2 The model functional description

Models of links and switches are necessary in modeling operation of computer networks in a real time. The link operation is simple and predictable: packet enters a link and the transfer time is related to the link's speed and the packet length. The two parameters are constant and can be used to calculate link transfer time in the link models.

The switch operation is far more complicated and related to a number of parameters. Currently there is a big range of switches on the market supplied by various manufacturers. Operation of a particular device depends on internal implementation; both hardware and software wise. This results in various characteristics for packets throughput and latencies. In our approach to build a parameterized model of a switch we tried to generalize operation of a switch and base it on a minimal number of measurable parameters. The parameters used in the model should allow to reproduce throughputs and latencies measured on a real device for various traffic patterns [2].

The OO technology has been chosen for the parameterized model implementation. The abstract base classes have been defined for various parts of the model. This allows using models of these parts with different policies for the traffic management and opens way for any new ideas the switch vendors may implement in the future.

The switch parameterized model internal organization reflects modular and hierarchical architecture of the contemporary switches. Modules house a number of input/output ports (at least one) and provide internal transfer medium for intra module communication (even for modules with shared memory, a reference to the packet data has to be moved from an input port to an output port). A number of modules (at least one) are housed in the switch chassis. The inter module transfers use a transfer medium in the chassis.

The operation of the parameterized model of a switch is based on 10 parameters [2]. The values of the parameters can be collected from a number of tests with a real switch. In addition to the parameters, the switch operation is related to the various traffic management policies the switch implements: bus or crossbar mechanism for backplane transfers, flow control, priority queues etc. The switch model can be seen as a set of resources necessary for packet transmission. Each packet arriving to the switch requests certain amount of resources (input buffer resource, transfer resource, output buffer resource) to get to the destination port. The number and type of resources depend on type of transfer: inter or intra module. During the transfer the packet keeps granted resources.

An arriving packet is buffered on input providing there is input buffering resource available in the module, otherwise the arriving packets are dropped. Depending on sources and destination addresses a type of transfer is defined: intra or inter module. Each module provides resources for intra module transfers and manages them internally. All inter module transfers have to go via a common backplane. When the type of transfer is known, a packet can request resources to get transferred to the destination port. If resources are available, the packet allocates resources and a transfer time is modeled. When the packet arrives to the destination module, the resources necessary for transfer are freed. The packet is queued for the output port. If the port is idle the packet starts leaving the switch. The link transfer time is modeled. When the last bit of a packet leaves the switch the output buffering resources are freed. If the destination port is busy a packet is queued. The output queue management operates the queue and decides which packet will go next.

In the case of lack of transfer resources or lack of output buffering resources at the destination module, an arriving packet has to wait until requested resources became available (other packets finish backplane transfers or leave the switch). If more packets have to wait they form a queue and it is up to the queue management to decide which packet will go next when certain resources become available.

In the switch model we implement two types of resources: passive and active. The packets can request and allocate the passive resources. It is up to the requesting packet to free the previously allocated resources. The buffering on input and output are examples of passive resources. The active resources can be allocated (not necessarily exclusively) and used for a certain time. The allocation time may depend on a number of parameters: requested amount, internal settings and a number of currently served requests. The active resource keeps a track of all packets which requests were positively acknowledged. It is entirely up to the resource to decide when the allocation will finish. The packet is then informed that the allocation time has finished and the active resource removes the packet's reference. The switch backplane for inter module transfers is an example of an active resource.

Wrapper is a class designed to guide packets through internals of the switch. There are as many wrapper types as many transfer types identified by the switch. The wrapper "knows" how to guide packet through the switch. It also "knows" where it has to queue in case there are not enough resources. If a model of a switch new module is to be developed implementing new structure of resources, a corresponding wrapper has to be build (it has to know how to route a packet). See more under description of the Wrapper class.

3 DESCRIPTION OF CLASSES

(refers to Figure 1. Switch parameterized model - TOP VIEW- Collaboration diagram)

3.1 intResource

This class is the smallest functional class in the switch structure. It models a simple passive resource. The resource can be initialized with a certain value. The requests for resource will be granted if the requested value is not bigger than the resource initial value minus a sum of all requests currently served. The class offers three methods. The method ***available()*** returns TRUE if there is enough resources to satisfy request, but it does not change any internal status. This method should be used to check whether a subsequent request

would be satisfied. The method ***allocate()*** allocates requested resources up to the internal limit. The method ***free()*** deallocates previously allocated resources.

3.1.1 memorySlots

It is used to model switch buffering resources. The buffering resources operate on memory slots. One memory slot is capable to buffer entire Ethernet packet despite of the packet's length (the Ethernet MTU is 1500 bytes – jumbo frames are not modeled). Each packet allocates one slot on entry and free one slot when leaving the switch.

The class instance is initialized with "Limit" number of integer resources. Requests will be granted as long as the allocated "Pool" would not reach the Limit.

3.1.2 accessBackplane

This class is used to model passive resources necessary for an inter module transfer. It represents switch ability to allow for a number of concurrent transfers from the source module to the backplane and from the backplane to the destination module. In the simplest case it may reflect number of links connecting module to the backplane. As the links may be unidirectional, a module may have different accessBackplane resources for transfers leaving the module and for transfers entering the module from the backplane. The class instance is initialized with bandwidth limit expressed in MB/s. The bandwidth limit represents an aggregated bandwidth of all links in each direction. Before the transfer can start, the packet has to check whether there is enough bandwidth resource on the access from the source module to the backplane and on an access from the backplane to the destination module.

3.2 transferMedium

This class represents an active resource. The main task of this class is to model transfer time. The class supports three basic methods to interrogate, allocate and free resources. The method ***availableRoute()*** should be used prior to the resource useage. The TRUE reply signals, that subsequent request to allocate resource will be granted. The method which models the time is ***route()***. The wrapper wishing to guide a packet, after request for resources availability, calls this method to model transfer time. When the transfer finishes the method ***routingFinished()*** will be called and subsequently, the transferMedium class will inform the wrapper that modeling of transfer time has elapsed.

