PMIF+ to QNAP transformation (Acceleo)

1. Introduction

This document describes the first stage of a Model-to-Text (M2T) transformation that generates QNAP models from PMIF+ models. This transformation is implemented using Acceleo (http://www.eclipse.org/acceleo/) which is a code generator based on templates that implements the OMG's Model-to-Text specification. Acceleo is fully integrated in the Eclipse's EMF framework.

2. Structure of the transformation

In this section, the structure of the Acceleo transformation is described at high level, in pseudo-code. The fine detail of the transformation is explained in the section 2.

```
/* declarations */

for each server, forkjoin node and sourcenode
        declare a queue
end for

for each servicerequest that has a ForkNode as server
        declare a queue // the splitter
        for each forkworload in the forknode
                declare a queue //the router
        end for
end for

for each passive entity
        declare a queue
end for

for each workload
        declare a class
end for

for each servicerequestplus
        declare a queue // for passive non blocking services
        for each service
                declare an auxiliary class //for routing between the active and the passive
                                                // stations
        end for
end for

for each servicerequest, openworkload and closedworkload
```

```
        declare a real // used for sample generation, some may be actually unused
end for

for each passive entity
        if is syncpoint
                declare two flags
                declare a customer reference
        else if event
                declare a flag
                declare two customer references
        else if timer
                declare a timer
        end if
end for


if exists a syncpoint
        declare attribute CLLRTRN //call return reference
        declare attribute QUEUED //whether the customer is queued for an event or not
end if

 /* procedure declarations */

for each workload and servicerequest
        generateRoutingProcedure(...) // see detailed explanation below
endfor


/*station declarations */

for each passive entity
        generate_PassiveEntity_Station
end for

for each openworkload
        generate_Source_Station
end for

for each closedworkload
        generate_ThinkDevice_Station
end for

for each servicerequest
        if service request is  ServiceRequestPlus
                if server is ForkJoin
                else

        else
                if server is ForkJoin
                else
                        generateStation
                end if
```

```
        end if
end for


procedure generateRoutingProcedure (prefix: String, num: Integer, transits, workload)
        if mixed TransitPlus and Transit
                ERROR
        else
                if all transits are TransitPlus
                        if mixed DepRoutingTypes
                                ERROR
                        else
                                declare_routing_procedure
                        end if
                else // all transits are Transit
                        declare_routing_procedure
                end if
        end if
end procedure
```

3. PMIF+ elements transformation

In this section we describe the details of the transformation of each PMIF+ element (or group of elements).

3.1. Workloads

For each workload (OpenWorkload, ClosedWorkload or ForkWorkload), a class, a queue and a routing procedure are declarated. The queue represents the associated node of the workload (a source for OpenWorkloads, a think device for ClosedWorkloads and a Fork node for Forkworkloads. The Routing procedure is a procedure that determines where a customer has to go after it has received service; this procedure is described in detail in section 2.3. In addition, a call to the routing procedure is generated for each transitFirst. This is summarized in Figure 1.

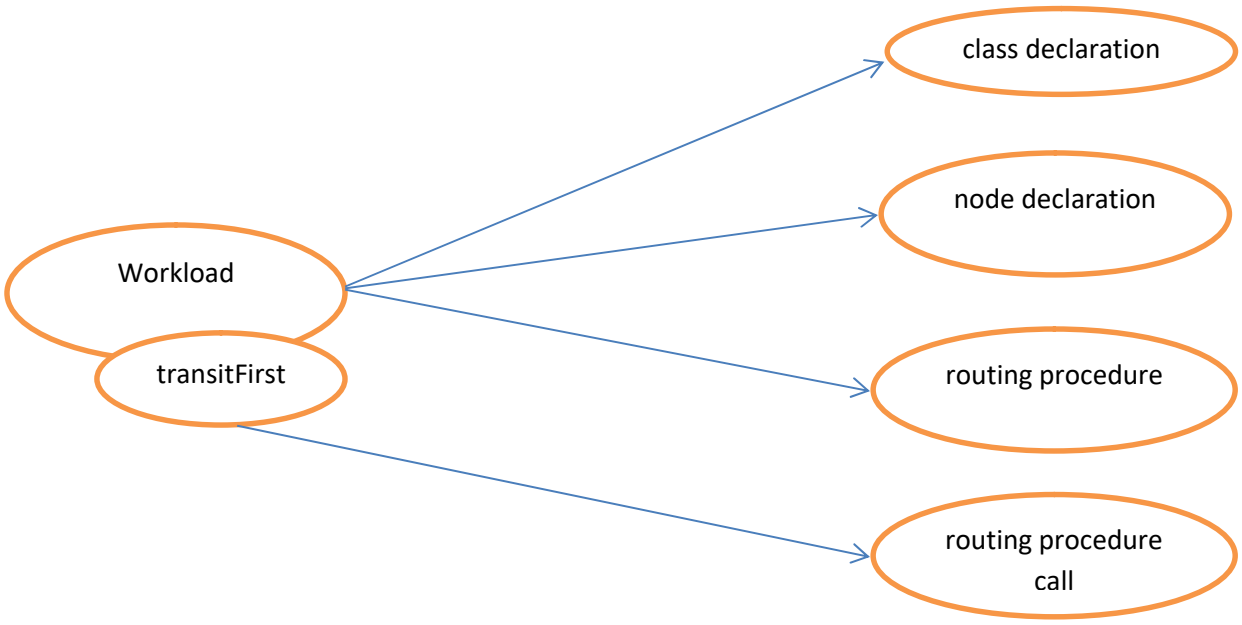

Figure 1 - Workload transformation.

3.2.Nodes

For each node, a queue is declared, as is shown in Figure 2. The specification of the corresponding station is done when the ServiceRequest/Workload element is transformed.

Figure 2 – Node transformation

3.3.Transit and TransitPlus

A procedure with the transit information is generated for each workload (openworkload, closedworkload, forkworkload) and for each service request. This procedure's name follows the pattern *prefixTnum*, where:

- *T* stands for Transit;
- *prefix* is '_OW' for openworkload, '_CW' for closedworkloads, '_FW' for forkWorkloads and '_SR' for serviceRequests;
- *num* is the workload/servicerequest position in the sequence of workloads/servicerequests.

