C h a p t e r 4

IV. PROPOSED MODEL

Section IV.1 - Introduction

In the previous chapters we have given an overview of the thesis background. First, we have presented the motivations of the thesis. Then we focused the work by presenting the concrete problem we are facing and particularly those aspects of more interest for us. In addition, we have reviewed the requirements we expected from the solution of the problem. We have also described several research initiatives that have already explored similar fields and that have proposed solutions from different points of view.

Along this chapter, we present the design of the solution we are proposing. More specifically, we conceive a policy-based management framework for the management of heterogeneous active, programmable and passive networks where policies will be expressed in XML [W3C00]. The reason for choosing XML is that it is becoming a de-facto standard for structured documents as policies. Moreover, XML has numerous advantages in the specific case of network and systems management. The possibility of using freely available validators for checking XML policies against XML Schemas before parsing is a powerful tool. Moreover, this facility might also be used for delegating management functionality. Furthermore, the separation of information represented in XML from the presentation, transmission and storage it makes it a flexible solution for the specification of policies. Finally, the use of XML eases the fulfilment of the portability and interworking requirements of the framework.

In this thesis, we have based the model description in the Unified Modelling Language (UML) diagrams [OMG01] accompanied with the necessary explanatory text to ease the comprehension of these diagrams. Accordingly, the overall functionality of the framework will be initially described with a group of supported use cases. Each use case will be afterwards further described with sequence and activity diagrams. Finally, we will provide detailed class models for each of the framework components and explain how such models develop the tasks expected from the component. The reason for choosing UML for the description of the proposed framework is that the UML has become the standard for system's description and specification. Thereby, its diagrams are widely used and understood. Hopefully, this will ease the readability and comprehension of the proposed framework.

The chapter is structured in four sections. After this introduction, we will provide an overview of our proposal to cope with the requirements and we will detail the functionality supported by the framework through the definition and description of several use cases. In the third section we will go into details in the description of the solution. Particularly, we will present in depth each framework component, its expected functionality, its interfaces and its interrelations with other components. The chapter is concluded with a summary outlining the solution.

Section IV.2 - Use cases description

After the definition of the system goals and requirements, we have already a clear idea of the expected functionality. In this section we suggest a general solution and describe, through use cases, how it copes with the functionality specified.

The solution is presented as a set of interacting components realising the system's functionality. A different set of components structured in a different way could have also served our purposes as long as they fulfilled the same functionality. Nevertheless, the components presented enclose the system's functionality in a structured and logical way. These components can be seen in Figure 4 - 1. The Policy Editor component contains all policy reception and edition functionality. The main policy-processing functionality is enclosed within the Policy Consumer Manager component that carries it out with the support of its surrounding components. These components are the Authorisation Check, the Policy Conflict Check, the Traffic Engineering Manager, the Decision-making Monitoring system, the Database and finally, the device-dependant components like the Policy Consumer, the Monitoring Meter and the SigDemux.

The Authorisation Check Component will verify that the user introducing the policy has the necessary access rights to realise that action. The Policy Conflict Check component has the responsibility of assuring the consistency of all policies introduced in the system and that two or more policies never request the same resources at the same time. For this reason, the Traffic Engineering Manager that will organise and assign network-wide resources to policy requests will be deeply interrelated with the Policy Conflict Check component. Decisions about when policies must be enforced are closely linked with policy condition monitoring that will be co-ordinated by the Decision-making Monitoring system component. The device-dependant components (Policy Consumer, Monitoring Meter and SigDemux) will carry out the policy enforcement, policy condition monitoring and signalling requests processing tasks over the underlying devices respectively. Finally, the Database component will give support to all others by storing Information Model Objects (IMOs) that will provide a picture of the system's status at any time.

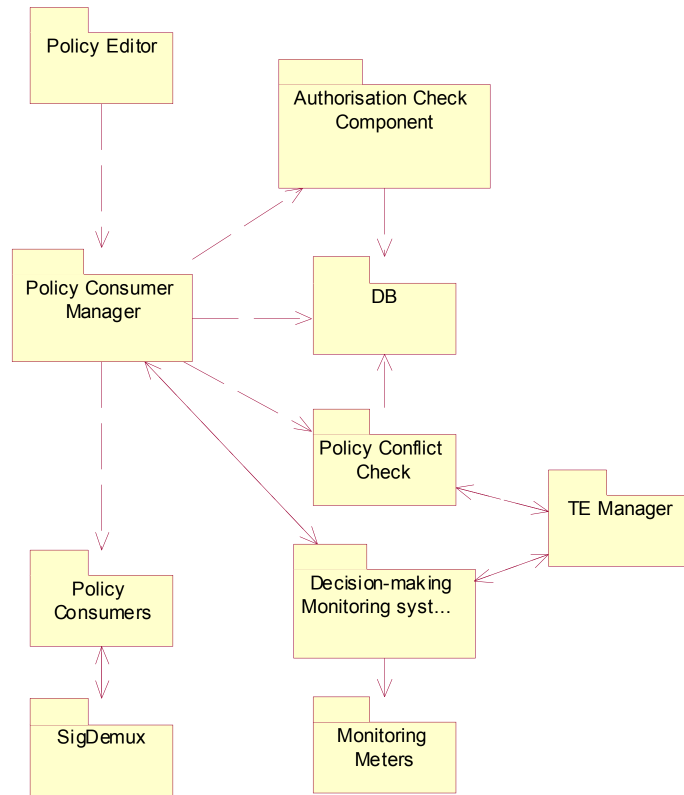

Figure 4 - 1. MANBoP framework

The behaviour of our management framework can be described with three main working modes or use cases. These three modes are started by different triggers. In short, the main use cases of the framework are policy-triggered, signalling-triggered or event-triggered. In addition to the use cases that describe these three working modes, the framework offers other capabilities and functionalities represented in three additional use cases, i.e. the bootstrapping, the add node and the remove node use cases. The use case diagram below shows these use cases and their relation with the framework environment.

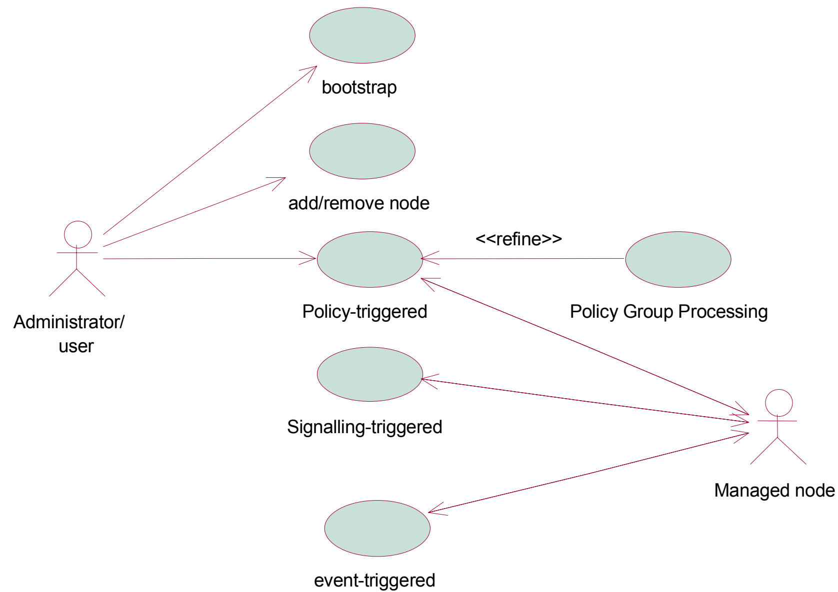

Figure 4 - 2. MANBoP Use case diagram

Each of these use cases will be described in detail in the following sub-sections. Nevertheless, before proceeding to their exhaustive description it is worth explaining when the three mentioned working modes apply.

In case the MANBoP management framework is working as an element-level management station, the three working modes can potentially coexist. The same applies if the framework is working as a network level management station, or sub-network management station, directly over the managed devices (without element level stations). The situation is not the same when the framework is working as a network-level management station with other lower-level management stations (i.e. sub-network or element). In such a case, the signalling-triggered working mode would not apply.

Note that an additional use case appears in the diagram. This is the Policy Group Processing use case consisting of a specialisation of the Policy-triggered use case. As such, it will be described within the Policy-triggered use case sub-section.

The use cases description covers only the basic functionality of the framework and its components. However, there are many other smaller tasks that, though important, would only be described inside the section of the component that realises them.

1st Policy-triggered Use Case

The working mode described in this use case is the most common of the three. Using IETF terminology for policy-based management [Westerinen01] it would coincide with the "provisioned policy" model.

Although the "provisioned policy" concept is similar to the ideas in the policy-triggered use case, they are not exactly the same. They have in common the fact that both cover the tasks that deal with policy processing (e.g. decision-making, policy enforcement…). The difference is that the "provisioned policy" approach from the IETF only covers the policy processing of those policies that will result in a managed device configuration when the conditions of the policy match. In contrast, the policy-triggered use case covers the policy processing of all policies in the framework, even if they are oriented to decide on signalling requests received afterwards. Hence, both the signalling-triggered and the event-triggered use cases need that the involved policies (i.e. those determining the signalling decisions to be taken, or those containing event-based conditions) are previously introduced in the framework through the policy-triggered use case.

To ease the description of the main tasks of this use case we provide below a high-level activity diagram.

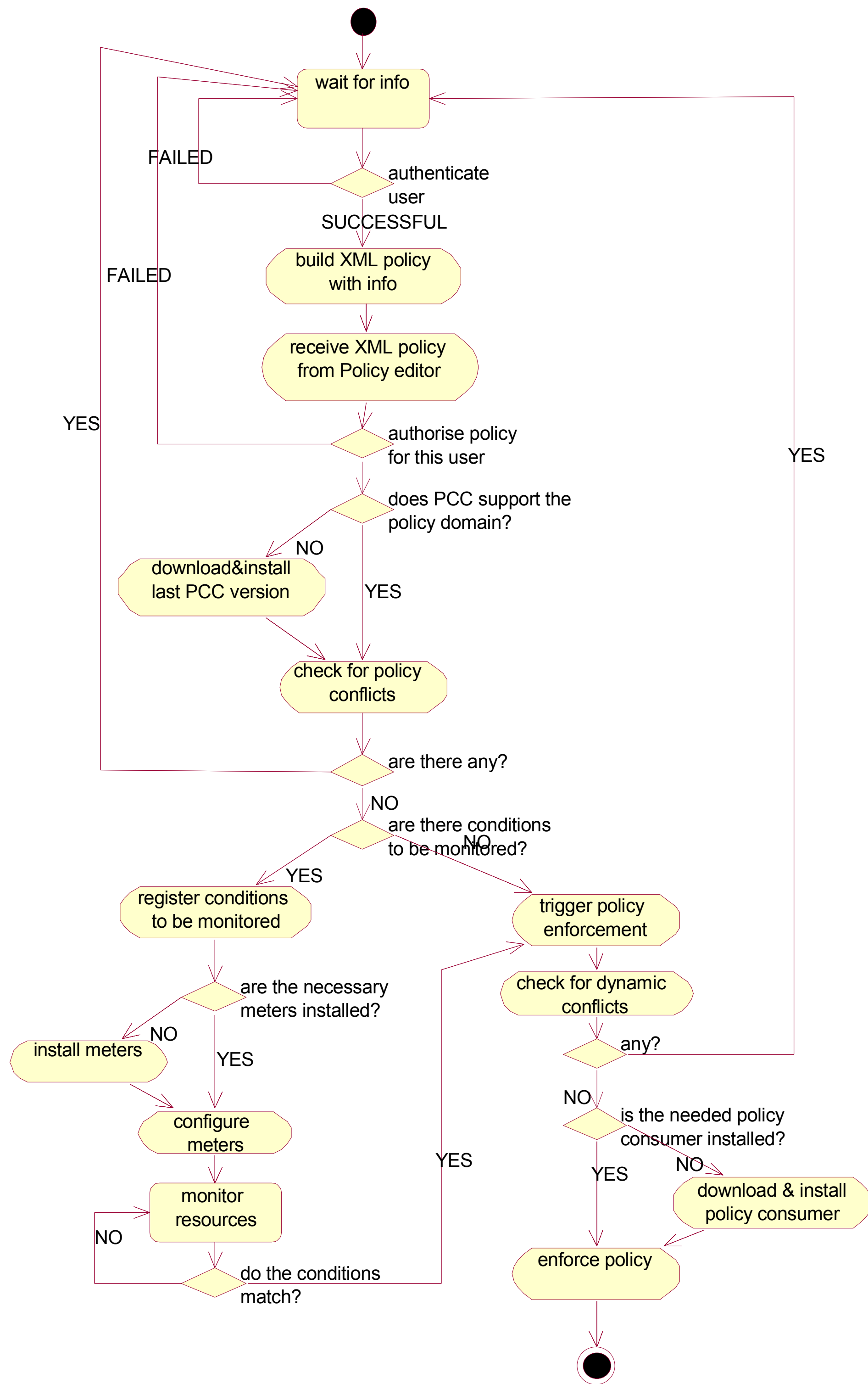

Figure 4 - 3. Policy-triggered activity diagram

The description of the activity diagram is split into four parts, namely the Policy Introduction, Policy Checking, Decision-making and Policy Enforcement.

Policy Introduction

In the activity diagram above, the policy information is introduced through a Graphical User Interface (GUI) within the Policy Editor. Nevertheless, the policy-triggered use case can also be initiated by the arrival of a policy within an active packet or from a higher-level management application.

The possibility of deploying policies using active packets adds more flexibility in the capabilities for the distribution of policies. It permits fast distribution of policies to the nodes where they should be enforced and even to link the distribution of policies with the network or node status. Additionally, it makes unnecessary to have a detailed knowledge of the network topology to distribute policies. The framework here presented should configure the managed devices to receive these policies, via active packets, correctly. The code that receives these active packets and forwards their policies to the management framework is not covered in this thesis.

Also service-level management applications according to the TMN layered structure [ITU00b], can introduce policies, or policy information, through an API given by the Policy Editor component. This allows the framework to be part of a complete management stack of applications.

Once the information is introduced, the first task realised by the framework is to authenticate[1] the user who is accessing the framework. The authentication task is realised by a special object within the Policy Editor component. Such an object-oriented approach allows the substitution of an obsolete authentication object with a new version.

The next step is to build an XML policy making use of the received information. Again, this task is developed within the Policy Editor component.

Policy Checking

The XML policy is then sent to the Policy Consumer Manager (PCM), which acts as core and coordinator of the tasks carried out within the framework. First, the XML policy is checked against the access rights of the user who is introducing the policy. That is, the policy information is compared against 'what management actions', and 'with which parameters' statements specifying the user rights. These authorisation checks are realised by the Authorisation Check Component (ACC) within the framework. For carrying out such tests, the Authorisation Check Component takes into account user information previously introduced in the framework.

[1] This thesis is not oriented to study the concrete security mechanisms; hence, they will only be introduced.

After the authorisation checks, and only in case they have been successful, the next step is looking for possible conflicts between the policy introduced and the rest of policies in the system. The first thing to be done within this step is to check whether the version of the component responsible for this task, the Policy Conflict Check (PCC) component, supports the new policy. Since the system is dynamically extensible with new functionality, it might be the case that the current version of the PCC component installed in the system cannot process the new policy. In that case, the Policy Consumer Manager will request to the Code Installing Application (CIA)[2] the installation of the newest version of PCC component. Such version should support the new incoming policy. Once we know that the Policy Conflict Check component is capable of realising the checks, the policy is forwarded to it. If any conflict arises, the PCC component will try to solve it based on mechanisms such as policy priorities. When the conflict cannot be solved, an exception is raised, in any other case the processing of the policy goes on.

Decision-making

The next task carried out by the Policy Consumer Manager is to check whether there are conditions that must be monitored to decide about the policy enforcement. If there are no conditions to be monitored the Policy Consumer Manager starts the enforcement process. If there are one or more conditions to be monitored, they will be registered by the PCM component in the Decision-making Monitoring system (DmMs). When the conditions are fulfilled, the DmMs contacts the Policy Consumer Manager to request the enforcement of the corresponding policy.

The Decision-making Monitoring system looks for the meters needed to monitor the registered conditions. If these Meter components are not installed, it requests their installation to the CIA. Each Meter informs the Decision-making Monitoring system when one of the conditions matches its value. Only when all the conditions of a policy are fulfilled the DmMs contacts the Policy Consumer Manager.

Policy enforcement

The Policy Consumer Manager prior to the policy enforcement does two tasks. The first one is requesting to the Policy Conflict Check a dynamic conflict check between the policy that will be enforced and policies currently enforced in the managed device. Unless a conflict is found, the PCM will proceed to the second task. In this one, the Policy Consumer Manager looks for the Policy Consumer that should enforce the policy. In case it is not already installed, it contacts the CIA to request its downloading and installation.

[2] The CIA is considered within this thesis as an external system.

Finally, the policy is forwarded to the Policy Consumer that enforces it on the managed device.

To ease the comprehension of what happens in the framework when developing these tasks, a sequence diagram containing the main interactions is given below:

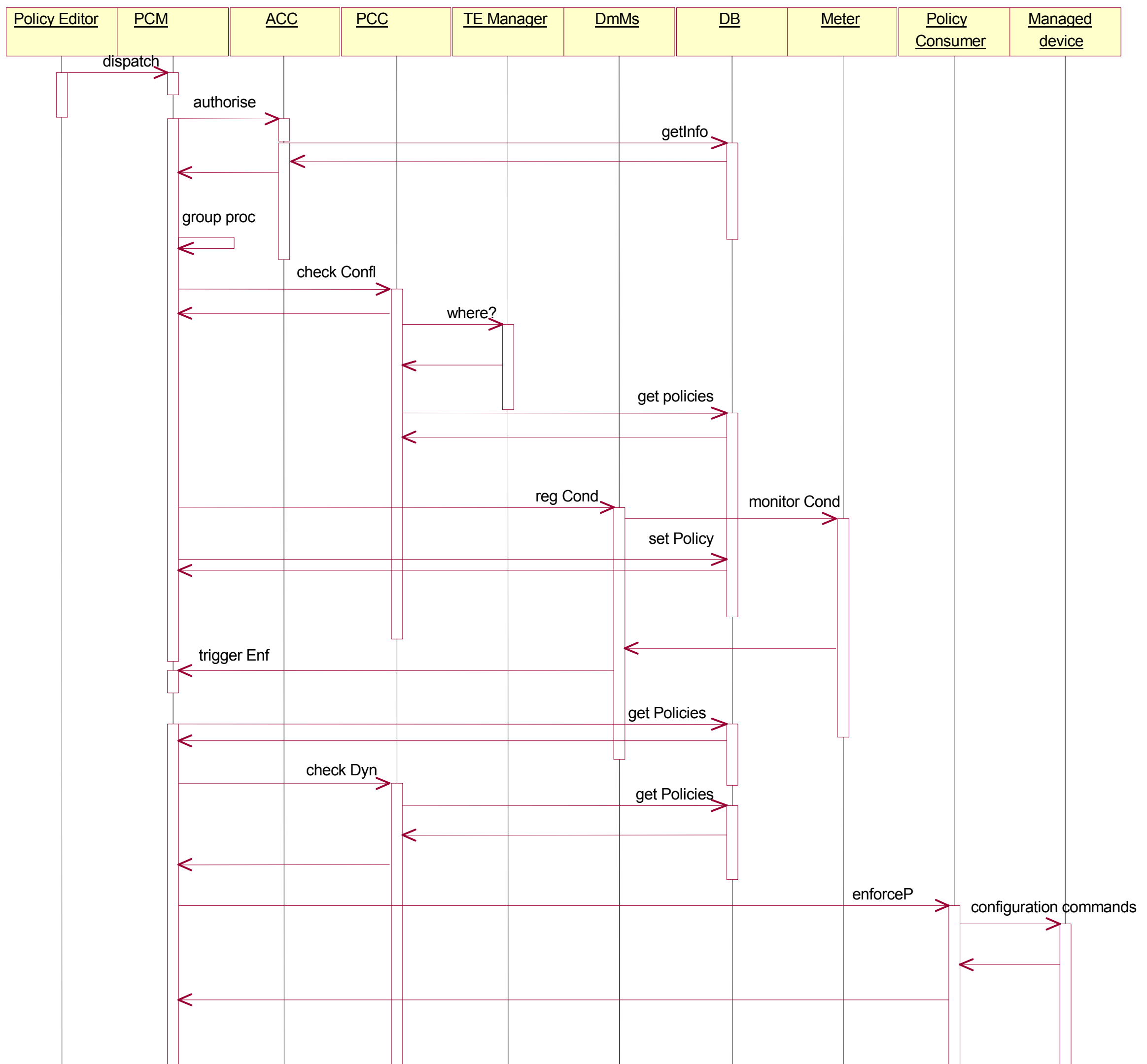

Figure 4 - 4. Policy-triggered sequence diagram

In the context of the sequence diagram of figure 4.4 is worthy to consider in detail the Traffic engineering manager activity and the Policy Group Processing task.

A Traffic-engineering

The sequence diagram of figure 4 – 4 shows a call to the TEManager component: “where?”. Such a call would only be realised by the Policy Conflict Check (PCC) component running at the network level. Using this method the PCC component asks the TEManager where in the network the processed policy can be enforced, based on the requested resources, network status, topology and other variables that will be described in more detail in the section describing the TEManager component functionality. The PCC uses this information to achieve efficient conflict checking at network level.

The routing information obtained is stored in the Database (DB) component together with the policy. When the policy must be enforced, this information, together with the policy, is forwarded to the corresponding Policy Consumer, so that it only configures the appropriate managed entities (or sub-network/element management stations responsible of them).

The reason for being the Policy Conflict Check the component making this call to the TEManager, instead of the Policy Consumer Manager component, is because the PCC is able to extract the information needed by the TEManager from the policies. The Policy Consumer Manager cannot develop such a task since it treats the policies as abstract entities; in other words, it is not capable of extracting information from conditions or actions of a policy.

Another possible candidate to call to TEManager would be the Policy Consumer. The problem in this case is that we would not be able to detect any resource conflict between policies until the policy is being enforced. This would degrade the system performance since someone introducing a policy in the management framework would not know if it is applicable until the policy enforcement time.

Nevertheless, it might happen that the PCC component need not to contact the TEManager when receiving the policy for the first time, but only when that policy should be enforced. This happens in policies with unpredictable conditions, such as policies based on alarms or performance degradations. These policies require some route calculation based on the available resources at their enforcement time. For example, the policy: if ‘congestion’ then ‘re-route flow X’ may never be enforced, since the condition might never be fulfilled. Moreover, if the conditions are ever fulfilled the new route should be calculated taking into account the network status at enforcement time.

Sometimes, these kind of policies aim to provide a backup route for a flow, to guarantee the bandwidth in case the main route falls. Such backup route could be pre-calculated upon policy arrival to the management framework or calculated only when the conditions are met. Each alternative has its advantages and drawbacks:

- Pre-calculated route: The advantage of this alternative is that it might, in theory, guarantee that the requested resources would be available

for the backup route. The main drawbacks are that the scheduling of resources would be inefficient, since there may be resources reserved for situations that might never occur; and that in congestion situations the guarantee might not be enough to assure resources in the pre-calculated route.

- Route calculated at enforcement time: The advantage of this approach is that it assures that it will find the optimum route for that flow at enforcement time. On the other hand, it might happen that at enforcement time there are no resources for the new route.

The management framework suggested in this thesis supports both alternatives, each of them being represented by policies from different functional domains. Additionally, the framework includes a conflict solution mechanism based on priorities. In consequence, in those cases when the TEManager component is not able to find a route with enough resources, it returns the minimum cost route based in a particular criterion (e.g. less number of hops, more bandwidth available…). Taking into account this information, the conflict solution algorithm inside the PCC component detects how many policies conflict with the new one in that route and solves these conflicts allocating the resources to the request with the highest priority. Obviously, the users whose requests have been removed will be reported immediately.

The MANBoP framework gives to the network operators more flexibility to specify the desired behaviour of the network, and to specify in the network the quality of service negotiated with its customers. It is the responsibility of the network operator to provide the agreed level of quality of service to each customer, setting and controlling the policies within the framework.

In conclusion, the Policy Conflict Check component needs to be smart enough to know when does it need to contact the TEManager and how to deal with the information it returns.

B Policy group processing

Also in the sequence diagram, we notice the presence of another task within the Policy Consumer Manager component: 'group Proc'. This name stands for the Policy Group Processing task.

An special situation within the policy-triggered working mode occurs when processing policy groups. A policy group as already stated is a set of policies that need to be processed in a concrete manner: atomically, sequentially, the first successful, etc.

The MANBoP management framework has been designed to enable the processing of policy groups. This capability adds more flexibility to the specification and deployment of policies and allows better determining the expected behaviour of managed entities. For example, a service provider might require several node resources in order to offer an active service to its

customers. These resources should be reserved in several policies that would form a policy group. Such a policy group should be enforced atomically because a single unreserved resource disables the service thus making unnecessary the reservation of the other resources.

The activity diagram that follows includes the tasks that cope with the policy group processing functionality.

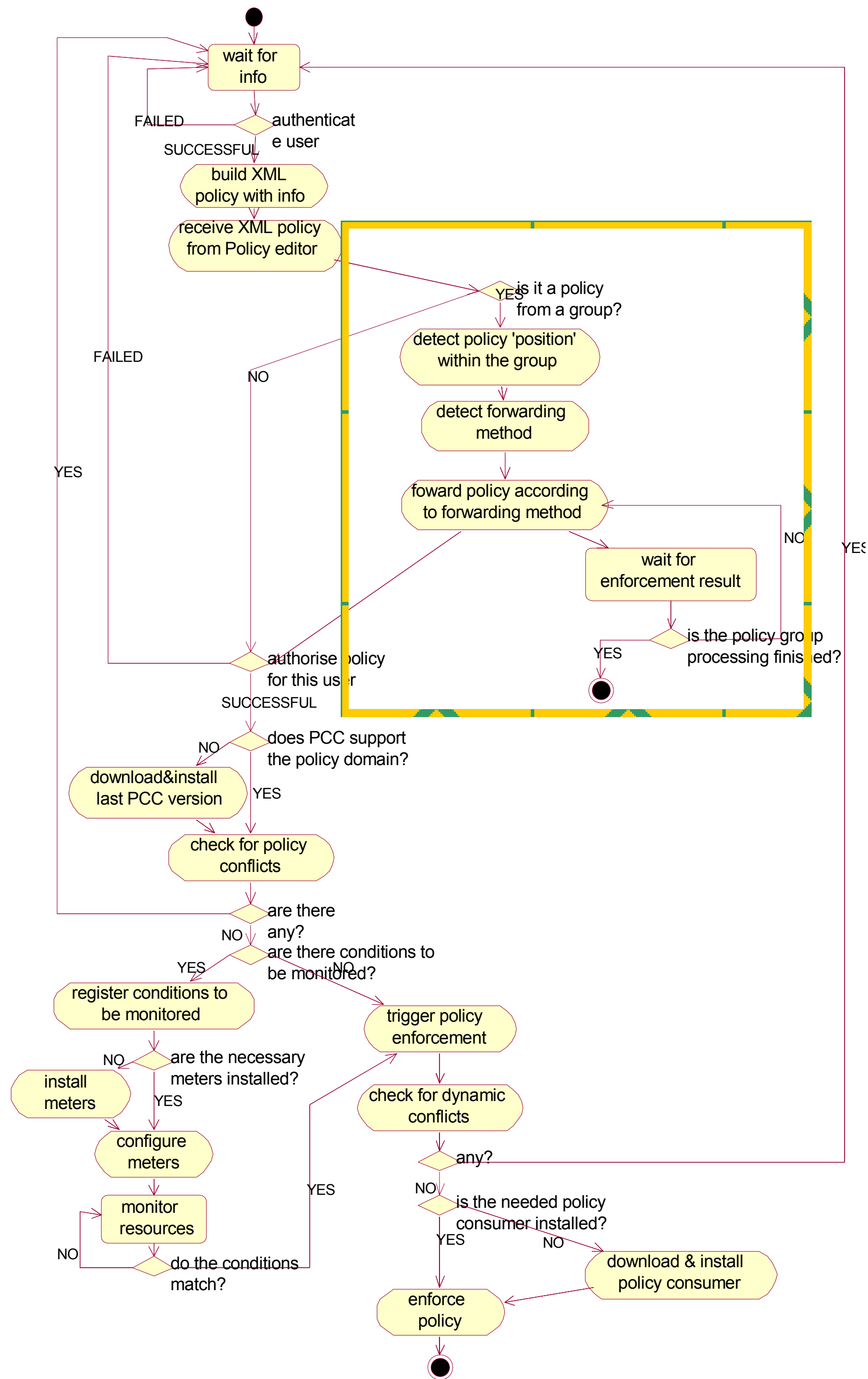

Figure 4 - 5. Policy Group Processing Activity diagram

The tasks enclosed within the square are those specific for policy group processing.

First of all, the system must detect whether the received policy is part of a policy group or it is an individual policy. When the latter, the policy would be processed as explained in the previous sub-section. Otherwise, we should identify the policy group, the processing status of that policy group, and the 'position' of that policy within the policy group. With 'position' we refer to the logical placement of the policy taking into account the forwarding method used. For example, in case it is a 'sequentially' forwarding method, the policy position value will determine the particular enforcement order of the policies within the group. Based on this information, the corresponding policy is processed as explained in the previous sub-section. The policy enforcement result will be used as feedback information to control the forwarding process.

Sometimes the impossibility of enforcing one of the policies of the group causes that others previously enforced should be now removed. When this happens, the system uninstalls these policies removing all configuration actions that might have been carried out as result of their enforcement.

Since all group processing functionality is realised within the Policy Consumer Manager component, we do not include the sequence diagram for this 'sub-use case'.

2nd Signalling-triggered Use Case

The working mode covered in this use case is the equivalent to the "outsourced policy" model using IETF terminology.

In this working mode, the management station acts as an access control entity. It mainly allows or denies resource reservation requests coming from the managed entity based on the policies available on the network at that time. The most common example for the outsourced policy approach is the usage of policies together with the Common Open Policy Service (COPS) [Durham00a] protocol and RSVP [Braden97] (Resource ReSerVation Protocol) described in RCF 2749 [Durham00b].

The framework supports this outsourced policy model, or signalling-triggered working mode, re-using as much as possible the functionality and tasks already developed for the policy-triggered use case. The objective is to make the framework as lightweight as possible and avoid duplicating functionality unnecessarily.

As in the previous use case description, we will base the explanation of the signalling-triggered working mode in its activity diagram shown below.

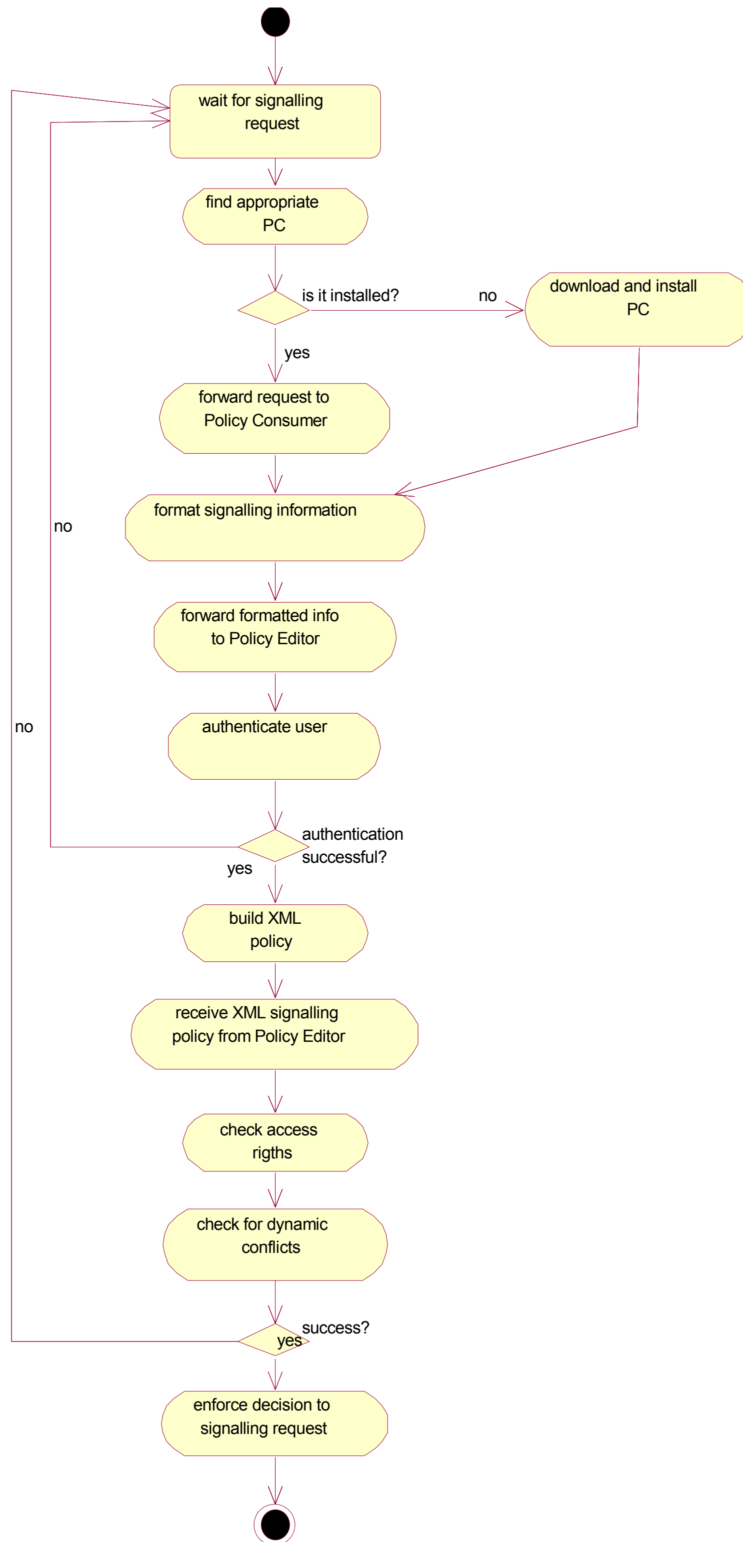

Figure 4 - 6. Signalling-triggered use case Activity Diagram

As above-mentioned, in this working mode the management framework receives the requests from the managed device. The component that receives the request is the SigDemux. When the request is received, the first task that must be realised by the SigDemux is to find the most appropriate Policy Consumer component to process that request. Each Policy Consumer will be able to process signalling requests of a particular functional type (e.g. COPS, proprietary…). The SigDemux should be able to detect which type is needed and, eventually, request the downloading and installation of the corresponding Policy Consumer.

The Policy Consumer receives the information in the request itself. Then, it formats this information before its introduction it in the Policy Editor. At this point, the process is quite similar to the policy-triggered use case except for two particularities. These are, that there are no conditions to monitor, since the decision has to be made immediately, and that the Policy Consumer is already installed. Hence, the tasks carried out are: first of all, the authentication of the user on whose behalf the request has been made. Immediately afterwards, if the authentication has been successful, an XML policy is built based on the information coming in the request and forwarded to the Authorisation Check component.

At this stage the access control is carried out; that is, the signalling request information is checked against the access rights information[3] of the user on whose behalf the request has been raised. Unless the authorisation check fails, since there are no conditions to be registered and the Policy Consumer is already running, the final decision over the request is only pending on the dynamic conflict checks. These checks are realised by the Policy Conflict Check against all other policies currently enforced in the system. If this final check is also successful, the Policy Consumer is asked to enforce the request.

Obviously, if any of the checks along the process fails, the Policy Consumer involved is notified and the request is rejected.

To bring more light into the whole process an explanatory sequence diagram is given below.

[3] The access rights information is introduced in the system by means of delegation policies as will be seen later on this document.

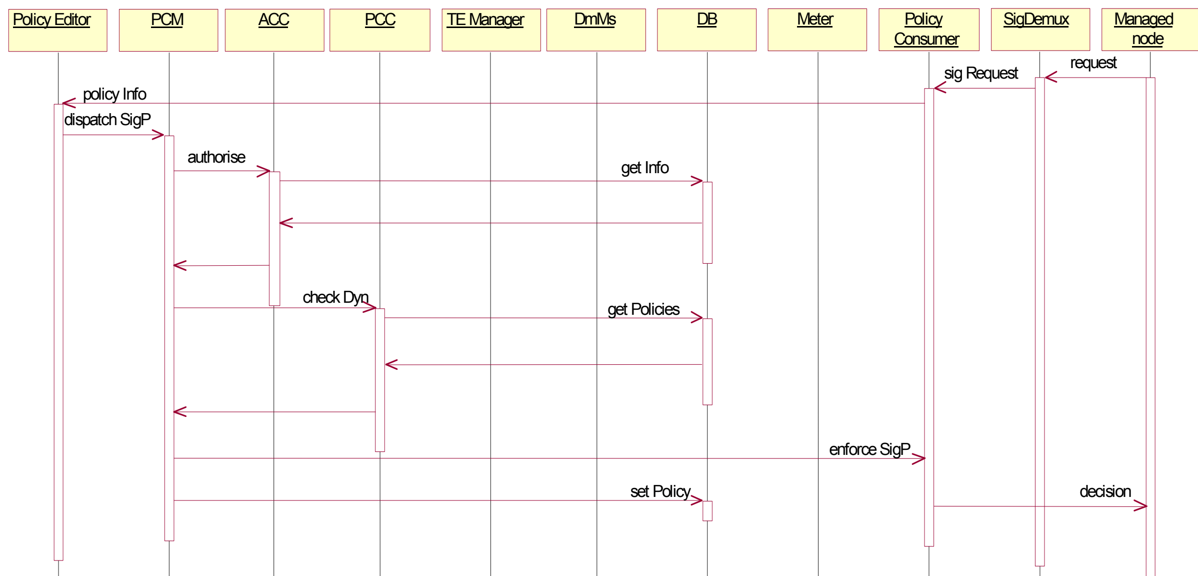

Figure 4 - 7. Signalling-triggered use case Sequence Diagram

3rd Event-triggered Use Case

The event-triggered use case presents the main tasks within this working mode. The event-triggered working mode is not explicitly foreseen in the IETF Policy framework.