3.3 DEEthSwitch_param

The DEEthSwitch_param class models the switch chassis. The DEEthSwitch_param class is the Ptolemy star. It inherits from the Ptolemy DEStar class.

The status of a star is defined by the state variables. The switch uses 10 state variables to hold values of the 10 parameters collected during the performance measurements on the real device. See description of parameters in [2].

The object communicates with other Ptolemy stars by receiving and exporting the GigEPacket objects, inheriting from the Ptolemy message class. The switch star receives information from other stars via the portholes (input:MultiInDEPort) and generates information for other stars using also the portholes (output: MultiOutDEPort). The portholes for communication are the multiportholes, and the exact number of input and output ports depends on a particular device the model represents. The chassis, when instantiated, creates

switch model internal structure. The structure is encoded in the star name. The number of letters 'F' or 'G' (corresponding to Fast or Gigabit Ethernet) in the star name defines the number of modules. For example, the string 'EthSw8F8F1G1G' found in the switch name, would create two modules with 8 Fast Ethernet ports each and two modules with single Gigabit port. The chassis keeps pointers to the created modules in vector ModuleIdToPter. The ports are added to the module by calling method ***addPort()*** from the module.

There are also two special portholes which the switch star uses to send information to itself ("self-triggers"; FeedBackIN: InDEPort and FeedBackOUT: OutDEPOrt). The FeedBackIN is connected to the FeedBackOUT.

Every packet arriving to the switch has to be recognized and the source and the destination addresses have to be analyzed. Based on that information a transfer type has to be defined. The chassis creates the abcVlan object responsible for maintaining routing tables and VLANs, and delegates to it the job to find the proper transfer type (see details under the abcVlan class). The chassis creates also the object the routingDecider, where modeling of time elapsed for taking the routing decision is done (see details under routingDecider class). To provide means for the inter module transfers, the chassis creates the TransferMedium object (see details under class TransferMedium). The chassis creates the WaitPool object responsible for maintaining a queue of all packets which were not able to start transfer to the destination port immediately after arrival (see details under WaitPool class).

To improve the switch model performance, there are four wrapper pools created for the four types of wrappers – separate pool for every type of transfer. The pools are initially filled with a small number of empty wrappers. When a new packet arrives and the type of transfer is defined, a wrapper from the pool corresponding to the transfer type is pulled out from the pool. The packet is then attached to the wrapper and the wrapper tries to gain switch resources necessary to bring the packet to the destination module. When the packet leaves the switch, the wrapper returns back to the pool. The wrapper pools are self expanding – if the pool becomes empty, with the last wrapper, the pool uses the last wrapper to clone it to the new set of wrappers of the same type (usually the quantum is 100) and initializes them to origin from the pool.

The DEEthSwitch_param class inherits and uses the ***start(), begin(), go()*** and ***wrapup()*** methods from the Ptolemy DEstar class. The ***start()*** method sets up the switch configuration and start the whole chain of instantiation of various objects. The ***begin()*** method is used to model the learning phase time (see more in the abcVlan description), however there is no strict timing modeling for the internal propagation of the GigEMarp packets. The method ***go()*** is executed whenever new Ptolemy particle arrives to the star (all Packets exchanged between nodes and switches inherit from the particle class). This mechanism is used to activate the star code for every packet, but also to activate the code for “self-triggers”. The switch star uses “self-triggers” to model time needed for internal transfers. The star uses method ReFireAtTime() to schedule requested time to the Ptolemy kernel. The method ***wrapup()*** is used to produce an output with statistics summary. The Ptolemy kernel activates this method on all stars when a user decides to dump statistics.

The ***UpdateRouting()*** method is used in the address learning phase. In the normal operation switch used either ***SwichPacket()*** or ***MakeMulticast()*** methods to model packet switching depending on the type of transfer.

3.4 Module

The module represents a set of resources necessary for packets to get to the destination port. The resources offered by a module are both passive and active. The buffering on input and output, access to and from the switch backplane for the inter module transfers are the passive resources. The transfer medium for the intra module transfers is the active resource. The buffering on input is necessary for new packets when they arrive to the switch and wait for decision on the transfer type. For the inter module transfers, they need access from the input buffer to the switch backplane, the switch backplane and an access from the backplane to the output buffer in the destination module. In the case of intra module transfers, the packets need intra module transfer medium to move from the input buffer of one port to the output buffer of another port in the same module. When the module object is instantiated, it creates another objects modeling resources. The module manages these resources, however the modeling of resources, their allocation and availability is delegated to the intResource class (see description of class intResources for details). Very often the input and output buffers are implemented as shared memory in the real devices. In the model with shared memory, the intra module transfer speed is very high and its contribution to the transfer latency is negligible.

The management of the resources is based on three ways for a resource manipulation. When a wrapper wants to use a resource it has to check whether the resource is available. If reply is positive the wrapper can allocate the resource, and free it if no longer needed. The module offers the following passive resources: input buffering - ***availableSrcStorage()***, output buffering: ***availableDstStorage()***, access from input buffer to the switch backplane:***availableSrcTransfer()***, and access from the backplane to the output buffer: ***availableDstTransfer()***. Depending on the wrapper (transfer type), the different resources would be needed. If the module replies positive on availability the wrapper can allocate the resources: ***storeSrc(), storeDst(), transferSrc(), transferDst()***. When the resources are not needed any longer, wrapper frees them: ***freeSrcStore()***, ***freeDstStore(), freeSrcTransfer() and freeDstTRansfer()***. For an intra module transfer, a module offers active resource: backplane. Communication between a packet and the backplane is relayed by the module and the three methods are: ***availableIntraRouting()***, ***routeIntra()*** and ***routingIntraFinished()***. The module manages the resource and relays requests and replies between wrappers and the resource. Actual testing availability, allocation and freeing is delegated to the specialized objects inheriting from intResources and TransferMedium.