This pattern makes easier to call the procedure later in the QNAP code. These calls are generated for each transitFirst or transitNext. The generation of routing procedures and routing procedure calls is depicted in Figure 3.

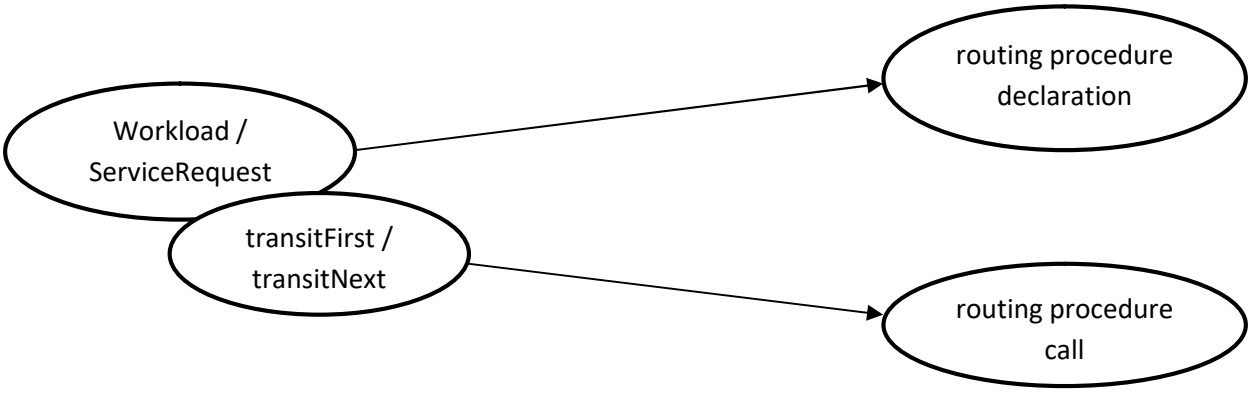

Figure 3 – Routing procedures and routing procedure calls

The transformation checks if the all the transits are of the same kind (TransitPlus or Transit); if not, the transformation generates an error message in the generated QNAP file.

The transit information could be embedded in the 'service' section, but it would be cumbersome. Furthermore, you can have local variables in a procedure so you avoid the declaration of a large number of global variables.

The following list of variables is declared locally in each procedure:

- Integer *N, M*: auxiliary indexes.

- REF QUEUE *station*(size): references to the transit's destination station. If it's the 'OUT' station, (QNAP complains when the reference is the 'OUT' station itself).

- REF CLASS *workload*(size): references to the transit's workload type which can be the incoming workload or another one if the transit is a TransitPlus.

- REAL *value* (size): values used for choosing the transition that actually happens. These values depend on which type of depRouting the transit is:

- for 'probabilities', value(i) is the probability (transit's 'Probability' attribute);

- for 'shortestQueueLenght', value(i) is the target station (transit's 'To' attribute) CUSTNB;

- for 'shortestResponseTime', value(i) is the target station MRESPONSE;

- for 'roundRobin', value(i) is unused;

- for 'fastestService', value(i) is the target station MSERVICE;

- for 'leastUtilization', value(i) is the target station MBUSYPCT.

If the transit is not a TransitPlus, *value* array will store probabilities.

The size of these arrays is equal to the cardinal of the Transit set. If there's only one transit, the aforementioned variables aren't declared as arrays of size 1 but as simple variables (because otherwise QNAP complains).

A global variable is also declared for each transit (INTEGER *prefixRRX*, where RR stands for Round Robin, and prefix and X are the same as in the procedure declaration); this variable is a counter used when *depRoutingType* is 'roundRobin'. If *depRoutingType* is 'probabilities', a random number is generated with the DISCRETE command. This random number will be used as the index for selecting the actual transit's target station and its workload. In the rest of the cases, the index is chosen depending on the *value*

3.4.ServiceRequests

For each ServiceRequest, a QNAP station is specified. As seen previously, a routing procedure and a several routing procedure calls are also generated (Figure 4).

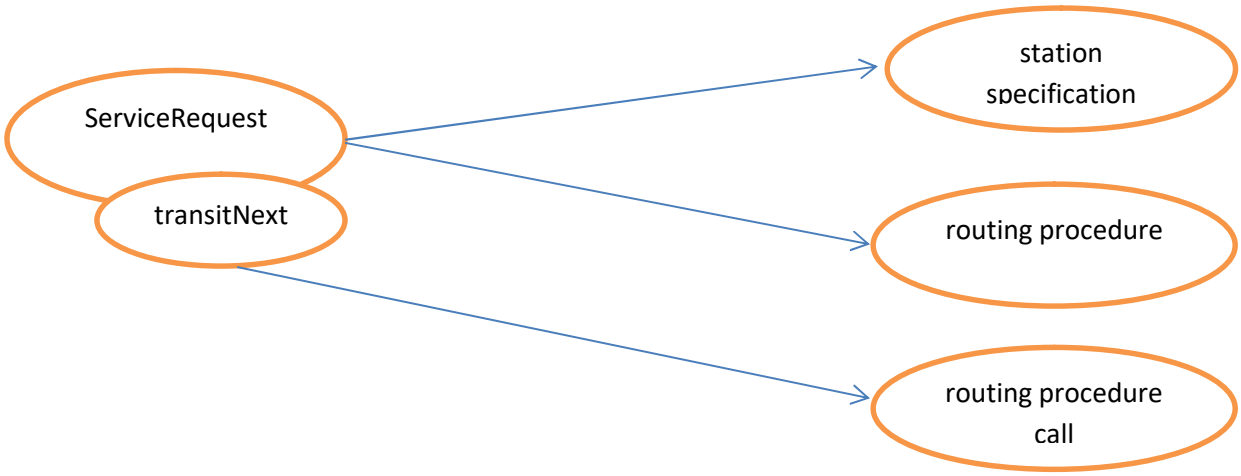

Figure 4 – Transformation of Service Requests

3.4.1. ServiceRequestPlus

ServiceRequestPlus is a kind of ServiceRequest that can contain several Service elements. This services can be active or passive, and the passive services can be blocking or non-blocking. In order to develop a correct transformation, several approaches were studied. Figure 5 summarizes the transformation of a ServiceRequestPlus. In the next subsections, these approaches are described as well as the problems that each of them arised.