The reason for including event-triggered functionality within our framework is to explicitly support automatic changes of the behaviour of the managed entity based on faults or performance events. This allows for a faster reaction to problems and a more autonomous management. In this way, when a fault or performance degradation occurs, we can determine and control which corrective actions should be taken in the managed entities.

These tasks can also be developed by continuously polling the appropriated variables, but it would result in a less efficient approach. The reason is that it would require more management traffic and the delay until the problem detection would be bigger.

This working mode can be seen as a special case within the policy-triggered working mode because the only difference resides in the type of monitoring information needed (pooled values or events), or to be more accurate, in the nature of the information to be monitored in that way. Faults and performance degradation are unexpected, infrequent and more suitable for event-based monitoring. On the other hand, configuration and security information happens to be more suitable for pooling.

Although the tasks expected for this use case are the same as those described within the policy-triggered use case after the conditions are met, we have included it as a separate use case to particularly highlight this capability of the framework. Nevertheless, to avoid redounding in the same information again, we will simply refer to the activity and sequence diagrams, and their description, given in the policy-triggered use case section.

4th Bootstrap Use Case

The bootstrap use case presents the tasks developed by the framework to initiate its main components with the appropriate initial parameters. These parameters are those needed by the components of the framework for identifying the functionality expected from them and for requesting the installation of other components when necessary. It is also important the information about the managed resources that allows framework components to make appropriate decisions as for example managed nodes, link capacities, node capabilities, etc.

The processes involved in the appropriate introduction and use of this information are those described in this use case.

The activity diagram shown below introduces the main tasks.

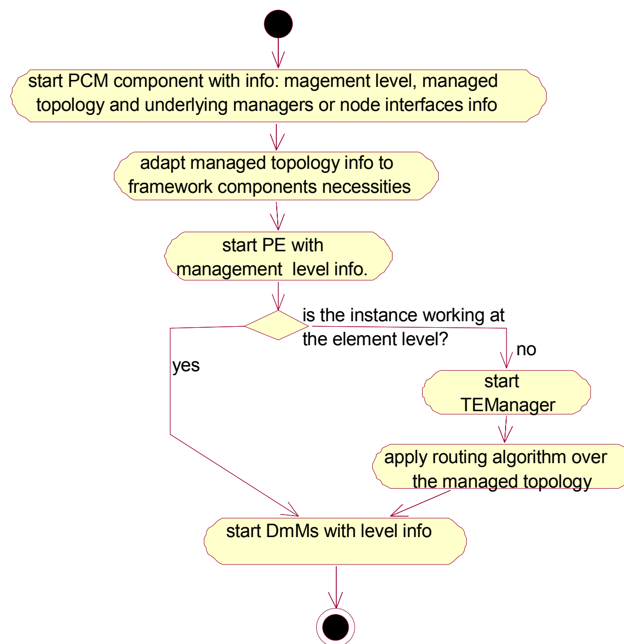

Figure 4 - 8. Bootstrap use case activity diagram

As can be seen in the activity diagram of figure 4 – 8, the core component in the bootstrapping of the framework is the Policy Consumer Manager. It is the responsible of receiving the initial information, creating the object instances that reflect the managed topology and starting other components.

The information received by the PCM is:

- *The management level at which the instance is acting:* This data specifies the management level at which the instance will be working [ITU00b], and thus, the expected behaviour of that instance. Hence, the information is used to select the most appropriate component modules to be installed when required. Possible values for this parameter are: network level, element level, network over element level and network over subnetwork level[4].

- *Managed topology information:* This information is mostly needed by the traffic engineering algorithm running inside the TEManager, when running at the network or sub-network levels, and by the Policy Conflict Check component. In the first case, the information is used to calculate routes with the requested resources. The route information is used by the Policy Conflict Check component as input. On the other hand, the Policy Conflict Check component uses the managed topology information to find, and if necessary resolve, the resource conflicts that might exist. The given information must be all that is needed by the TEManager and PCC components for realising their respective tasks. For example: IP address, node identifier, link capacity…

- *Underlying managers or node interfaces info:* The last information introduced at the bootstrapping of the framework is related with the underlying devices. It can be either information about MANBoP instances running under the current instance in the management infrastructure or information about device interfaces directly managed by the instance being booted. Obviously, the type of information supplied depends on whether this instance will run directly over managed devices or instead, over other lower-level MANBoP instances. The underlying devices information is used by both the DmMs and the PCM for requesting the installation of the most appropriate Monitoring Meter or Policy Consumer respectively.

The Policy Consumer Manager component, after the correct reception of the booting information, instantiates the corresponding Information Model Objects (IMOs) based on the received information. These instances are stored in the database from where they can be retrieved by other

---

[4] A subnetwork instance of the MANBoP framework will be instantiated as network, network over element or network over subnetwork. Since its expected behaviour will be the same as for a network instance, though just over a subset of the managed topology.

components. A detailed description of these objects will be given in the Information Model section. After creating the appropriate instances, the PCM initiates the rest of the framework with the corresponding data. In particular, it will start the Policy Editor component with the “management level” information, so that it can be extended appropriately to receive new policy information at that level. In addition, the DmMs will be started with the “management level” information which is kept in an attribute of this component. As already mentioned, this information, together with the underlying topology information obtained from the IMOs stored in the database, is used for the correct election of the Monitoring Meters to be installed and for the registration of Notification Services as explained below. Finally, the TEManager is only started if the MANBoP instance being booted works at the network or subnetwork levels.

Afterwards, other components of the framework such as the Authorisation Check Component or the Database will be equally started, although they do not need any initial data.

All nodes in the management infrastructure will be booted in the same way. The management infrastructure should be instantiated starting from the lower levels and ending with the upper ones. At the bootstrapping a simple synchronisation process (which is out of the scope of this thesis) with the lower-level MANBoP instance would be advisable. Additionally, the Notification Service included as part of the Decision-making Monitoring system, should register as event consumer on the Notification Services of the lower-level instances at bootstrap. For that reason, when started, the DmMs takes into account the management level at which it is acting so as to retrieve from the database, when working over other MANBoP instances, the underlying managers location.

At the time when the first policy is introduced in the management infrastructure, all MANBoP instances should be up and running, since the different interactions for the decision and enforcement of a policy will occur between them.

5th Add/Remove node Use Case

I have grouped these two use cases into a single section because they have obvious common aspects. Indeed, the tasks that should be done in the Add node use case should be undone in the Remove node one.

The interest of these capabilities in the framework is justified by the most likely progressive deployment of active nodes inside legacy IP networks. Thus, in order to avoid the need of changing or re-initiating the whole management infrastructure every time a node is added or removed in the network, the framework includes the capability to add or remove nodes dynamically to the managed topology.

Nonetheless, to keep the whole process feasible, certain conditions should be taken into account:

- If the addition of a new node requires a new MANBoP instance for its management (e.g. an element manager) then the new instance will be booted, as described in the previous use case, by the network administrator.
- The same applies if the removal of a node leaves one MANBoP instance unused.
- A node can only be removed as long as there are no policies, or reservations, applicable to that node.
- In case there are one or more framework components (e.g. Policy Consumer or Monitoring Meter) that were installed in the management station exclusively for the removed node, they will be removed dynamically via the default lifecycle logic for these components.

Figures 4 – 9 and 4 – 10 show the activity diagrams for the addition and removal of a node to the managed topology:

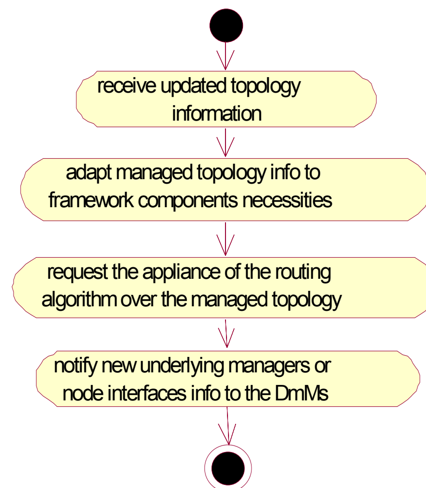

Figure 4 - 9. Add node activity diagram

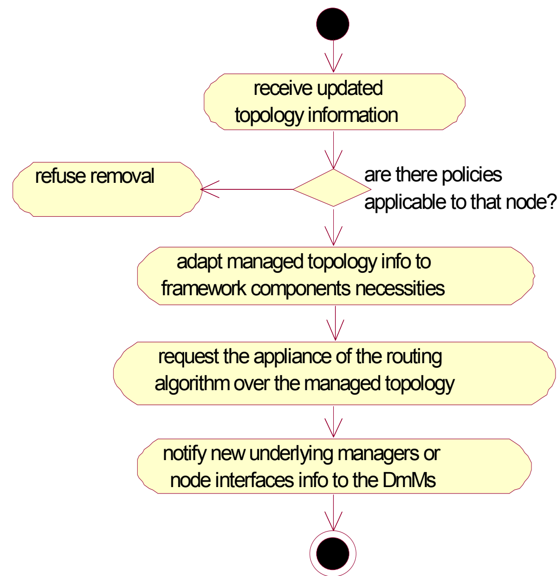

Figure 4 - 10. Remove node activity diagram

As can be seen in the activity diagrams both use cases need the realisation of the same activities, although in opposite sense. The only change is that in the remove node use case, before any removal task is done, a check is executed to verify that there is no policy applied, or that should be applied now or in the future in that node. This check is realised by the Policy Consumer Manager accessing the information stored in the Database component.

The rest of tasks are also controlled by the PCM, which requests the different activities to the corresponding components. In particular, it first maps the managed topology information to the IMOs that will be accessed by the framework components. Then, it requests to the TEManager to run the routing algorithm over the new managed topology to establish the paths and their costs.

Finally, the PCM component will also contact the DmMs to notify the addition or removal of a new node. Then, the DmMs when working over other MANBoP instances will request to the Notification Service to register, or unregister as event consumer in the new underlying MANBoP instance.

6th Summary of components and tasks

In order to summarise the ideas described above, and organise the tasks described in components of the management framework, we provide below a table mapping tasks with components and interfaces. This table pretends to be useful both as a fast checkpoint of the framework functionality as well as a preliminary guide to the component description of the next section.

Task	*Requested*		*Interface used*
	To	*By*	
Policy-triggered			
Wait for Info	Policy Editor	user	GUI
Authenticate user		Policy Editor	Internal interface
Build XML Policy with Info		Policy Editor	Internal interface
Authorise policy	Authorisation Check Component	Policy Consumer Manager	authorise()
Check for dynamic conflicts	Policy Conflict Check	Policy Consumer Manager	checkConfl()
Check for Policy Conflicts		Policy Consumer Manager	checkDyn()
Receive XML Policy from Policy Editor	Policy Consumer Manager	Policy Editor	dispatch()
Does PCC support the policy domain?		Policy Consumer Manager	Internal interface
Are these conditions to be monitored?		Policy Consumer Manager	Internal interface
Is the needed Policy Consumer installed?		Policy Consumer Manager	Internal interface
Trigger policy enforcement		Decision-making Monitoring system	triggerEnf()
Register conditions to be monitored	Decision-making Monitoring system	Policy Consumer Manager	regCond()
Are the meters installed?		Decision-making Monitoring system	Internal Interface
Do the conditions match?		Decision-making Monitoring system	Internal interface
Configure Meters	Meter	Decision-making Monitoring system	monIS()
Monitor resources		Meter	Internal interface
Download and install last PCC version	Code Installing Application	Policy Consumer Manager	dwCode()
Download and install meters		Decision-making Monitoring system	dwCode()
Download and install Policy Consumer		Policy Consumer Manager	dwCode()
Enforce Policy	Policy Consumer	Policy Consumer Manager	enforceP()
Policy Group processing			
Is it a policy from a group?	Policy Consumer Manager	Policy Consumer Manager	Internal interfaces
Detect policy 'position' within the group			
Detect forwarding method			
Forward policy according to forwarding method			
Is the policy group processing finished?			
Wait for enforcement result		Policy Consumer	event
Signalling			
Wait for Signalling request	SigDemux	Managed device	event
Find appropriate Policy Consumer		SigDemux	Internal interfaces
Is it installed?			
Download and install Policy Consumer	Code Installing Application	SigDemux	dwCode()

Send formatted info to Policy Editor	Policy Editor	Policy Consumer	policyInfo()
Authenticate user		Policy Editor	Internal interfaces
Build XML policy			
Receive XML Signalling policy from Policy Editor	Policy Consumer Manager	Policy Editor	dispatch()
Check Access Rights	Authorisation Check component	Policy Consumer Manager	authorise()
Check for dynamic conflicts	Policy Conflict Check	Policy Consumer Manager	checkDyn()
Forward request to Policy Consumer	Policy Consumer	SigDemux	sigRequest()
Format signalling info		Policy Consumer	Internal interface
Enforce decision to signalling request		Policy Consumer Manager	enforceP()
Bootstrapping			
Start PCM component with info	Policy Consumer Manager	Network Administrator	main()
Start PE with management level info	Policy Consumer Manager	Policy Consumer Manager	Internal Interface
Is the instance working at the element level?			
Start TEManager			
Start DmMs with level info			
Adapt managed topology info to framework component necessities			
Apply Routing algorithm over the managed topology	TEManager	TEManager	Internal Interface
Add/Remove node			
Receive updated topology information	Policy Consumer Manager	Network administrator	addN()
Adapt managed topology info to framework component necessities		Policy Consumer Manager	Internal Interface
Request the appliance of the routing algorithm over the managed topology	TEManager	Policy Consumer Manager	updateTop()
Notify new underlying managers info or node interfaces info to the DmMs	Decision-making Monitoring system	Policy Consumer Manager	upUnI()
Remove node [5]			
Receive updated topology information	Policy Consumer Manager	Network administrator	removeN()
Are there policies applicable to that node?		Policy Consumer Manager	Internal Interface
Refuse removal			

Table 4 - 1. Table of components and interfaces

[5] Only the additional tasks in relation with the Add Node use case are included in the table.

Section IV.3 – Description of the MANBoP components

1st Policy Editor

A Component Behaviour

From the previous sub-section we can extract and summarise the tasks and interfaces that the Policy Editor must offer. Namely, the tasks previously listed for this component are:

- *Wait for info*: Provide a GUI that can be utilised by a user (e.g. a network operator or service provider) to introduce policies for managing his resources. The interface for this task is the GUI itself.
- *Authenticate user*: The Policy Editor should authenticate all users that try to enter the system, either by means of the GUI or by any other means that we will comment later on this sub-section. Since authentication is an internal task of the Policy Editor component, the interface is not accessible from the outside.
- *Build XML Policy with info*: Also, the Policy Editor must format the received information, in case it is not already received with the correct format [6] to be forwarded appropriately to the Policy Consumer Manager component. As in the previous case the corresponding interface is not accessible from the outside.
- *Send formatted info to Policy Editor*: This task of the signalling-triggered use case describes another behaviour to receive information by the Policy Editor component; that is, from Policy Consumers dealing with a signalling request. The information received will start the decision-taking processes. The interface offered by the Policy Editor component for such cases is the policyInfo() interface. The concrete parameters of the interface and their justification will be provided along this sub-section.

Besides the above tasks obtained directly from the tasks enumerated in the use cases descriptions, there are other tasks and interfaces that the Policy Editor must offer and that have not been yet explicitly commented.

The first one is the possibility of receiving XML policies. Such policies should only be authenticated before being forwarded to the Policy Consumer Manager. This capability will be provided through the recvXPolicy() interface. Such an interface will allow the framework to receive XML policies from higher-level management applications (e.g. an element level policy coming from a network manager) or policies deployed by means of active packets.

[6] The Policy Editor can also receive XML policies.

There are many other tasks defining the properties and capabilities of the Policy Editor, however, they are all internal tasks of the component. Hence, they determine the behaviour of the component but they do not modify the external interfaces offered by this component. These functionalities are dictated by the requirements imposed both to the Policy Editor (i.e. receive, authenticate and format information to be understood by the framework) and to the framework as a whole (e.g. support of dynamic extensibility of functionality, flexible management infrastructure…).

We will show the expected behaviour of the framework components using UML activity diagrams. Specifically, for the Policy Editor three activity diagrams will be provided, one for each external interface offered by the component (i.e. GUI, policyInfo() and recvXPolicy()). Each activity diagram will be described in a separate sub-section, namely GUI, Signalling and XML.

a GUI

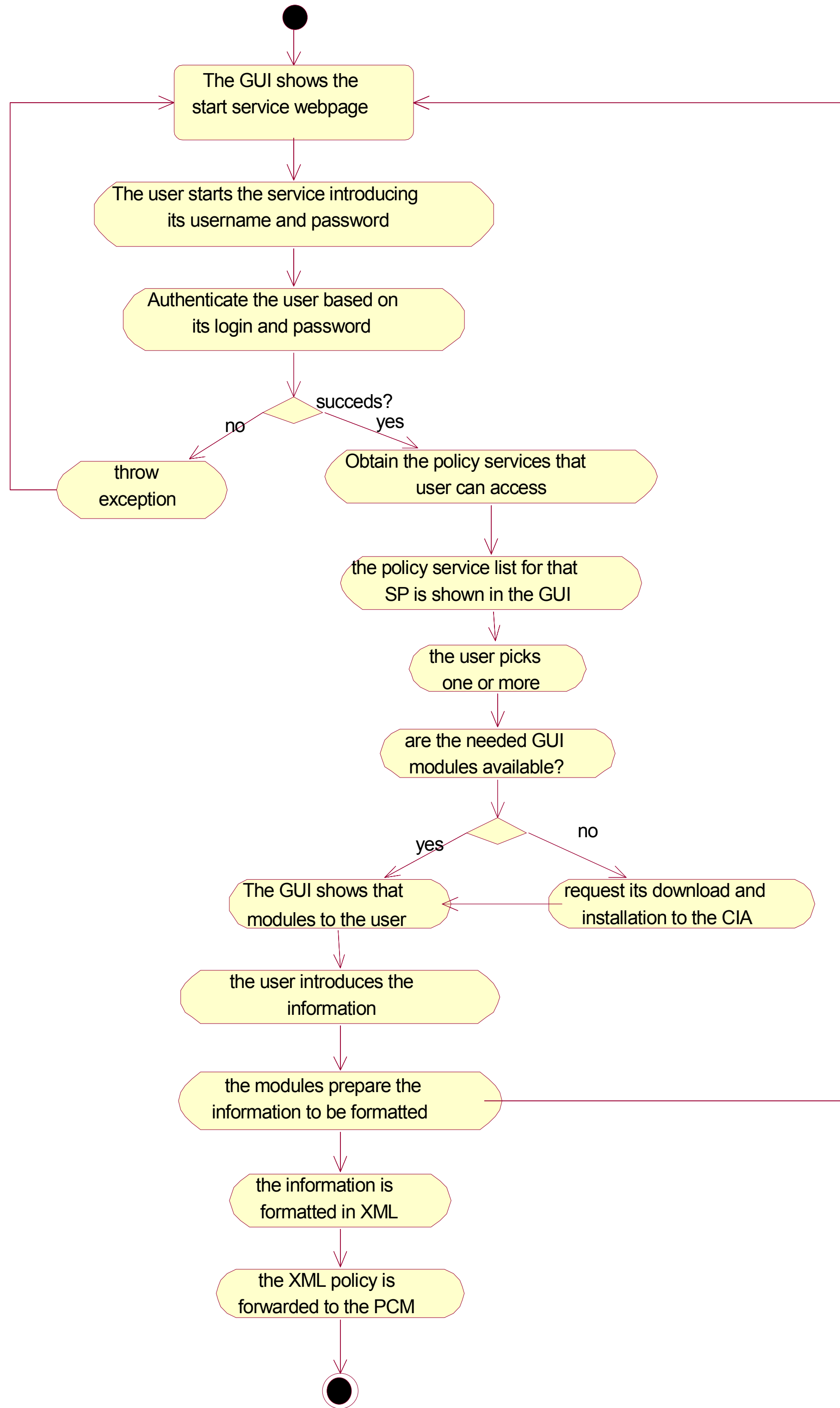

Figure 4 - 11. GUI-initiated activity diagram

The first activity diagram introduced for describing the expected behaviour of the Policy Editor is the *GUI-initiated activity diagram*. As easily deducible from the title, this diagram specifies the expected behaviour of the Policy Editor component when a user wants to manage its resources through the GUI.

In order to start the process, the user first introduces his login and password in the GUI for initiating a session. This information will be used by the component to authenticate the user. We have opt for a such a simple authentication method instead of a more elaborated one, because the focus of this thesis is the framework as a whole and not the concrete security algorithms.

If the authentication of the user succeeds, the component must now identify what policy services or functional domains is the user allowed to access. These services will then be listed in the GUI to allow the user to pick one or more of them. The list of policy services accessible for a user (e.g. service providers) is set by the management infrastructure administrator (e.g. network operator) after the negotiation of the corresponding Service Level Agreement. This process can be seen as an initial discrimination of access rights for that user, as a low-granularity authorisation check complemented by the high-granularity authorisation check realised afterwards.

Once the user has chosen the policy services, the Policy Editor has to check whether the GUI modules that will permit the user to introduce the corresponding information for those policy services are installed. In case it receives a negative answer, it requests to the Code Installing Application (CIA) the download and installation of the necessary GUI modules before proceeding to the next step.

In the next step, the user introduces the information in the GUI modules, which on their turn, prepare the received information to be formatted in XML and finally, send it to the Policy Consumer Manager (PCM) component.

b *Signalling*

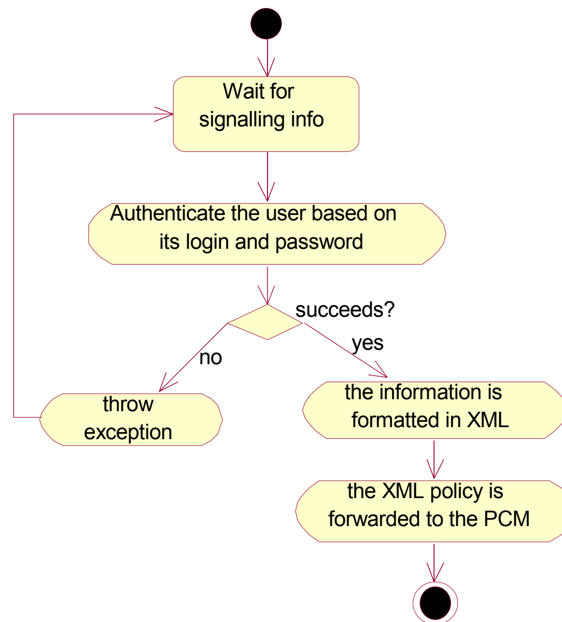

Figure 4 - 12. Signalling-initiated activity diagram

When the information arrives from a signalling request forwarded by a Policy Consumer, many of the previous tasks of the Policy Editor component are not needed. A clear example is all GUI related processes, which are obviously skipped. Also, since the Policy Consumer component that has forwarded the signalling request has already prepared the information to be formatted, this step can also be skipped within the Policy Editor. Thereby, only the three tasks shown above specify the behaviour of the component for signalling support.

First, the signalling information forwarded has to be authenticated. The framework checks whether the user on whose behalf the request is made has privileges to access the framework. Only when succeeding in the authentication, the signalling information received will be formatted in XML and forwarded to the Policy Consumer Manager for deciding about the request.

c *XML*

When the Policy Editor receives an XML policy from an active packet or a higher-level management application, its expected behaviour would be as simple as just authenticating the user on whose behalf the XML policy has been sent and forwarding the policy to the Policy Consumer Manager component. It is obvious that other tasks of the Policy Editor such as GUI related processes or XML formatting processes do not apply in this case.

The simplicity of the component behaviour in this case makes unnecessary its representation in an activity diagram that we have omitted.

B *Component Design*

From the component behaviour described in the previous sub-section, we can extract and divide the expected functionality in four main groups of tasks. These are:

- GUI-related processes: Processes needed to provide to a user a pleasant graphical interface to manage his resources.
- Authentication processes: Verify that the user who is 'entering' the management framework has rights to do so.
- Information-formatting processes: Related with the creation of the corresponding XML policies with the information received.
- Coordination processes: The core functionality of the component, that is, to coordinate the different tasks to be developed based on the type of request received.

Consistently with the four main groups of tasks, the component design is based on four classes, each of them developing one of the task groups. These classes and their interfaces are shown in the class diagram below.

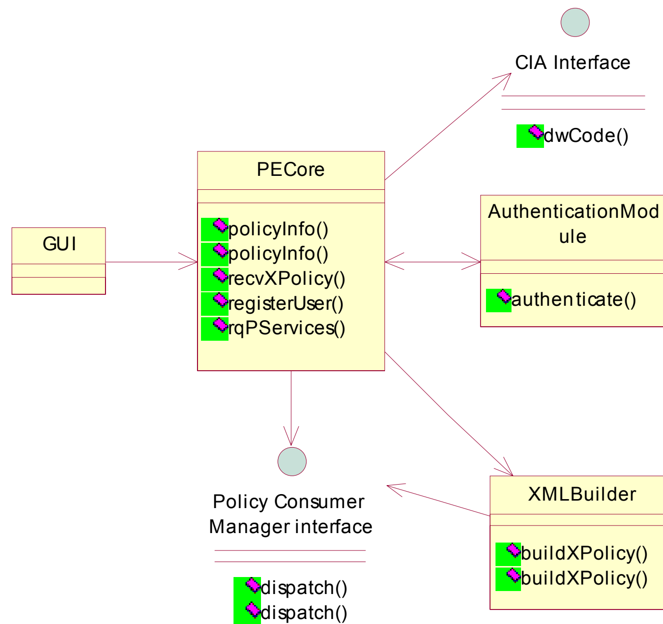

Figure 4 - 13. Policy Editor component class diagram

a PECore class

The PECore class is the main class within the Policy Editor component. It coordinates the whole component behaviour using the GUI, XMLBuilder and AuthenticationModule classes. It also makes use of the CIA to request the download and installation of GUI modules, when necessary, as briefly stated before.

The class offers five public methods, namely: policyInfo() (overloaded in two methods with different input parameters), recvXPolicy(), registerUser() and rqPServices(). The input and output parameters as well as the functionality within these methods are listed and explained in the table below:

Interface	Input parameters	Output parameters	Functionality
registerUser()	credential User	string[] PolicyServices	The GUI uses this method for retrieving the policy services that the user is able to access, and list them on the screen. The 'User' parameter identifies the user who introduces the information. 'Credential' is a structured type that contains both the username and password. As return parameter 'PolicyServices' is a list of strings one per policy service accessible by that user. When receiving a call to this method the PECore class will first request to the AuthenticationModule the checking of the credential for that user. If successful it returns the policy services this user can access. It retrieves the policy services for each user from a local table.
rqPServices()	string[] PolicyServices	handle[] GUIModule	After the selection of policy services by the user, the GUI uses this method to request the handles of the needed GUI modules for these policy services. The 'PolicyServices' parameter lists the services chosen by the user. The returned parameter is an array of handles[7], one per each GUI module. The PECore class, when receiving a call to this method, requests the download and installation of the needed GUI modules, in case they are not already installed, and returns the handles of all modules to the GUI.
policyInfo()	string Info, credential User	-	This method is used to introduce the policy information coming from the GUI in the framework and to create the corresponding XML policies out of it. The 'Info' parameter is the information itself that will be, afterwards, formatted in XML. Finally, the 'User' parameter identifies the user who introduces the information. When receiving a call to this method the PECore class will forward all parameters to the buildXpolicy() method of the XMLBuilder class to be formatted and forwarded to the Policy Consumer Manager. The user would have been previously registered and authenticated in the framework using the registerUser() method above. Thus, there is no necessity to repeat the authentication process at this point.
policyInfo()	string sigRqId, string Info, credential User	-	The previous method has been overloaded to introduce the information of a signalling request to take a decision. The new parameter added is the 'sigRqId' parameter. It is used for identifying the request to which a decision should be applied. The other two parameters are left unchanged. When receiving a call to this method, the PECore class will first request to the AuthenticationModule the checking of the credential. If succeeded, all parameters are forwarded to the buildXpolicy() method (which is also an overloaded method) of the XMLBuilder class to be formatted and forwarded to the Policy Consumer Manager. In this case the authentication is needed because the information arrives from the Policy Consumer, and the involved user would not have been previously authenticated.
recvXPolicy()	credential User, string XPolicy	-	Used to introduce XML policies coming from higher-level management applications or even from active packets. The 'User' parameter is used for authenticating the principal that sends the policy. The policy is included in a serialised form in the Xpolicy parameter. When receiving a call to this method the PECore class will request to the AuthenticationModule the checking of the credential. If successful the XML policy will be forwarded to the Policy Consumer Manager through its dispatch() method. Both 'Xpolicy' and 'User' will be passed as parameters of this method.

Table 4 - 2. PECore interface description table

[7] The concrete handle type depends on the implementation

b AuthenticationModule class

The AuthenticationModule class develops the authentication tasks within the framework. The current design is generic enough for supporting several types of authentication algorithms. The 'credential' structured type can be adapted to the necessities of the chosen algorithm and the AuthenticationModule itself can be replaced with a newer version thank to the modular design of the framework. Nonetheless, as previously justified, the authentication algorithm we have considered as proof of concept is simply based on usernames and passwords.

The AuthenticationModule class offers only one public method, namely the authenticate() method. In the table below we can see the input and output parameters as well as the functionality expected in this method.

Interface	*Input parameters*	*Output parameters*	*Functionality*
authenticate ()	credential User	boolean Result	This method is used by the PECore class to request the authentication of a user. As input parameter the credential of the user is submitted. The output parameter is a boolean that indicates the result of the authentication: 'true' (success), 'false' (fail). When a call to this method is received, the AuthenticationModule will apply the authentication algorithm over the credential. This class may utilise user information, such as credentials, associated to users in a local table or database.

Table 4 - 3. AuthenticationModule interface description table

c XMLBuilder class

The XMLBuilder class realises all information formatting tasks. It mainly formats the information received into XML policies that will be afterwards forwarded to the Policy Consumer Manager (PCM) component.

The class offers, through its interface, one overloaded buildXPolicy() method with two possible input and output parameters. These methods will always be accessed by the PECore class and will result in the forwarding of the XML Policy to the PCM component. The concrete description of the XMLBuilder interface is given in the table below:

Interface	Input parameters	Output parameters	Functionality
buildXPolicy()	credential User, string Info	-	This method will be used by the PECore class to request the creation of an XML policy based on the 'Info' parameter. The 'Info' parameter provides the policy information introduced by the user. The 'User' parameter, as in the previous cases, provides the credentials for that user. When receiving a call to this method, the XMLBuilder class will try to create an XML policy using templates and the received information. In case it does not succeed, an exception would be raised. If successful, the XML Policy together with the credential will be forwarded to the PCM component.
buildXPolicy()	string sigRqId, credential User, string Info	-	The only difference with the previous method is that, in this case, the forwarded information to the PCM component is the XML policy, the credential of the user and the 'sigRqId' parameter.

Table 4 - 4. XMLBuilder interface description table

d GUI class

The GUI class includes all the functionality needed to interact with the user by means of a graphical interface. This functionality can be split in:

- Mechanisms to graphically represent information.
- Mechanisms to recompile user information through the graphical interface.
- Mechanisms to communicate the received information to the PECore class.
- Mechanisms to dynamically show in the graphical interface new installed GUI modules (e.g. JAVA applets) that would have been requested by the user.

All these mechanisms are realised internally inside the GUI class therefore, no public method is offered by this class to the other classes inside the Policy Editor component.

e Sequence diagram

For easing the comprehension of the Policy Editor component we include below the sequence diagrams with the interactions between the classes for all possible behaviours of the Policy Editor component. These sequence diagrams complement the activity diagrams given before for the description of the component expected functionality.

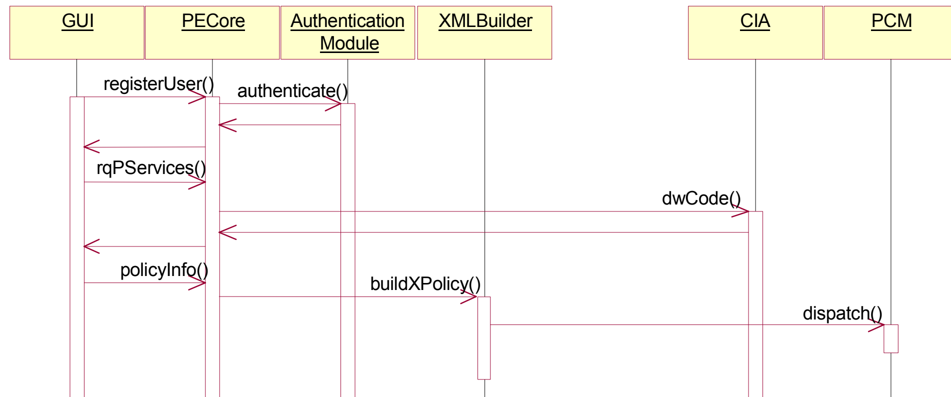

Figure 4 - 14. GUI-initiated Policy Editor behaviour sequence diagram

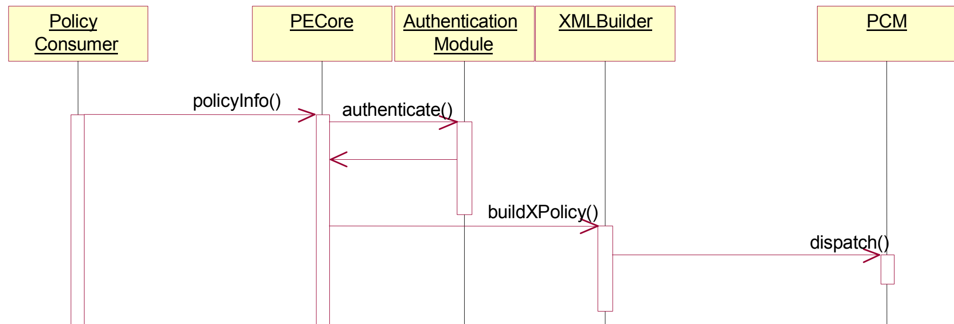

Figure 4 - 15. Signalling-initiated Policy Editor behaviour sequence diagram

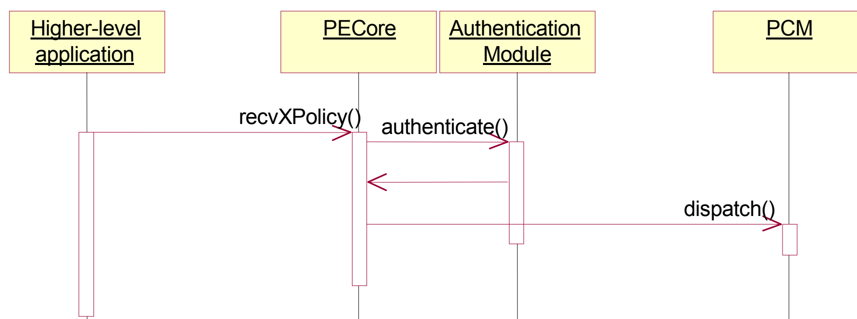

Figure 4 - 16. XML Policy-initiated Policy Editor behaviour sequence diagram

2nd Policy Consumer Manager

A Component Behaviour

Taking into account Table 4 - 1, the main tasks and interfaces expected from the Policy Consumer Manager component are:

- *Receive XML Policy from Policy Editor*: This task is realised through the dispatch() interface offered by the component. There is no real algorithm behind this task, just the ability of this component for correctly receiving policy information from the Policy Editor.

- *Does the PCC support the policy domain?:* The PCM must find out the current version of the Policy Conflict Check component installed in the system, and the functional domains that this version supports. In case the version does not support the functional domain of the policy under process, the Policy Consumer Manager will also request the download and installation of the newest version of the PCC component, which must support this functional domain, and probably even newer ones. No external interface is offered by the component for realising this task.

- *Are these conditions to be monitored?*: The PCM component must detect whether the conditions within the policy under process should be monitored to take a decision about the enforcement of the policy, or instead the policy enforcement must start immediately. As in the previous case, this is an internal functionality of the component and hence, no external interface is linked to this functionality.

- *Is the needed Policy Consumer installed?*: This task is quite similar to the second one. The aim is to check whether the appropriate Policy Consumer component, for the policy under process, is installed. In case it is not, the PCM component requests the download and installation of the corresponding Policy Consumer component. The Policy Consumer requested is identified not only based on the functional domain to which the processed policy belongs, but also based on the position of the management station within the management infrastructure. That is, the Policy Consumer for a QoS domain working at the network level over element management stations in the management infrastructure, will certainly be different from the Policy Consumer for the same domain if there weren't element management stations in the infrastructure. During the bootstrapping of the system the position of the instance within the management infrastructure is introduced. The Policy Consumer Manager component has to take into account this information, not only for the downloading of Policy Consumers but also for the downloading of Policy Conflict Check components. The Decision-making Monitoring system must also take into account its position

within the infrastructure to request the downloading of the most appropriate Meter components.