3.5 routingDecider

This class represents process of defining transfer type. The packet latency measurements demonstrated, that the fixed offset in latency does not depend on the packet length. It depends on the transfer type: shorter for intra module transfers than for inter module. In the switch model, we assume that this fixed offset is spent on defining the transfer type. As the wrapper defines type of transfer, it holds value of the fixed overhead. When the newly arrived packet is attached to the wrapper and wrapper start it's guidance through the switch, the very first step is to contact the routingDecider object to model time needed for the transfer type identification. The routingDecider offers two methods for standard operation: ***decide()*** and ***decisionCompleted()***. The first is used by a wrapper to start modeling time for a transfer type definition. The routingDecider consults the wrapper to get amount of time for fixed offset and schedules to Ptolemy a request to "self-trigger" when requested time elapses. Wrappers

waiting on completion time are held in the internal container. When the time elapses, the switch calls the decisionCompleted() method and the routingDecider finds a proper reference to a wrappe. It then instructs the wrapper to continue the packet guidance.

3.6 abcVlan

This class helps the switch to find a routing for a newly arrived packet. The class is configured via the ***addPort()*** method, where the VLAN assignment for a port is passed. The routing table is build and updated during the learning phase, when the control packets (GigEMarp packets) are generated by the nodes attached to the network. Every new packet arriving during that time is inspected for the source address and this address is assigned to the port where the packet arrived. The destination address during that phase is irrelevant. The switch floods copies of this packet onto all other ports. This way all switches in the network learn correlation between addresses of nodes attached to the network and port numbers. In the GigEMarp packets nodes can send a list of multicast addresses (identifiers) they wish to listen to. The switches use this list to extend number of VLANs identifiers assigned to the port where the GigEMarp packet arrived

.

Here are some points specific to the ATLAS strawman architecture.

1. The ATLAS strawman architecture creates network loops that clash with the Spanning Tree operation. To avoid the Spanning Tree switching off redundant links, the strawman architecture heavily relays on VLANs. In modeling, the switches are configured with VLAN definition at the initialization phase (before any packets will flow through the network). Therefore the GigEMarp packets (using extended Ethernet with the VLAN tagging) will be allowed to flow only through the network within the VLAN boundaries. If the node is attached to a port that is a member of many VLANs, it has to generate the GigEMarp packets for every VLAN it communicates through.
   The VLAN configuration files (one for each port) are generated automatically after the Ptolemy creates the architecture and all connections are known. The files are the plain ASCII files listing pairs of a port number and VLAN number assigned to that port.
2. To improve the model performance, the GigEMarp packets will be directed to the Gigabit links connecting switches only (the Fast Ethernet links connect to the end nodes and the GigEMarp packets will be dropped there anyway).

When switching unicast packets, the switch uses the ***lookup()*** method to retrieve routing information and decide where to send the packet. The first check is made on the inbound port, whether the VLAN identifier form the packet's tag is one of the identifiers assigned to the port. If the match is not found a packet is dropped. The switch then makes another check, whether the source and destination ports belong to the same VLAN. The method ***checkCommonVlan()*** can be used to check whether two ports belong to the same VLAN. In case of negative reply, packet would be dropped. The method ***producePortList()*** produces a vector with all ports belonging to a specified VLAN ID. Similarly, the method ***produceVlanList()*** produces a vector of all VLANs sharing the same port. When switching the multicast packet the same checks are performed – packet is not delivered to the ports from different VLANs. In the internal routing, the switch creates as many copies of the packet as many different modules have at least one destination port. The copies are delivered serially to the modules, and later are distributed by a module to all ports, which have the VLAN tag of the original packet.

3.7 OutputQueue

The packets arriving to the destination port need buffering resource before they will be granted an output port. The class OutputQueue is responsible for managing the queue for the output port. The class takes wrappers using the ***store()*** method, and passes them to the output port if the port is free. The method ***retrieve()*** moves one packet from the queue to the head position and returns pointer to the wrapper guiding the packet. Once the wrapper is in the head position it can be removed from that position (and from the queue) with method ***accessHead()***, leaving the place empty (the head of the queue).

3.8 WaitPool

The WaitPool class is designed to hold references to the wrappers which were not able to guide their packets due to the lack of resources in the switch. It provides two methods: holdWrapper() and activateWrapper().

The holdWrapper() method is used by wrappers in case they do not have enough resources to start of transfer instantly

The activateWrapper() method is used by the switch to inform the WaitPool, that resources allocation has changed. During execution of this method the WaitPool scans internal container with wrappers and calls their method retryGuidance(). The Wrappers have a possibility to check whether the resources they need became free.

The WaitPool object can implement various methods for manipulating with wrappers waiting for resources. The simplest is the "longestWait". Whenever the WaitPool is actiavted with the activateWrapper() method, the wrappers are scaned in timely order with the first one waiting the longest. The other method implements the round robin scheme among wrappers originating from different modules.

3.9 Wrapper

The Wrapper class is designed to guide packets through internals of the switch. The transfer is mainly based on availability of the switch/module resources. Each transfer type has the corresponding wrapper: unicast intra module, unicast inter module, multicast intra module and multicast inter module. Depending on the transfer type the corresponding Wrapper executes methods offered by Modules and backplane, to get resources necessary for its mission.

There are four basic methods executed by a Wrapper in response to the switch status change. The method ***waitForDecision()*** informs the Wrapper, that it can model time necessary to find a route to the destination port. The object responsible for modeling that time is the routingDecider. The wrapper calls the method of the routingDecider to hold it for a time necessary to model the routing decision process.

The method ***guidePacket()*** informs the Wrapper, that it should seek for the resources necessary for transfer to the destination module. Using the methods offered by the Modules and backplane, wrapper tries to allocate necessary resources. If they were found the Wrapper calls method of the backplane to model time necessary for transfer. In case resources were not available, the Wrapper calls method of the WaitPool to put itself in the hold and wait for moment when the resources will became available.

The ***retryGuidance()*** informs the Wrapper, that the WaitPool has been informed that the resources allocation switch wide have changed and there is a chance, that some wrappers

held by the WaitPool may be able to start transfer. The WaitPool scans all Wrappers (the order of scan depends a the policy the WaotPool implements) and executes the Wrappers' retryGuidance() method to allow them to check the resources availability.