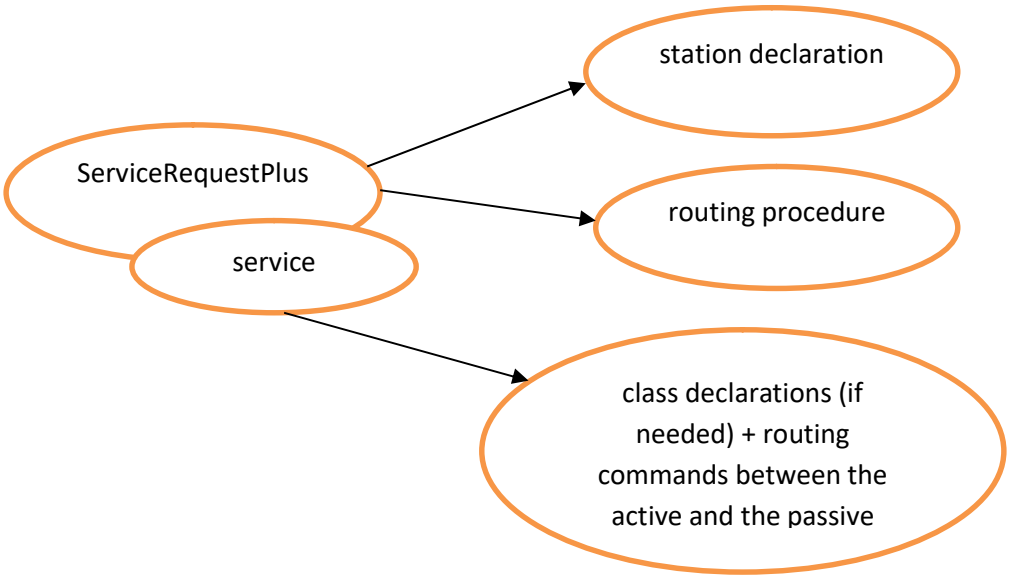

Figure 5 – ServiceRequestPlus transfomation

3.4.1.1. First approach

The ServiceRequestPlus (SRP) to be transformed is composed by active and/or passive services. For each SRP, a station is generated (figure 6); and for each service, the corresponding QNAP commands are generated in the station's SERVICE block.

Figure 6 - Transformation of a SRP (one station for all the services)

Problem: QNAP gives an error ((0R040J) ==>ERROR (SIMUL) : NO SYNCHRONISATION ON STATION WITH PS SCHEDULING) when a PS station is blocked (i.e. at a semaphore).

3.4.1.2. Second approach

A station is created for each SRP service (Figure 7). The first one receives the name of the SRP's server. The names of the remaining stations are automatically generated: *_asXY* for active services and *_psXY* for passive services, with **X** being an integer corresponding to the

position that the SRP in its ordered and **Y** being the number of the service. These name patterns are due to QNAP restrictions (identifiers can be 8 characters long at the most).

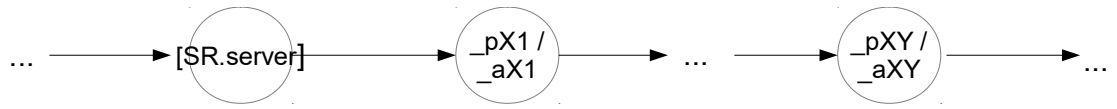

Figure 7 - Transformation of a SRP (one station for each service)

Passive stations have infinite servers and FIFO policy whereas Active stations are defined as the service request indicates (scheduling policy and number of servers).

Problems: the model that this approach generates isn't equivalent to the original model, because when a SRP has multiple active services, there could be customers in every station (_aXY) and this is not what is represented in the original model.

3.4.1.3. Third approach

We changed the PMIF+ model introducing several new stations (syncX) where the blocking commands where performed. However, this change was dismissed because we were adapting the PMIF+ model to a concrete tool (QNAP).

3.4.1.4. Fourth approach

In this approach (working with the original PMIF+ model, not the one of the previous approach) an "entrance station" was introduced. This station receives the server workload's name because this way is easier to route the incoming workload to this service request. Besides this station, two new stations are created, one that processes the active services (_asX) and one that processes the passive services (_psX), as is shown in Figure 8.