- *Trigger policy enforcement*: The Decision-making Monitoring system launches this task through the triggerEnf() interface to inform the PCM component that the conditions of a particular policy have changed their status. Therefore, the policy enforcement (or removal) of that policy should be initiated. When triggered, it initiates and coordinates the processes realised until the actual forwarding of the policy to the Policy Consumer component (in charge of actually enforcing the policy on the managed device). These processes are mainly two: request to the Policy Conflict Check component a check for dynamic conflicts against other enforced policies, and to check that the appropriate Policy Consumer is installed as described above.
- *Is it a policy from a Policy Group?*: The Policy Consumer Manager checks whether each processed policy pertains to a policy group. In case it does, the PCM will initiate the policy group processing as described in the Policy-triggered Use Case sub-section. Otherwise, the policy is processed normally. This is an internal functionality of the PCM component, thus it is not linked to any external interface.
- *Detect policy 'position' within the group:* The goal of this functionality is to determine the relative enforcement order, taking into account the forwarding algorithm, or Policy Group Execution Strategy (PGES), chosen. This determines whether the policy must be immediately enforced or, instead, it must be stored to be enforced afterwards; depending on the enforcement result of other policies within the group. For example, if the position is three, the forwarding method is sequential, and the first and second policies have not yet arrived, the policy is stored, until a correct enforcement result of the previous policies of the group arrives. As in the previous case, this functionality is internal to the PCM component and thus there is no external method offered for this functionality.
- *Detect forwarding method:* This functionality complements the previous one. Together, they form the input information needed by the Policy Group Execution Strategy to take a decision of whether the actual policy should be enforced or instead should be stored to be enforced later. The concrete task carried out is to extract the PGES information from the received policy.
- *Forward policy according to forwarding method:* The forwarding algorithm already mentioned a couple of times before is, indeed, represented by this task. This algorithm will use the PGES and policy position information to control the dispatching of the policies from the group. The enforcement result of policies is also needed as input to decide when the group policies must be dispatched.

- *Is the policy group processing finished?*: When receiving an enforcement result from a policy pertaining to a policy group, the Policy Consumer Manager component checks whether the enforcement of that policy concludes the enforcement of the policy group (then, the higher-level application or GUI should be informed), or instead, the Policy Group Execution Strategy of the policy group continues and a new policy must be processed.

- *Wait for enforcement result:* This task represents the ability of the Policy Consumer Manager to receive the enforcement result of policies. When the policy pertains to a group, the arrival of the result might trigger the PGES as described above. The result is notified to the PCM by the Policy Consumer components.

- *Receive XML Signalling policy from Policy Editor:* This task is very similar to the first one. Indeed the method linked to this functionality is also the dispatch() method (overloaded to support different arguments). The particularity of this task in relation with the first one is that in this case the policy logically represents a signalling request. The policy is processed by the framework to take a decision about the signalling request. The Policy Consumer that raised the request links an identifier to the request, so that when it receives the decision it can easily map it with the request. Hence, the Policy Consumer Manager must forward the identifier of the request together with the decision to the corresponding Policy Consumer.

- *Start Policy Consumer Manager component with info*: When a network operator wants to start a MANBoP instance, it first runs the Policy Consumer Manager component with the adequate input information. In particular, this information is the management level at which the MANBoP instance will work, the topology managed by this instance and the managed node interfaces (when running directly over managed resources), or lower-level managers below this instance. The main() method of the PCM component uses this information to bootstrap the system.

- *Start PE with management level info*: One of the booting processes developed by the Policy Consumer Manager is the initiation of the Policy Editor component. The management level at which the instance is working is introduced as parameter in the instantiation of the Policy Editor component, so that it can extend its functionality appropriately.

- *Is the instance working at the element level?:* The PCM checks the value of the management level parameter, since it determines whether the TEManager should be started or not. Only when working at the element level the TEManager component will not be instantiated in the system.

- *Star the TEManager:* The PCM component is responsible of instantiating the TEManager when necessary. During the booting process, the TEManager applies the routing algorithm over the managed topology, retrieving the topology Information Model Objects from the database.

- *Start DmMs with level info:* Another booting task realised by the PCM is the instantiation of the Decision-making Monitoring system with the management level info. The DmMs updates an attribute with this parameter that uses to determine whether the Notification Service should register as event consumer in underlying MANBoP instances.

- *Adapt managed topology info to framework component necessities:* Each time the component receives new managed topology information it has to adapt it to the format required by other framework components. This functionality is internal to the PCM component, hence no external interface is involved.

- *Receive updated topology information:* This functionality covers the ability of the Policy Consumer Manager to receive new managed topology information. This new managed topology information can differ with the old one only in the effects of including (or removing) one more node to the managed topology. In case the MANBoP instance is running at either the network or subnetwork level, the addition or removal of a node might not be directly visible in the new topology (i.e. the access points of the network or subnetwork might be the same), but only in the capacity of the links between the access points. This happens when the added or removed node is not an edge node of the managed subnetwork. The reception of the new topology by the PCM triggers the topology update processes. When a new node is being added, the involved method in the PCM manager is the addN() method. When a node is being removed is removeN().

- *Are there policies applicable to that node?:* Before attempting to remove a node from the managed topology, the PCM component must check if the specified node is used by any policy in the system, enforced or not. This task is realised internally by the PCM using information stored in the Database.

- *Refuse removal:* In case the previous check task determines that there are one or more policies involving this node, the PCM refuses the removal of the node sending an exception.

As one could easily deduce by the amount of tasks listed for this component, the Policy Consumer Manager is the core component of the MANBoP framework. It controls and keeps updated all the functionality within the framework. Nevertheless, there are other functionalities that should be carried out by this component that, although not listed above, are equally important.

The first one is to control the lifecycle of Policy Consumer components. The aim of this functionality is to avoid keeping many Policy Consumers running within the system, which might have not been active for a certain time. In this way, we try to make the whole framework more lightweight, dynamic and autonomous. The criteria to control the lifecycle of Policy Consumers might be diverse e.g. time since the last policy enforcement, presence of policies within the database from the functional domain covered by the Policy Consumer, etc.

The Policy Consumer Manager component is also responsible of keeping the lifecycle of policies. All policies entering in the system will specify a validity period after which they should be removed from the system as well as the existing configurations in managed devices related with that policy. When this validity period expires, the PCM should check and completely remove from the system the policy and its associated configurations.

In addition, the Policy Consumer Manager needs to report to higher-level managers the enforcement result of a policy, as well as additional data related with the enforcement such as resources reserved. To keep this functionality independent of the higher-level management software and allow different applications to subscribe for this kind of information, we will use a CORBA-like notification service. That is, the Policy Consumer Manager component will send 'enforcement-result' type of events to the Notification Service, which will forward them to the applications that have requested their reception. More details about the Notification Service will be provided afterwards, within the Decision-making Monitoring system description section.

Finally, to develop all these tasks easily, the PCM component will parse the XML policy into a JAVA[8] object in order to simplify the access to policy information.

In the next sub-sections a set of UML activity diagrams are provided so as to simplify the comprehension of the expected behaviour of the Policy Consumer Manager component. One activity diagram is given per each of the external interfaces of the component, except for the main(), addN() and removeN() interfaces since their activity diagrams will not differ much from those shown in Figure 4 - 8, Figure 4 - 9 and Figure 4 - 10 respectively. Thereby, the activity diagrams shown, namely Policy processing, Signalling processing and Policy enforcement trigger, correspond to the dispatch(), dispatch() (overloaded) and triggerEnf() external interfaces respectively.

a *Policy processing*

Many of the tasks appearing in the activity diagram shown in the figure 4 - 17 are almost the same as those given in the Policy Group processing use case.

---

[8] JAVA is the programming language chosen for the implementation of the framework.

Nevertheless, the interest of figure 4 – 17 is to illustrate in more detail all tasks realised by the PCM component when processing policies. Nonetheless, to avoid redundant repetition of information we will just briefly mention those tasks that have already been described in previous chapters.

When the PCM component receives a policy, the first task it realises is to parse the XML policy into a JAVA object to simplify and speed up the handling of information by the framework. Immediately afterwards, it checks whether the policy pertains to a policy group. If so, the policy group processing tasks (see page 63), are realised. Otherwise, the PCM stores the policy in the DB and requests to the ACC component an authorisation check for that policy. If the policy actions requested are not authorised, the policy is removed from the database and the policy processing is stopped. In case the check is successful, the PCM revises the version of the installed PCC component before requesting to it the policy conflict checks. Again, only if these second checks are successful, the policy processing goes on. Otherwise, the policy is removed and the policy processing stopped. The next step realised by the PCM is registering the policy expiration date to uninstall the policy when it expires. Afterwards, the PCM proceeds to the registration of policy conditions that should be monitored in the DmMs component. If there are no conditions to monitor, the policy enforcement processes start immediately.

First, the PCM requests to the PCC the realisation of a dynamic conflict check against policies already enforced in the system. If successful, the Policy Consumer expiration date of the involved Policy Consumer components (related with the policy expiration date as explained later), is updated if necessary. Then, the PCM demultiplexes the policy to the appropriate Policy Consumer component to be enforced. When not installed, the PCM component requests its installation.

After the enforcement has been requested, the PCM component waits for the policy enforcement results. Based on this information it updates the policy status information in the database, continues the processing of a policy group when appropriate and informs higher-level managers about the enforcement result.

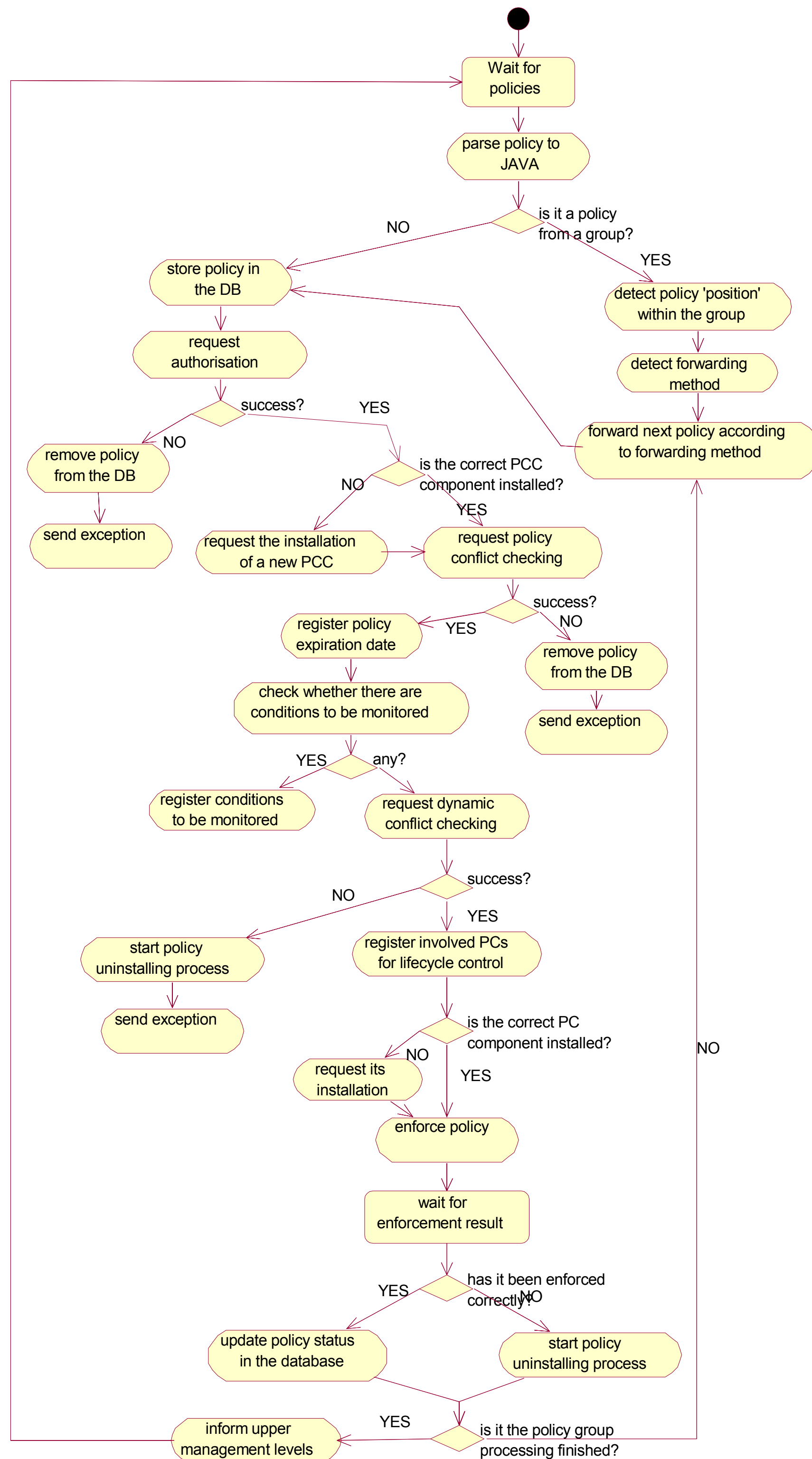

Figure 4 - 17. Policy processing inside the PCM component: activity diagram

b Signalling processing

The signalling processes realised by the PCM component are slightly different from the policy processes described in the previous section. First, there is no policy group processing since signalling requests are always 'individual' requests. In addition, the policy conflict checks are skipped since a signalling request is not a policy although it might be syntactically expressed in a similar way for processing convenience. Since there are no conditions in a signalling request, because of the nature of the signalling request itself, there is no need to contact the DmMs and the enforcement process can start directly. As with policies, a dynamic conflict check is requested before actually giving to the Policy Consumer component the decision about the signalling request. However, prior to requesting this check to the PCC component the PCM has to make sure that the correct version of the PCC is installed. In the policy processing case, this check is done before. The last difference with the policy processes is that now there is no need to check if the appropriate Policy Consumer component is installed, because it is the Policy Consumer component itself the one that initially raises the signalling request.

The other tasks in signalling processing have already been described in the previous section; thus, they will not be repeated here.

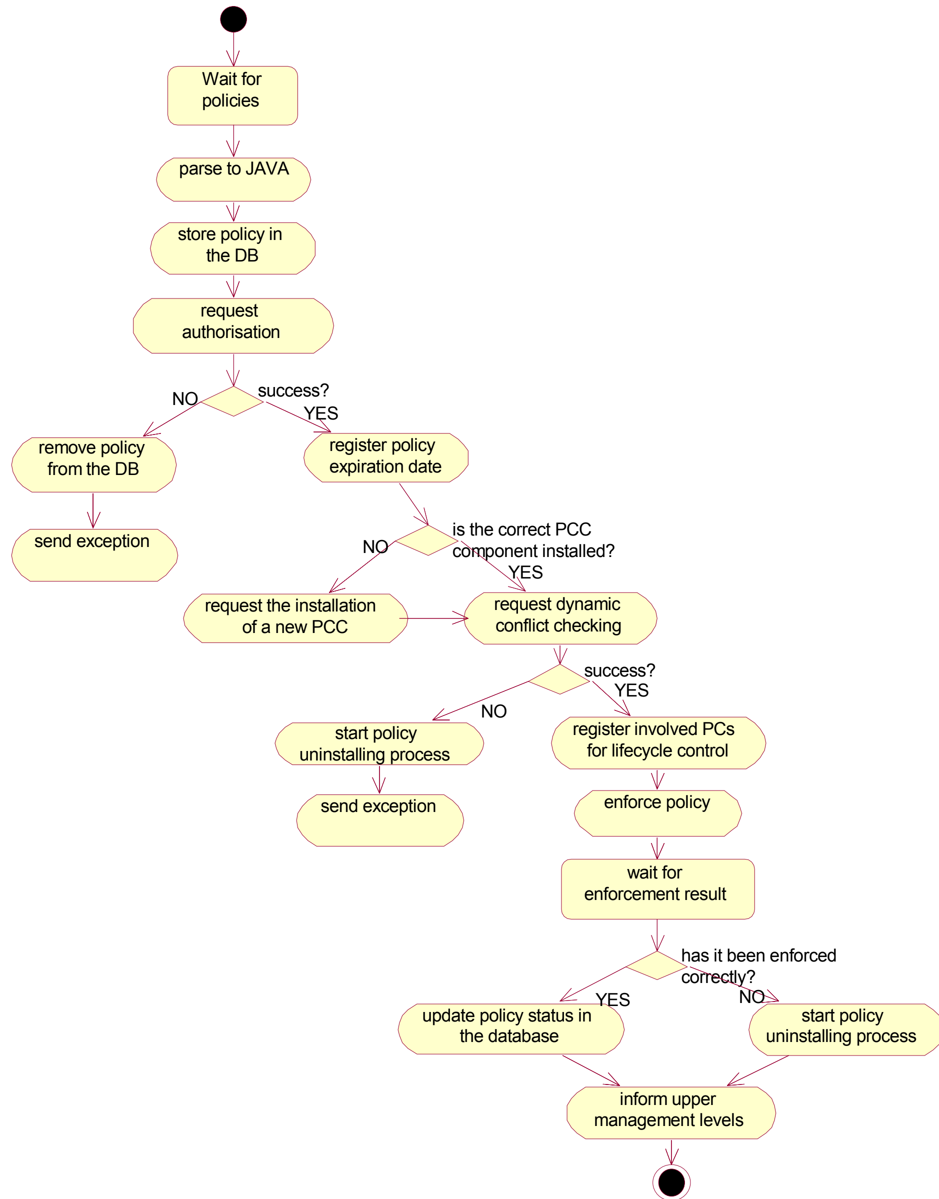

Figure 4 - 18. Signalling processing inside the PCM component: activity diagram

c *Policy enforcement trigger*

The trigger enforcement case is slightly different from the previous two, because it begins when conditions of received and processed policy are either fulfilled (initiating the policy enforcement), or are no longer fulfilled (causing the removal of the enforcement configurations related with that policy). The processes are initiated when the DmMs makes use of the triggerEnf() method (offered by the PCM component) to request the beginning of the policy enforcement when the conditions are fulfilled or the removal when not. The

conditions evaluation value is indicated in a parameter included within the method itself.

The first task realised by the PCM component is to retrieve the mentioned policy from the database for initiating the corresponding processes. The call cause (enforcement or removal) is indicated in a parameter introduced in the call itself by the DmMs component. In case the call indicates that conditions are fulfilled, the enforcement process starts. As we have already commented, the first step in the enforcement process is requesting to the PCC component the realisation of a dynamic conflict check. As with the policy processing case, the PCM does not need to check now that the correct version of the PCC is installed, because it has done it before (when the policy was first received and processed). If there are no dynamic conflicts, the PCM makes sure that the correct Policy Consumer component is installed before requesting to it the enforcement of the policy.

If the call indicates that the conditions are no longer fulfilled the component will simply change the *'act'* field of the policy with the 'Remove' value and request its enforcement. Such policy enforcement will cause the removal of all policy-related configurations in the underlying devices. Afterwards, the PCM requests to the PCC component the update of the resource information related with the enforced policy. The dynamic checks as well as the assessment of the availability of the responsible PC component are not necessary. The first one because the removal of a policy will not cause any resource conflict, since the resources are being freed, nor a consistency conflict because we assume that if a group of policies is consistent all possible subgroups will also be consistent (for more information about conflicts see pag.123). Additionally, it is not necessary to check the availability of the responsible PC component since this component should, at least, have been installed for enforcing the policy.

After the enforcement, or removal, has been requested the component waits for the result form the PC component. In case the enforcement or removal has been successful, the component updates the policy status in the database before starting the policy group processing tasks (or informing to higher-level managers if the policy group processing is finished). Otherwise, if the enforcement or removal could not be realised, on the one hand, an error is sent to the higher-level managers and, on the other hand, the policy group processing is started if needed.

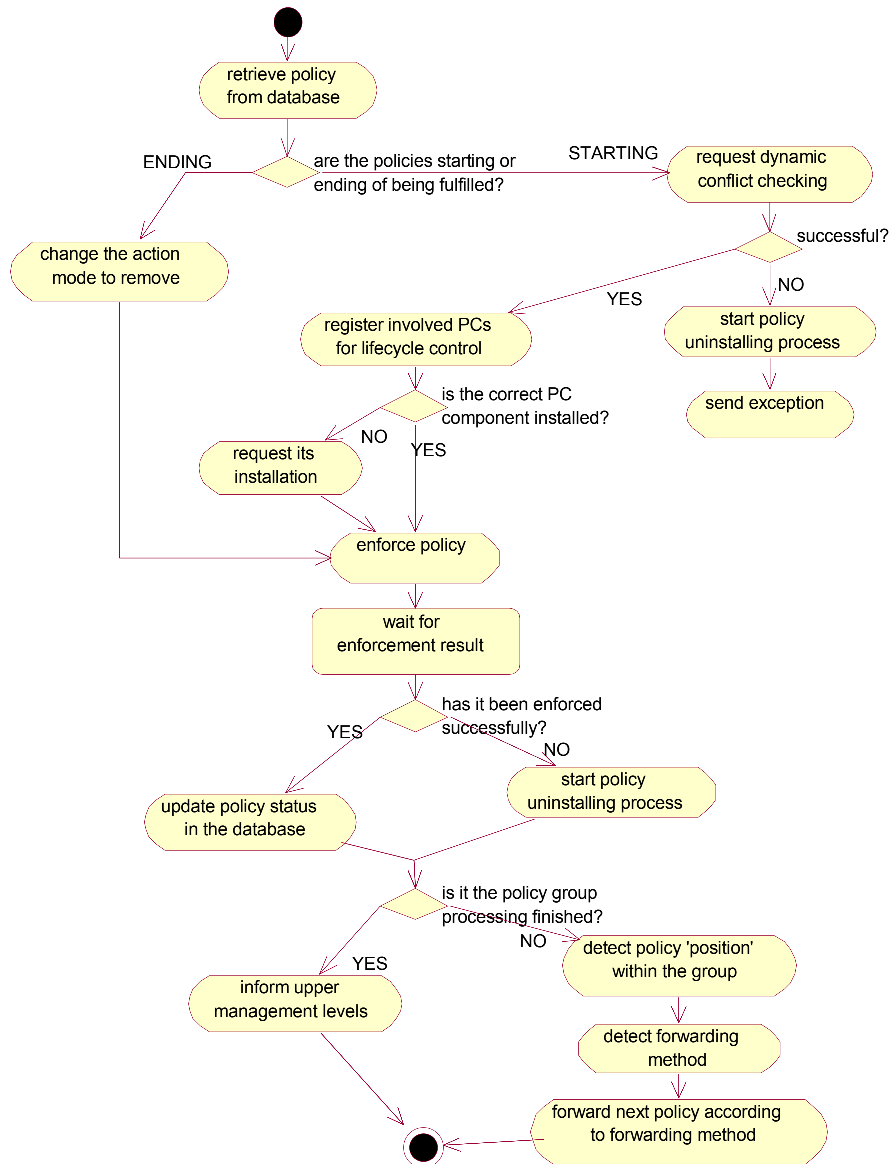

Figure 4 - 19. Policy enforcement trigger: activity diagram

B *Component Design*

The list of tasks described in the previous chapter can be joined in groups of functionalities, which summarise the overall functionality of the Policy Consumer Manager component:

- Functionalities related with the bootstrapping and management infrastructure configuration (e.g.. add/remove node related issues).

- Tasks related to the adaptation of managed topology information received to IMOs that can be used by the other framework components.
- Tasks for the correct reception, processing and enforcement of each individual policy (either signalling or 'normal' ones).
- Notification to upper management levels of the enforcement result of a policy or policy group.
- Functionalities related with the correct reception and processing of policy groups.
- Tasks related with the storage and maintenance of policies.
- Functionality dealing with the lifecycle of Policy Consumer components.
- Control of Policy Consumers and demultiplexing of policies to the correct one.
- Tasks dealing with the control of the Policy Conflict Check component.

To cope with these groups of tasks, the PCM has been designed with the classes shown in the UML class diagram below:

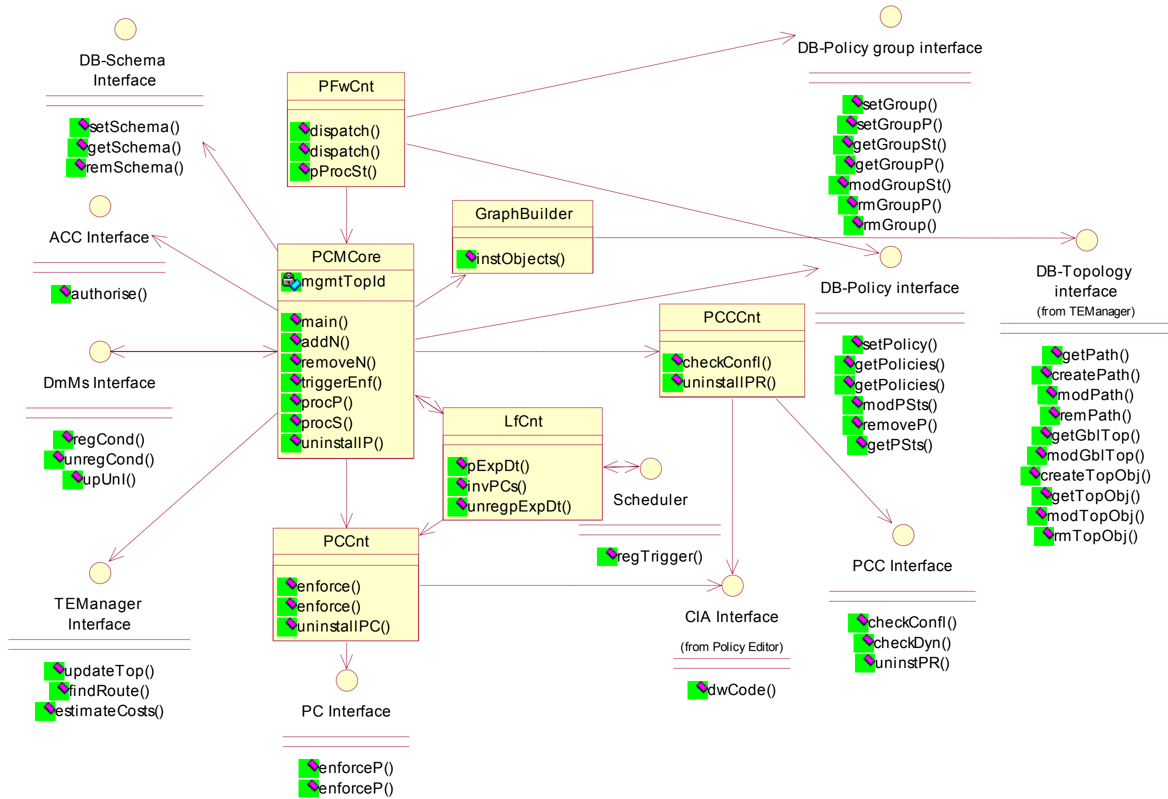

Figure 4 - 20. Policy Consumer Manager class diagram

In the next sections, the description of each of the classes in the diagram above will show how do they cope with the expected behaviour of the PCM component.

a PFwCnt class

The Policy Forwarding Control (PFwCnt) class is in charge of receiving policies and signalling requests from the Policy Editor component and processing them accordingly. In particular, this class examines the received information and detects whether it is a policy or a signalling request. When is a policy, the PFwCnt checks if it pertains to a policy group and processes it accordingly.

For realising this functionality, the class receives the policies and their enforcement results. This information is used as input in the Policy Group Execution Strategy (PGES) and is kept in the database together with other group information.

The policy enforcement result is received by the class through the pProcSt() method. The results might be introduced by the PCMCore class or Policy Consumer components. The only Policy Consumer components that will

introduce enforcement results directly to the pProcSt() method are those working at the network-level over element-level MANBoP managers. The reason for this behaviour is that the enforcement at this level is the forwarding of element-level policies. Thereby, the enforcement time of these element-level policies is unknown. Hence, to avoid that the PCM component waits for an unknown period of time for the enforcement result, Policy Consumer components working over element-level managers respond immediately to the PCM component an undefined enforcement result. Such enforcement result does not have any effect over the processing of other group policies, neither over the status of the policy. The only effect over the system is that the PCM waits no more for the enforcement result. Later on, when the enforcement result is finally known by Policy Consumers, because they are warned by element-level managers, they will directly contact the pProcSt() method to introduce the final enforcement result. Inside the pProcSt() method the usual tasks will be done. More details are given later on this sub-section.

The policy group processing functionality is not straightforward. Hence, we describe it based on two activity diagrams to ease its comprehension.

The PFwCnt functionality can be triggered because of the reception of a policy (or signalling request), or after the reception of an enforcement result. Consequently, the two activity diagrams shown below reflect these two possibilities.

Figure 4 - 21 shows the main activities carried out when a policy arrives to the PFwCnt class.

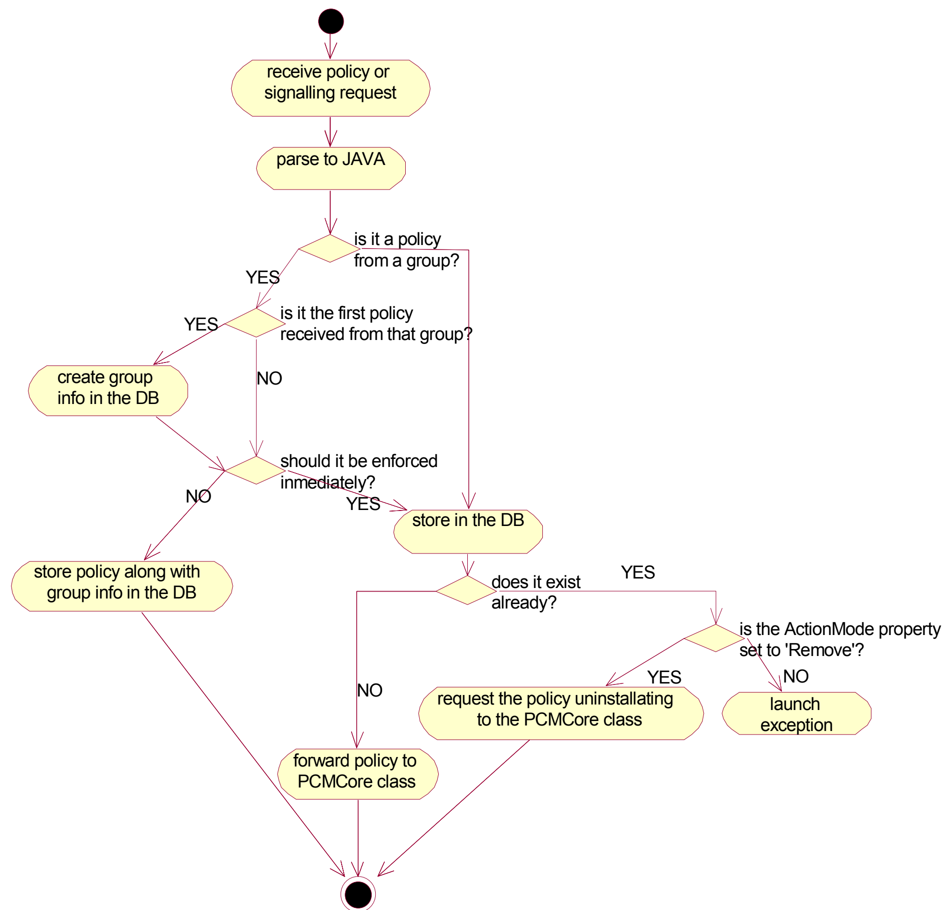

Figure 4 - 21. PFwCnt class: Policy triggered activity diagram.

First, the class parses the XML information into a JAVA object. This is done to ease the handling of the policy information by the framework components. Then, it checks whether the received policy is part of a policy group or not. For realising this check the class simply accesses the group-related information within the policy (see Information Model section in chapter 5). In case it isn't, the policy is simply stored in the DB (in both XML and JAVA) and the policy identifier is forwarded to the PCMCore class. Although this will be the most common behaviour, it might be also the case, as reflected in the diagram, that when trying to store the policy in the DB it detects that another policy with the same identifier is already stored in the system. This situation might happen either because an error has occurred or because the operator is trying to uninstall the corresponding policy. If so, the received policy should have the *'act'* field set with the 'Remove' value. When this happens the PFwCnt class requests the uninstalling of the policy using the uninstallP() method offered by the PCMCore class.

When the policy is part of a policy group, the PFwCnt class essays to get group information from the DB. If there is no information for this group, the PFwCnt deduces that the policy is the first one received policy from that group and creates the group information in the DB. Then, based on the group information, it decides whether the received policy should be enforced immediately (and thus, forwarded to the PCMCore class) or instead, it should be enforced after other policies from the group. When the latter, the class simply stores the policy in the DB along with the group information.

When receiving an enforcement result, the activities realised by the class are a bit more complex than the ones just described, as can be seen in the activity diagram below:

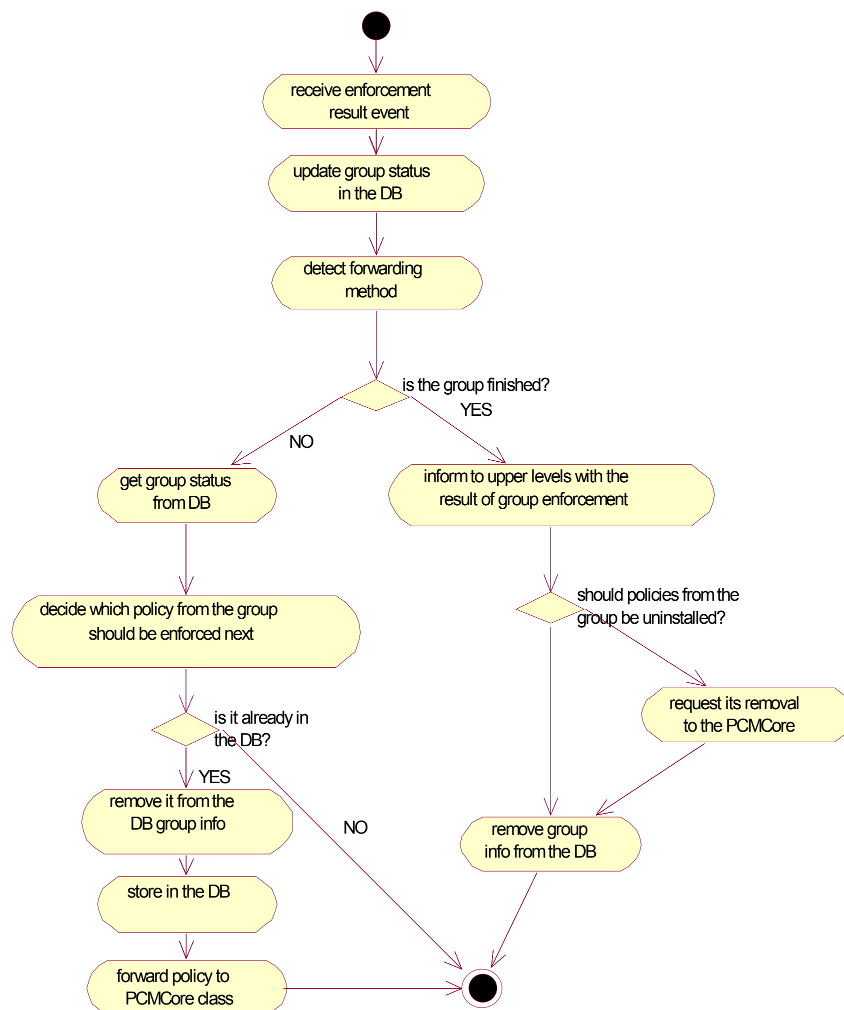

Figure 4 - 22. PFwCnt class: Enforcement result triggered activity diagram.

After the enforcement result arrival, the first step taken by the PFwCnt class is the update of the policy group status information present in the DB with

the policy enforcement result. The class also gets the PGES applied to that group. Then, based on the current group status and the execution strategy, the PFwCnt decides if this last enforcement concludes the policy group processing, or instead, if a new policy should be processed.

When the policy group processing is finished, the PFwCnt class informs the upper management layers about the policy group enforcement result. This report can even specify what group policies have been enforced (many times only some of them will be enforced). Afterwards, it checks if any group policies that were correctly enforced should be now removed because of the last enforcement result received (i.e. an 'atomic' forwarding method). If so, the class requests to the PCMCore to remove those policies. Finally, the information related with that policy group status is removed from the DB, since it is only kept there during the actual processing of the policy group.

Alternatively, when the policy group processing is not finished, a new policy needs to be processed. The PFwCnt retrieves the group status from the DB to decide which should be the next policy to be forwarded to the PCMCore class. As soon as the decision is made, the PFwCnt class looks for that policy in the DB. If the policy has not yet arrived to the framework, the PFwCnt does nothing. Otherwise, retrieves the policy from the DB and forwards it to the PCMCore class to be processed.