The ***transferFinished()*** method informs the Wrapper that transfer through to the destination module has finished, and it is a time to book for the output port. During the ***transferFinished()*** method Wrapper frees resources necessary for transfer and calls method queueForOutput() of the destination module. The module relays the request to the OutputQueue object which runs it's policy to access the destination port.

The ***frameLeft()*** method informs the Wrapper the last bit of the packet left the switch. During this method wrapper frees any resources, which were still allocated for the packet held in the output port queue, and returns itself to it's own WrapperPool.

4 SEQUENCE DIAGRAMS

4.1 packetArrival

The diagram (Figure 2) presents interactions between various classes when an event of arriving new packet is to be modeled. The packet is an object of GigEPacket class inheriting from the Ptolemy Message class which in turn inherits from the Ptolemy Particle class – the basic class for the inter-star communication. The arrival of a packet provokes the method go() of the DEEthSwitch_param. The switch implements network portholes for communication with other stars, but also it implements feed-back portholes to schedule “self-triggers”. The method go() check in the first instance, which porthole has got a data. The network packets arrive on the network portholes. If the input message is a regular Packet, the switch checks whether the packet is a unicast or multicast. For the unicast packets the switch consults the vlan object (via the lookup() method) to find a destination port. When the destination port is known, the switch decides on the type of traffic (it has a map of all ports and modules) and pulls a wrapper (via the giveWrapper() method) from the proper pool. The wrapper is then initialized with proper settings for source and destination ports and modules. Finally the newly arrived packet is attached to the wrapper and the method waitForDecision() of the wrapper is executed.

The wrapper knows what resources and when are necessary for the packet to reach the output port. At the first step it tries to get input buffering resources. It checks with the input module whether there are enough resources to buffer the packet (via the availableSrcStorage() method of the input module). If the input memory is fully occupied the packet is dropped and the wrapper returns to it’s pool. In case the source module offers a buffering on input, the wrapper allocates the resource (via the storeSrc() method of the input module). In the next step, the wrapper contacts the routingDecider object to model time the real switch uses to establish transfer between the source and destination module. This time can be measured on real devices as a fixed overhead in transfer latencies. The value of this time is set to the switch model as parameters P9 and P10 depending on the trasfer type (intraModuleFixedOverhead_P9 or interModuleFixedOverhead_P10). The value of this parameters is passed to the wrapper at initialization time. When the wrapper contacts the routingDecider to model the fixed overhead, it passes the value to the routingDecider. The routingDecider adds the fixed overhead to the current modeling time and schedules “self-trigger” ROUTING_DECIDED with the Ptolemy kernel. There may be a number of routing decisions processes active at any given moment, however, they do not need to be queued and

their times run independently. The routingDecider puts the pointer to the wrapper into the internal container.

4.2 routingDecided

The Ptolemy kernel delivers the “self-trigger” message to the DEEthSwitch_param star (diagram in Figure 3). The “self-trigger” has been booked to model constant time spent on making a routing decision after arrival of a new packet (see packetArrival sequence diagram for details).

In the “self-trigger” the Ptolemy kernel delivers a self-generated particle to the special input porthole and invokes method go(). The method go() is activated any time a new particle arrives to the switch. In the first step is has to recognize what type of particle caused method go() to run. If there is a particle at the feedback port, an additional check has to be made on a type of the feedback (the switch uses the feedback port to receive “self-trigger” events generated on various conditions). If the code of “self-trigger” is “ROUTING_DECIDED”, the routingDecider object is consulted to find a wrapper, for which the routing decision completed. Once the wrapper has been found it is instructed to perform the guidePacket() method. The guidePacket() method is the next step in guiding a packet through the switch internals.

The guidePacket() method is the main method for modeling packet transfer through the switch. It has to check and allocate all necessary resources for a transfer. For the intra-module transfers resources are located within the same module and their assessment is slightly easier than for the inter-module transfer. The sequence diagram from Figure 3 presents the inter-module transfer.

To get to the destination module, a wrapper checks buffering resources on the destination module: availableDstTransfer(), access resources on input module to get to the backplane: availableSrcTransfer() and on destination module to get from the backplane to the module’s output memory: accessDstTransfer(). It also needs to check if there is enough bandwidth capacity in the main switch backplane: availableRoute(). If all the checks return positive, the wrapper can allocate required resources and ask an active resource, the backplane, to model transfer across the switch. The allocation of buffering to the destination module is made with storeDst() method of the destination module. The access resources are allocated via the transferSrc() method on an input module to allocate access to the backplane. The transferDst() method on the destination module allocates access resources from the backplane to the output memory. In the call to the route() method of the backplane, the wrapper passes a pointer to itself. The route() method of the backplane, takes the pointer and stores it in an internal container. It also calculates the moment when the transfer will finish and schedules the “self-trigger” INTER_TRANSFER_COMPLETED. The two other parameters are used in the timing calculations: interModuleBandwidth_P7 and intraModuleBandwidth_P8 (see [1] for more details on parameters).

If the check on resources returns negative results, the wrapper has to wait until the resources became available. Which of the waiting pool is selected depends on the transfer type (inter or intra module). For the intra-module transfer, each module manages its private pool and the packets waiting in different modules do not interfere. However, the packets waiting in the same pool may compete for resources and it is up to the pool management policy to decide which packet would go next. For the inter-module transfers, the switch provides a common wait pool where packets from different modules may compete for the same resource (for example output buffering in the same destination module). The wrapper

executes its own method waitOnInput(), which passes the wrapper's pointer to the WaitPool object in the method holdWrapper(). The WaitPool takes the pointer and puts it in its internal container. Packets wait in the pool until there is reallocation of the switch resources. Every time a packet finishes its transfer across switch internals it frees some of the access, buffering and transfer resources. Similarly, when a packet leaves the switch it frees output buffering. At each of these cases the WaitPool is contacted to check whether there are any packets waiting for resources just freed (for more details see explanation for the interTransferCompleted sequence diagram).