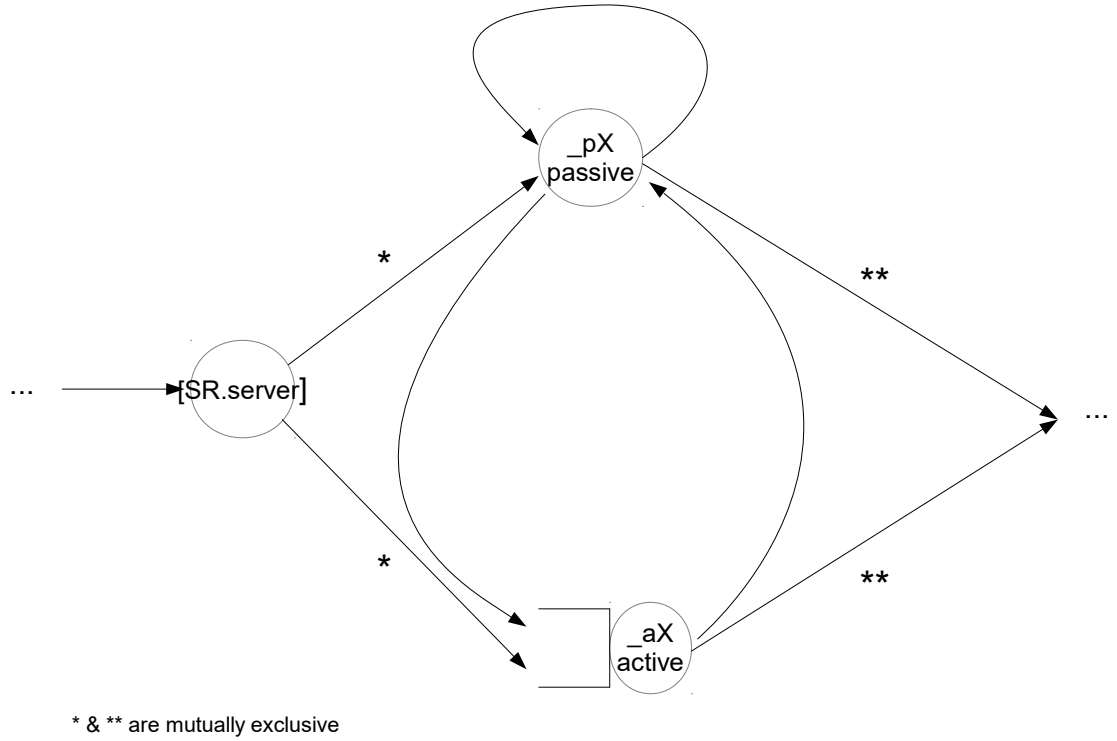

Figure 8 - Transformation of a SRP (entrance station, active station and passive station)

The first station receives the original workload. Then the workload name changes in each step of the route in order to prevent having the same station and workload combination repeated multiple times because QNAP's simulation would fail. The workload names follow the pattern _wlXY, where X is the SRP number and Y is the service number.

Problems: the entrance station is unnecessary and complicates the resulting model. Moreover, the results are hard to find, because the performance indexes of the active station are usually the subject of interest but the may be lost/hidden between the results of several stations named _asX. It would be preferable that the active stations could keep the SRP server name.

3.4.1.5. Fifth approach

In this approach, the entrance station is eliminated. Now the transformation only generates two stations for each SRP, one for the active services (this one receives the name of the SRP server and its scheduling policy and number of servers are defined as the SRP states) and one for the passive services (this one named _psX with a FIFO scheduling policy and infinite servers). The active station is always the one that receives the incoming workload. If the first service is a passive service, the workload receives no service and it is routed to the passive station (Figure 9). As in the previous approach, the workload name is changed in every transition (_wlXY).

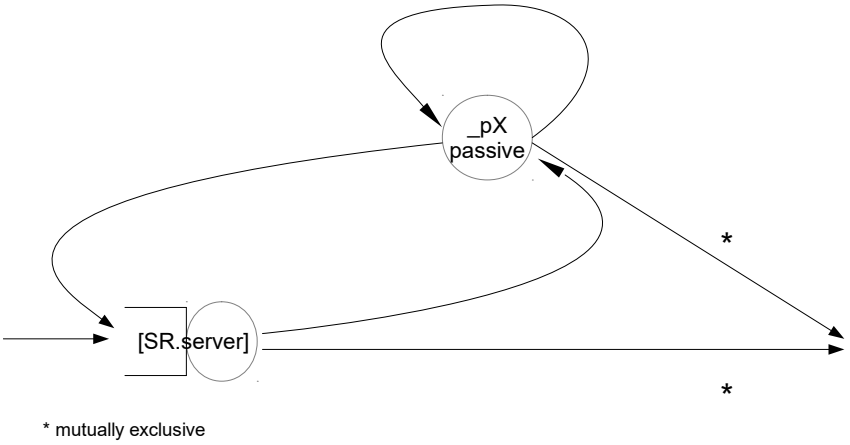

Figure 9 - Transformation of a SRP (active station and passive station)

Problem: how to treat sequences of services in which after an active service there is a number of passive services that block the server (it can be repeated any number of times, i.e.: active-passive(blocks)-passive(blocks)-active-passive(blocks)-passive(blocks)) without letting other customers take the server?

3.4.1.6. Sixth approach

To solve the problem found in the previous approach, the consecutive services that are either active (though there can't be two consecutive active services) or passive with the attribute blocksServer *true* are grouped and performed in the active station whereas the passive services (with blocksServer *false*) are performed one by one in the passive station. In other words, the resulting structure of the transformation is the same as the previous approach but the transits between the active station and the passive station are determined by the presence of non-blocking passive services (Figure 10).

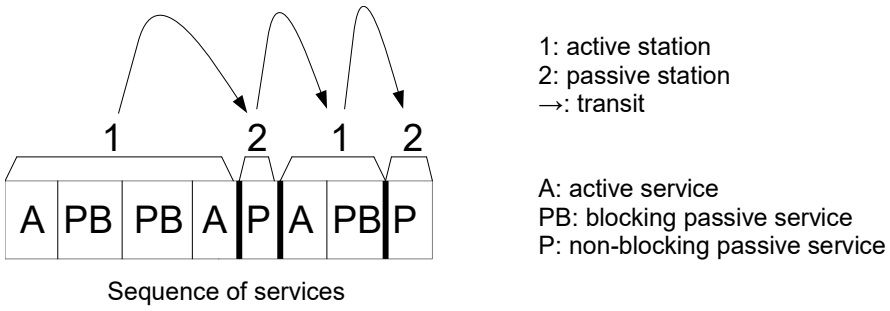

Figure 10 - Grouping of services

3.5.ForkJoin

If a ServiceRequest's server is a Forknode, the transformation generates:

- A "splitter" station. This is the station where the SPLIT command is performed. Its name follows the pattern _sX, with X being the SR number.

- A "router" station for each kind of customer created with the SPLIT station. Without this station the transformation would have to generate all the possible combinations of target stations and number of customers. Its names follow the pattern _rXY, with X being the SR number and Y being the transit number.

- A "fork" station. This station's name is the SR server name. The aim of this station is to model the exit of the Fork node.

Both _sX and _rXY stations have infinite servers and FIFO policy, whereas the "fork station" has its characteristics defined as the SR indicates.

When the **WillJoin** attribute is *true*, no *father* customer is created, and a MATCH operation is performed in the "fork station" (Figure 11). If ***WillJoin*** is false, the *sons* exit the system after its itinerary has ended and the *father* is sent from the _sX station to the "fork station" (Figure 12).