The PFwCnt class offers three public methods, namely the dispatch() method overloaded to accept both policies and signalling requests, and the pProcSt() method for receiving enforcement results. The input and output parameters, as well as the functionality within these methods, are listed and explained in the table that follows:

Interface	Input parameters	Output parameters	Functionality
dispatch()	credential User, string XPolicy	-	This method is used by the PE component to request the processing of an XML policy. The XPolicy parameter provides the policy in XML. The User parameter gives the credentials for the user that introduced the policy. When receiving a call to this method, the PFwCnt class will realise the tasks described in Figure 4 - 21. These tasks might eventually derive in the processing of the policy, which is requested to the PCMCore class through the procP() method.
dispatch()	string sigRqId, struct decision, credential User, string XPolicy	-	The difference of this overloaded dispatch method with the previous one is that the sigRqId string and the decision structure are also introduced as input parameters. sigRqId univocally identifies the signalling request so that when notifying the decision taken to the Policy Consumer component, it can easily link that decision with the signalling request raised. The decision structure specifies the decision taken. This structure is initialised by the Policy Editor with an acceptance value. The structure is defined as: struct decision {Boolean decision; string reason;}; The Boolean expresses whether the request has been accepted or refused, while the string expresses the reason for the refusal when appropriate. As previously mentioned, the signalling request is received as a special type of XML policy for processing convenience. The PFwCnt class, as it is not part of a policy group, will simply forward the request to the PCMCore class through the procS() method.
pProcSt()	string policyId, int result, string error	-	This method is used by either the PCMCore class or Policy Consumer components. The method is used for notifying, either an enforcement result or a policy removal for a particular reason (specified in one of the parameters of the method). The input parameters are just a string identifying the policy, an integer that determines the result and a string that provides more details in case an error has occurred. The possible values of the integer are: (0) enforced, (1) enforcement removed, (2) policy removed, (3) enforcement error. Enforcement removed applies when due to the policy conditions the configurations in the managed device related to this policy are removed. Policy removed applies when the policy is uninstalled due to its expiration, a conflict or any other reason. The PFwCnt class when receiving a call to this method will update (or remove) the policy information from the database, detect (and act accordingly) if the policy is part of a group not yet concluded and inform to upper management levels when necessary. For more details see Figure 4 - 22.

Table 4 - 5. The PFwCnt class interface description table

b PCMCore class

The PCMCore class is the central point in the Policy Consumer Manager and probably the most important class of the whole MANBoP framework. Among its tasks, the most important ones are the bootstrapping of the framework, the control of the managed topology update and the coordination of the policy processing functionality.

In order to facilitate the comprehension of how the PCMCore class behaves, a set of activity diagrams are given below. These activity diagrams show the class behaviour when processing policies (or signalling requests), or when uninstalling a policy. The activity diagrams for the bootstrapping of the system, or the addition/removal of network nodes are not included here because they remain mainly as they are in figures 4 - 8, 4 - 9 and 4 - 10.

The information provided by these activity diagrams is complemented at the end of the PCM description section by a set of sequence diagrams showing how the different classes of the PCM component interact to achieve their objective.

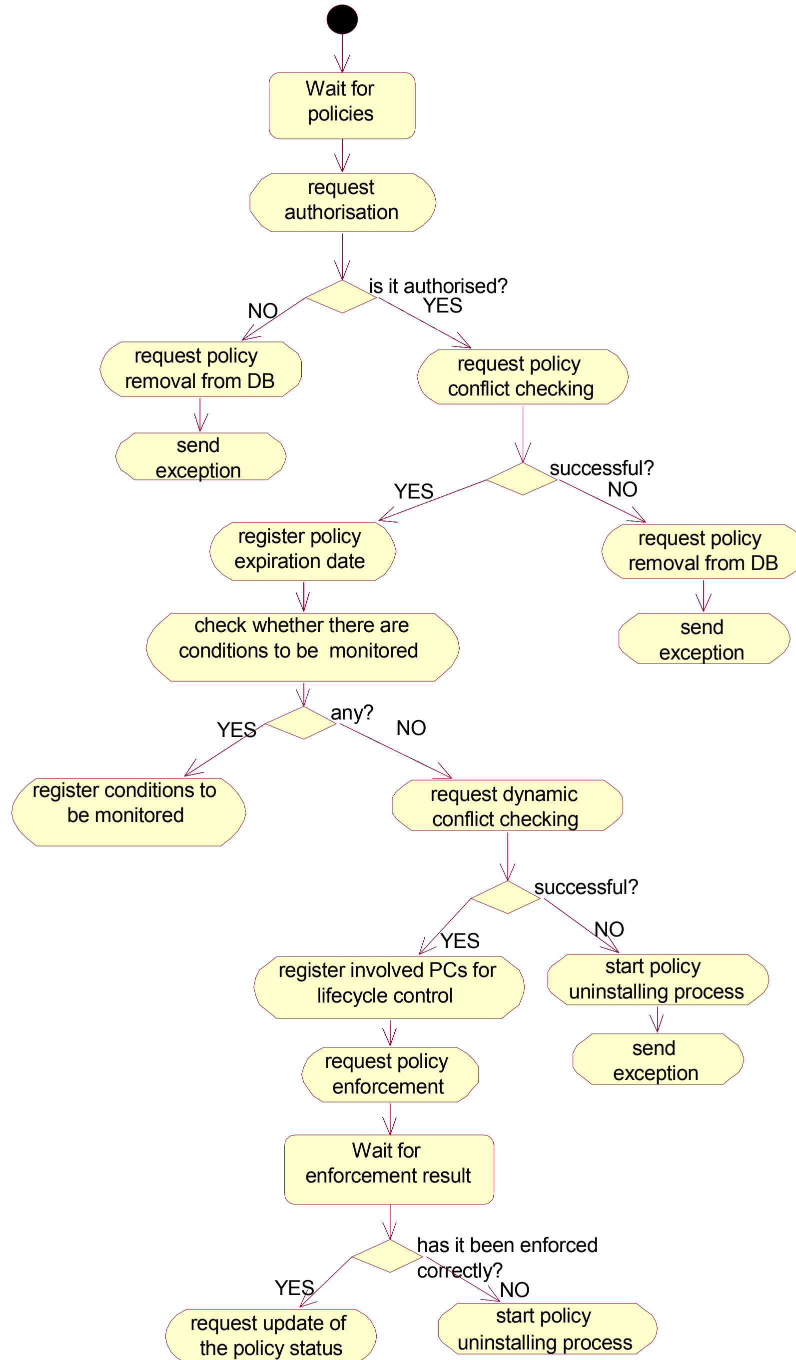

Figure 4 - 23. PCMCore class: Policy processing activity diagram

The figure above shows the main tasks carried out by the PCMCore class when processing a policy. First, it contacts the ACC component to request an authorisation check to the received policy. If the policy is allowed, the PCMCore class requests to the PCCCnt class the execution of policy conflict checks.

The PCCCnt class, as will be seen later, maintains the correct version of the PCC component and interacts directly with it. As result from the conflict checks, both policy and dynamic conflicts, a list of nodes where the policy should be enforced might be returned. The PCMCore class will forward this list to the PCCnt class when requesting the enforcement of a policy. In case no conflicts are found, the PCMCore component asks to the LfCnt class the registration of the policy expiration date.

Then, it decides whether the policy has any condition that needs to be monitored based on the policy attributes. To access to these policy attributes, the PCMCore class retrieves the policy from the database. If there are conditions to monitor, the PCMCore class registers them through the DmMs interfaces.

When no policy conditions need to be monitored, the class assumes that the policy should be enforced immediately. The first pre-enforcement task, developed by the PCMCore is requesting to the PCCCnt the realisation of dynamic conflict checks. Only if the checks are successful, the PCMCore registers in the LfCnt class the involved PCs before requesting to them the policy enforcement. As the PCCCnt does with the PCC component, the PCCnt class maintains the appropriate PC components in the system. More details on the concrete functionality of the PCCnt class will be given later.

Finally, after requesting the policy enforcement the PCMCore class waits for the reception of the enforcement result. In case it has been successful it contacts the PFwCnt class to request the update of the policy information in the database accordingly. Otherwise, it starts the policy uninstalling processes, or does nothing if an undefined result is received. The uninstalling tasks are those shown in Figure 4 - 25.

Complementary, the behaviour of the PCMCore class during the trigger enforcement process is the one shown in the activity diagram within the component behaviour description section (see Figure 4 - 19). The only tasks appearing in that diagram which are not carried out by the PCMCore class, but by the PFwCnt class, are those shown after the checks to determine if the enforcement (or removal) has been realised successfully. Based on this information the PCMCore class accesses the PFwCnt class to realise the remaining tasks shown in the diagram as has already been described.

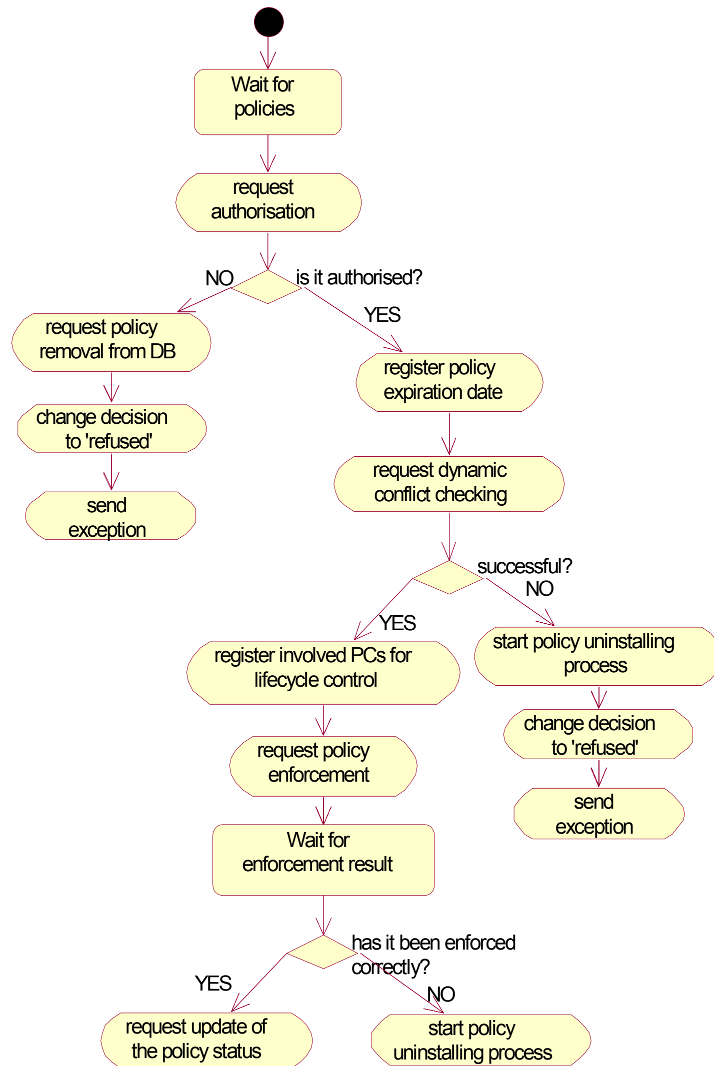

Figure 4 - 24. PCMCore class: Signalling processing activity diagram

The activity diagram above shows the behaviour of the PCMCore class when processing a signalling request. The tasks developed are a subset of those realised when processing a policy, because the conflict checks, as well as the evaluation of conditions to be monitored, need not be done.

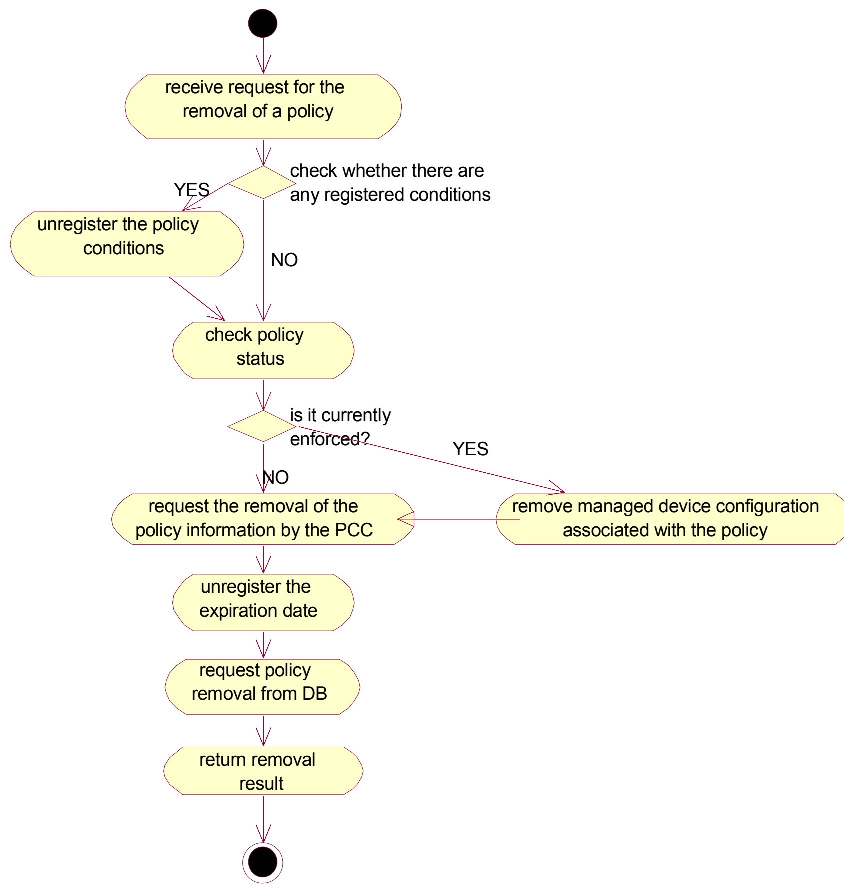

Figure 4 - 25. PCMCore class: Policy uninstalling activity diagram

Finally, the last activity diagram described for the PCMCore class details the tasks to be developed when a policy expires, and hence, must be uninstalled from the system.

The tasks shown in the diagram might be triggered, either by the LfCnt class, the PCC component or by the PFwCnt, accessing the uninstallP() method in the PCMCore class interface. When triggered by the LfCnt class, the removal of the policy is caused by its expiration. When triggered by the PFwCnt the cause is an operator request or the policy group processing method applied. Finally, when requested by the PCC component the reason is the necessity of removing that policy to solve a conflict.

The PCMCore class receives the removal request and checks, considering policy attributes, whether there are conditions registered in the DmMs. If so, requests through the DmMs interface the obliteration of these conditions (that is, the removal of the conditions registered).

The next task developed by the PCMCore class is retrieving the policy status from the database to see whether the policy is currently enforced on the managed devices. If so, it requests to the PCCnt class the enforcement of a policy, equal to the one that should be uninstalled except for the *'act'* policy field, which is set with the 'Remove' value. The enforcement of such a policy by the corresponding PC component will cause the removal of policy-associated configurations in the underlying level and finally on the device.

Finally, the PCMCore class asks to the PCC component the removal of the policy-related information, obliterates the expiration date from the LfCnt class and requests to the PFwCnt class the removal of the policy information from the database.

For carrying out all these tasks enumerated along the subsection, the PCMCore offers seven public methods, namely main(), addN(), removeN(), triggerEnf(), procP(), procS() and uninstallP(). The input and output parameters as well as the functionality within these methods, are described in the table below:

Interface	*Input parameters*	*Output parameters*	*Functionality*
main()	int mgmtTopId, file MgdTop, file UndInt	-	This method is called by the administrator to request the instantiation of a new MANBoP instance. The method requires three input parameters. The first one is an integer (i.e. mgmtTopId) that establishes the position within the management infrastructure at which this new MANBoP instance is going to work. The possible values are: 0 (network level), 1 (element level), 2 (network over element level) and 3 (network over subnetwork level). The second input parameter introduced is a file with information about the managed topology. The information provided by this file is IP addresses, available link bandwidth, etc. Finally, the third parameter is another file with information about the underlying interfaces. It provides information about how to contact and configure the lower-level devices, which either can be lower-level MANBoP instances, or managed devices. When receiving a call to this method, the PCMCore class realises the tasks described in Figure 4 - 8, together with the GraphBuilder class that realises the adaptation of the managed topology information. These tasks will derive in the instantiation of a new MANBoP manager.
addN()	file MgdTop	-	This method is used by the administrator to request the addition of one node to the topology currently managed by this MANBoP instance. The method only needs two input parameters. These are, the file with information about the managed topology and the file with the underlying interfaces information (both described for the main() class method). When receiving a call to this method, the PCMCore class develops the tasks described in Figure 4 - 9, together with the GraphBuilder class that realises the adaptation of the managed topology information .
removeN()	file MgdTop	-	This method is used by the administrator to request the removal of one node from the topology managed by this MANBoP instance. The method only needs one input parameter. That is, the file, with information about the managed topology, as described before for the main() method. When receiving a call to this method, the PCMCore class carries out the tasks described in Figure 4 - 10, together with the GraphBuilder class that realises the adaptation of the managed topology information, as well as verifying that no policy applies to the node that must be removed .
procP()	credential User, string policyId	-	Used by the PFwCnt class to request the processing of a policy within the system. The input parameters introduced are the credentials of the User, which will be used for the Authorisation Check requested to the ACC class, and the policyId string that identifies the policy and allows to easily retrieve it from the DB. The PCMCore class, when receiving a call to this method, coordinates the interactions between the different components of the system as described in Figure 4 - 23.
procS()	string sigRqId, decision dec, credential User, string policyId	-	Used by the PFwCnt class to request the processing of a signalling request policy within the system. The input parameters are the same as in the procP() method plus the sigRqId string and the decision structure. The sigRqId string identifies the signalling request; so that the Policy Consumer component, when it receives the decision, can match it with the corresponding request. The decision structure specifies the decision taken. This structure is initialised by the Policy Editor with an acceptance value. The structure has been already defined in Table 4 - 5. The PCMCore class, when receiving a call to this method, coordinates the interactions with the ACC, PCC, DmMs and Policy Consumer components for correctly processing the signalling request policy (see Figure 4 - 24). If the decision received as parameter already indicates a refusal value, the class simply forwards the decision to the PCCnt class through the appropriate method.
triggerEnf()	string policyId, Boolean reason	-	Used by the DmMs when the conditions of a policy change their global evaluation value. It indicates that the policy should either be enforced or removed. The parameters needed in this method are two. The policyId string identifies the policy whose condition evaluation value has changed. The reason parameter is a Boolean that indicates (true) when the conditions

			match or (false) when they don't match any longer. The PCMCore class coordinates the enforcement processes: retrieves the policy from the DB, requests the dynamic conflict checks and requests also to the PCCnt class the enforcement of the policy.
uninstallP()	string policyId, int reason	boolean Result	This method can be called either by the PFwCnt class, the PCC component or by the LfCnt class to request the removal of a policy. When requested by the PFwCnt class, the cause of the removal can be either because of the Policy Group Execution Strategy applied or because there is an explicit request from the policy owner. When requested by the PCC component, the removal is requested to solve a conflict. Finally, when requested by the LfCnt class, the reason is the policy expiration. The removal request cause is given as input parameter in the reason integer. The possible values are: (0) explicit request from the owner, (1) due to the policy group processing method, (2) policy expiration, (3) to solve a conflict. The other input parameter introduced in the method is the identifier of the policy to be removed. The output parameter is a Boolean that determines the result of the actions taken. The PCMCore class requests to the PCCnt class the removal of the policy configuration from the managed device, removes the policy from the database and removes the expiration trigger from the LfCnt class when needed. In addition, it informs the owner about the cause of the removal. These tasks can be seen in Figure 4 - 25.

Table 4 - 6. The PCMCore class interface description table

c GraphBuilder class

The GraphBuilder class is in charge of the creation, or removal, of topology Information Model Objects (IMOs) whenever the topology is updated and during the bootstrap of the framework. Every time a topology IMO should be removed, the GraphBuilder class must verify that no reservation is using the resources of the element represented by that IMO. The reason for this has already been described in the Add/remove node Use case section (see pag.72). For realising this task, it simply accesses the resource information linked with this element and verifies that no resources are used or scheduled to be used in the future.

In particular, during the MANBoP instantiation the GraphBuilder class gets the MgdTop and UndInt files as well as the mgmtTopId (see Table 4 - 6) from the PCMCore class and creates the IMOs related to the underlying devices that will be stored in the Database. Among all the information stored, the one relevant to this chapter is the assignation of nodeSets to each underlying topology device. NodeSets are specified within the UndInt file. They are artificial sets of underlying topology devices grouped based on Policy Consumer component criteria. NodeSets are identified by the pointer to where the attached Policy Consumer will be installed. To clarify a bit more the concept we enumerate below the rules followed to determine these nodeSets during the bootstrapping:

- When the MANBoP instance is not working directly over the managed resources (i.e. mgmtTopId values 2 or 3), all underlying topology devices are grouped under a single nodeSet. This implies that there will be a single Policy Consumer component per functional

domain for all underlying topology devices, probably co-located with the MANBoP instance.

- If the MANBoP instance is working directly over the managed resources, (i.e. mgmtTopId values 0 or 1), there might be more than one nodeSet:
  - From the MgdTop file the GraphBuilder extracts the active nodes that permit the installation of a PC component in an EE. Each of those nodes will form a single nodeSet. The value of the nodeSet, which is the pointer to where the PC should be installed, will be obtained from the UndInt file.
  - The other nodes will be grouped in those with the same interface. Each of these groups will form a single nodeSet and thus, they will have a single PC for each group and functional domain. All these PC components will probably be co-located within the MANBoP instance.

The nodeSet information, as we will see on the DmMs description section, is also used to determine the Monitoring Meter components that are needed and where they should be installed.

The class must also modify the IMO containing the global topology information before concluding. The interface offered by the class is described in the table below.

Interface	*Input parameters*	*Output parameters*	*Functionality*
instObjects()	file newTop, file UndInt, int type	boolean Result	This method will be called by the PCMCore component to request the creation or removal of topological IMOs, their linked resource information IMOs and finally, the update of the global topology IMO. The method specifies three input parameters and a boolean that indicates the result of the process as output parameter. The first input parameter is a file indicating the new topological information and the resources linked to it. The second parameter is a file with information regarding the interface and access parameters for the new nodes. The third one is an integer 'type' that indicates if the topological information should be added (0) or removed (1) to the managed network. When receiving a call to this method the GraphBuilder class obtains from the newTop file the topological elements (nodes and links) added (or the ones that should be removed). Based on this information it creates the corresponding IMOs. The resource IMOs for these elements are also created based on the information available in the file. Finally, the global topology IMO is updated with the new topology. The class returns 'true' if no error has been found during the process; otherwise, it returns 'false'.

Table 4 - 7. The GraphBuilder class interface description table

d PCCCnt class

The main task of the PCCCnt class (PCC Controller class) is the maintenance of the correct version of the PCC component at any time.

The class receives requests for conflict checks (either dynamic or policy conflicts) from the PCMCore class. The PCCCnt class keeps a list of the functional domains supported by the current version of the PCC component. When a conflict check request is received, it verifies if the functional domain of the checked policy is in the list. If not supported, it requests to the CIA system the installation of the newest version of the PCC component and updates the list of supported functional domains. When the PCCCnt is sure that the PCC is capable of processing the policy, it forwards the request to the new PCC component. When the check is finished, the return parameters are forwarded back to the PCMCore class.

As with the LfCnt class, the functionality of the PCCCnt has already been described in a previous paragraph and the introduction of activity diagrams will not provide any extra information.

The PCCCnt class offers an interface with just two methods: the checkConfl() is used for requesting both dynamic and (static) policy conflict checks. The uninstPR() method is used to request the removal of the resource information associated with a policy. A detailed description of these methods is provided in the table below:

Interface	*Input parameters*	*Output parameters*	*Functionality*
checkConfl()	string policyId, credential User, string domainId, int chType	boolean Result, string[] nodeId	The PCMCore class uses this method to request a policy conflict check. The method includes four input and two output parameters. The first input parameter is a string representing the policyId that uniquely identifies the policy. The second one are the credentials of the user, needed for checking that the resources allowed to the user are not overridden. The third is another string that identifies the functional domain to which that policy pertains. Finally, the last input parameter is an integer, i.e. chType, with two possible values: 0 is used for requesting a policy conflict check, and 1 is used for requesting a dynamic conflict check. Finally, the method returns a boolean that indicates whether the checks have been successful or not and a list of nodeIds that identify where should be enforced the policy. This last parameter is different from null only when a dynamic conflict checking is requested. When receiving a call to this method, the PCCCnt class carries out the PCC controlling functionalities already described at the beginning of the section.
uninstPR()	string policyId, boolean cause	boolean Result	This method is called by the PCMCore component to request the removal or update of the resource information related with a policy. The method specifies two input and one output parameter. The input parameters are the policyId that identifies the policy (whose resource information should be either removed, or updated when the policy is de-enforced). The boolean 'cause' specifies which of the two options (removal when true or update when false) applies in this case. The output parameter is just a Boolean that indicates if the requested action has been realised correctly. When receiving a call to this method the PCCCnt class contacts the PCC component to request the realisation of the expected functionality.

Table 4 - 8. The PCCCnt class interface description table

e PCCnt class

The functionality developed by this class is somehow similar to the one in the PCCCnt class in the sense that its main role is to maintain the lifecycle of Policy Consumer components. Nonetheless, its responsibilities are broader, and the complexity of its tasks is slightly higher.

The main task of this component is the maintenance of Policy Consumers. The PCCnt receives enforcement requests from the PCMCore class and looks for the needed Policy Consumers[9]. If not installed, it requests their installation at the location(s) pointed by the involved nodeSets. In addition, the PCCnt retrieves the interfaces of the involved nodes from the database to include them in the request to the PC. When the PCCnt knows that the needed PC components are installed, it forwards the enforcement requests to them.

Another task realised by this component is the removal of an unused Policy Consumer component when requested by the LfCnt class.

The PCCnt offers an interface with three methods for developing these tasks. These methods are: the enforce() method overloaded to support policy and signalling enforcements and the uninstallPC() method. The logic behind these methods, as well as the input and output parameters specified are detailed in the following table:

[9] In case of a signalling request, the PCCnt simply retrieves the Policy Consumer that raised the request from the request itself and links its identifier with its interface.

Interface	Input parameters	Output parameters	Functionality
enforce()	string policySer, string[] nodeId	int result, string error	The PCMCore class requests, through this method, the enforcement of a policy. The method specifies two input and two output parameters. The policy that must be enforced, serialised ('policySer') in a string, and the list of nodes where this policy must be enforced, are the input parameters. The output ones are an integer that specifies the result and a string that provides more details if an error occurs. The possible values of the integer are: (0) enforced, (1) enforcement removed, (2) policy removed, (3) enforcement error, (4) undefined. Enforcement removed applies when due to the policy conditions the managed device configurations related to this policy are removed. Policy removed applies when the policy is uninstalled due to its expiration, a conflict or any other reason. When receiving a call to this method, the PCCnt class does the tasks described before for assessing the availability of the PC component before forwarding to it the request.
enforce()	string policySer, string[] nodeId, string sigRqId, struct decision	int result, string error	The PCMCore class requests, through this method, the enforcement of a signalling request. The parameters specified in the method are those for the previous one plus the identifier of the signalling request and the decision structure. This identifier is used by the PC component to link the decision made by the system with the signalling request raised. The decision structure, already described for the PCMCore component, expresses the decision that should be forwarded to the PC component. In this case the class forwards directly the enforcement request to the appropriate PC component.
uninstallPC()	string nodeSet, string PCId	boolean Result	This method is used by the LfCnt class to request the removal of an unused Policy Consumer component. The method specifies two input parameters and one boolean parameter as output, which indicates if the requested action has been realised correctly. The input parameters are the 'nodeSet' where the PC component to be removed can be located, and the PC identifier 'PCId'. Both parameters are strings. When receiving a call to this method, the PCCnt class removes from the system the requested Policy Consumer component. However, the component code might be kept in a local cache to avoid the need of downloading it again in the future.

Table 4 - 9. The PCCnt class interface description table

f LfCnt class

The LfCnt class (i.e. Lifecycle Control class) is responsible of controlling the lifecycle of both policies and Policy Consumer components. The lifecycle of policies is dictated by its expiration date property, while the Policy Consumer lifecycle is dictated by its use within the system. That is, when a PC is not needed by any of the policies within the system the LfCnt will remove it in order to save resources. Nevertheless, the removal of Policy Consumers is only realised once a day, e.g. at midnight, to avoid an unnecessary high number of installations and removals of Policy Consumers during the day.

Nevertheless, the lifecycle control of Policy Consumer components is not straightforward, particularly when the MANBoP instance is working at the network level directly over the managed resources (i.e. mgmtTopId value is 0). In such a situation, the underlying managed devices might be grouped in more than one nodeSet, as we have seen in the GraphBuilder class description section, and thus, there might be more than one instance of the

same type of Policy Consumer component, one per nodeSet. Thereby, the LfCnt, in order to keep the lifecycle of the PC components, needs to know not only the policy identifier and the expiration date but also the nodes where this policy will be enforced. Based on this information the LfCnt will be able to extract from the database the nodeSets involved and thereby, the PC components involved.

The main problem is that sometimes, as justified in the traffic engineering section (see pag. 62), the nodes involved in the enforcement of a policy might not be known until enforcement time. On the other hand, the expiration date of the policy should be registered when the policy is received. This process cannot be done at enforcement time because some policies might never be enforced. Hence, when the expiration date is registered in the LfCnt class, when the policy is received, we might not know yet which are the involved nodes.

The solution to this problem is doing the lifecycle control in two steps. The first one is, when policies are received in the system, the registration of the expiration date in the LfCnt class. The second step is, when the policy is going to be enforced, the registration of the involved PC in the enforcement of this policy. These two steps are represented in the LfCnt interface by two methods, i.e. the pExpDt() for the first step and the invPCs() for the second one.

To execute these tasks, the LfCnt class keeps a table with all policies, their expiration date, the type of PC component they need and a list of involved nodes. The first two properties of each row of the table, i.e. policyId and expiration date, are filled during the first step, while the second couple of properties are left blank until the second step. Additionally, the class accesses and modifies the underlying topology information available in the database (created by the GraphBuilder class during bootstrap as previously described). In particular, it accesses the nodeSet information in order to find and modify the list and number of policies processed by a particular Policy Consumer component instance. The actual structure of this information within the database will be seen in detail in the Information Model description section in chapter 5.

Additionally, the LfCnt class makes use of the Scheduler service, considered as an external service in this thesis, for registering the time triggers for each policy expiration date. Each time the class receives a trigger from the scheduler, it gets the involved policies and requests their removal to the PCMCore class. Also, in case the list of the involved nodes is not blank in the local table (meaning that the second registration step has been done at least once), the LfCnt checks if these policies are the last ones that needed a particular PC component instance and if so, marks that instance in the table to be uninstalled at midnight. On the other hand when a new policy expiration date is registered it also checks whether the involved PC is marked to be uninstalled and if so, clears the mark.

The functionality of this component is not very complex; thus, we have not considered necessary the introduction of activity diagrams to enhance the description of its behaviour. The LfCnt class interface is composed by three methods, pExpDt(), invPCs() and unregpExpDt(), which are further described in the following table:

Interface	*Input parameters*	*Output parameters*	*Functionality*
pExpDt()	string policyId, string date	boolean Result	This method is used by the PCMCore class to register the expiration date of a received policy. The method requires just two input parameters: the policyId that uniquely identifies the policy and the expiration date. The method returns a boolean that indicates if the registration has been successful. When receiving a call to this method, the LfCnt class creates a new entry for the policy in the local table kept by this class and registers in the Scheduler service the corresponding trigger.
invPCs()	string policyId, string PCId, string[] nodeId	boolean Result	This method is used by the PCMCore class to register the PC, linked with a received policy, for lifecycle control. The method requires three input parameters. The first one is the policyId that uniquely identifies the policy. The PCId identifies the Policy Consumer component type needed to enforce this policy. Finally, a list of nodes where the policy should be enforced is introduced The method returns a boolean that indicates whether the registration has been successful. When receiving a call to this method, the LfCnt class updates the policy entry in the local table kept by this class, and modifies the DB with the lifecycle information for the involved PC components.
unregpExpDt()	string policyId	boolean Result	This method is called by the PCMCore class to request the removal of an expiration date of a policy. The method only needs one input and one output parameter. The input parameter is the policyId that identifies the policy whose expiration date should be removed. The output parameter is just a Boolean that indicates if the requested action has been realised correctly. When receiving a call to this method, the LfCnt class will just remove the corresponding row from the local table kept by this component and if necessary update the corresponding PC components information from the DB.

Table 4 - 10. The LfCnt class interface description table

g *Sequence Diagrams*

With the sake of completing the description and comprehension of the Policy Consumer Manager component we include below sequence diagrams, with interactions between the classes of the component, for all possible use cases described in the 'Component Behaviour' section, with an additional one for the policy uninstalling process.

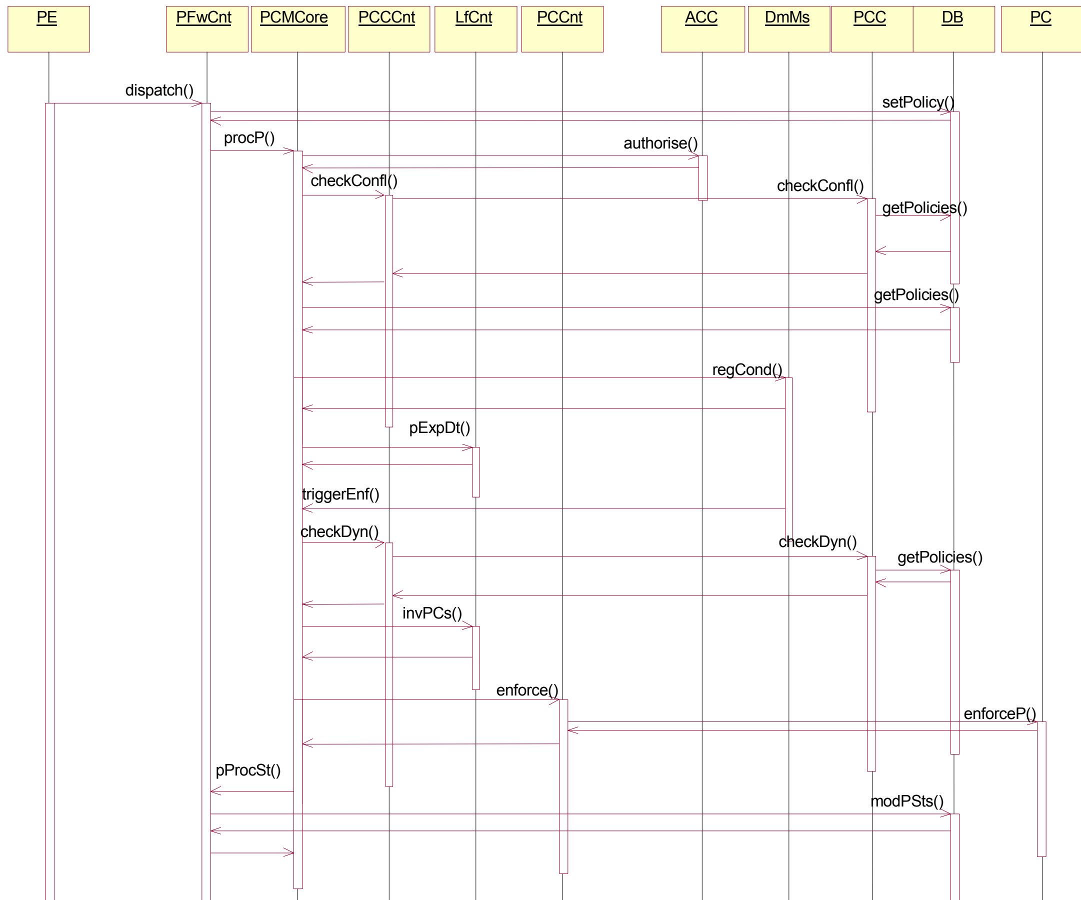

Figure 4 - 26. Policy Consumer Manager: Policy processing sequence diagram

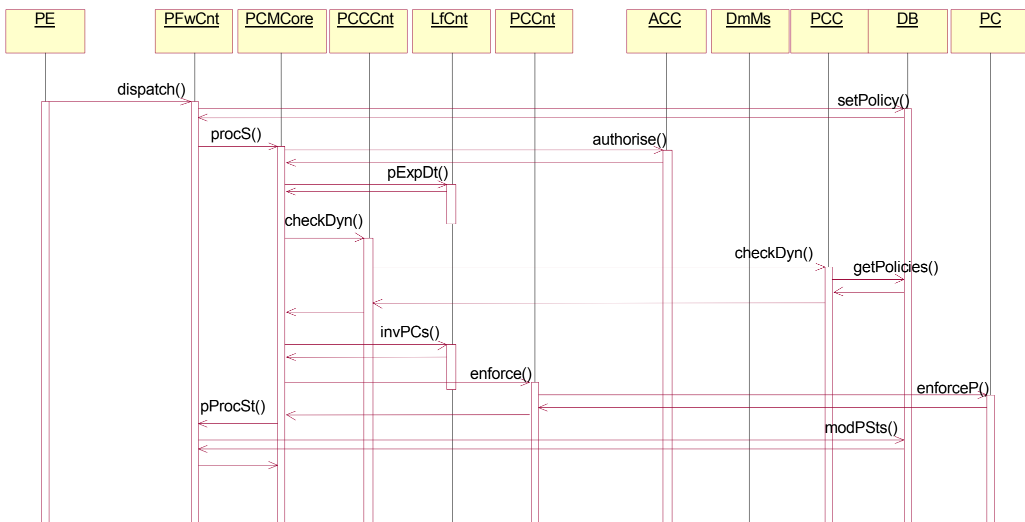

Figure 4 - 27. Policy Consumer Manager: Signalling processing sequence diagram

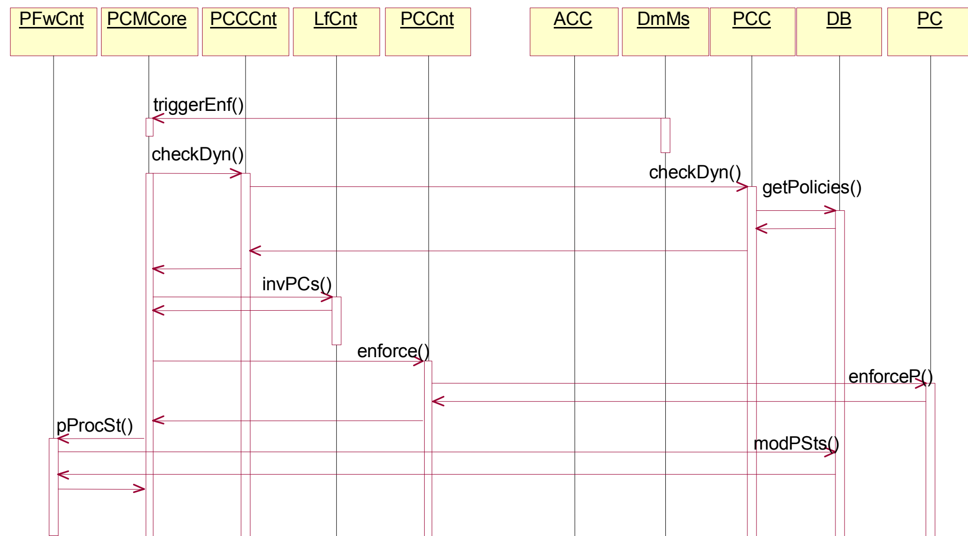

Figure 4 - 28. Policy Consumer Manager: Trigger Enforcement Sequence Diagram

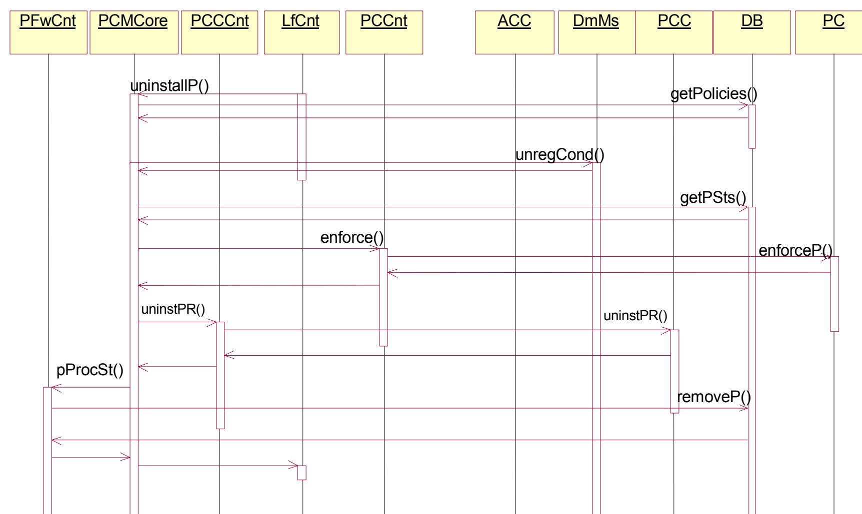

Figure 4 - 29. Policy Consumer Manager: Policy uninstalling sequence diagram

3rd Authorisation Check Component

A Component Behaviour

The table of components and interfaces given in section IV.1 (done after a system behaviour analysis based on use cases), assigns one task to the Authorisation Check Component (ACC):

- *Authorise policy*: This task is requested through the authorise() component interface. It authorises the policy by comparing it against the access rights of the user.