4.3 interTransferCompleted

The Ptolemy kernel delivers the "self-trigger" message interTransferCompleted to the DEEthSwitch_param star. The "self-trigger" has been booked to model time spent on transferring a packet between the input memory of the source module and output memory of the destination module.

In the "self-trigger" the Ptolemy kernel delivers a self-generated particle to the special input porthole and invokes method go(). The method go() is activated any time a new particle arrives to the switch. In the first step is has to recognize what type of particle caused method go() to run. If there is a particle at the feedback port, an additional check has to be made on a type of the feedback (the switch uses the feedback port to receive "self-trigger" events generated on various conditions).

If the code of self-trigger is "INTER_TRANSFER_COMPLETED", the switch instructs the backplane object (via the routingFinished() method) to find a wrapper, for which the transfer completed. Once the wrapper has been localized (in the internal container), its method transferFinished() is activated. This method is the next step in guiding a packet to the destination port.

Because the transfer completed some resources previously allocated can be freed. The wrapper contacts the source module to free input buffering and access from the input memory to the backplane: via the freeSrcStore() and freeSrcTransfer() methods respectively. In the destination module the access from the backplane to the output memory can be freed as well: via the method freeDstTransfer().

Once the packet arrived to the destination module it has to queue for the output port. Each module provides output buffering resources and it is up to it to decide how to queue the packet for output port. The wrapper executes the queueWrapperFotrOutput() method of the destination module. As the module only manages buffering and relays requests, it delegates the request to the OutputQueue object. The module executes the store() method of the OutputQueue to pass a pointer to the wrapper waiting for the output port. The module checks whether, there is any other packet being transmitted at the moment (via the method headAllocated()), and if so, the new request is put into the queue. If the output port is free, the module puts the wrapper at the head of the queue (via method accessHead()) and executes method sendData() of the GigELink object (responsible for modeling transfer over the Ethernet links). The packet is detached from the wrapper, but the wrapper remains in the head of the queue. The GigELink object schedules to the Ptolemy kernel a "self-trigger" with the LINK_SEND_END code when the last bit of the packet would leave the switch.

In the second part of the switch go() method, the switch checks whether there are any packets waiting for resources which just became available. It executes the activateWrapper() method of the WaitPool object. The Pool has a container with pointers to all wrappers waiting

for resources. It scans the container and executes the retryGuidance() method of each wrapper to check whether there are enough resources to start transfer. The order in which the scan is done depends on the policy the waitPool implements. The succeeded wrappers allocate resources in a standard way and continue modeling transfer time through the switch internals. These wrappers are removed from the WaitPool queue.

4.4 linkSendEnd

The Ptolemy kernel delivers the “self-trigger” message linkSendEnd to the DEEthSwitch_param star. The “self-trigger” has been booked to model time spent on transferring a packet on an Ethernet link.

In the “self-trigger” the Ptolemy kernel delivers a self-generated particle to the special input porthole and invokes method go(). The method go() is activated any time a new particle arrives to the switch. In the first step is has to recognize what type of particle caused method go() to run. If there is a particle at the feedback port, an additional check has to be made on a type of the feedback (the switch uses the feedback port to receive “self-trigger” events generated on various conditions).

If the code of self-trigger is the “LINK_SEND_END”, the switch calls the method frameLeft() of the module, to inform it that the packet finished transfer over the Ethernet link. This information is passed further to the OutputQueue object as it still holds a pointer to the wrapper in the head position. The OutputQueue object returns pointer and the wrapper executes the method frameLeft(), the final step in packet guidance. It frees the output buffering resources (via the freeDstStore() method of the module object). In the last action the wrapper gives its own pointer to the wrapper pool (via the method returnToPool()).

In the second part of the go() method, the switch checks whether there are any packets waiting for resources which just became available (output buffering). It executes the activateWrapper() method of the WaitPool object and procedure repeats as for the INTER_TRANSFER_COMPLE-TED case (see description above).