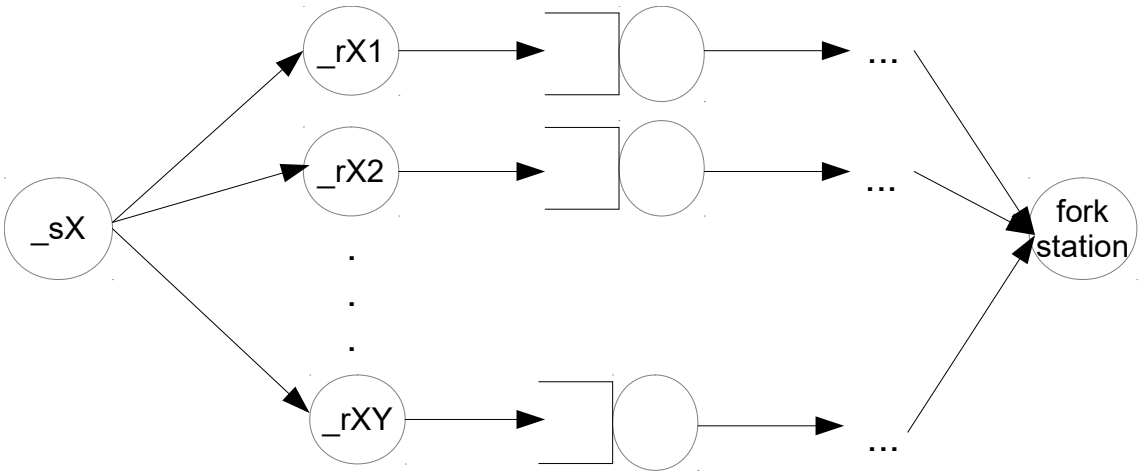

Figure 11 - Transformation of a Fork (WillJoin = true)

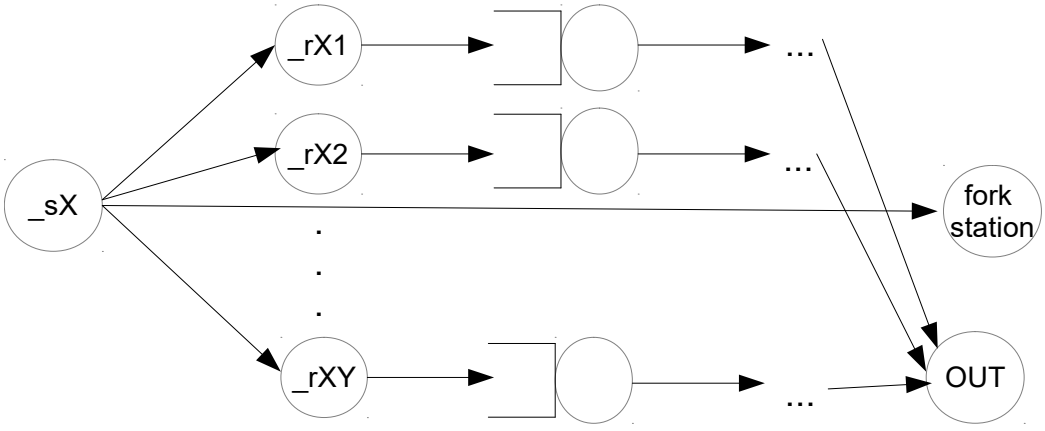

Figure 12 - Transformation of a Fork (WillJoin = false)

The transformation currently disregards any service time related data present in the SR. It would be easy to add this service time in case of Time Service Requests or Demand Service Requests, though it should be decided whether the service time should be represented in the splitter station or in the fork station. If the SR is a Service Request Plus, it would be harder because to transformation because this structure (splitter-router-fork) should be combined with the one shown in the previous section.

The use of automatically generated names is mandatory because there are more stations in QNAP model than in nodes in the PMIF+ model) but it complicates the transitions between nodes, because the transformation can't assume that a transit will go to a station with the SRP server name, as sometimes it will have to transit to a station with an automatically generated name.

When the 'to' attribute of a Transit element (PMIF+ model) is a ForkNode, it can be either a transit corresponding to a son customer returning to the fork station (for the MATCH operation) or a transit corresponding to the entrance to a Fork structure. In the second case, the transit in QNAP must send the customer to the appropriate _sX station (splitter).

As the forks can be nested, whether the workload is a forkworkload or not doesn't determine which case we are dealing with. In other words, if forks couldn't be nested, if the workload were a forkworkload it would mean that the customer comes from a fork so the transformation would have to send it to the fork station. Since the aforementioned restriction isn't true, the customer could also be a customer that comes from a fork and that it's entering into a new fork.

The solution to this problem is to check if exists a SR with that node as server and that workload. If it exists, the customer must be sent to a splitter, otherwise it must be sent to the fork station.

3.6. Transformation of the Passive Entities and its related commands

Most of the passive entities are represented by queues. Resources, Buffers, Memories, Syncpoints and Tokens are associated to Semaphores whereas Mailboxes are associated to Resources. Events and Timers are a little different: an event is represented by flags and a Timer is represented by QNAP's timer.

Some Passive Entities need auxiliary structures in order to be able to implement their behavior: customer attributes and customer references are needed for Syncpoints and Events.

This is summarized in Figure 13. In the next subsections, the details of the implementation of each Passive Entity and its related commands are explained.