The ACC is focused on realising this task that is key for enabling the delegation of management functionality to users.

The authorisation checks determine what types of policies a user can introduce in the system. That is, what conditions, actions as well as their values and, sometimes, where and when policies can be applied. At the same time, the ACC also checks that the policy is syntactically correct.

To do these tasks, the ACC accesses the DB to obtain the user's access rights formatted in a particular way. This information has been introduced previously in the DB as result of delegation policies enforcements. In our case, this information is formatted as XML Schemas. Hence, the ACC accesses the DB to obtain, based on the functional domain and the user's credentials the XML Schema defining what the user is allowed to do within that functional domain. No XML Schema means that the user is not allowed to introduce any type of policies of that functional domain. In the activity diagram shown below we can see in more detail how the component does this task:

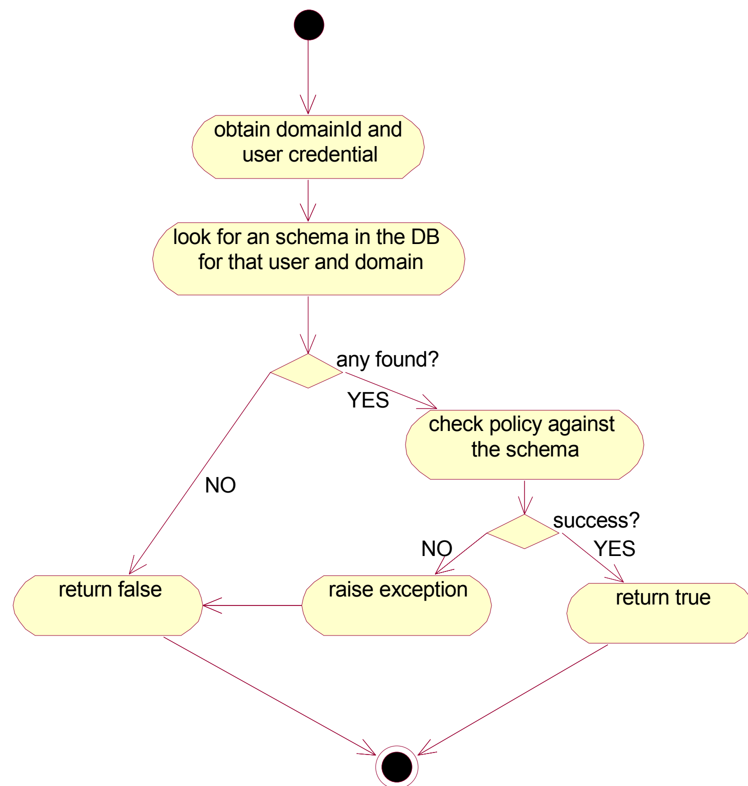

Figure 4 - 30. Authorisation Check Component Activity diagram

B Component Design

As we have seen in the Component Behaviour sub-section there are two major tasks to be developed: the logic to obtain the necessary information for the authorisation checks and the checks themselves. The class diagram shown below reflects this division.

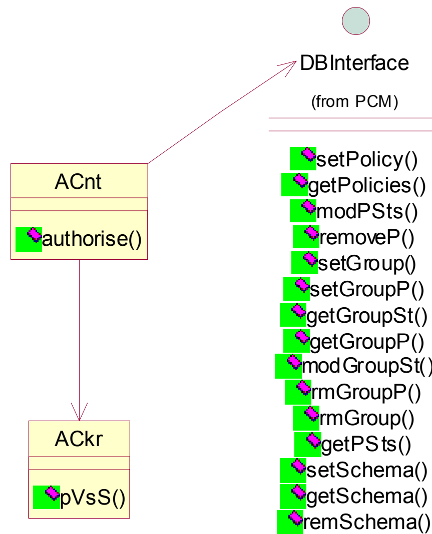

Figure 4 - 31. Authorisation Check Component class diagram

The ACnt (Authorisation Control) class is in charge of receiving the request, getting the schema from the DB and, if necessary, forwarding the schema and the policy to the ACkr class to do the checks. The concrete functionality and method description for these two classes are given in the sub-sections below.

a ACnt class

The Authorisation Control class (ACnt) gathers the needed information from the Database and input parameters and, if a schema is found, retrieves from the DB the policy in XML format and the XML schema before requesting to the ACkr the realisation of the checks.

The class interface is described in the table below:

Interface	*Input parameters*	*Output parameters*	*Functionality*
authorise()	string policyId, credential User, string domainId	boolean Result	This method is called by the PCM component for deciding whether a user is authorised to introduce a policy. The method includes three input parameters: the policyId, which uniquely identifies the policy, the user's credentials and the functional domain identifier. The method returns a boolean that indicates whether the user's policy is authorised or not. When receiving a call to this method, the ACnt class carries out the tasks already described.

Table 4 - 11. The ACnt class interface description table

b ACkr class

The task done by the ACkr class is validating the policy against the access rights of the user who introduced it. We are not going to enter in detail how this functionality is developed, because we are simply taking one of the freely available XML validators [W3Ctools] that realise this functionality (i.e. the validation of an XML document against a schema).

The class interface is described in the table below:

Interface	*Input parameters*	*Output parameters*	*Functionality*
pVsS()	string Xpolicy, string XMLSchema	boolean Result	The ACnt class requests the validation of the policy against the user schema through this method. The input parameters are the XML policy and the schema needed by the class. The method returns a boolean that indicates whether the user is authorised or not to introduce that policy. In case he is not, an exception is raised. When receiving a call to this method, the ACkr class develops the XML validation tasks just described.

Table 4 - 12. The ACkr class interface description table

c Sequence Diagrams

As in the previous component descriptions, we provide hereafter a sequence diagram to facilitate the description of the component behaviour.

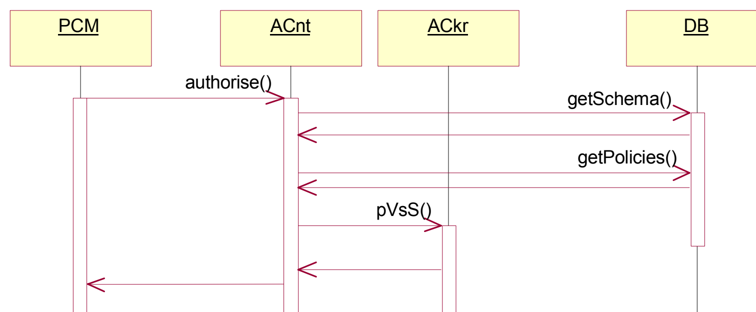

Figure 4 - 32. Authorisation Check Component sequence diagram

4th Policy Conflict Check

A Component Behaviour

When a set of policies do not contradict each other (they do not conflict), the set is considered consistent. In [Verma00] they define a policy conflict as:

"A set of policies is consistent if it can be shown that no contradictory policies will ever be found"

In MANBoP, the PCC component will have the responsibility of detecting, and if possible solving, these contradictions so that the set of policies introduced in the system are always consistent.

Taking into account the table of components and interfaces given in section IV.1 (Table 4 - 1) the main tasks that must be fulfilled by the Policy Conflict Check (PCC) component are:

- *Check for policy conflicts*: This task is requested through the checkConfl() method. It consists on verifying that the new policy does not break the consistency when introduced in the system. Thus, the check is done at policy introduction time.
- *Check for dynamic conflicts*: In this case the task is requested through the checkDyn() method. The PCC component checks if the policy that is going to be enforced conflicts with other policies previously enforced. This might also include delegation policies that determine the resources allowed for a user. This check is done at enforcement time because in some cases is not possible to predict when a policy is going to be enforced, and thus, whether there is going to be a conflict.

In addition to these two tasks, the PCC component must essay to resolve any detected conflict. To do this task, the component uses a priority property included within policies. That is, if a conflict between two policies is found, the one with the highest priority is introduced and the other one removed. This might be particularly useful, for example, for guaranteeing that network operators fault management policies (e.g. policies oriented to congestion avoidance), will be enforced even if they conflict with lower priority user policies. Obviously, whenever a user policy is removed because conflicting with a higher priority policy, the user must be notified.

Another important remark is that, when the TEManager component is not able to find a route with enough resources for a flow, it returns the route with the minimum cost and a list of the conflicting resources in that route. Hence, based on this physically-possible route and policy priorities, the PCC component will take a decision.

One additional task that should be carried out by the Policy Conflict Check component is the recompilation, analysis and reporting of resource scheduling and consumption status in underlying devices. These reports are generated periodically and introduced within the notification channel. PCC components, except those working directly over the managed devices, will register at bootstrap in the Notification Service for receiving these reports from the underlying MANBoP managers.

Despite the apparently simple functionality of this component if we just take into account the above description, the realisation of these checks based on the available resources, network topology and user rights is not at all straightforward. The activity diagrams of the component, given hereafter,

reflects the complexity of these tasks and complement the textual description just given.

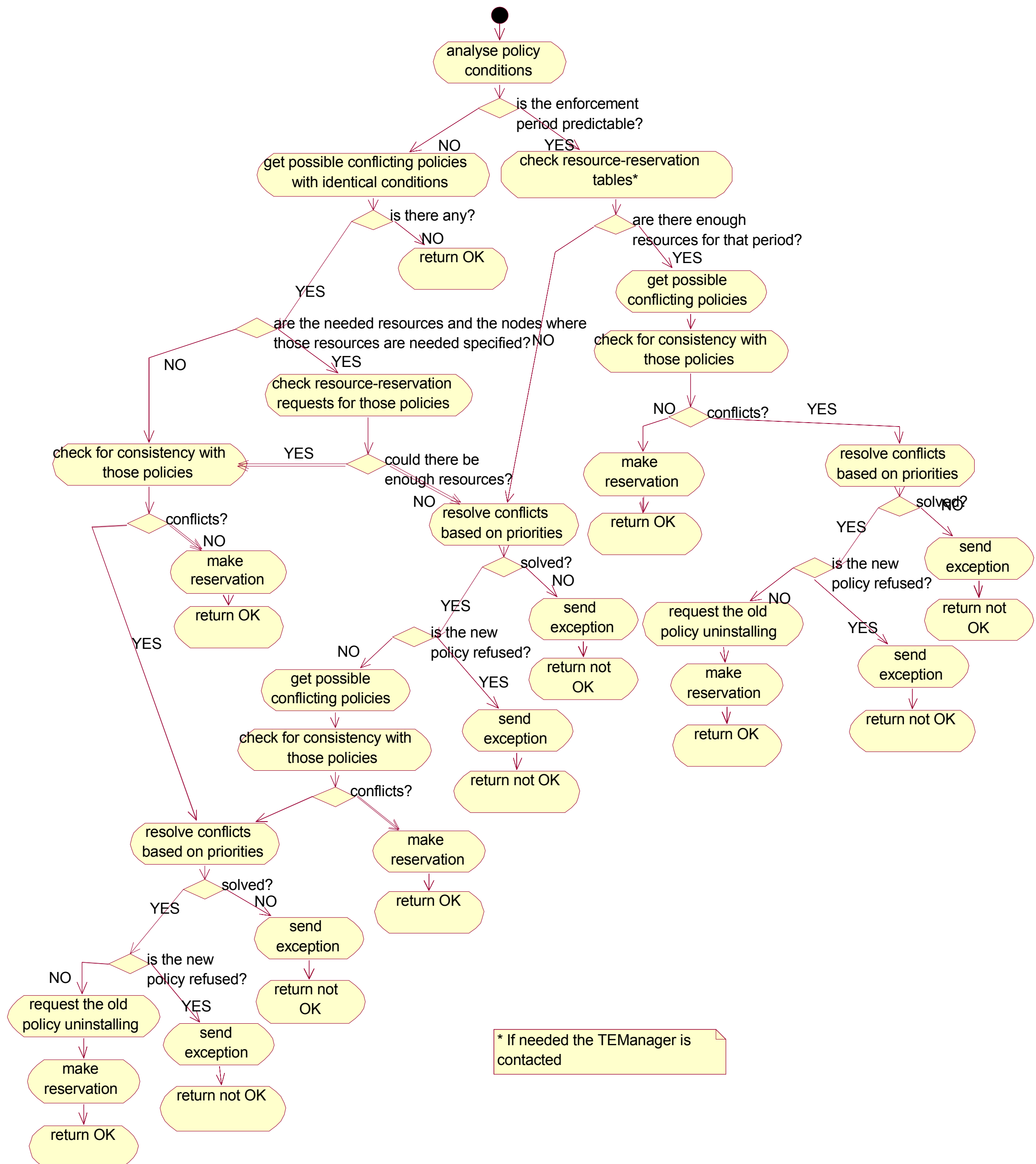

Figure 4 - 33. Policy checking activity diagram

Figure 4 - 33 shows the tasks carried out by the PCC when a new policy conflict check request is received. It guides the rest of the process and establishes the time interval during which the new policy will be enforced (the enforcement interval). Such information is needed to assess the availability of the requested resources at that time and to schedule the allocation of resources. Nevertheless, sometimes the enforcement interval is not foreseeable and therefore this scheduling task cannot be realised.

When the enforcement interval can be established the PCC component finds out if the requested resources are available, not only globally but also for that particular user. These resources might be needed all along two end-points of the managed network. In this case, the concrete route with the available requested resources should be found by the TEManager[10] component, which would be contacted by the PCC. If there are not enough resources, it tries to solve the conflict based on the requests priorities.

Once the resource scheduling is solved, and obviously only if the new policy has not been rejected along this process, the component looks for policies previously received that might create an inconsistency with the new one. Then, it checks if such inconsistency exists. Again, if a conflict is found, it tries to solve it based on the requests priorities. Finally, depending on the process result, the resource scheduling information might be modified and some policies might be requested to be uninstalled to avoid detected conflicts.

In those situations when the enforcement interval is not foreseeable, the component looks for policies that share the same conditions with the new one: those are the only ones that we can assure that will be enforced at the same time. Then, the PCC checks whether the requested resources (e.g. including nodes where these resources are requested in) are already known, or they will only be known at enforcement time. This check is realised to assess the feasibility of making an initial check of resource availability for those policies. Anyway, all these policies are checked for consistency, and if a conflict is found, the component tries to solve it. Finally, the resource information is updated accordingly.

As already described in the introductory paragraphs of the component, before the enforcement of a policy another check is realised: a dynamic conflict check. The activity diagram below shows the main tasks realised by the component when such a check is requested.

[10] The TEManager could be seen as a subcomponent of the PCC since its functionality is complementary. Nonetheless, both its complexity and its necessity just at network or sub-network levels have moved us to design it as a separate component.

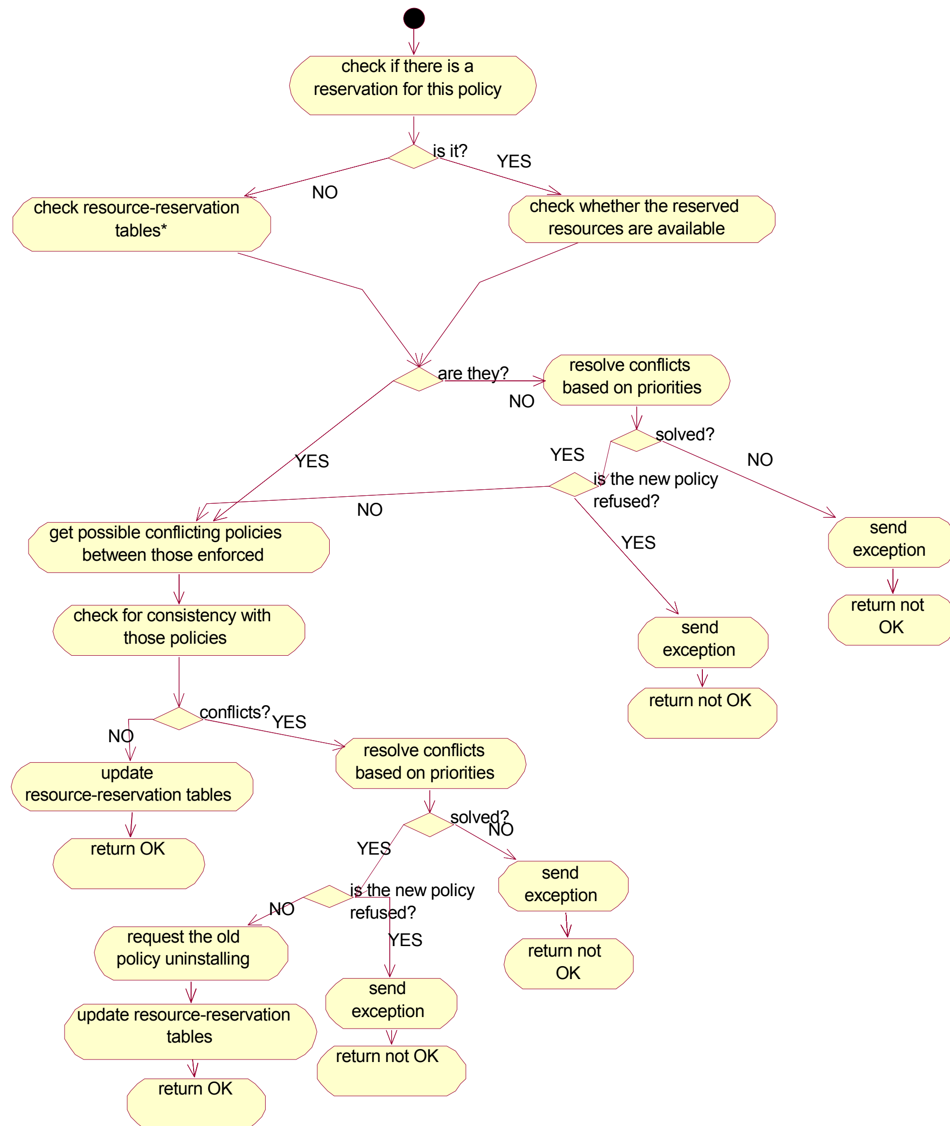

Figure 4 - 34. Dynamic conflict checking activity diagram

Since all policies should have been previously checked for conflicts by the PCC component, the first task realised in the dynamic check is finding out if a reservation has been scheduled in that process (i.e. if the enforcement interval was predictable). When there is such a reservation, the component just verifies that the scheduled resources are still available, and solves the conflict if they are not. On the other hand, when there is no scheduled reservation, the component checks whether the requested resources are physically available for that user within the network. Again, if not enough resources are found, the system tries to resolve the conflicts based on the requests priorities. Once the resource conflicts are solved, the system gets the policies

that might create an inconsistency with the one that is going to be enforced (i.e. all enforced policies from a potentially conflicting domain) and looks for such an inconsistency. If any is found, it tries to resolve it. Otherwise, the resource information is updated accordingly and, if necessary, conflicting policies are requested to be uninstalled.

Although not shown in the activity diagrams, the PCC component must also be capable of removing all resource reservation requests information related with a policy when this policy is removed. Therefore, each time a policy is removed, the Policy Consumer Manager requests to the PCC the removal of this information. Then, the PCC retrieves from the database the resource information linked with the policy, and it modifies the resource information kept in the database accordingly.

Finally, the last important task that must be executed by the PCC, not shown in the activity diagrams, is report processing. That is, receiving reports from underlying devices, updating resource information based on these reports, requesting when appropriate the re-calculation of paths costs to the TEManager component and building reports for higher-level instances. These tasks will be described in more detail within the RpProc class description sub-section.

B Component Design

The Policy Conflict Check (PCC) component design presented in this section does not pretend to suggest an innovative conflict-checking algorithm but just fulfilling the expected functionality for proving the concepts of the management framework presented. We should not forget that the focus of this doctoral thesis is the proposal of a framework oriented to the management of active and programmable networks. Hence, the concrete policy-related logic such as policy repository, policy conflict checking and policy translation is faced just from the functional point of view (we do not consider the performance properties of these mechanisms) to prove the global framework. Indeed, an advanced work within each of the above-mentioned topics itself could be the subject of a doctorate thesis.

Complementarily, the implementation of the PCC component can be faced in two ways; looking for an easy as possible component upgrade with new functional domains, or looking for a better component performance properties.

We should point out also that the logic of the PCC component, as already described in previous chapters, depends on the management level at which the component acts, as well as on the functional domains it supports. Thus, the design presented here should be particularised to the real environments where the different versions of the component are placed and needed.

In the component behaviour description above, we have enumerated a number of tasks the component must develop. These tasks can be grouped as shown below:

- Tasks related with an analysis of policy conditions and checking requests for identifying the next steps to be taken.
- Functionality that copes with the search of requested resources within the managed devices and the scheduling of resources.
- Maintenance of the correct resource information within the database.
- Functionality aimed to look for potentially conflicting policies within the database.
- Logic that certifies that a set of policies is consistent.
- Tasks that in case of conflict (either because of resource unavailability or consistency) try to resolve such a conflict based on requests priorities.
- Tasks related with resource information reports (i.e. their construction and processing).

These groups of tasks lead us to the component design shown in Figure 4 - 35. Each class in the diagram will mainly cope with one of the task groups above stated. The concrete description of how they handle with that functionality is given in the class description sub-sections hereafter.

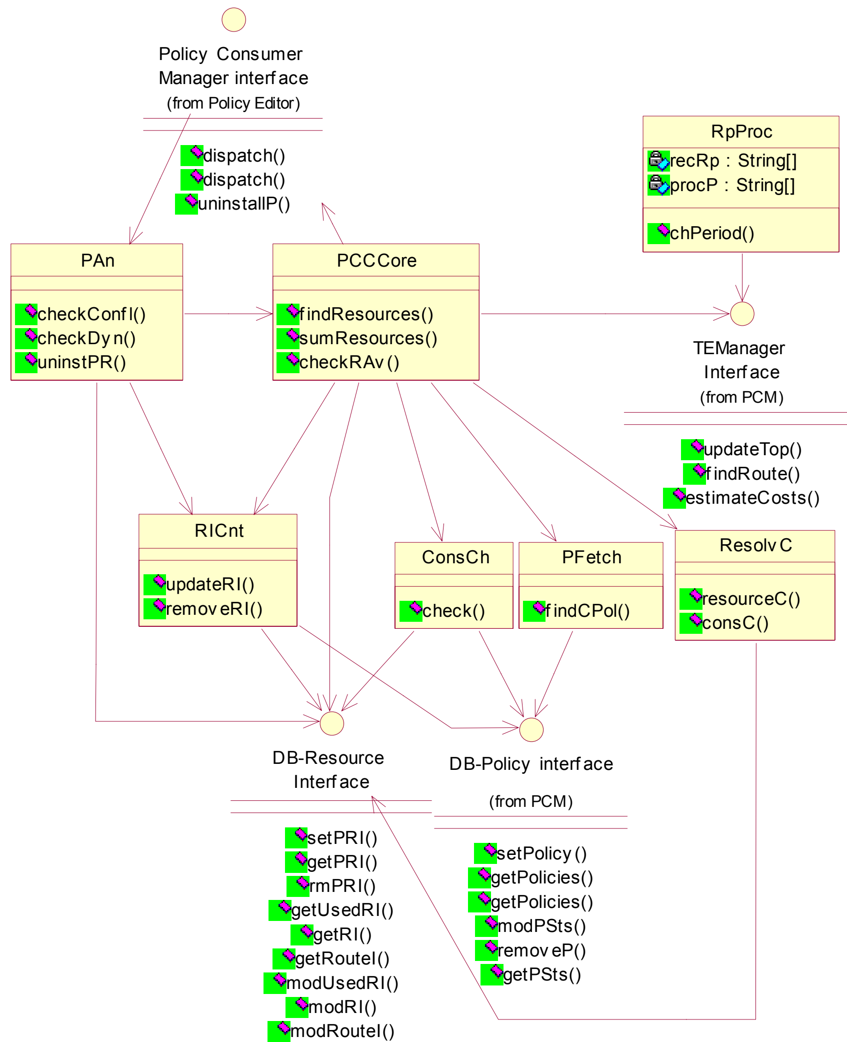

Figure 4 - 35. Policy Conflict Check class diagram

a PAn class

The Policy Analysis (PAn) class is mainly responsible of receiving check requests from the Policy Consumer Manager. Then, analyse these requests (i.e. analyse policy conditions to find out, when possible, the enforcement interval) and request the appropriate conflict checks and resource reservation schedules based on this analysis. The class is also responsible of returning the check result to the PCM component.

The logic designed within this class is not extraordinary complex. We describe this logic for each one of the methods offered by this class in the following table.

Interface	Input parameters	Output parameters	Functionality
checkConfl()	string policyId, credential User	boolean Result	The PCM component will make a call to this method to request the conflict check of a new policy received. The method is defined with two input parameters and just one boolean parameter as check result. The first input parameter is a string representing the policyId that uniquely identifies the policy to be checked. The second one represents the credentials of the user, needed for checking that the user-allowed resources are not overridden. When receiving a request through this method the PAn class tries to establish the enforcement period for this policy and request the corresponding checks to the PCCCore class accordingly.
checkDyn()	string policyId, credential User	boolean Result, string[] nodeId	The Policy Consumer Manager uses this method to request a dynamic conflict check to the PCC before the actual enforcement of the policy. The method includes two input and two output parameters. The input parameters are the same as the ones described for the previous method. As output parameters, in addition to the boolean result (mentioned briefly in the previous method), the method returns a list of nodes where the policy should be enforced. If not applicable (i.e. when working at the element level) this parameter is simply ignored and returned with null value. The PAn class checks whether the policy to be checked has obtained, in a previous policy conflict check, any kind of resource reservation schedule. Based on this information it requests the corresponding checks to the PCCCore class.
uninstPR()	string policyId, boolean cause	boolean Result	This method is used by the PCM component to request the removal, or update, of policy-related resource information. The method specifies two input and a single output parameter. The input parameters are the policyId that identifies the policy whose resource information should be removed (or updated when the policy is not removed but de-enforced). The boolean 'cause' specifies if the policy is being removed or if it is being de-enforced. The output parameter is just a Boolean that indicates if the requested action has been realised correctly. When receiving a call to this method, the PAn class contacts the RICnt class to request the realisation of the expected functionality.

Table 4 - 13. The PAn class interface description table

b PCCCore class

The PCCCore class is probably the most important class within the PCC component. It is in charge of coordinating the whole checking process, finding requested resources, maintaining resource schedules, contacting the TEManager component if necessary, deciding what policies should be uninstalled (based on the results from the ResolvC class) and requesting their removal to the PCM component.

The functionality, briefly summarised in the previous paragraph, is not as straightforward as it might seem, especially the functionality related with resource searching and scheduling. For that reason we provide below an activity diagram together with a detailed description for each of the three methods that this class incorporates: findResources(), sumResources() and checkRAv().

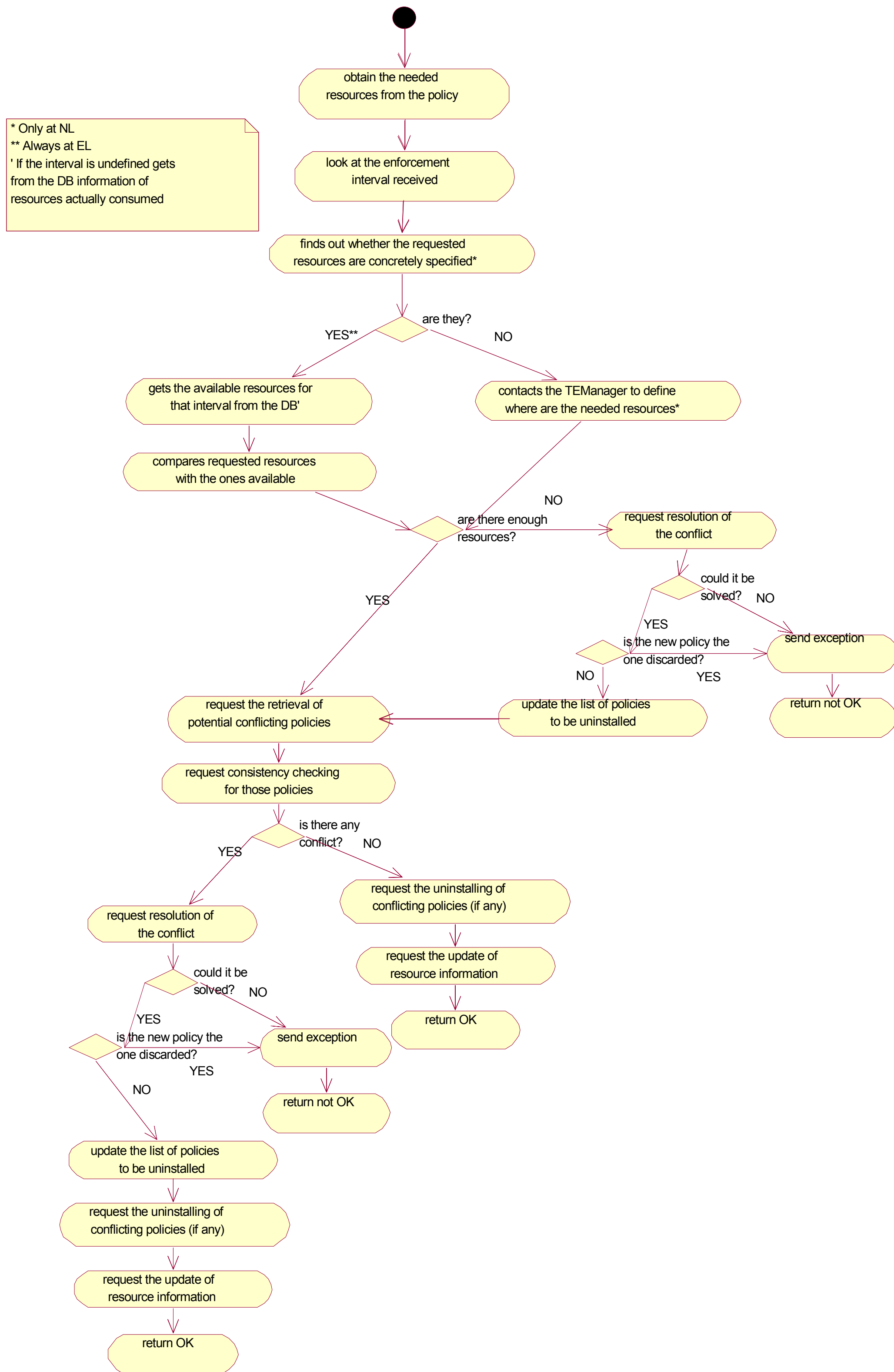

Figure 4 - 36. PCCCore class: findResources activity diagram

The activity diagram above shows the main tasks that the PCCCore class realises when a request is received through the findResources() method.

First, the class obtains the requested resources information from the policy being checked. Moreover, the PCCCore class identifies if the requested resources must be reserved or instead, they must be allocated directly[11]. This information might be forwarded to the TEManager, when reached later in the process, and to the RICnt class at the end of the process. The PCCCore class also finds out whether the resources are provided to the level of specifying also the nodes where they are requested. That is, it assesses whether the TEManager should be contacted to determine the nodes within the network where the resources can be obtained from or, instead, these nodes are already specified in the policy itself. Together with the enforcement interval (received as parameter), the class either finds out itself if the needed resources are available based on the resource information in the database, or contacts the TEManager to realise this task. The class that will carry out this task is chosen based on the result of the previous assessment. It is important noticing that, when the enforcement interval is undefined and the findResources method is requested, the class will compare the requested resources against those currently used in the managed system since this situation will only happen in a dynamic conflict checking.

In case not enough resources could be found to fulfil the request the PCCCore class requests to the ResolvC class the resolution of the conflict detected. If a solution is found, the PCCCore class receives a list of one or more policies that should be uninstalled to solve the conflict. When the new policy is the one discarded, the class simply raises an exception and returns not OK to the PAn class. Otherwise, it keeps the list of policies to be uninstalled and continues the process.

Once we know that there are enough resources for the new request the PCCCore class asks to the PFetch class the database retrieval of policies that might potentially create an inconsistency with the new one. When a list of policies is returned by PFetch, and if at least one policy is in that list, the PCCCore class sends the list of potentially conflicting policies to the ConsCh class to realise the consistency checking between those policies. If a conflict is found the ConsCh returns the list of conflicting policies to the PCCCore that will forward them to the ResolvC class to request the solution of the conflict based on priorities.

Again, if a solution is found, the ResolvC class returns a list with the policies that should be uninstalled. The PCCCore class checks that neither in this second list the new policy is discarded. In case it is, the class raises an exception and returns not ok. Otherwise, updates the list of policies that must be removed making sure that no policy is repeated.

[11] This information is obtained from the *'act'* field of the policy.

Finally, after the whole checking process is completed the PCCCore class requests to the Policy Consumer Manager component the removal of all policies discarded to solve conflicts (if any) and to the RICnt class the update of the resource information in the database.

Not included in the diagram, to keep it as simple as possible, are the tasks related with the processing of policies with the *'act'* policy field containing a *'Modify'* value. The processing of this kind of policies is quite similar to the other ones, with some particularities.

First, after the request, the first thing the PCCCore realises is checking the *'act'* policy field of the policy received. If the value of this *'act'* field is *'Modify'* then the PCCCore retrieves from the database the policy being modified. It compares both policies, particularly the resources requested. Then, if necessary it tries to accommodate within the network the resource difference between both policies. When there is no resource conflict, it makes the consistency checking against all potentially conflicting policies except the one that is being modified. Finally, after assuring that there are no conflicts the PCCCore requests to the RICnt the update of the resource information related with the policy being modified. Afterwards, the PCCCore changes the status of the policy that has been modified to *'not enforced'* and request its removal to the Policy Consumer Manager, because the new one has already been successfully introduced. The reason for this is that the enforcement of the modify policy will cause the modification of all configurations in the managed devices. Therefore, we do not desire to remove these configurations when uninstalling the old policy. This removal is requested to keep the whole group of policies received consistent and to simplify processes such us the policy expiration and others.