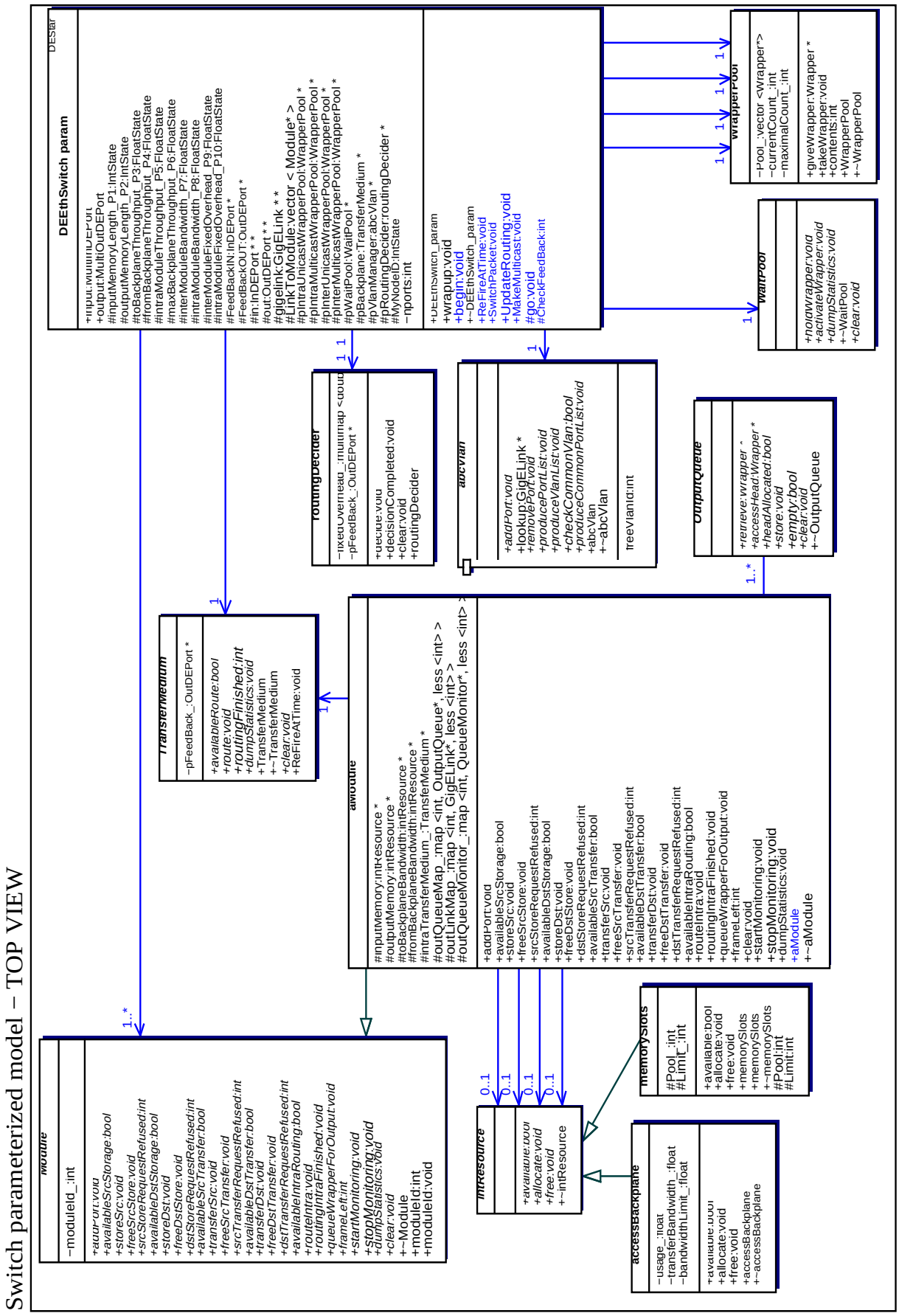

Figure 1 Switch parameterized model TOP View collaboration diagram

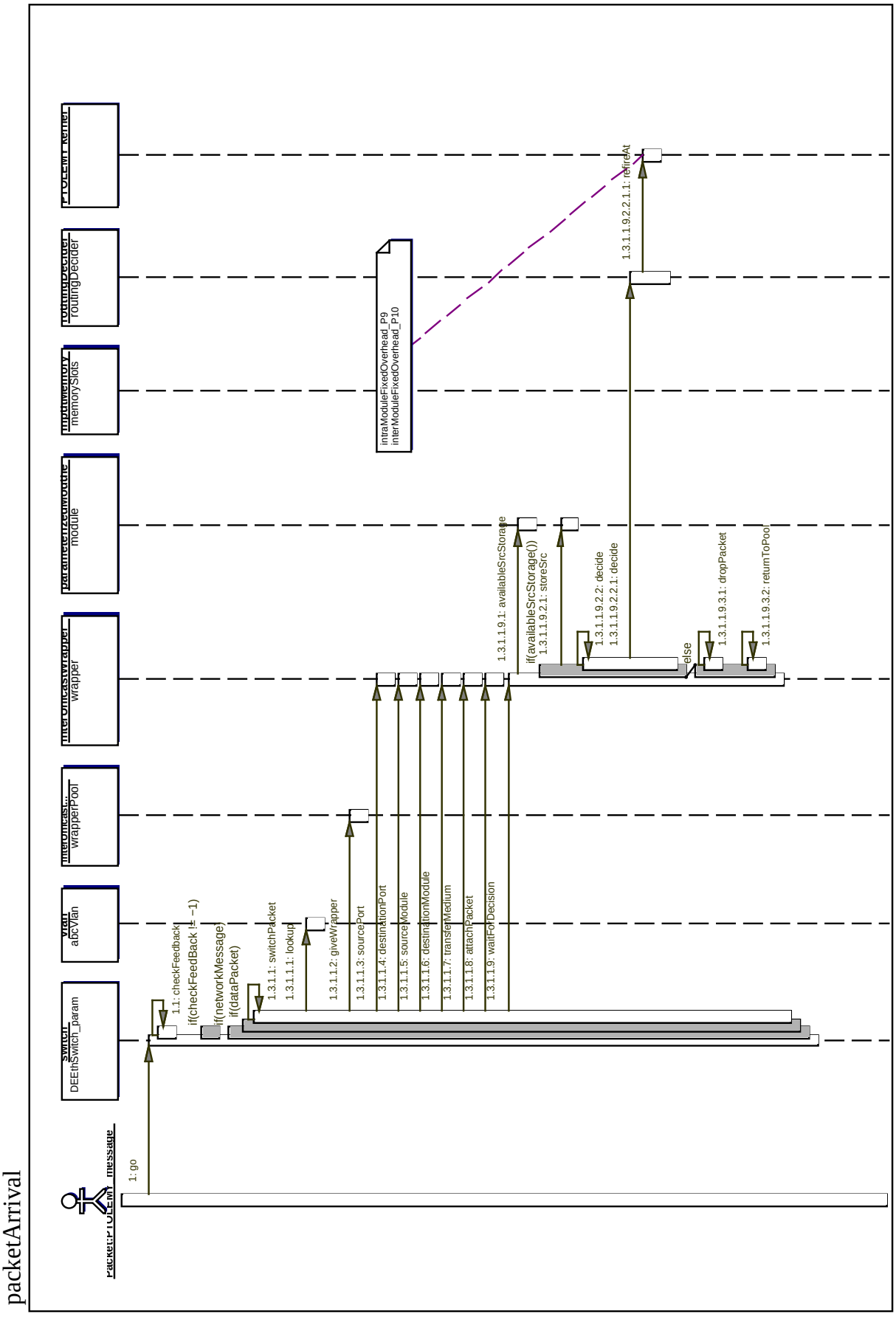

Figure 2 Packet arrival sequence diagram

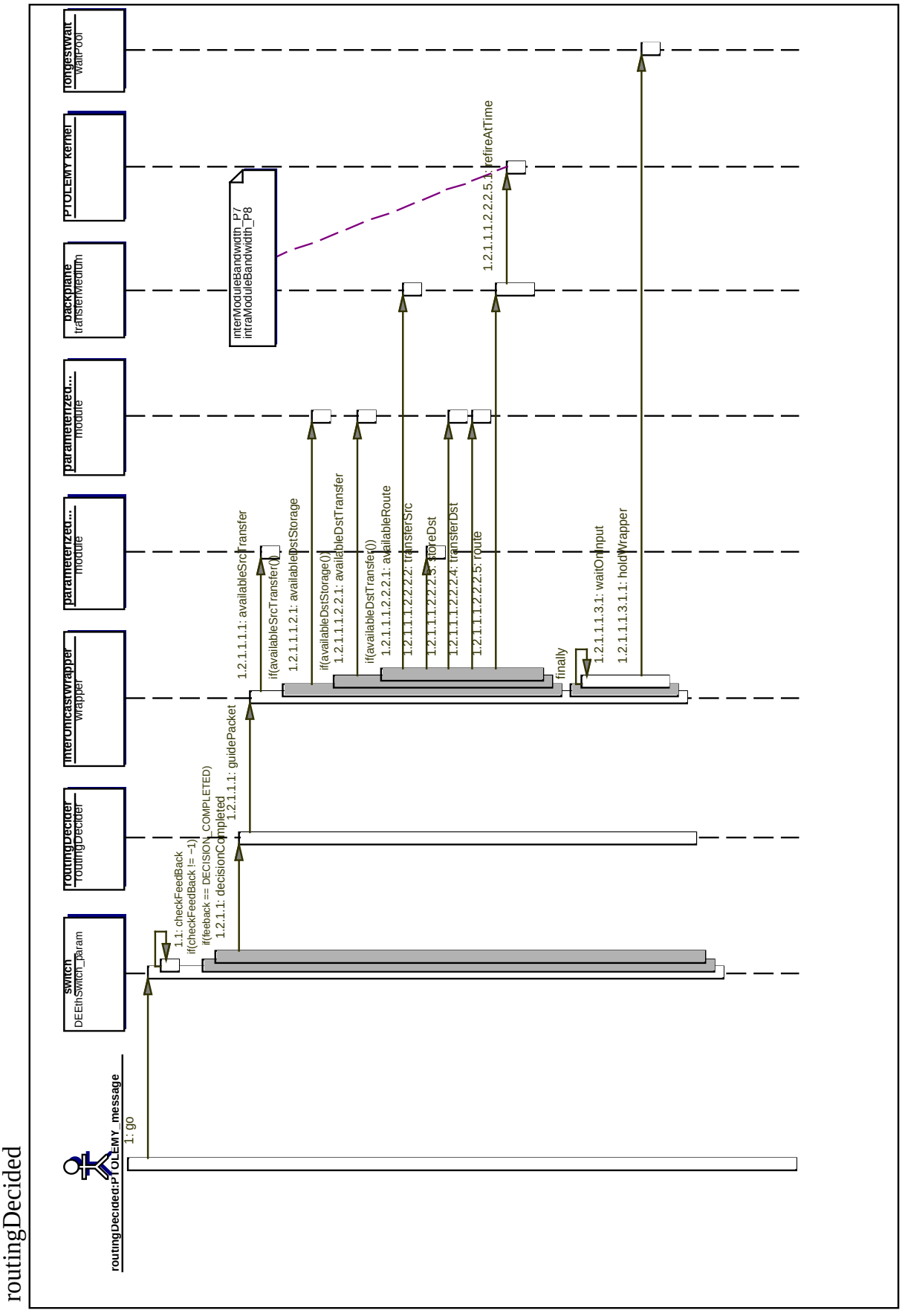

Figure 3 Routing decided sequence diagram

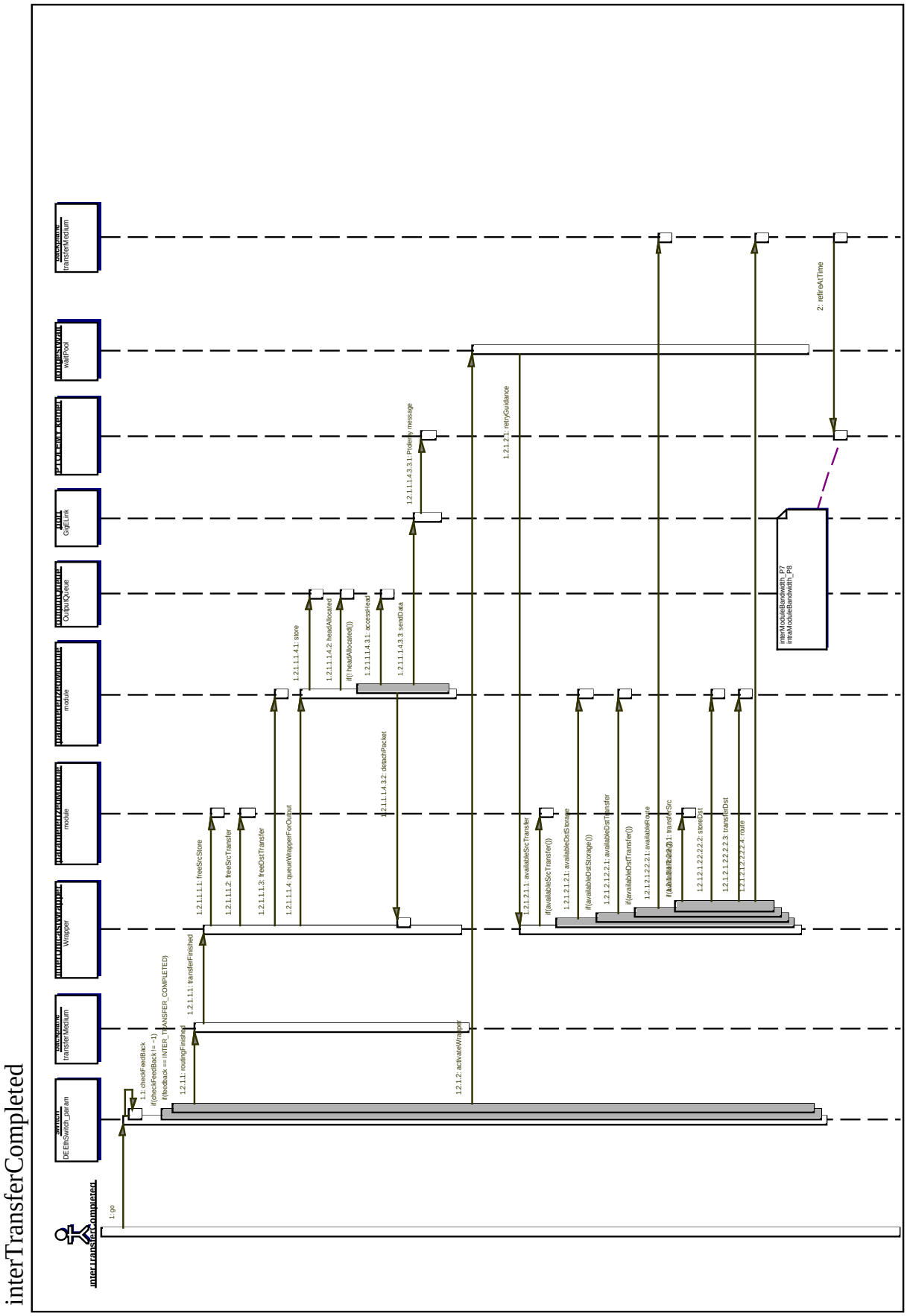