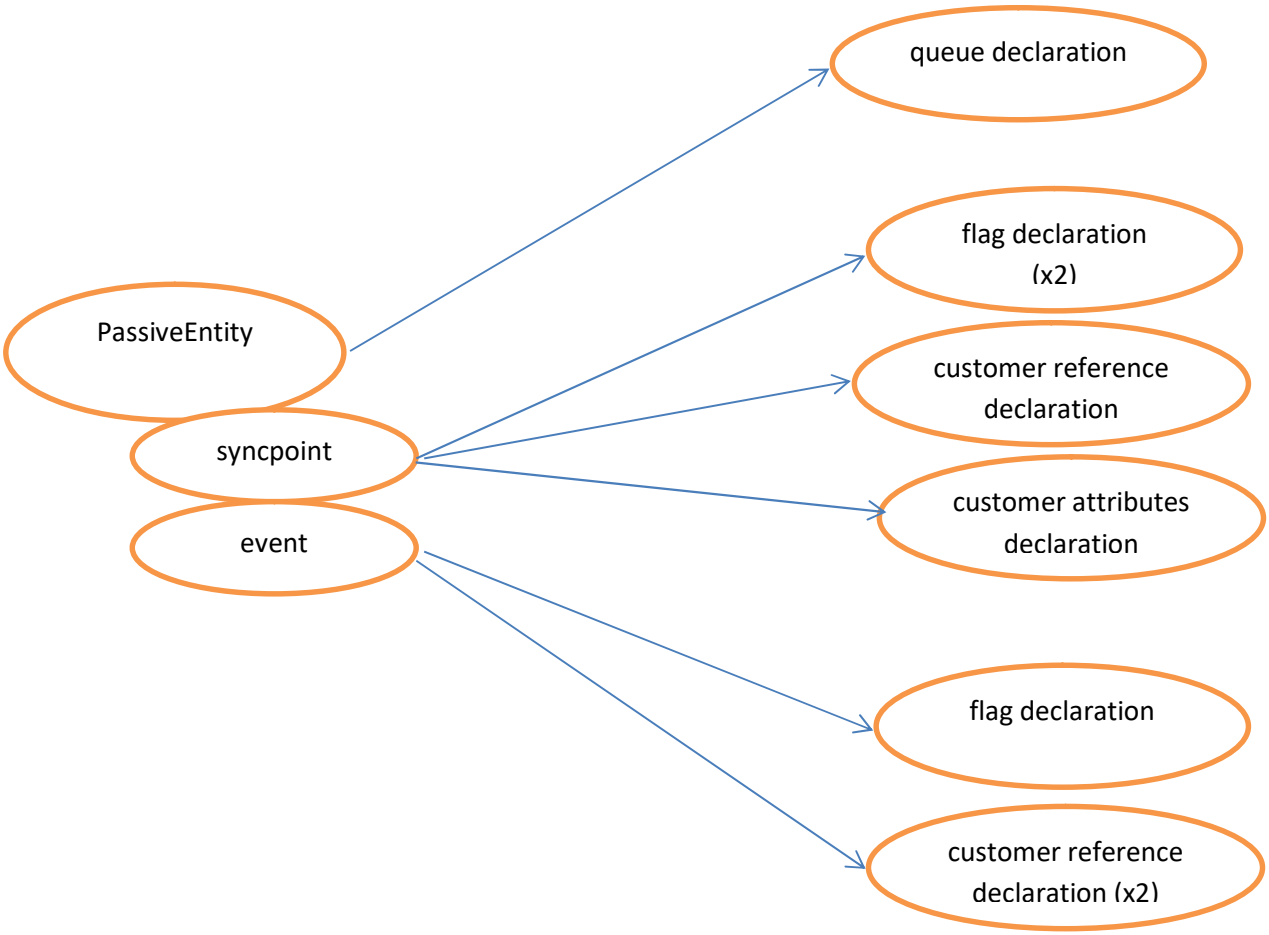

Figure 13 – Transformation of Passive Entities

3.6.1. Resource – Allocate / Deallocate

A resource is represented by a QNAP semaphore, its capacity is determined by the initAvailable attribute (TYPE = SEMAPHORE, MULTIPLE (initAvailable)). The allocation and deallocation of resource units is performed by PMUL and VMUL commands, respectively.

3.6.2. Mailbox – Send / Receive

A mailbox is represented by a QNAP resource. The allocation and deallocation of the mailbox is performed by P and V commands, respectively. CATI: Això hauria de ser “send” and “receive” i “send” hauria de ser la V I receive la P. Això crec q esta be a la transformació

3.6.3. Event – Wait / Queue / Set / Clear

In order to represent an Event and its operations in QNAP, a Flag (FlgEvX, where X is the number of the event), two Customer references (CX and CqX) and a Boolean Customer attribute (queued) are declared for each event. A customer can either wait or queue for an event. If it waits, the *queued* attribute is set to FALSE whereas if it queues, the *queued* attribute is set to TRUE, as is shown in figure 14. In both cases, the customer performs a QNAP Wait operation on the flag.

When an event is Set there are two cases: the first case is that there are no customers waiting or queued for the event, the second case is that there are customers waiting or queued. In the first case, the event is set by performing the QNAP Set operation (SET(FlgEvX)). In the second one, the list of waiting and queued customers (FlgEvX.LIST) is traversed using the CX customer reference: all the waiting customers are freed by performing the QNAP FREE operation whereas only the first queued customer is freed. The customer reference CqX is used to distinguish the first queued customer from the rest.

When an event is cleared, a QNAP RESET command is performed.

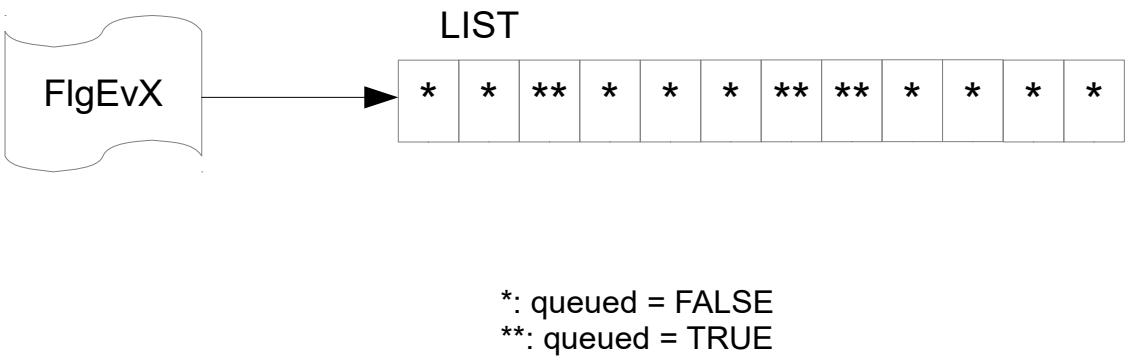

Figure 14 – Event flag and its waiting and queued customers

3.6.4. Timer – Start / Stop

The PMIF+ Timer element is transformed to a QNAP Timer. The operations Start and Stop have been implemented with the commands SETTIMER:ABSOLUTE(timer, 0.0) and SETTIMER:CANCEL(timer), respectively.