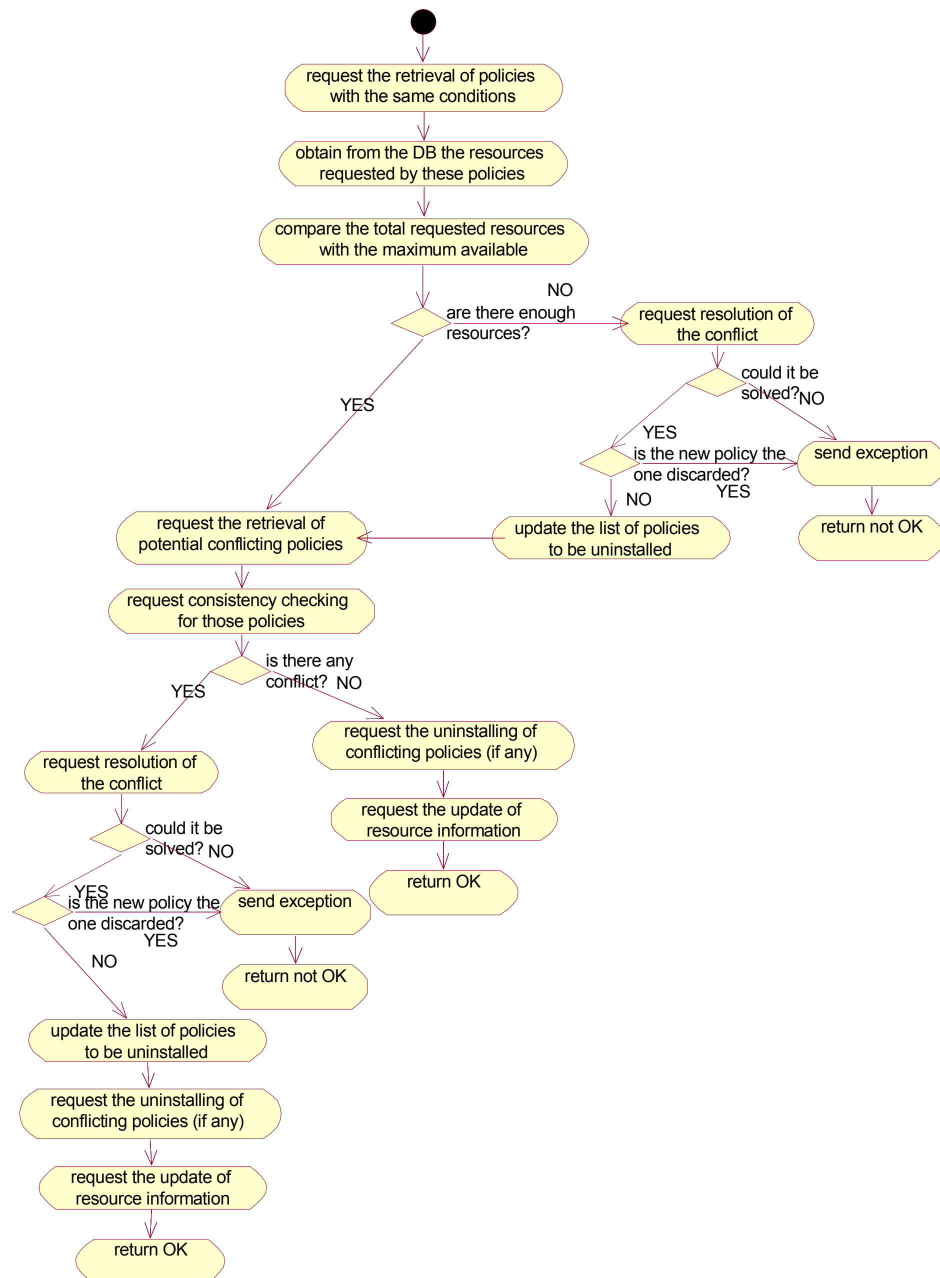

Figure 4 - 37. PCCCore class: sumResources activity diagram

The diagram above shows the main tasks realised within the PCCCore class when a call is made to the sumResources method. The first task realised is to extract from the received policy the conditions to ask to the PFetch class the retrieval from the database of all policies with the same conditions. The rationale behind such task is that we assume that policies with same conditions will be enforced at the same time and on the same managed nodes (if applicable). Hence, they are potentially conflicting policies. Afterwards, it

retrieves from the database the resources requested by these policies and adds them. Once we have the result, the PCCCore class compares it against the maximum capability value for these resources. If the added value is higher, a conflictive situation will occur at their enforcement time. Therefore, the ResolvC is contacted to resolve such a conflict as already described for the findResources activity diagram.

Additionally, once possible resource conflicts have been solved, the PCCCore class will request for a consistency checking between the remaining policies. From this point, up to the end of the process, the tasks developed are the same as those described for the findResources activity diagram.

Again, as in the findResources case, there are a number of tasks within the PCCCore, for the processing of 'Modify' policies, which have not been considered in the activity diagram for sake of simplicity. When a request is received through the sumResources method, the first thing the PCCCore class carries out is checking the *'act'* field of the policy. In case its value is 'Modify' then the PCCCore gets all policies with the same conditions (except the one that is being modified), and assess whether there could be enough resources for them. If so, checks whether there is any inconsistency within the same group of policies. Finally, unless a conflict is found, the PCCCore requests to the RICnt class the update of the resource information related to the policy being modified, and requests to the Policy Consumer Manager the removal of the modified policy after having changed its status to 'not enforced'.

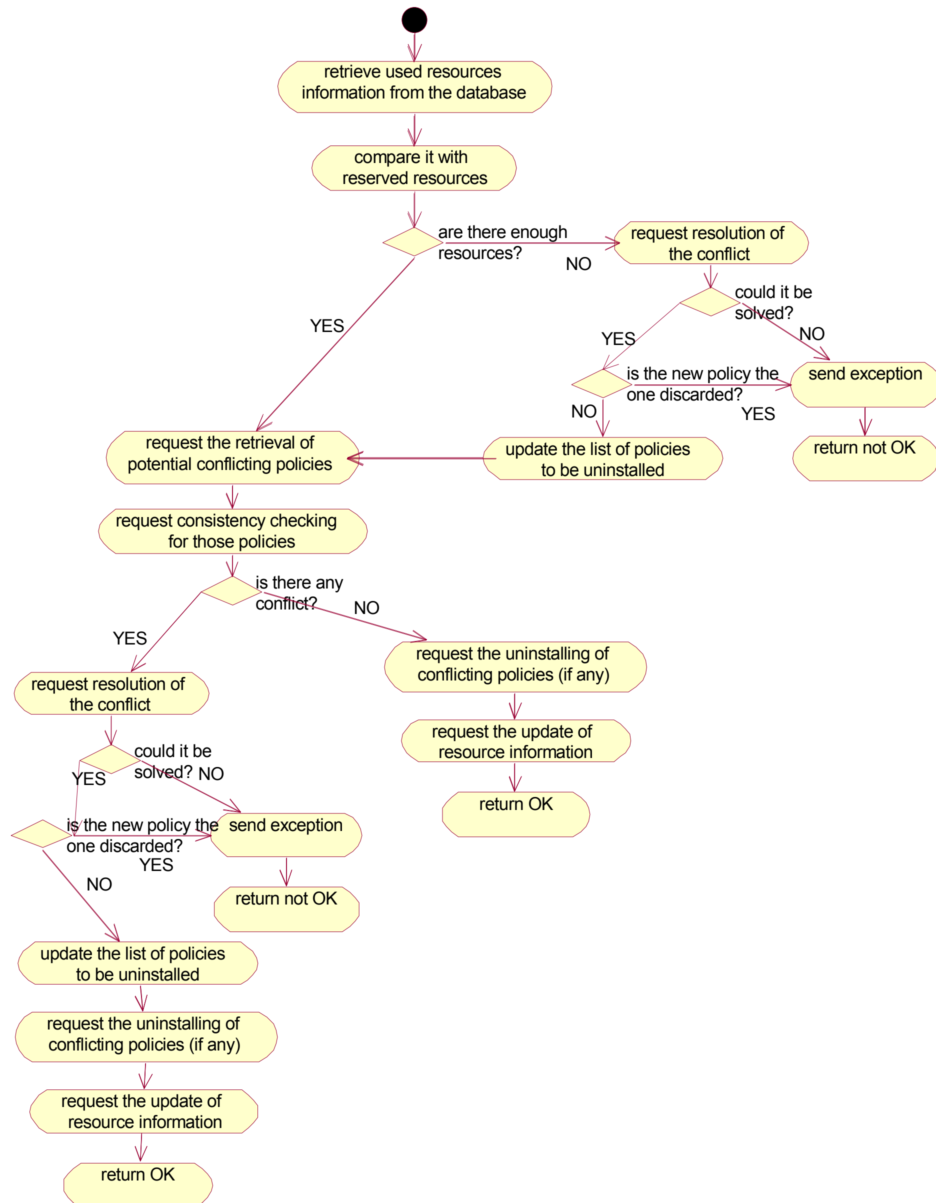

Figure 4 - 38. PCCCore class: checkRAv activity diagram

The checkRAv method will be requested by the PAn class when a dynamic conflict checking is realised for a policy with a previous reservation already scheduled. The tasks realised by the PCCCore class are thereby oriented to confirm the availability of the scheduled resources and solve potential conflicts that might appear.

First, the PCCCore class retrieves from the database the information about used resources on the managed devices and compares it with scheduled resources for that policy. In case the scheduled resources are not available, the

PCCCore class finds out, using the resource information in the database, which are the reservations (and associated policies) conflicting, and requests to the ResolvC class the resolution of such a conflict.

When the scheduled resources are available, or once the resource conflicts have been solved, the PCCCore class requests the retrieval of policies potentially conflicting with the one that is going to be enforced. The PCCCore asks to the PFetch the retrieval of these policies but only among those that are currently enforced. The rationale behind this is that consistency with policies whose enforcement time is detailed has already been done in the policy conflict checking developed when the policy was first introduced in the system. Thereby, we must only do consistency checks against those policies whose enforcement time cannot be foreseen. Additionally, it is only useful to check only against the subgroup that is already enforced, because the others might never be enforced. Even if they do, the corresponding dynamic check will be realised before their enforcement, thus detecting any potential inconsistency.

The other tasks carried out within the PCCCore class for this method have already been described before.

The concrete methods, parameters and behaviour of the interface offered by this class are described in the following table:

Interface	Input parameters	Output parameters	Functionality
findResources()	string policySer, credential User, interval enfInt	boolean Result, string[] nodeId	The PAn class accesses this method for requesting the resources needed by a policy either for an established interval or at enforcement time. The method is defined with three input parameters and two output parameters. The first input parameter is a string with the policy requesting the resources serialised. The second one are the credentials of the user, needed for checking that the user resources are not overridden. The third input parameter is the enforcement interval described in detail below. The output parameters are just a boolean with the result and a list of nodes that will be different from null only when the TEManager has been accessed. interval is an structure defined as: struct interval{int start_time; short moymask; int dommask; byte dowmask; string[] tday; int end_time;};[12] start_time and end_time are 'int' types with the following structure: YYYYMMDDhhmm (Year, Month, Day, hour, minutes) moymask is a mask that indicates the valid months of the year. It is formatted as an octet string of size 2, consisting of 12 bits identifying the 12 months of the year, beginning with January and ending with December, followed by 4 bits that are always set to '0'. dommask is a mask indicating the valid days of the month. This property is formatted as 31 bits identifying the days of the month counting from the beginning. dowmask is a mask that specifies the valid days of the week. This property is formatted as 7 bits identifying the 7 days of the week, beginning with Sunday and ending with Saturday, followed by 1 bit that is always set to '0'. tday specifies the valid times of the day. Each member of the array has the structure: hhmm/hhmm. When receiving a request through this method the PCCCore class will look for the resources requested within the managed devices as described in the Figure 4 - 36.
sumResources()	string policySer, credential User	boolean Result	This method will be used for requesting the checking of possible conflicts due to the introduction of a policy whose enforcement time cannot be predicted. The method includes two input parameters and just one Boolean result as output parameter. The input parameters are a string with the incoming policy serialised and the credentials of the user. The PCCCore class will realise the tasks described in the activity diagram represented in Figure 4 - 37.
checkRAv()	string policyId	boolean Result	This method is called by the PAn class to verify the availability of the scheduled resources for a policy. The method only needs on input and one output parameter. The input parameter is the policyId that identifies the policy whose scheduled resources availability should be verified. The output parameter is just a Boolean that indicates if the requested action has been realised correctly. When receiving a call to this method the PCCCore class will realise the tasks drawn in Figure 4 - 38.

Table 4 - 14. The PCCCore class interface description table

c RICnt class

The Resource Information Controller (RICnt) class develops two tasks: updating the resource information in the database whenever a new reservation is made, and removing the policy-related resource information as part of the policy removal process. In addition, the RICnt also updates the

[12] This structure for identifying the enforcement period of a policy is extracted from the IETF PCIM RFC [Moore01], [Moore03].

policy-related resource information when it is modified by a new policy or when the policy is de-enforced.

The RICnt class stores the resource information in the database taking into account the type of request parameter. This parameter might get one of the following values:

- Type 0 (reservation schedule): A reservation schedule occurs when the enforcement interval of the request is foreseeable and resources can be assigned and scheduled in advance (i.e., when the policy is introduced in the system). The scheduling "reserves" some resources. That is, these resources will be split between multiple posterior schedules (either reservation or allocation schedules) from the same user. It can be seen as a virtual tunnel (when forwarding resources), or device (when computing resources), with certain Quality of service.
- Type 1 (allocation schedule): An allocation schedule is exactly the same as type 0 except that resources are allocated to one or more flows which make the physical use of this resources. Thus, the resources cannot be used by other flows.
- Type 2 (undefined schedule): Occurs when a policy whose enforcement interval cannot be foreseen is introduced in the system. At introduction time, the requested resources are simply stored in the database. No scheduling can be done until enforcement time.
- Type 3 (scheduled reservation enforcement): A scheduled reservation enforcement occurs whenever a policy whose enforcement interval was predictable (thus, obtained a reservation schedule at introduction time), is enforced.
- Type 4 (scheduled allocation enforcement): Exactly as the previous type except that the schedule was an allocation schedule instead of a reservation one.
- Type 5 (unscheduled reservation enforcement): It occurs when a policy whose enforcement interval was not predictable is enforced and, in addition, the policy requests that resources are reserved, not allocated.
- Type 6 (unscheduled allocation enforcement): As before except that the resources are allocated instead of reserved.

The activity diagram below shows the tasks carried out by the RICnt class whenever an update of the resource information of type 0 is requested.

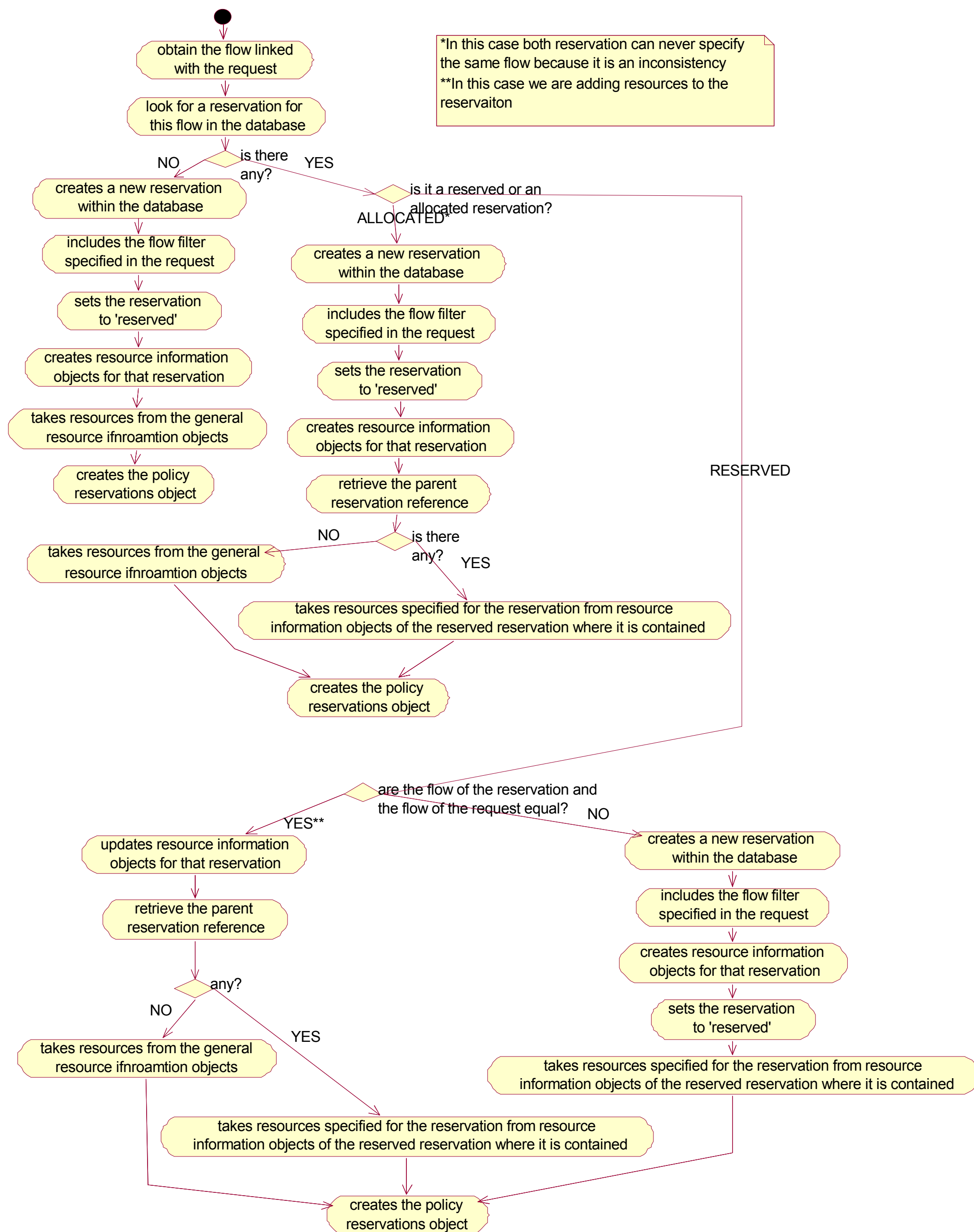

Figure 4 - 39. RICnt class: Reservation Schedule activity diagram

All tasks realised are oriented to establish what information is already in the database and should be modified only (such as a previous schedules linked to this one either because this one extends the resources assigned to the previous one or because it gets some of the reserved resources), and what information does not exist yet, and thus, should be created now.

Schedules are linked to flows assigned to them. Hence, the RICnt class looks for a schedule that applies to the flow specified in the request. If the flow in the previous available schedule is the same as the requested one, and the schedule is of the reservation type, we assume that we are adding resources to the previous reservation schedule. Hence, former reservation schedule information is updated (i.e. a new reservation information is not created in the database). If it is a sub-flow contained within the flow filter specified in the reservation schedule, then we create a new reservation and put the former one as parent reservation. We should also create the resource information for that reservation schedule and update, with the requested resources, the parent reservation resource information, if any exist, or the overall resource information instead.

When the schedule retrieved from the database based on the requested flow is an allocation schedule, hence its resources cannot be used by a new resource schedule, we get the parent schedule of the allocated one (it would be the same parent schedule to the current one) and update its resource information. We also create, as explained before, a new reservation schedule with all the necessary resource information in the database.

A parent schedule can only be of the reservation type, because it is the only one that permits, indeed it is its goal, to split its resources into multiple resource schedules.

The same concepts apply when the request is of type 1. However, the tasks vary slightly. These tasks are shown in the activity diagram below.

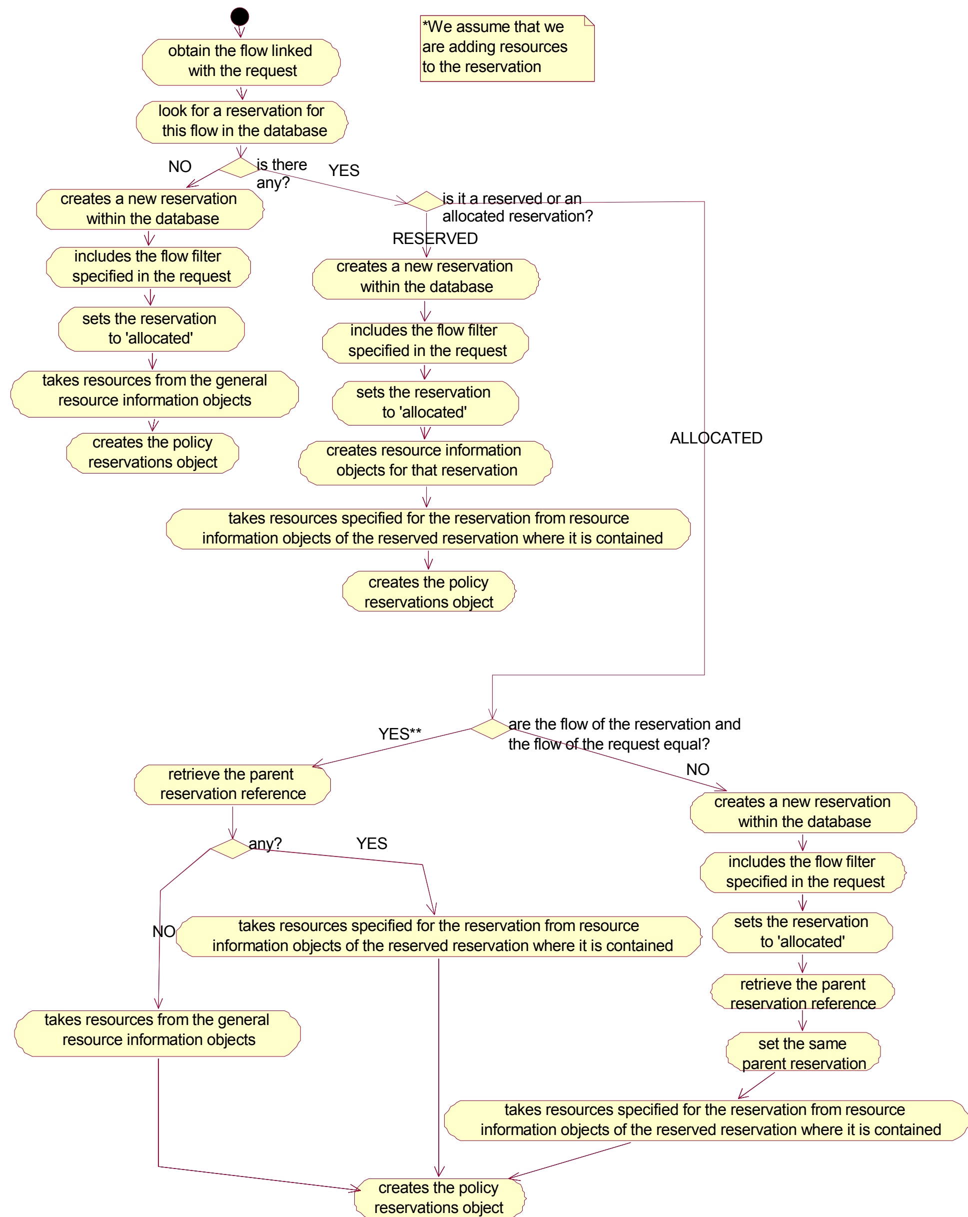

Figure 4 - 40. RICnt class: Allocation schedule activity diagram

When the RICnt class receives a 'type 2' request it simply create the policy schedule objects[13] file with the requested resource types.

In case of type 3 and type 4 requests the RICnt class retrieves the resources specified in the policy schedule objects and updates the used resource information of the parent information as well as the overall used resource information.

Finally, for type 5 and 6 requests the behaviour of the RICnt class will be exactly the same as the one shown in previous activity diagrams (i.e. Figure 4 - 39 and Figure 4 - 40 respectively) except for the fact that instead of dealing with the scheduled resource information, the RICnt class will create the schedule with the used resource information.

So as to realise these two tasks the class offers two methods, namely updateRI() and removeRI(). The concrete description of goal, functionality and parameters for these methods is given in the table below:

[13] Policy schedule objects keep the mapping between a policy and the schedules caused by that policy

Interface	*Input parameters*	*Output parameters*	*Functionality*
updateRI()	String policySer, flow assocFlow resource[] resources, credential User, string[] nodeId, int type, boolean modify	boolean Result	The PCCCore class accesses this method for requesting the update of the resource information available in the database after both policy and dynamic conflict checks are completed. The method specifies seven input parameters and a boolean as result. The 'type' parameter is an integer that determines the status of the scheduling process after the checks and therefore the resource information that should be updated. The possible values are: (0) reservation schedule, (1) allocation schedule, (2) undefined schedule, (3) scheduled reservation enforcement, (4) scheduled allocation enforcement, (5) unscheduled reservation enforcement and (6) unscheduled allocation enforcement. The first three values might be used after a policy conflict check, while the last three might be used after a dynamic conflict check. This value also determines which of the incoming parameters should be taken into account by the RICnt class. Another input parameter is a string identifying the policy that has been checked. The credentials of the user, needed for updating user's resource information are also introduced. When working at the network or sub-network levels, the list of nodes where the policy should be enforced might be also introduced in the system. The flow associated to the request is introduced in a flow structure. The structure is defined as: struct flow{string srcIP; string destIP; string srcPort; string destPort; string Prot;}; All fields inside the structure, namely source and destination IP addresses, source and destination ports and protocol are specified as strings to allow the use of filtering characters such as '*'. Another input parameter introduced in the function is the list of resources. This parameter is an array of resource structures. The structure is defined as: struct resource{int resourceType; string resourceId; interval enforcementInt; int reqValue;}; The resourceType is an integer that identifies the kind of resource requested (e.g. bandwidth (0), CPU (1)...). The resourceId is a string that uniquely identifies the resource requested, the interval parameter described before (pg. 131) defines the scheduled enforcement interval. Finally, the reqValue is an integer that identifies the requested quantity of this resource. Finally the modify parameter is a boolean that indicates whether the resource information is caused by a new policy (0) or by a modification of an existing one (1). When receiving a request through this method the RICnt class will update the resource information available in the database based on the 'type' parameter as described throughout the section. Additionally when the modify parameter is set to (1) the RICnt class will first remove the resource information of the 'old' policy and then create the appropriate resource information corresponding to the new one.
removeRI()	string policyId, boolean cause	boolean Result	The PAn class will call this method to request the removal from the database (or update when de-enforced) of all resource information related with a policy. As input parameters of the method the policy identifier and a boolean specifying whether the policy is being removed (0) or de-enforced (1) are given. The output parameter is a Boolean indicating the result of the process. The RICnt class, based on the policy identifier, retrieves the policy requested resources and user credentials, as well as the policy status to know whether the policy is currently enforced or not. With this information it updates the resource information in the database: removing the resources linked to this policy when it is being removed, or just removing the corresponding used resources information when it is being de-enforced.

Table 4 - 15. The RICnt class interface description table

d PFetch class

The Policy Fetch (PFetch) class is responsible of searching for policies that might potentially conflict within the database in order to realise the conflict

checks over a limited set of policies and not over all policies received in the system. It decides which are the potentially conflicting policies based on the interval when the policy should be enforced, the role of the policy to be checked, the functional domain identifier and the resources affected by the policy request.

The interval will be used by the PFetch class to determine the method used for searching the potentially conflicting policies. When the enforcement interval is clearly specified, the PFetch class retrieves from the database those policies wit the same enforcement interval and resources, as well as those with potentially conflicting roles and functional domains. In case the start_time of the interval has a '0' value (i.e. specifies that the policy is going to be enforced immediately and for an undefined period of time), the PFetch also gets those policies sharing the same properties as above but being also currently enforced. Finally, when the start_time of the interval has the maximum possible value (the enforcement period is unknown), the PFecth class gets from the database those policies with identical conditions from potentially conflicting roles and functional domains.

So as to realise this functionality the PFecth offers an interface with the findCPol() method. The goal, functionality and parameters in this method are described in table below.

Interface	*Input parameters*	*Output parameters*	*Functionality*
findCPol()	string policySer, interval enfInt, string[] resourceID	boolean Result, string[] policyId	The PCCCore class accesses this method for requesting the retrieval of potentially conflicting policies from the database. The method specifies three input and two output parameters. The first input parameter is the serialised policy being checked. The enforcement interval is also passed as parameter. The interval structure has already been described in pag.139. Finally, the last input parameter introduced in the function is the list of resource identifiers involved in the policy request. As output parameters, a Boolean indicates the result of the operation and an array of strings contains the list of policy identifiers of potentially conflicting policies. When receiving a request through this method the PFetch class carries out the task already mentioned in the class description above.

Table 4 - 16. The PFetch class interface description table

e ConsCh class

The Consistency Checking (ConsCh) class is responsible of determining whether a group of policies is consistent or not.

The logic within this class is clearly dependant on the management level and functional domains supported by the current PCC component version. Hence, we will not describe in detail the tasks realised by this class but just give some hints about them. The main task is to determine whether the action properties and values are consistent or not. Therefore, the class should be able to interpret the policy actions and establish which potentially inconsistent actions are really conflicting based on the action properties. The ConsCh

receives only those policies that are potentially conflicting because of their enforcement time, involved resources, roles and domains, thus the ConsCh does not need to realise this task but just assessing if there is a real inconsistency between those policies based on the action values.

The class offers an interface with the check() method to realise this functionality. The goal, functionality and parameters of this method are described in the table below.

Interface	*Input parameters*	*Output parameters*	*Functionality*
check()	string[] policyId	boolean Result, string[] policyId	Either the PCCCore class or the ResolvC might access this method for requesting the consistency check among a group of policies. The method specifies one input and two output parameters. The input parameter is an array with the group of policy identifiers that must be checked for consistency. As output parameters, a Boolean indicates the result of the operation and an array of strings contains the list of policy identifiers of conflicting policies. When receiving a request through this method the ConsCh class does the tasks already mentioned in the class description above.

Table 4 - 17. The ConsCh class interface description table

f ResolvC class

The functionality of the Resolve Conflict (ResolvC) class is mainly resolving either resource or consistency conflicts between policies based on the priorities of the requests.

The algorithm implemented by this class first looks for the lowest priority policy and, after verifying that is lower than the corresponding one from the policy checked, checks whether removing this policy the conflict is solved. When the conflict is a resource-sharing conflict, it is inside the class where this first check is realised. Otherwise, that is, when the conflict is of the consistency type, the ResolvC class contacts the ConsCh class to realise the consistency checks with the remaining policies.

In case the conflict remains, the policy with the lowest priority among those remaining is removed (only if its priority is lower than the one of the policy checked). Then, the class checks again whether with the remaining group the conflict is solved. This process is repeated until the conflict is solved. Then, it returns 1 and the list of policies to be uninstalled[14]. If for whatever reason it cannot be solved the ResolvC returns 0.

The algorithm can be tuned a bit more if once the conflict is solved, it checks if it is really necessary to remove all policies in the list. In this case, the ResolvC might check whether not uninstalling the one before the last policy

[14] When the policy to be removed is the new one because removing policies with lower priority does not solve the conflict, it will always be the only policy in the list. The rationale is that without the new policy there was not any conflict so there is no need to remove also lower priority policies.

selected the conflict is solved and if so, repeating the process with the others except for the last policy selected. The reason for this is that it might eventually happen that the removal of a policy with higher priority than others solves on its own the conflict and thus all the lower priority policies previously selected are unnecessarily removed.

When the conflicting resources are scheduled, i.e. 'type' parameter value 0 (see below), it might happen that during the future enforcement interval of the request one or more policies will get and free a certain amount of these resources. Therefore, it might be necessary to uninstall different lower priority policies to free the missing amount of resources during the whole enforcement interval. For this kind of checks, the ResolvC will need to access the resource information available in the database.

The interface offered by this class defines two methods, namely the resourceC() and the consC() methods. These methods are used for requesting the resolution of a resource conflict or a consistency conflict respectively. The goal, functionality and parameters of these methods are described in the table below:

Interface	*Input parameters*	*Output parameters*	*Functionality*
resourceC()	string policySer, cRes[] conflRes, int type	Boolean Result, string[] policyId	The PCCCore class uses this method to request the resolution of a resource usage conflict. The method specifies three input and two output parameters. The input parameters are the serialised policy being checked, an array of cRes structures with information about the conflicting resources and the involved policies, and an integer determining whether the conflicting resources are those enforced(1) or scheduled(0). As output parameters, a Boolean indicates the process result and an array of strings determines what policies that must be uninstalled to solve the conflict. The cRes structure is defined as: struct cRes{string resourceId; int misValue; interval conflInt; string[] policyId;}; ResourceId identifies the conflicting resource. misValue identifies the quantity of resources that are missing. 'conflInt' specifies the interval during which the specified resource is missing in the misValue quantity (for the interval structure description see pag.139). Finally the array of policies that request the resources during that enforcement interval is given. Whenever the resourceC() method is called the ResolvC class will realise the tasks already mentioned in the class description above as resource conflict solution techniques.
consC()	string policySer, string[] policyId	boolean Result, string[] policyId	The PCCCore class accesses this method for requesting a consistency conflict resolution based on the policy priorities. The method specifies two input and two output parameters. The input parameters are the policy being checked serialised and an array with the policy identifiers of the conflicting policies. As output parameters, a Boolean indicates the result of the operation and an array of strings contains the list of policy identifiers of those policies that should be removed to solve the conflict. When receiving a request through this method the ResolvC class will realise the tasks already mentioned in the class description above.

Table 4 - 18. The ResolvC class interface description table

g RpProc class

The report processing functionality consists of two main groups of tasks: one in charge of periodically building resource information reports, and one responsible of updating resource information with reports received from underlying MANBoP instances. Obviously, the later is only developed when the MANBoP instance is not working directly over the managed devices.

To carry out periodically the resource information report construction, the component retrieves the needed resource information from the database. The information that should be included in the report is that needed by a higher-level. That is, the resource information from a network should be just that of minimum cost paths between each two end-points since for the higher-level instance it would be mapped as two "nodes" and a "link". To achieve a more efficient resource management, we also include the information about the maximum available resources individually requested between two end-points[15]. All this information is given both as actually used resources and as resource reservations scheduled for the future. Finally, the identifier of the last policy processed in the system (which is kept within the component), is also included, so that higher-level instances can re-adapt the resource information received in the reports if they have processed other policies after that one.

When the component receives a resource information report, it updates the resource information stored in the database accordingly. Additionally, it compares the policy identifier received in the report with the list of policies processed since the last cost re-calculation. If any policy has been processed after the policy identified in the report, the resource information is also updated based on the schedules that these policies might have caused. The next step carried out by the component is updating the list of received reports since the last cost re-calculation. When the report was the last one among all reports from all underlying devices (it compares the number of received reports since the last cost re-calculation with the number of underlying devices), the TEManager is requested to re-calculate the costs of the paths within the managed network. Finally, also the lists of received reports and processed policies are restarted.

The activity diagram that follows reflects these tasks:

[15] For example, paths with lower computational costs and paths with lower forwarding costs.

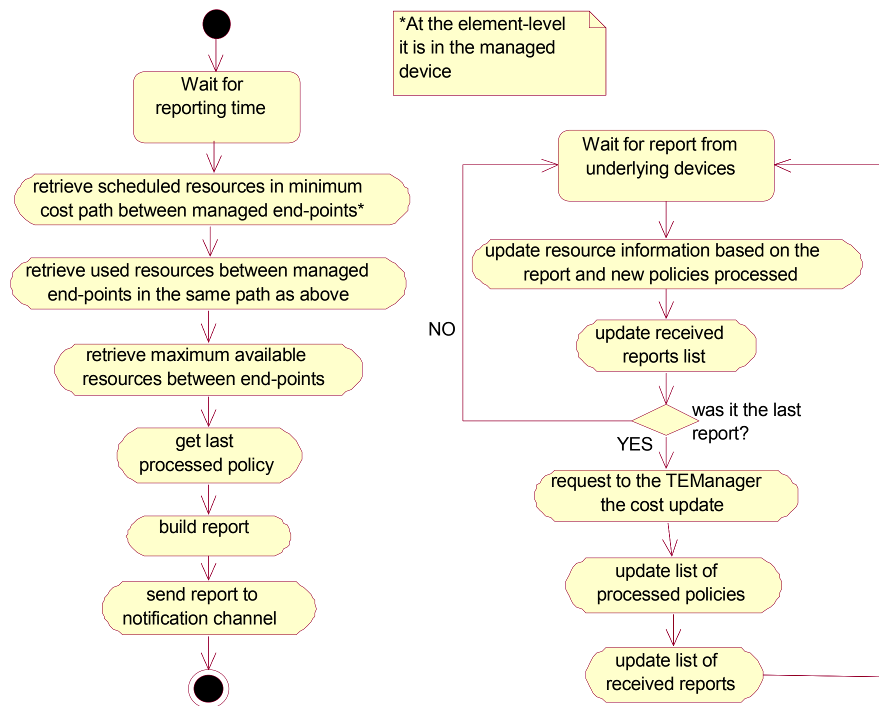

Figure 4 - 41. RpProc class: Reports processing activity diagram

The RpProc class is started whenever a new PCC component is instantiated within the MANBoP framework and works without interruption during the whole life of the PCC component.

When the PCC component is removed (it might be removed when replaced by a new one), the arrays with the received reports and processed policies lists are stored in the database from where they can be retrieved by the RpProc class of the new PCC component.

The RpProc class interface offers only one method. This method can be used directly by the network operator or by a Policy Consumer component specifically designed for the configuration of this facet of the MANBoP instance itself. The method changes the period at which the information reports must be re-calculated and introduced in the notification channel.