Figure 4 Inter-module transfer completed sequence diagram

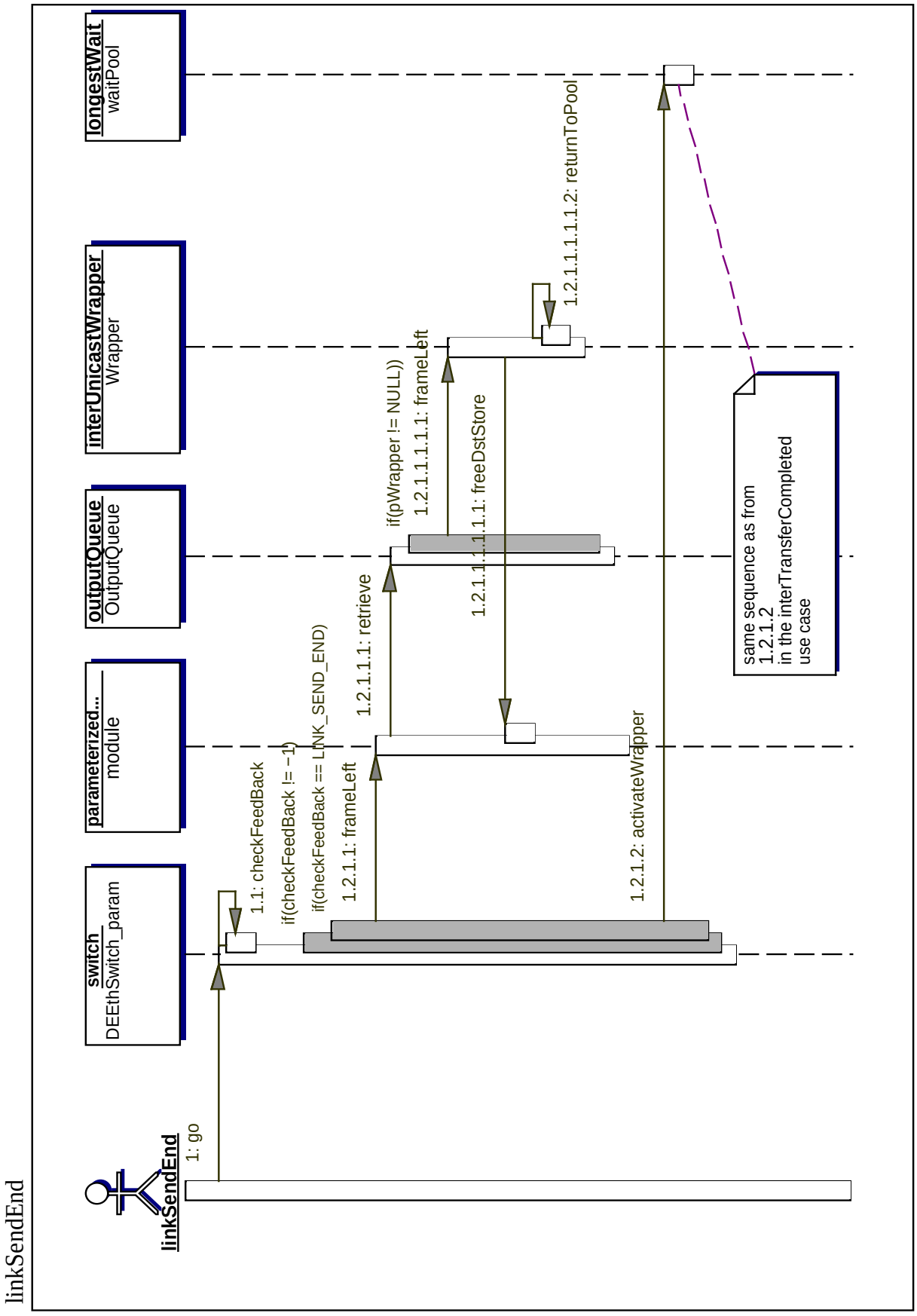

Figure 5 Frame left sequence diagram

This document has been prepared using the Short Note Template provided and approved by the ATLAS TDAQ and DCS Connect Forum. For more information, go to http://atlas-connect-forum.web.cern.ch/Atlas-connect-forum/.

This template is based on the SDLT Single File Template that has been prepared by the IPT Group (Information, Process and Technology), IT Division, CERN (The European Laboratory for Particle Physics) and then converted to MS Word. For more information, go to http://framemaker.cern.ch/.