3.6.5. Buffer – Get / Put / Create / Destroy

A buffer is represented by a QNAP semaphore, its capacity is determined by the initAvailable attribute (TYPE = SEMAPHORE, MULTIPLE (initAvailable)). The get and destroy operations are performed by a PMUL command whereas the put and create operations are performed by a VMUL command.

3.6.6. Memory – Allocate / Deallocate / Add

A memory is represented by a QNAP semaphore, its capacity is determined by the initAvailable attribute (TYPE = SEMAPHORE, MULTIPLE (initAvailable)). The allocation of memory units is performed by a PMUL command whereas the deallocation and the addition of memory units are performed by VMUL commands.

3.6.7. Syncpoint – CallReturn / Accept / Return

Figure 15 represents the flow of customers when there is a **callreturn** command. The workload1 customer waits the node “callreturn” until the workload2 customer performs a return. This customer is previously waiting in the node accept until the workload1 customer performs the **callreturn** command.

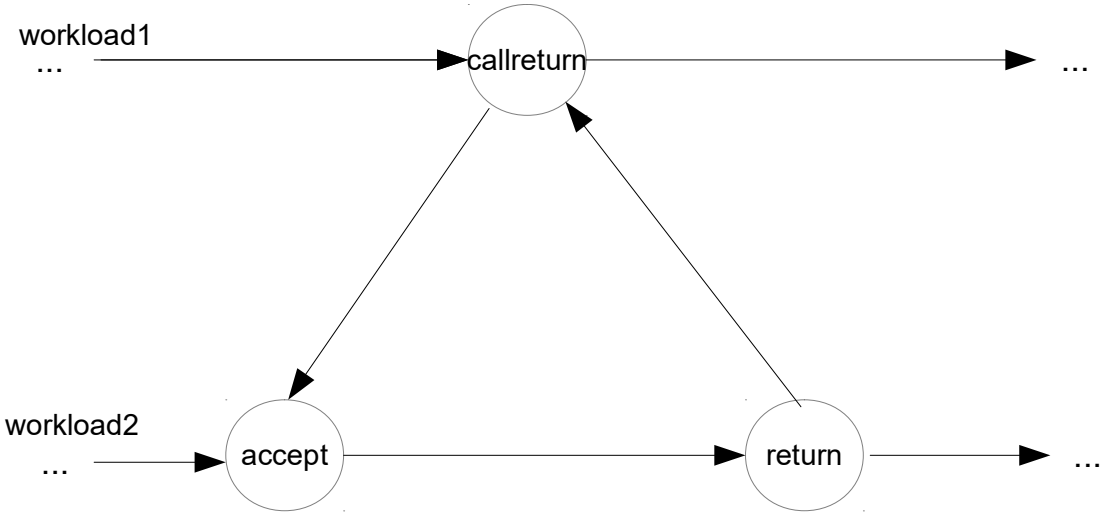

Figure 15 - Syncpoint

The three commands (**callreturn**, **accept**, **return**) where implemented by two flags (*flagAccept*, *flagReturn*) that initially would be unset. In addition, a semaphore has to be declared in order to assure that the first customer can perform the callreturn operation.

The implementation, summarized in Figure 16.

a) callreturn:

1 - when a customer (WRKLD1 class) arrives to the callreturn node, it waits for the semphore SEM1;

2 - when the semaphore SEM1 is released (V), the reference of the first customer (WRKLD2 class) in accept node (they are blocked because they are waiting for FLAG_ACCEPT) is fetched and saved in the attribute cllrtrn;

3 - the first customer waiting in accept node is unblocked (FREE command);

4 - the WRKLD1 customer gets blocked waiting for flag FLAG_CALLRETURN.

b) accept:

1 - when a customer (WRKLD2 class) arrives to the accept node, it releases the semaphore SEM1;

2 - the customer waits gets blocked waiting for flag FLAG_ACCEPT

c) return:

1 - when a customer (WRKLD2 class) arrives to the return node, the customer referenced in its cllrtrn attribute is unblocked (it's waiting for flag FLAG_ACCEPT).