Interface	*Input parameters*	*Output parameters*	*Functionality*
chPeriod()	int period	Boolean Result	The network operator or an appropriate Policy Consumer can use this method to change the period at which reports should be built and introduced in the notification channel. The method specifies one input parameter, an integer with the new reporting period, and one output parameter, a boolean that indicates if the change has been done successfully or not. Whenever the chPeriod() method is called the RpProc class will update the reporting period accordingly.

Table 4 - 19. The RpProc class interface description table

b *Sequence diagrams*

With the sake of completing the description of the Policy Conflict Check component, we include below the most common sequence diagrams among the classes of the component.

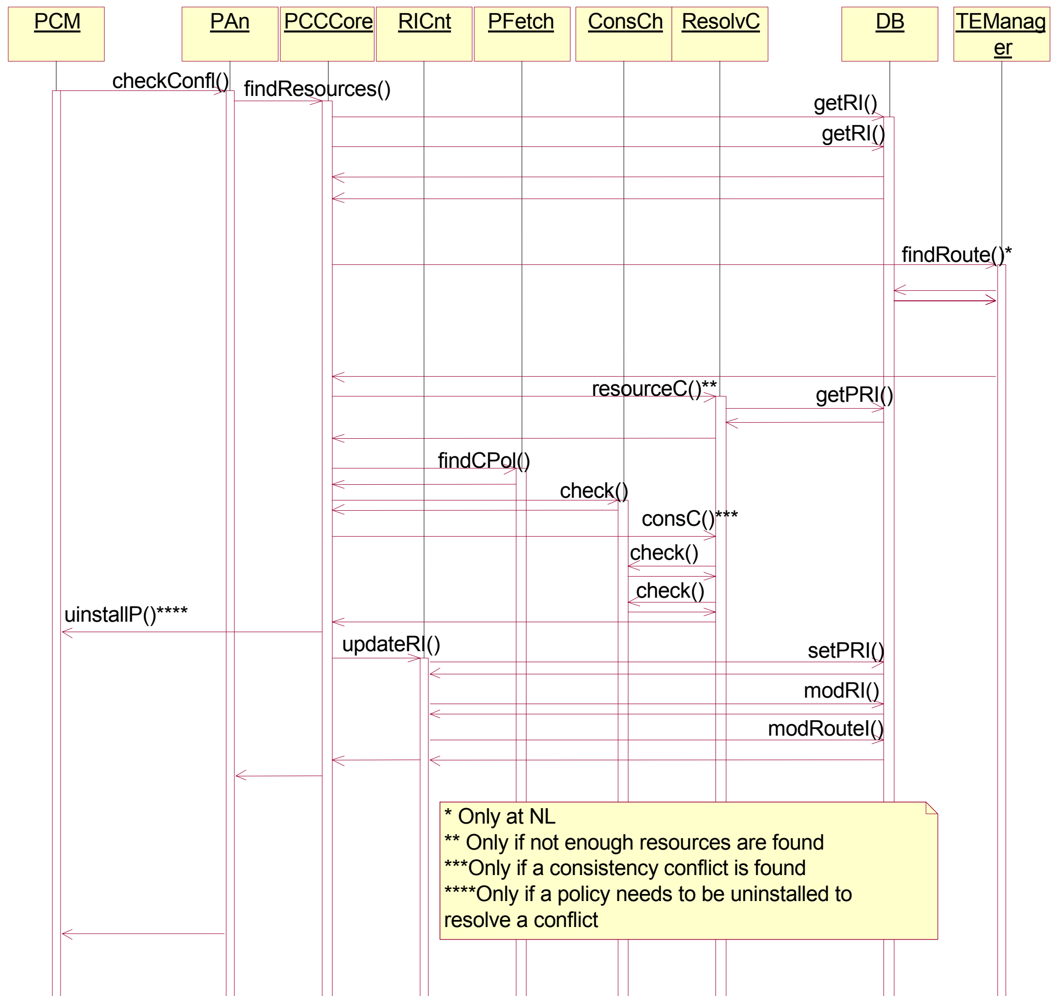

Figure 4 - 42. Policy Conflict Check: Policy conflict checking (predictable enforcement interval)

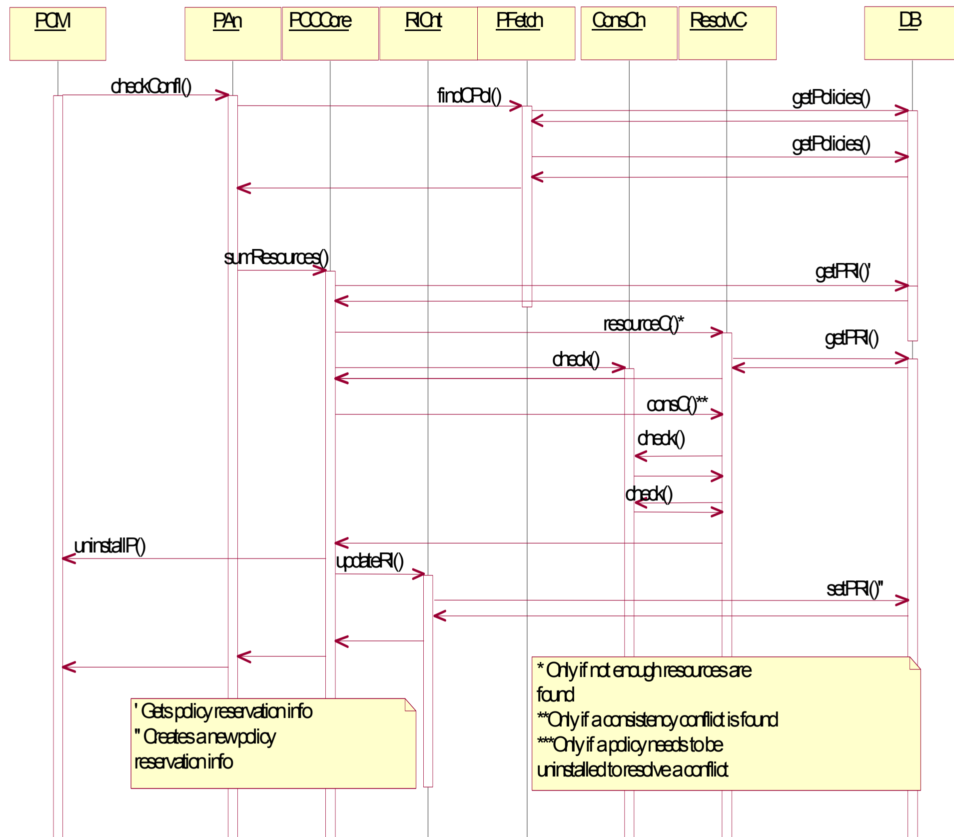

Figure 4 - 43. Policy Conflict Check: Policy Conflict Checking (unpredictable enforcement interval)

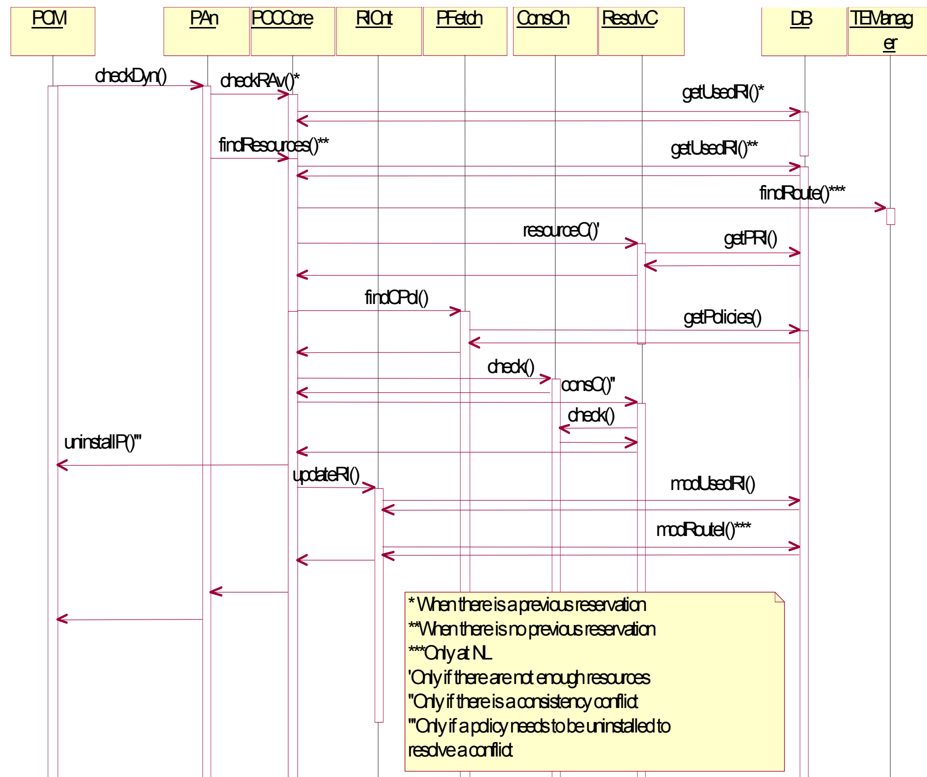

Figure 4 - 44. Policy Conflict Check: Dynamic conflict checking

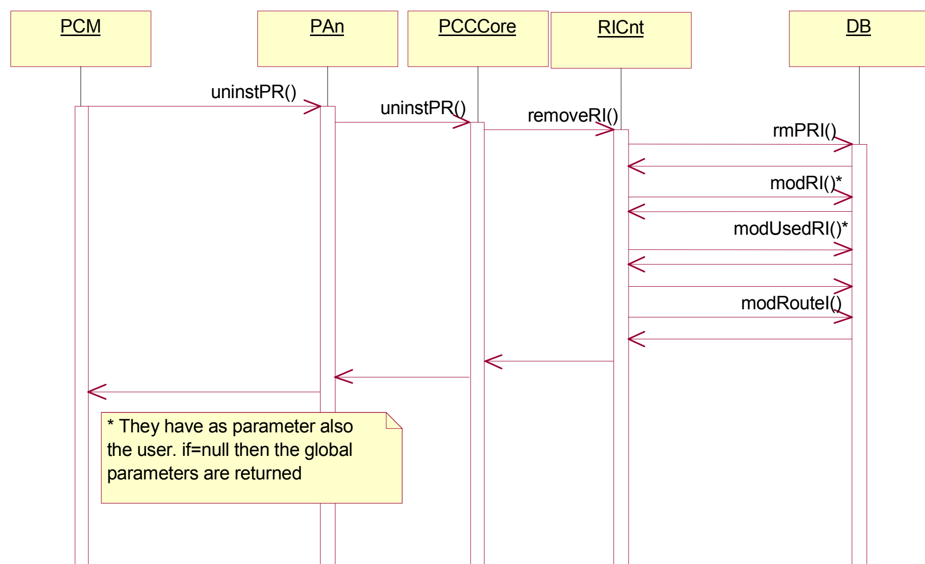

Figure 4 - 45. Policy Conflict Check: Removal of policy related data

5th TE Manager

A Component Behaviour

The TEManager component is in charge of finding out where in the managed network are enough resources to satisfy a request. It is a special component within the MANBoP framework since it is the only one not always instantiated. As already mentioned in previous chapters, the TEManager is only instantiated when the MANBoP instance is running at either network or sub-network levels because its tasks are not necessary at the element level.

The TEManager can be seen as a sub-component of the Policy Conflict Check since it participates in the process of assuring that a new policy does not conflict with the others due to resource sharing. Nevertheless, the complexity and importance of this component, and the fact that it is not always instantiated within the framework, has lead us to design it as a completely separated component.

Taking into account Table 4 - 1 (see pag.76), the main tasks that must be carried out by the TEManager component are:

- *Request the application of the routing algorithm over the managed topology:* After the reception of new managed topology, the PCM requests to the TEManager component the execution of the routing costs re-calculations over the new topology. This request is realised through the updateTop() method offered by the TEManager component.
- *Apply the routing algorithm over the managed topology:* The same task as above except that it is realised automatically at the bootstrapping of the component.

The main task of the TEManager component, that is, to find requested resources within the managed network, does not appear in this table because it gets shadowed by the conflict check tasks as we have argued before. For realising this task, the TEManager uses the topological information stored in the Database.

We have to take into account that the goal of this doctorate thesis is not the design of a Traffic Engineering manager, which is a rather complex task. The design included in this document is given for sake of completeness and to assess the overall framework concepts.

The topological information is composed by nodes and links. Each object describing a node will contain lists of incoming and outgoing links as well as references to resource schedule objects. These resource schedule objects will

be used to calculate the "scheduled costs" for each path[16]. That is, costs foreseen in time, taking into account the schedules assigned to different paths.

All possible paths between all end-points of the managed network are calculated at bootstrap and each time the topological information is updated.

The routing algorithm used is based on a Dijkstra algorithm [Halabi01] modified to find out all possible paths from one end-point to another in the network, and not just the one with lower cost.

The reason for choosing the Dijkstra algorithm as basis for the routing algorithm is simply because it is the most widely available and known.

In a first instance, the cost estimation of the routing algorithm is just based on the number of hops. However, immediately after concluding the routing algorithm the TEManager calculates all costs for all paths found within the managed network. These two processes are realised separately to re-use the logic of the cost calculation algorithm that is done periodically by the TEManager as justified below. For the cost calculation we should take into account not only the costs of the links (related mainly with bandwidth) but also the costs of the nodes (related mainly with computing power).

In order to ease the description of the routing algorithm that will be used in the TEManager we give an example.

Let's assume that we have a simple topology of 4 nodes. Three of them are end-points and the fourth one is an internal router within the managed network. In the figure below we can see this simple topology:

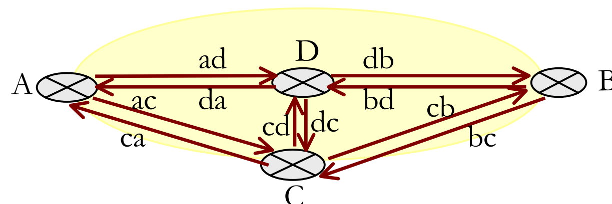

Figure 4 - 46. A network topology example.

The links are always considered unidirectional, that's why we have created one in each direction between each pair of nodes.

The TEManager first gets the list of end-points that form the network (A, B and C) from the network topology object. The network topology object also contains an array of references to all nodes and links that form the managed network.

[16] In this thesis, we differentiate between paths and routes in the next way. A path is physical 'circuit' to arrive from one end-point to another in the managed network. On the other hand, a route is a set of resources from a path that have been already allocated to one or more flows.

In the next step, the TEManager gets the object with the information concerning the first end-point of the list (end-point A). This object, as well as all other node objects, contains:

i) Node topology identifier: a string that uniquely identifies this element within the managed topology.

ii) Array of references to outgoing links objects.

iii) Array of references to incoming links objects.

iv) Available scheduled and actual resources: All the resources that will be available in this node in different intervals of the future and the resources that are currently available. The first ones are those that are left after all allocations caused by the enforcement of policies whose enforcement interval is predictable. The second ones are the resources that are left after considering the allocations because of scheduled policies, unpredictable (not scheduled) policies and signalling requests. The resources considered at this point depend greatly in the PCC and TEManager implementations and the interests of the network operator. The most common resource information for a node will be the type of node (e.g. passive:0, active:1, programmable:2) and for active and programmable nodes: the available CPU, memory, execution environments...

The TEManager retrieves the list of outgoing links and gets the link information from the link object:

i) Link topology identifier: A string that uniquely identifies the link within the managed network.

ii) Source node: A reference to the node object of the node originating this link.

iii) Sink node: A reference to the node object of the node where this link ends.

iv) Available scheduled and actual resources: As described for the node object. These resources are taken into account by the TEManager together with the ones from the node for calculating the costs of the paths.

Finally, it also gets the sink node instance. With all this information, it creates a path object instance with the corresponding values:

i) Path identifier: a string that uniquely identifies this path with the this end-point as source.

ii) Array of links: In this array we include the links that already form the path.

iii) Array of nodes: The nodes that form the path.

iv) Cost: Structure with the scheduled costs taking into account the different resources and intervals and actual available costs as well as the number of hops. During the routing algorithm only the number of hops value is updated.

v) Looping: A boolean that indicates if the last node added to the array of nodes was already in the array and therefore we are entering in a loop.

vi) Parent path: A reference to the path from where this path comes from. That is the parent path of a path going through A, C and B is the one that went in the previous step from A to C.

In the example two path objects would be instantiated: Path1 associated to link ad and node D and Path2 associated to link ac and node C:

<u>Path1</u>

- Path identifier: Path1
- Array of links: ad
- Array of nodes: A, D
- Cost: The number of hops attribute is set to 1.
- Looping:0
- Parent path: null

<u>Path2:</u>

- Path identifier: Path2
- Array of links: ac
- Array of nodes: A, C
- Cost: The number of hops attribute is set to 1.
- Looping:0
- Parent path: null

After instantiating all path objects the TEManager checks if any of the paths instances has a value of 1 in the looping property. If any has, then it checks if the parent path is already stored in the database and if not it stores it. Finally, it removes all path instances that had a looping value of 1 and all parent path instances (in this case null). If no path instances remain the process is finished, otherwise the algorithm is applied again.

The TEManager gets from the node instances the references to the outgoing links for that node and creates for each one of the links a new path instance with the information of the parent path updated taking into account the information for that link and its sink node.

Following the example we would have:

Path3:

- Path identifier: Path3
- Array of links: ad, db
- Array of nodes: A, D, B
- Cost: The number of hops attribute is set to 2.
- Looping:0
- Parent path: Path1

Path4:

- Path identifier: Path4
- Array of links: ad, dc
- Array of nodes: A, D, C
- Cost: The number of hops attribute is set to 2.
- Looping:0
- Parent path: Path1

Path5:

- Path identifier: Path5
- Array of links: ad, da
- Array of nodes: A, D, A
- Cost: The number of hops attribute is set to 2.
- Looping:1
- Parent path: Path1

Path6:

- Path identifier: Path6
- Array of links: ac, cb
- Array of nodes: A, C, B
- Cost: The number of hops attribute is set to 2.
- Looping:0
- Parent path: Path2

Path7:

- Path identifier: Path7

- Array of links: ac, cd
- Array of nodes: A, C, D
- Cost: The number of hops attribute is set to 2.
- Looping:0
- Parent path: Path2

Path8:

- Path identifier: Path8
- Array of links: ac, ca
- Array of nodes: A, C, A
- Cost: The number of hops attribute is set to 2.
- Looping:1
- Parent path: Path2

Path5 and Path8 have a looping value of 1. Thus, we store in the database their parent paths (Path1 and Path2 respectively) and delete Path5 and Path8 instances. Finally, we remove all parent path instances (Path1 and Path2) (not from the database but just from the algorithm).

The path instances that remain are Path3, Path4, Path6 and Path7. So we repeat the process again.

Following the process:

Path9:

- Path identifier: Path9
- Array of links: ad, db, bd
- Array of nodes: A, D, B, D
- Cost: The number of hops attribute is set to 3.
- Looping:1
- Parent path: Path3

Path10:

- Path identifier: Path10
- Array of links: ad, db, bc
- Array of nodes: A, D, B, C
- Cost: The number of hops attribute is set to 3.
- Looping:0

- Parent path: Path3

Path11:

- Path identifier: Path11
- Array of links: ad, dc, ca
- Array of nodes: A, D, C, A
- Cost: The number of hops attribute is set to 3.
- Looping:1
- Parent path: Path4

Path12:

- Path identifier: Path12
- Array of links: ad, dc, cd
- Array of nodes: A, D, C, D
- Cost: The number of hops attribute is set to 3.
- Looping:1
- Parent path: Path4

Path13:

- Path identifier: Path13
- Array of links: ad, dc, cb
- Array of nodes: A, D, C, B
- Cost: The number of hops attribute is set to 3.
- Looping:0
- Parent path: Path4

Path14:

- Path identifier: Path14
- Array of links: ac, cb, bd
- Array of nodes: A, C, B, D
- Cost: The number of hops attribute is set to 3.
- Looping:0
- Parent path: Path6

Path15:

- Path identifier: Path15

- Array of links: ac, cb, bc
- Array of nodes: A, C, B, C
- Cost: The number of hops attribute is set to 3.
- Looping:1
- Parent path: Path6

Path16:

- Path identifier: Path16
- Array of links: ac, cd, da
- Array of nodes: A, C, D, A
- Cost: The number of hops attribute is set to 3.
- Looping:1
- Parent path: Path7

Path17:

- Path identifier: Path17
- Array of links: ac, cd, dc
- Array of nodes: A, C, D, C
- Cost: The number of hops attribute is set to 3.
- Looping:1
- Parent path: Path7

Path18:

- Path identifier: Path18
- Array of links: ac, cd, db
- Array of nodes: A, C, D, B
- Cost: The number of hops attribute is set to 3.
- Looping:0
- Parent path: Path7

The paths Path9, Path11, Path12, Path15, Path16 and Path17 have the looping value 1. Therefore, the corresponding parent paths Path3, Path4, Path6 and Path7 are stored in the database and all parent path instances, as well as paths 9,11,12,15,16 and 17, are removed.

Since we have still Path10, Path13, Path14 and Path18 the algorithm is applied again.

The next path instances obtained are:

Path19:

- Path identifier: Path19
- Array of links: ad, db, bc, ca
- Array of nodes: A, D, B, C, A
- Cost: The number of hops attribute is set to 4.
- Looping:1
- Parent path: Path10

Path20:

- Path identifier: Path20
- Array of links: ad, db, bc, cd
- Array of nodes: A, D, B, C, D
- Cost: The number of hops attribute is set to 4.
- Looping:1
- Parent path: Path10

Path21:

- Path identifier: Path21
- Array of links: ad, db, bc, cb
- Array of nodes: A, D, B, C, B
- Cost: The number of hops attribute is set to 4.
- Looping:1
- Parent path: Path10

Path22:

- Path identifier: Path22
- Array of links: ad, dc, cb, bd
- Array of nodes: A, D, C, B, D
- Cost: The number of hops attribute is set to 4.
- Looping:1
- Parent path: Path13

Path23:

- Path identifier: Path23

- Array of links: ad, dc, cb, bc
- Array of nodes: A, D, C, B, C
- Cost: The number of hops attribute is set to 4.
- Looping:1
- Parent path: Path13

<u>Path24:</u>

- Path identifier: Path24
- Array of links: ac, cb, bd, da
- Array of nodes: A, C, B, D, A
- Cost: The number of hops attribute is set to 4
- Looping:1
- Parent path: Path14

<u>Path25:</u>

- Path identifier: Path25
- Array of links: ac, cb, bd, dc
- Array of nodes: A, C, B, D, C
- Cost: The number of hops attribute is set to 4.
- Looping:1
- Parent path: Path14

<u>Path26:</u>

- Path identifier: Path26
- Array of links: ac, cb, bd, db
- Array of nodes: A, C, B, D, B
- Cost: The number of hops attribute is set to 4.
- Looping:1
- Parent path: Path14

<u>Path27:</u>

- Path identifier: Path27
- Array of links: ac, cd, db, bd
- Array of nodes: A, C, D, B, D
- Cost: The number of hops attribute is set to 4.

- Looping:1
- Parent path: Path18

Path28:

- Path identifier: Path28
- Array of links: ac, cd, db, bc
- Array of nodes: A, C, D, B, C
- Cost: The number of hops attribute is set to 4.
- Looping:1
- Parent path: Path18

The result of this last step of the algorithm is that all new path instances calculated have the looping value 1. Hence, the parent paths are stored in the database (i.e. Path10, Path13, Path14 and Path18), the path instances with looping value of 1 are removed and all parent paths are also deleted.

Finally, there are no path instances left so the process is finished. The resulting paths with source end-point A are: Path1, Path2, Path3, Path4, Path6, Path7, Path10, Path13, Path14 and Path18. So if we want to go from A to B the possible routes are:

- Path3: A, D, B
- Path6: A, C, B
- Path13: A, D, C, B
- Path18: A, C, D, B

The same process should be realised again for each one of the end-points forming the managed network.

The routing algorithm designed is quite resource consuming but we only execute it at bootstrap and when the managed topology is updated. When a re-calculation of costs is requested to update them to the actual resource status the process is much simpler and therefore less resource consuming as described below[17].

The PCC, as described in the previous section, will periodically introduce resource-scheduling information in the notification service and register to receive these reports from lower level instances (except for the PCC at the element level). Each time the PCC receives the reports from all underlying MANBoP instances, it requests to the TEManager the re-calculation of "scheduled costs" in all paths.

[17] The scalability of the algorithm can be enhanced introducing sub-network manages inside the management infrastructure, thus distributing the algorithm.

The information included inside these reports describes the resources within the managed nodes (when coming from element-level MANBoP instances) or between edge-nodes in a managed sub-network (when coming from sub-network managers). The resource information given is: scheduled resources and used resources (only takes into account those resources being used but not scheduled, because of signalling requests or unpredictable policies) for each one of the possible paths between end-points (when coming from sub-network managers).

Moreover, this information is duplicated in resources available when requested individually (that is, only bandwidth or just CPU...) and resources available when more than one resource is requested at the same time (e.g. bandwidth and a certain level of CPU and memory). When reports are coming from an element-level MANBoP instance both types of information will be the same. Oppositely, when reports are coming from a sub-network level MANBoP instance both types of resource values would normally be different. The reason is that it might happen that a path within a sub-network with more bandwidth resources is the one with less computing capabilities. Thereby, if reports just indicate the available resources in the minimum cost path between two end-points, this information could be misleading, because the minimum cost would be estimated taking into account an average of all resources. Hence, other paths with less average but more available resources of a particular type would not be taken into account. For that reason, the individually available resources specified in the report show the maximum available resources, maybe through different paths, between two end-points of the sub-network.

The reason for the exchange of resource usage information between the different MANBoP instances through these reports is that, when a MANBoP instance is working over sub-network MANBoP managers, a request for an allocation schedule through the managed sub-network produces a change in the available resources within the affected end-points of the sub-network. Such change cannot be estimated by the upper MANBoP instance. A safe estimation is to remove the allocated resources from the ones that were available before, however this is only true if the path within the sub-network where the resources have been allocated is still the path with the maximum available resources of that type. If after the allocation of the resources in that path another path becomes the one with more available resources between those end-points, the estimation made removing the allocated resources from the available ones would be incorrect. The only way a MANBoP instance working over sub-network managers can know if its 'safe' estimations about the available resources in the sub-network after allocating resources are correct is receiving periodical information reports from the managed sub-networks specifying the concrete available resources at a particular time.

Reports also include the last policy processed by the MANBoP instance whose processing had an effect on the available resources. This information is

used by the instance receiving the reports to take into account the schedules that might have been realised by policies processed after that one.

Reports include only information about resource schedules and resources used by signalling requests or unpredictable policies that have been enforced at a particular time. There is no information about the total used resources because these depend on best effort resource consumption. The network operator can introduce policies determining a minimum percentage of resources to be used for best effort connections. However, other best effort connections above these percentages are simply not taken into account by the TEManager and the PCC components for the scheduling of resources (the resources used by these best effort users might be assigned to resource schedules). Therefore, this information is not necessary for these processes. Nevertheless, the network operator could introduce a performance-type of Policy Consumer component that periodically retrieves this information from the managed devices and updates the corresponding resource objects if it still prefers to introduce this information within the used resources reports. The tasks of the Decision-making Monitoring system (DmMs) are different from that one (i.e. monitoring of used resources). It is limited to the monitor of policy conditions to detect when a policy enforcing should be triggered, thus participating in the decision-making mechanism.

When all resource information from underlying MANBoP instances has been received, the TEManager re-calculates the costs of the paths. We assume that the PCC component has updated correctly all resource information (specially the links between end-points of the underlying sub-networks) based on the received reports. To carry out this task the TEManager, for each one of the end-points in the managed network, gets all possible paths. For each one of the paths, it gets the list of nodes and links that form it. Finally, it gets the resource scheduling costs for these nodes and links and establishes the costs based on them. The scheduling costs might be calculated differently depending on the resource. For example, the total bandwidth costs for a route will be the value of the link with less available bandwidth. Contrarily the total delay in a path will be calculated summing up all partial delays along the paths.

The figure below shows in detail how these tasks are carried out by the TEManager component.

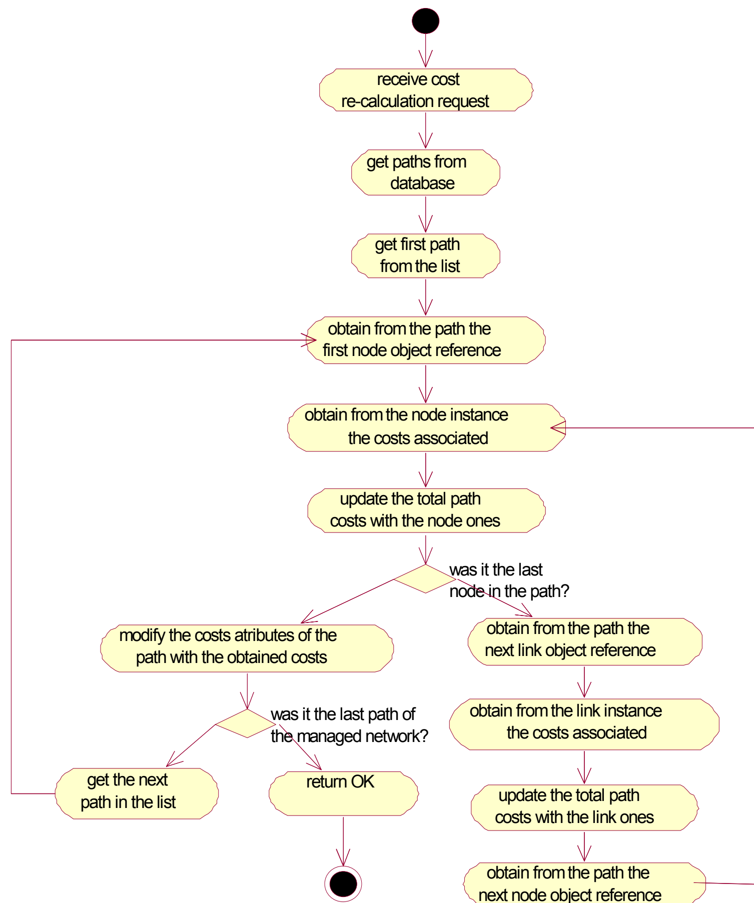

Figure 4 - 47. TEManager: Cost re-calculation activity diagram

Each time the TEManager receives a request to find a path with a set of available resources between two particular end-points in the managed network, the first thing it realises is to check whether a route is already assigned to the flow specified in the request. If so, it checks if the route found is a reserved route[18] or an allocated route. In the allocated route case, since resources from an allocated route cannot be used, the TEManager gets the parent route from that one (if no parent route exists then it takes the general resources) and just verifies that the requested resources are available along the parent route. In case they are not available, it returns the parent route together with the conflicting resources. Otherwise, it simply returns the path. The same procedure is followed if first, the route found is a reserved route whose resources can be used; second, the flows of the request and of the route are

[18] We are using the term 'Schedule' when we talk about resources generically and 'route' when we refer to end-to-end forwarding and computing resources assigned to one or more flows.

exactly the same; and third, the request is a reserve request. Again, the resources from the parent route are taken because we assume that the user wishes to add more resources to the reserved route found. In any other case, the TEManager gets the resources from the route found and checks whether the requested resources can be found among them. In all cases, if the resources are found, the TEManager returns the corresponding route and if not, it returns the route with a list of conflicting resources.

When there is no route already assigned to the related flow, the TEManager gets the information about the paths calculated between those two end-points that are stored in the database. It receives the type of resources requested, as well as where and when they are requested. The TEManager then considers whether there is a single resource requested individually or instead, a group of resources are requested jointly. In the first case, it uses the scheduled resource information for individual requests. In the second case, it uses the scheduled resource information for several resources requests. The TEManager compares this information with objects about possible paths retrieved from the database. Each path object contains the available scheduled costs for that path and a list of references to nodes and links that form the path. The TEManager chooses the path object with more available resources (among those requested), during the requested interval. From this path object, it retrieves the nodes and links forming the path and verifies that the requested resources are available where they are needed (e.g. computing resource at concrete places in the network). If so, it simply returns a reference to the path object to the PCC component that made the requests. Otherwise, it checks if the second possible path fulfils the resource requirements.

In case that at the end of the process no path is found with the requested resources, the TEManager returns a reference to the path object with more available resources, as well as the list of nodes and links of that path where the requested resources could not be fulfilled.

It is also important remarking that in all cases the TEManager, if the interval specified is undefined, takes the used resources instead of the scheduled ones since it assumes that the request received is going to be enforced immediately if no conflict is found.

In the activity diagram below we can see all these tasks realised by the TEManager drawn.

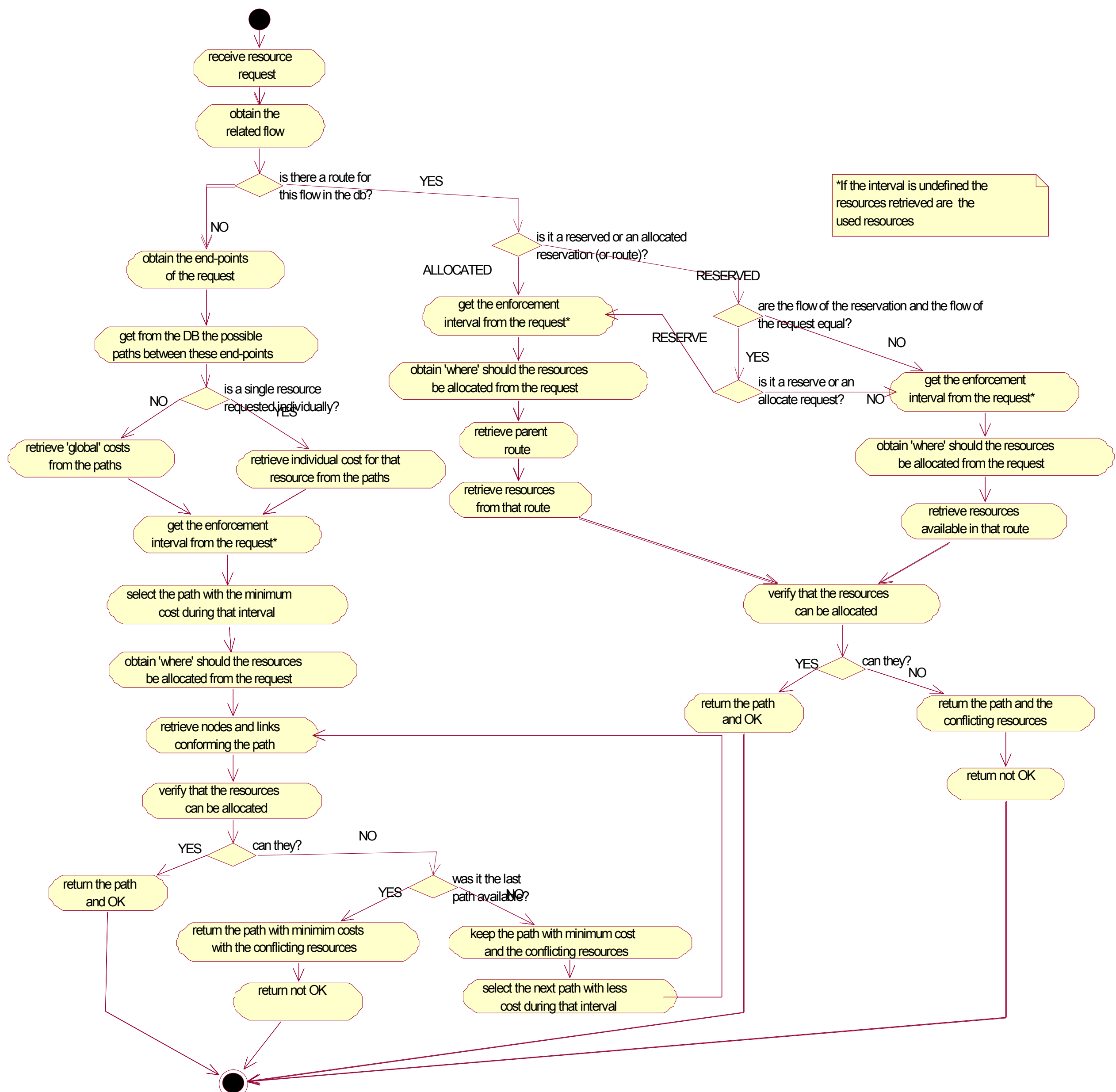

Figure 4 - 48. TEManager: Resource request activity diagram

B *Component Design*

Summarising the component behaviour section, the TEManager must develop three groups of tasks:

- Calculate all paths within the managed topology: At system bootstrap or when the topology is updated, the TEManager must calculate all possible

paths between the end-points of the managed network. The obtained paths are stored in the database.

- Re-calculate the path costs periodically: After a reception of a request from the Policy Conflict Check component, the TEManager has to re-calculate the costs of all paths within the managed network with the current resource information.

- Find a path with the requested resources: Again, when requested by the PCC, the TEManager component has to find out whether the requested resources can be fulfilled with the paths available between two end-points. If no adequate path is found, the TEManager returns the path with the minimum cost and the conflicting resources.

For realising these groups of tasks, we have designed within the TEManager component three classes, namely the TECore, CostCalc and Routing, which are shown together with their interfaces in the class diagram below.