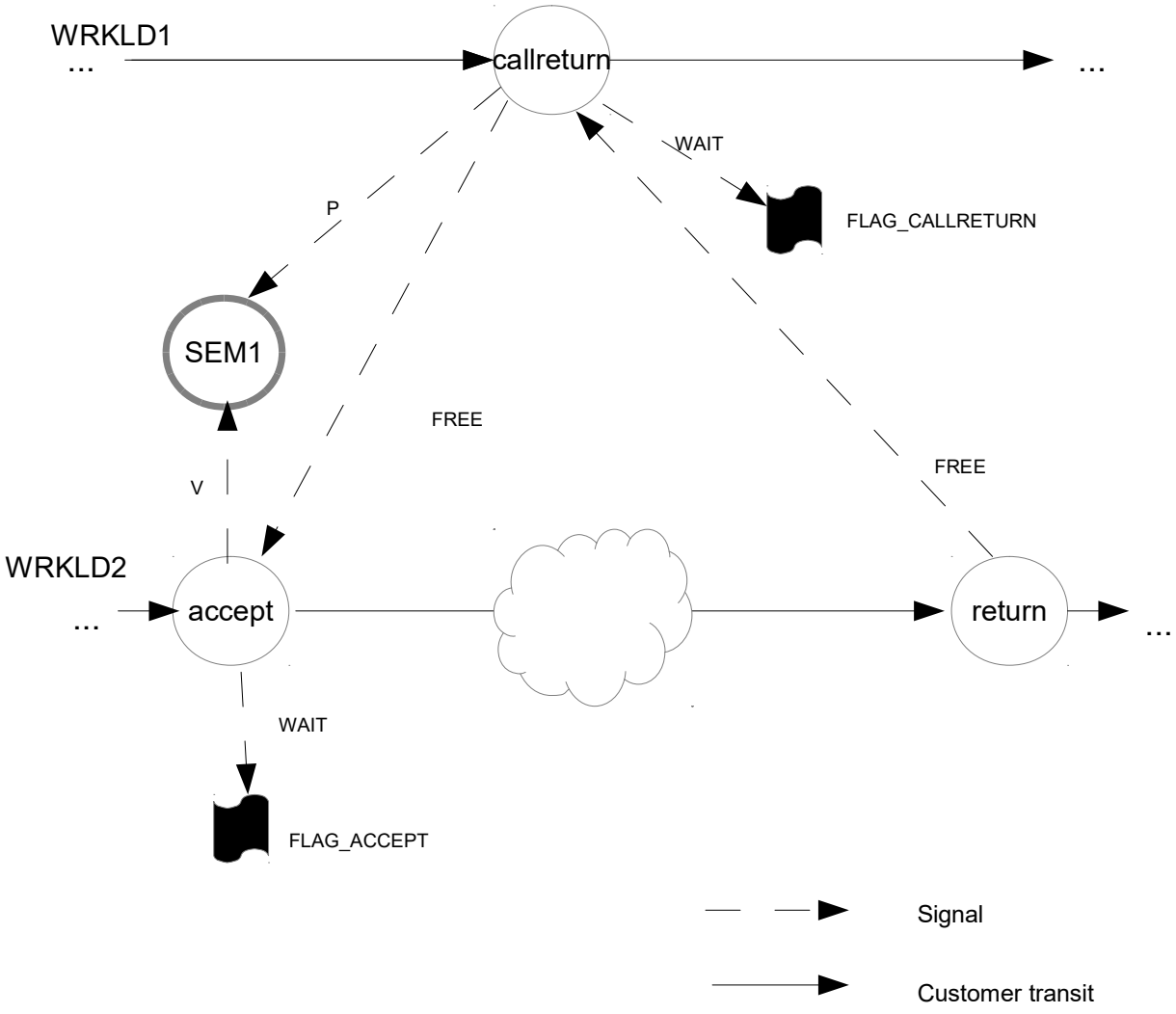

Figure 16 - Callreturn, Call and Return implementation

3.6.8. Token – Request / Release / Create / Destroy

A token is represented by a semaphore. If the *initAvailable* attribute is 1, a V operation is performed on the semaphore at the beginning of the simulation in order to make available the token. Otherwise, the token needs to be created before it can be requested. The creation of a token (Create command) is performed by a V operation and its destruction (Destroy

command) is done by means of a P operation. Likewise, Request command is implemented by a P operation whereas the Release command is implemented by a V operation.

4. Assumptions and restrictions

In this section, the list of assumptions? and restrictions the transformation is based on is given (refer la frase).

- OpenWorkload:

  - **arrivalRate** and **arrivalDistribution** are incompatible.

  - (1/**arrivalRate**) is the parameter of an exponential distribution.

- ClosedWorkload:

  - **thinkTime** and **thinkTimeDistribution** are incompatible.

  - **thinkTime** is the parameter of an exponential distribution.

- ActiveService:

  -**serviceTime** and **serviceTimeDistribution** are incompatible.

  - **serviceTime** is the parameter of an exponential distribution.

- Distributions:

  - *Exponential*, *Hyperexponential*, *Uniform*, *Erlang* and *Constant* distributions can be used directly in QNAP's Service command. *Normal* and *'other'* distributions need a variable to generate the service time and then it's set as a constant service time (CST).

- Service:

  - **sequenceNumbers** can be non-consecutive. This doesn't change the result of the ordering.

  - Services without **sequenceNumber** are executed after services with **sequenceNumber.**

- Wait (command):

  - If there are several clients waiting for an *event* or a *buffer*, the chosen client will be the one with highest priority whereas if all the clients have the same priority, the client will be chosen following the FCFS rule.

- PassiveEntity:

  - **initAvailable** is the number of free elements in the *buffer/resource/message* at the start of the simulation.

  - **quantity** is the number of units to *get/put allocate/deallocate*.

- ForkJoin nodes:

  - When **willJoin** attribute is *true*, two stations are created in order to avoid a TRANSIT/SPLIT incompatibility. Thus, the first station acts as a 'splitter' station and the MATCH command is performed in the second station where the TRANSIT command routes the customers to the next station. Without the 'splitter' station all the commands (SPLIT, MATCH and TRANSIT) would be performed in the same station and QNAP would ignore the SPLIT because of the presence of the TRANSIT command.

  - When **willJoin** attribute is *true*, the SPLIT operation doesn't create a customer that represents the 'father' of the fork, because it's not needed (it would just wait in the MATCH station).

  - When **willJoin** attribute is *true*, the sons created by the SPLIT operation cannot change its workload class, i.e. there can't be a **TransitPlus** with a *newWorkload* atribute in the son's route.

  - The fork operation can be specified by any kind of ServiceRequest (ServiceRequestPlus, TimeServiceRequest, DemandServiceRequest, WorkUnitServiceRequest) but the service time/demand time/ passive and active services are disregarded (*at least for now*).

  -There can be nested forks. To differentiate whether a son goes back to the station where the MATCH is performed or whether it goes to a new 'splitter' station (for a new fork), we look for the existence of a ServiceRequest for the Fork Join node and the son's workload class.

- ForkWorkload:

  - If there are several **Transit** for a given ForkWorkload and its **quantity** attribute has a value greater than 1, a potentially large number of destination combinations should be generated by the Acceleo template, along with the probability of each specific combination, which is a complex task. To solve this problem in a simpler fashion, a new station is created and each customer generated by the fork operation (determined by the **quantity** attribute) is send to this station and then it is routed to the next station applying the **Transit** information found in the ForkWorkload.

- ServiceRequestPlus:

  - Two stations are created, one for **Active Services** and the other for **Passive Services**. The reason for these two stations is a QNAP limitation regarding blocking operations when the scheduling policy is PS.

  - There can't be two consecutive Active Services (it makes no sense), at least a Passive Service must be in between.

- Transits:

All the transits in a **TransitNext** or **TransitFirst** are of the same type, there can't be **Transit** and **TransitPlus** mixed. In case of TransitPlus, all the **depRouting** attributes must obviously have the same value.

- General:

  If there are attributes in the source element that are not needed to generate the target element (i.e. a **quantity** attribute when a Passive Service's **command** is *callreturn*), these attributes are disregarded.

5. To-do

- Find out how to represent FCFSRS and FCFSR in QNAP.
- Define how Timer and Mailboxes work. The mailboxes' messages haven't been implemented yet.
- Transform PMIF+ experiments.