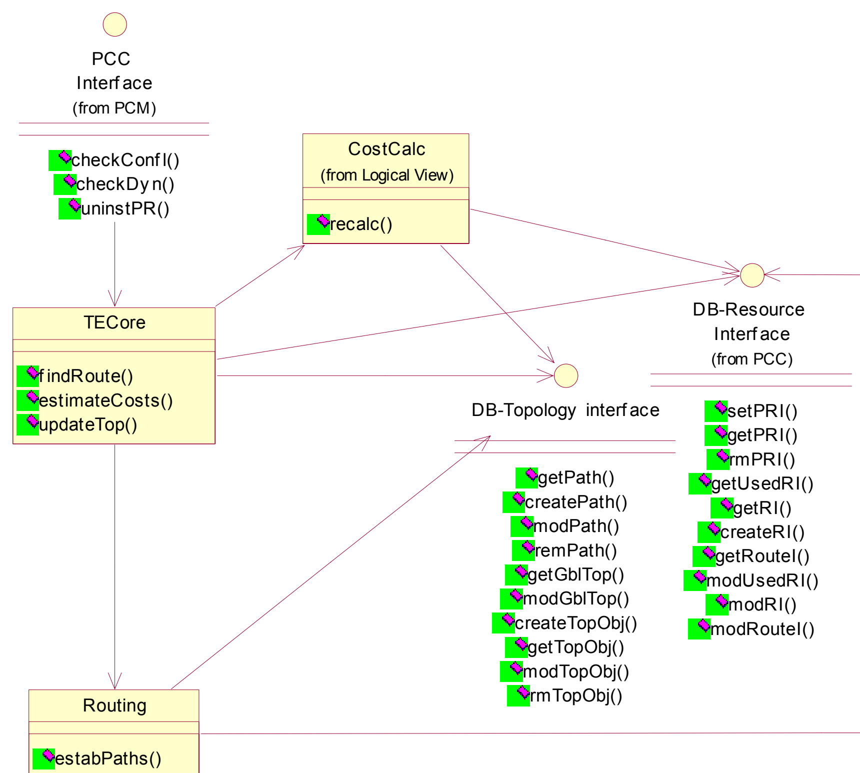

Figure 4 - 49. TEManager class diagram

In the following sub-sections we will describe in detail the functionality and interfaces offered by these classes. Additionally, the last sub-section shows

some sequence diagrams that reflect how the different classes of the component interact in order to fulfil the received requests.

a TECore class

The TECore is the class responsible of finding the requested resources within the managed network. This functionality has already been extensively described in the previous section and shown in Figure 4 - 48. Additionally, the TECore also coordinates all processes of the component and interacts with the Policy Conflict Check component.

To realise this functionality the TECore class offers an interface with three methods: findRoute(), estimateCosts(), updateTop(). Table 4 - 20 describes the functionality and parameters of these methods.

Interface	Input parameters	Output parameters	Functionality
findRoute()	flow assocFlow, resource[] resources, string endPointA, string endPointZ, boolean directionality, boolean type	boolean Result, string[] nodeId, string pathId, resource[] cRes	The Policy Conflict Check (PCC) component uses this method to ask for the search of requested resources within the managed network. The method specifies six input and four output parameters. The first input needed is the flow associated to the requested. The flow structure, which has already been described in the RICnt class interface description table (Table 4 - 15.), is used to determine where to retrieve resources from. The resource structure is used as input parameter for specifying the requested resources. Two strings, the endPoint A and Z, specify the two end-points of the managed network between which the resources are requested. The boolean 'directionality' specifies whether the requested resources should be allocated just from A to Z (0) or both from A to Z and from Z to A (1). Finally, the 'type' input parameter is a boolean that identifies whether the requested resources are to be reserved (0) or allocated (1). The output parameters are a boolean that indicates if the resources could be found, or if there were not enough resources and a conflict must be solved The nodeId parameter is a list of strings that identify the nodes forming the assigned route. Another string, the 'pathId' identifies the path object linked to the assigned route. Finally, in case that a conflict is found the 'cRes' parameter, a list of resource structures, identifies the conflicting resources and the amount of resources missing at particular intervals (see page 145). When receiving a request through this method, the TECore class will look for the requested resources within the managed network as shown in the activity diagram included in Figure 4 - 48.
estimateCosts ()		boolean Result	The PCC periodically, also at bootstrapping or when the managed topology is updated, requests to the TEManager through this method a cost re-calculation between the end-points of the managed network. The method includes just one boolean with the result as output parameter. The TECore class will request to the CostCalc class the realisation of the process.
updateTop()	file newTop, boolean type	boolean Result	This method is called by the PCM component to request the update (addition or removal of a node) of the topological information managed by this MANBoP instance. The method specifies two input parameters, and a boolean indicating the process result as output parameter. The first input parameter is a file indicating the new topological information object and the resources linked to it. The boolean 'type' indicates if the new topological information object should be added (1) or removed (0) to the managed network. When receiving a call to this method the TECore class contacts the Routing class to re-calculate all possible paths within the managed network. Finally, when the Routing class finishes, the TECore requests the cost re-calculation for all paths. If an error is found in any of the steps in the process, the TECore returns 'false'.

Table 4 - 20. The TECore class interface description table

b *Routing class*

The Routing class uses the topology instances previously created by the GraphBuilder class inside the PCM component to calculate all possible paths between all end-points of the managed network.

Interface	Input parameters	Output parameters	Functionality
estabPaths()		boolean Result	The TECore requests to the Routing class at bootstrapping, or when the managed topology is updated, the application of the routing algorithm implemented by this class over the managed topology information of this MANBoP instance. The method includes just one boolean with the process result as output parameter. When a request is received in this method, the Routing class first accesses the global topology IMO to retrieve all end-points within the managed network. Then, it carries out the routing algorithm tasks described along the component behaviour section of the TEManager component (see page 154) with an example.

Table 4 - 21. The Routing class interface description table

c CostCalc class

The CostCalc class carries out the costs calculation for all paths within the managed network periodically, at bootstrap or when the managed topology is updated.

Interface	Input parameters	Output parameters	Functionality
recalc()		boolean Result	The TECore periodically, also at bootstrapping or when the managed topology is updated, requests to the CostCalc class, a cost re-calculation between the end-points of the managed network. The method includes just one boolean with the process result as output parameter. When a request is received through this method the CostCalc class carries out the tasks reflected in the activity diagram drawn in Figure 4 - 47.

Table 4 - 22. The CostCalc class interface description table

d Sequence diagrams

To complement the description and comprehension of the TEManager component, we include below sequence diagrams reflecting the interactions between the component classes, related with the three public methods offered in the component interface.

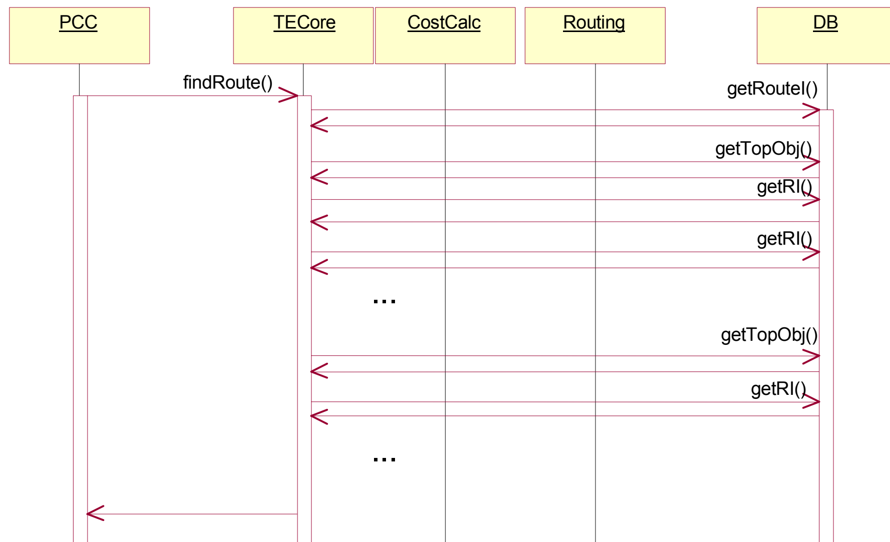

Figure 4 - 50. TEManager: findRoute sequence diagram

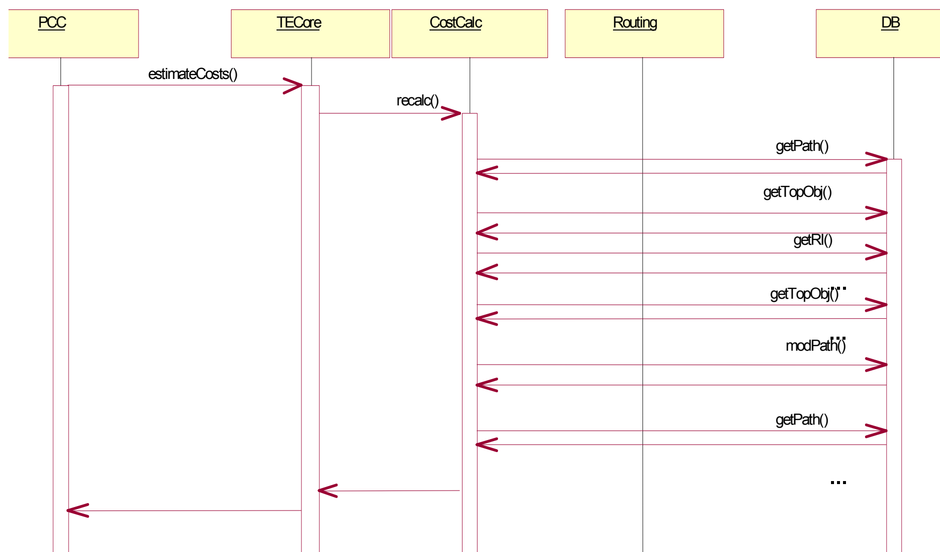

Figure 4 - 51. TEManager: estimateCosts sequence diagram

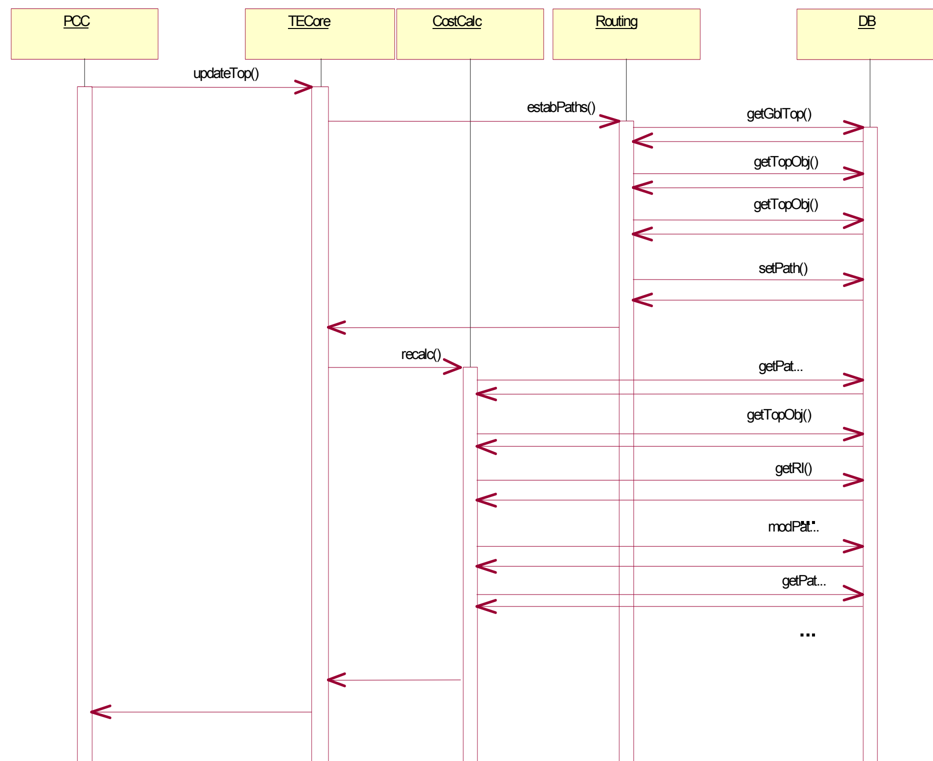

Figure 4 - 52. TEManager: updateTop sequence diagram

6th Decision-making Monitoring system

A Component Behaviour

The Decision-making Monitoring system (DmMs) component plays an important role within the process of deciding whether a policy should be either enforced or de-enforced. This component receives the policy conditions from the Policy Consumer Manager and executes, together with the appropriate Monitoring Meter (MMs) components, the necessary tasks to determine when the overall policy conditions are satisfied. When the conditions are not fulfilled anymore, the DmMs also warns the Policy Consumer Manager to de-enforce the policy from the managed devices.

The component has many other tasks that were highlighted, in part, at the end of the Use Cases section (see Table 4 - 1). These tasks are listed below:

- *Register conditions to be monitored:* Basic functionality that should be offered by the DmMs in order to trigger the monitoring of the policy conditions process. The task is requested by the Policy Consumer Manager component through the regCond() method. The request causes the storage of policy conditions linked to the processed policy and the

creation of a logic expression that represents those policy conditions, so that the component knows when the overall condition is true based on its Individual Statement (IS) values (a policy condition can be decomposed in multiple Individual Statements). As an example, if a request to register a policy with the following policy condition arrives: if (Tuesday AND between 8h and 10h AND usedBW in link X<10) OR (usedBW in link X >= 10). Then, it would be expressed as a boolean expression of Individual Statements such as: if (A AND B AND C) OR (!C).

- *Are the meters installed?:* Once the policy condition is registered in the component, it divides the overall policy conditions into the Individual Statements that form the boolean expression of the condition. Then, it looks for the Monitoring Meter components in charge of processing them (in the example above meters for A, B and C). If a needed Monitoring Meter is not found, the DmMs requests to the Code Installing Application (CIA) its download based on the Individual Statement to be monitored, the position of the MANBoP instance within the overall management infrastructure and the underlying devices.

- *Do the conditions match?:* This is, obviously, the central task to be developed by the component. Every time a Monitoring Meter component detects a change in the status of an Individual Statement, it notifies this change to the DmMs component through the ISValue() method. Then, the DmMs introduces this change in the logic expression representing the policy condition and checks whether the overall expression is fulfilled or not. When a change in the status of the overall expression occurs, the component notifies the Policy Consumer Manager, so that it can react accordingly.

- *Notify new underlying managers info or node interfaces info to the DmMs:* The DmMs receives this notification from the PCM through the upUnI() interface with the instance identifier of the new underlying device, or the one removed, as parameter. Then, only if the DmMs is working over other MANBoP instances, it obtains the underlying device IMO from the database and retrieves the interface of its Notification Service to register, or obliterates if the managed device is being removed, as event consumer in the new underlying MANBoP instance.

In addition to these tasks listed above, the Decision-making Monitoring system develops other functionalities such as obliterating the policy conditions when a policy is being removed. The unregCond() method of this component is used for requesting this obliteration (or unregistration).

Furthermore, the DmMs keeps the lifecycle of the Monitoring Meter components. When a Monitoring Meter component is not processing any IS, the DmMs component uninstalls it to save computing resources in the management station. Monitoring Meters are installed based on the

functionality to be monitored, the position of the MANBoP instance within the management infrastructure and the underlying devices.

Additionally, in this section we include a brief description of the Notification Service defined by the OMG [OMG], which is used by several components of the framework. Since the Notification Service has not been developed as part of this thesis, we will just introduce the main concepts of the notification service and explain how it is used within the framework. For sake of completeness, we will provide some references that provide details about the design and implementation of this service along the section.

The by-default communication in CORBA is synchronous. Nonetheless, in many applications an asynchronous message exchange is a prerequisite for scalability and for the ability to add and remove event consumers or suppliers dynamically without affecting existing clients [Iona01]. The OMG group detected this necessity and developed the Event Service to handle it. The Event Service [OMG01b] does not provide yet a decoupled, asynchronous, multicast communication, but provides the building blocks to define an Event Channel, which does satisfy these requirements. The Event Channel is an entity that receives all events from event suppliers and delivers them to all connected consumers. The Notification Service [OMG02b] extends the Event Service by letting programmers associate filters that precisely specify in what events each consumer is interested. Moreover, the Notification Service extends the Event Service in many other ways such as: allowing suppliers to obtain the kind of events consumers are interested in, as well as allowing consumers to obtain the kind of events that a channel offers, introducing quality of service parameters for the event transmission, etc.

At bootstrap, when the DmMs is started with the underlying topology, and if the current instance is working over other MANBoP instances, the Notification Service is started and requested to register in the Notification Service of each underlying MANBoP instance. Additionally, every time that an underlying manager is added, the Notification Service is requested to register to the new underlying manager Notification Service as event consumer.

The Notification Service is used within MANBoP for the communication between MANBoP instances at different levels by means of events. The Policy Conflict Check component uses it for sending resource information reports to higher-level instances, the PFwCnt class within the Policy Consumer Manager introduces policy group enforcement result events, Policy Consumers that might be oriented to performance, accounting or billing could also use the Notification Service to send its periodic reports to higher-level instances or even to interested users. All these components would act as suppliers in the Notification Service within their MANBoP instances. As long as the Notification Services of higher-level MANBoP instances (or even other applications in behalf of authorised users) are connected as event consumers, they will receive the events. In the higher-level instance, interested

components are registered as event consumers with a filter specifying the events they are interested in. The main event consumers would be the Policy Consumers (to know the enforcement result of policies), Monitoring Meters (to receive monitoring information) or the Policy Conflict Check component (to receive resource information reports).

The above introduction to the main functionalities covered by the DmMs is complemented with some activity diagrams that provide a higher level of detail.

The activity diagram below describes in more detail the tasks realised by the DmMs when a request for monitoring a policy condition is received.

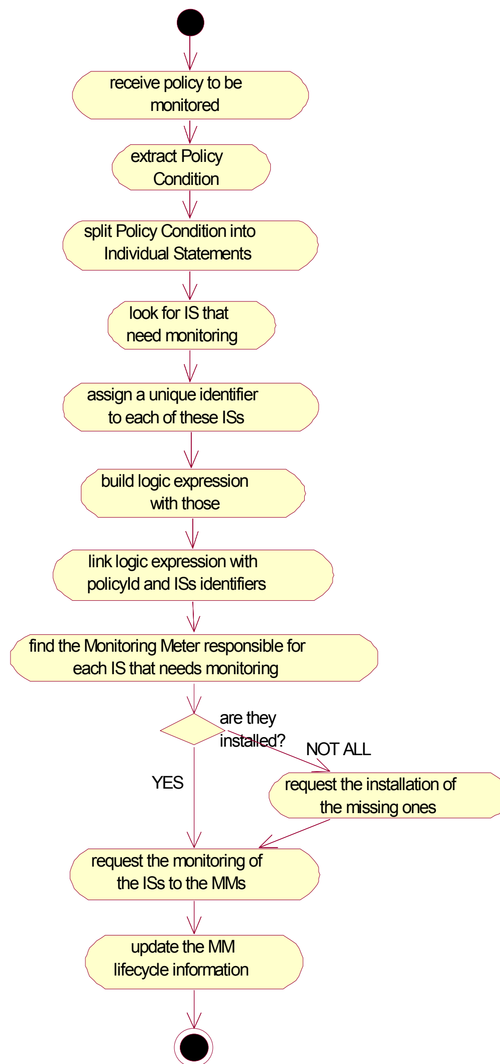

Figure 4 - 53. DmMs: Policy condition monitoring request activity diagram

The first thing that the component does is extracting the policy condition from the policy received and split it into Individual Statements (ISs). Once we have converted the overall policy condition into Individual Statements, we look among them, for the ones that need some monitoring to be applied. Only with those Individual Statements, we build a logic expression that represents the overall policy condition. When the overall logic expression value is 'true' the overall policy condition is fulfilled, when false it is not. The logic expression is linked with the policyId to which the policy condition pertains and with the unique identifiers assigned to the Individual Statements obtained.

The next step is to verify if the Monitoring Meter (MM) components that are needed to realise the monitoring tasks implied in the ISs are installed. If any of them is not, the DmMs requests to the Code Installing Application (CIA) their installation taking into account the management level at which the instance is acting, the monitoring task to be realised and the underlying device. Once we know that all Monitoring Meter components are installed, we request them the monitoring of their respective Individual Statements. Finally, the DmMs updates the Monitoring Meter lifecycle information kept within the component. This information keeps the number of Individual Statements that are being processed by each Monitoring Meter. These values are checked periodically and those MMs that are not processing any IS are uninstalled.

Also the DmMs must handle requests to obliterate a policy condition because the policy is being removed for whatever reason. The activity diagram below shows the tasks that are carried out by the component when that occurs.

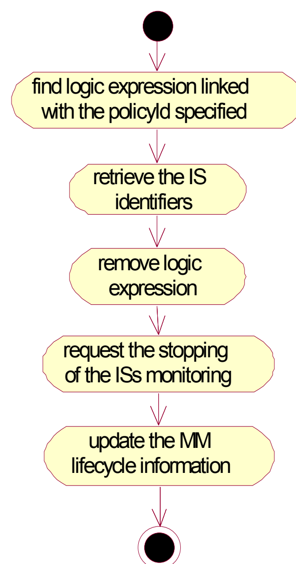

Figure 4 - 54. DmMs: Unregistration of a policy condition monitoring request activity diagram

First, the DmMs uses the PolicyId submitted in the request to find the logic expression linked to it. With the logic expression, it retrieves the Individual Statement identifiers that are being processed by the Monitoring Meter components.

Then, the logic expression is removed and a request is launched to the Monitoring Meters (MMs) to stop processing the involved ISs.

Finally, the Monitoring Meters lifecycle information kept within the component is updated accordingly. The next time the component checks the lifecycle information, those MMs that are not processing any IS will be uninstalled.

The main functionality developed by the DmMs component is to assess when a policy condition is met, or no longer met, and inform the Policy Consumer Manager about it. The tasks done by the DmMs to handle this functionality are shown in the activity diagram below:

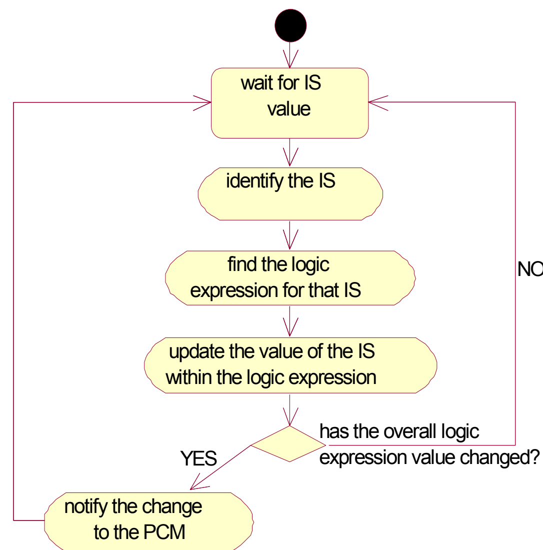

Figure 4 - 55. DmMs: Policy Condition monitoring activity diagram

The component waits for information about the Individual Statements (ISs) received from the Monitoring Meter components. Every time a MM informs about a change in the status of one of these ISs the DmMs looks for the corresponding logic expression based on the IS identifier and updates the logic expression with the new value. When this change causes that the entire logic expression value changes, the DmMs warns the Policy Consumer Manager about it. Otherwise, it waits for another change in an Individual Statement.

Finally, the DmMs is notified when the managed topology is updated so that the Notification Service might connect (or disconnect) to lower-level

Notification Services if necessary. The tasks carried out by the component every time the managed topology is updated are reflected below:

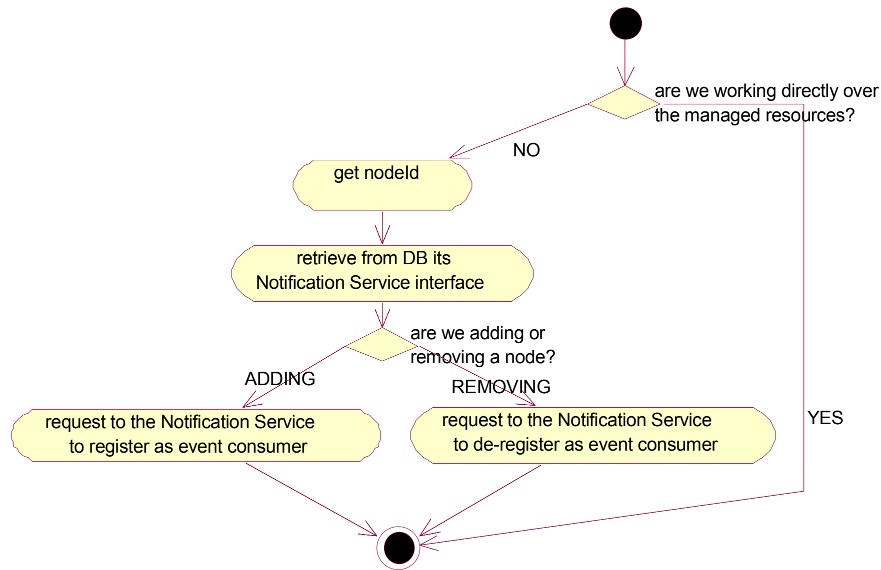

Figure 4 - 56. DmMs: Managed topology update processing activity diagram

Only if the instance is not working directly over the managed resources a change in the managed topology affects to the Notification Service. In that case, the component uses the node identifier received in the request to retrieve from the database the interface of the Notification Service of the appropriate underlying MANBoP management node. Finally, the DmMs requests to the Notification Service its connection (or disconnection in case the node is being removed), to the specified Notification Service as event consumer.

B Component Design

In the component behaviour section above, we have described the tasks that the Decision-making Monitoring system must develop. These tasks can be easily joined in four groups:

- Register and obliterate policy conditions to be monitored: These tasks form the basic functionality offered to other components in the framework. They represent the most important communication methods between the Policy Consumer Manager and the DmMs component.

- Monitoring of policy conditions: This one is the most important group of tasks, because it represents the core functionality expected from the component. The policy condition monitoring carried out in cooperation with the MMs is an essential functionality in the decision-taking process.
- Monitoring Meter lifecycle controlling: The component also carries out a group of tasks aimed to uninstall those MMs that are not used so as to keep their number in a reasonable scale and save computing resources.
- Notification Services interconnection: These tasks are possibly the less frequent in time (they are only realised at bootstrap or when the underlying topology is updated), but very important for the correct behaviour of the whole component. These tasks are aimed to guarantee that, when working over other MANBoP instances, the Notification Service of the current instance is connected, as event consumer, in all underlying Notification Services.

These groups of tasks listed above are handled within the component by two classes that are shown in the class diagram below: the DLgc and the MMCnt class. The Notification Service is a CORBA-service and therefore is not described exhaustively within this thesis.

The concrete description of how each class copes with the corresponding functionality is given in the class description section hereafter.

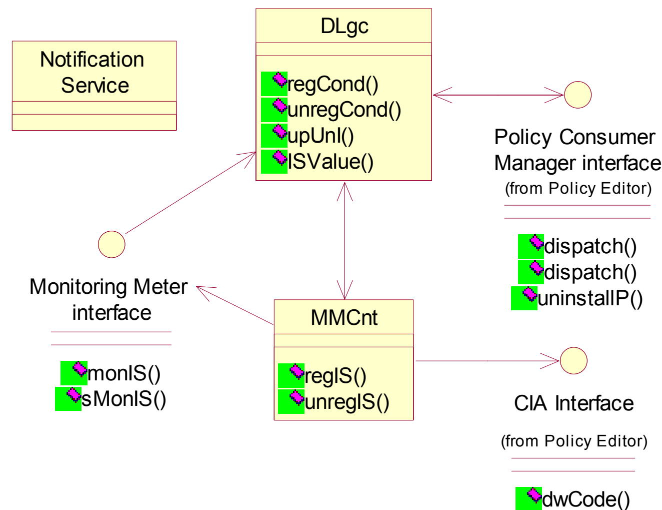

Figure 4 - 57. Decision-making Monitoring system class diagram

a DLgc class

The DLgc (Decision Logic) class carries out the Notification Services interconnection control and participates, together with the MMCnt class in

the registration/obliteration of policy conditions and their monitoring. Particularly, the class is in charge of: splitting the policy condition into Individual Statements (ISs), creating the logic expression, maintaining it and warning the Policy Consumer Manager component when the logic expression changes its value.

The concrete tasks developed by this class are explained in more detail in the following interface description table. The DLgc offers an interface with four methods, namely the regCond(), unregCond(), upUnI() and ISValue(). The first three methods will be used by the PCM component, while the last one is used by Monitoring Meters.

The table below provides a concrete description of these methods, their functionality and parameters.

Interface	*Input parameters*	*Output parameters*	*Functionality*
regCond()	string policySer, credential User	boolean Result	The PCM component uses this method to request the monitoring of a policy condition. The method specifies two input parameters and a boolean specifying the result as output. The input parameters are the serialised policy whose condition should be monitored and the credentials of the user who introduces that policy. The user's credentials might be needed in case the resources to be monitored are assigned to a user. When the DLgc receives such a request, it carries out the tasks described in Figure 4 - 53 up to 'link logic expression to policyId and IS identifiers'. Then, it requests to the MMCnt class the realisation of the remaining tasks in that figure.
unregCond()	string PolicyId	boolean Result	When a policy is removed, the Policy Consumer Manager uses this method to request to the DLgc class the obliteration of its policy condition. The method only specifies the policyId of the policy whose condition should be obliterated as input parameter and a boolean specifying the result as output parameter. The DLgc, when receiving a call to this method, realises the tasks drawn in Figure 4 - 54 up to 'remove logic expression' and request to the MMCnt class the realisation of the remaining tasks in the figure.
upUnI()	boolean type, string[] nodeId	boolean Result	Whenever the managed topology is updated, the PCM component accesses this method to notify it to the DLgc class. Thus, the Notification Service can be always appropriately interconnected. The method includes two input parameters and a boolean as result of the operation. The input parameters are a boolean that indicates whether the node identified in the second parameter is being added (0) or removed (1). The DLgc class carries out the tasks listed in Figure 4 - 56 when a request is received in this method.
ISValue()	string ISId, boolean Value		The Monitoring Meters (MMs) use this method to inform to the DLgc class about changes in the value of an Individual Statement they are monitoring. The method specifies two input parameters: a string identifying the IS that has changed and a boolean that indicates the new value (i.e. whether the IS is met or not). When a request is received through this interface, the DLgc class carries out the tasks reflected in the activity diagram drawn in Figure 4 - 55.

Table 4 - 23. The DLgc class interface description table

b MMCnt class

The MMCnt (Monitoring Meter Controller) class collaborates with the DLgc class in the tasks related with the registration and obliteration of monitoring requests and realises, on its own, the MMs lifecycle controlling tasks.

Summarising, the main responsibility of this class is controlling the whole lifecycle of the Monitoring Meter components, starting from their installation when they are needed, the demultiplexing of Individual Statements to them, and the periodic verification of the number of ISs processed by each of them for uninstalling those not processing any.

The MMCnt class offers an interface with two methods so as to realise these tasks, namely the regIS() and unregIS() methods. The concrete goal, functionality and parameters of these methods are analysed in the table below.

Interface	*Input parameters*	*Output parameters*	*Functionality*
regIS()	string ISId, string ISSer, credential User	boolean Result, boolean initV	The DLgc class makes a call to this method to find the most appropriate Monitoring Meter to process an Individual Statement and to install it if not already running. The method specifies three input and two output parameters. The input parameters are a string identifying the Individual Statement to be processed, another string with the IS serialised and the credentials of the user (they might be needed by the corresponding MM in case the resources requested are allocated to a user). The output parameters are a boolean that indicates the result of the operation and a boolean with the initial value of the IS. When the MMCnt receives such a request, it carries out the tasks described in Figure 4 - 53 starting from 'find the Monitoring Meter responsible for each IS that needs monitoring'. Its tasks conclude with a request to the corresponding Monitoring Meter for the processing of the IS.
unregIS()	string ISId	boolean Result	When a policy is removed, the DLgc class uses this method to request to the MMCnt class the finalisation of the Individual Statements monitoring associated to the policy being removed. The method only specifies a string identifying the IS that should be no longer processed as input parameter and a boolean that returns the result as output parameter. The MMCnt, when receiving a call to this method, realises the tasks drawn in Figure 4 - 54 starting from 'request the stopping of the ISs monitoring'. The method functionality concludes with a request to the MMs for stopping the processing of the corresponding IS.

Table 4 - 24. The MMCnt class interface description table

c Sequence Diagrams

To complement the description of the DmMs component we include hereafter sequence diagrams showing the interactions between the component classes linked with three out of four public methods offered in the component interface. The sequence diagram representing the interactions occurring in the upUnI() method is not shown, since it does not provide any extra information (it is just a call and a response between the PCM component and the DLgc class).

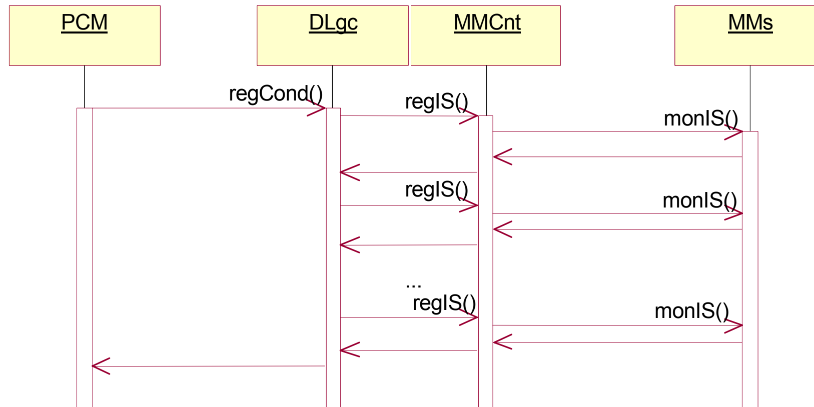

Figure 4 - 58. Decision making Monitoring system: regCond sequence diagram

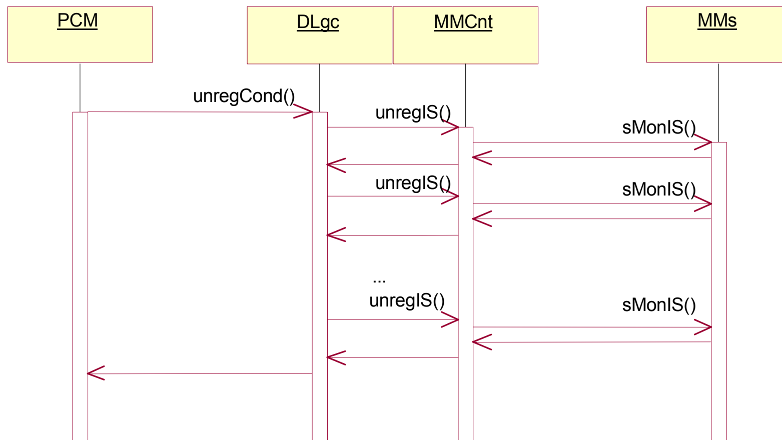

Figure 4 - 59. Decision making Monitoring system: unregCond sequence diagram

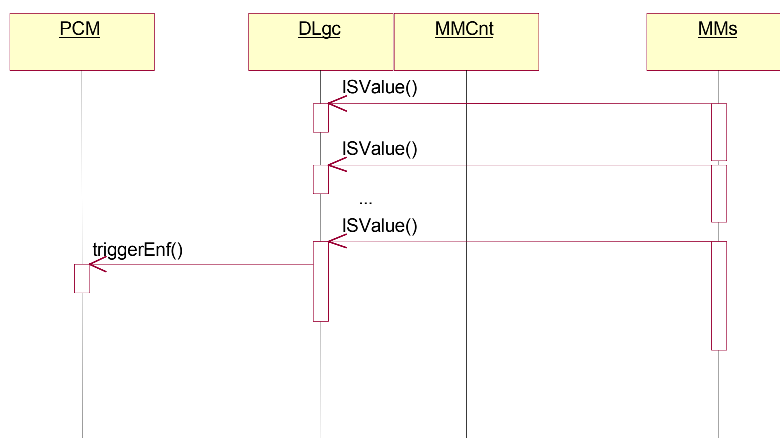

Figure 4 - 60. Decision making Monitoring system: ISValue sequence diagram

7th Monitoring Meter

A Component Behaviour

The Monitoring Meter (MM) components are responsible for the monitoring of Individual Statements (ISs). The MMs are installed based on the management level at which the instance is acting, the IS to be monitored (each MM will process one or more types of ISs) and the underlying devices. The logic behind it is the same as for Policy Consumers: even if we have a MM component capable of processing all present types of ISs, it might occur that future functionalities within the managed network might require the processing of new types of ISs and, therefore, new MM components.

Besides the functionalities enumerated above, we have identified in the Use Cases section some other functionalities for this component that were summarised in Table 4 - 1. We list below those functionalities identified for the MM:

- *Configure Meters:* As briefly explained before, when it receives a monitoring request through the monIS() method, it must analyse the individual statement to decide what resources from the underlying devices should be monitored.
- *Monitor resources:* Each meter class realises the Individual Statements monitoring on the indicated device resources. As result of the method call, the initial value of the IS is returned. From there on, only the changes in this value will be notified until the IS is obliterated. The MMs, when possible, will configure the underlying device to inform about a change in the monitored resources that affects the IS monitored. Otherwise, they poll the underlying device periodically to obtain the value of the monitored resources.

In addition to all functionalities already commented for the Monitoring Meter component, there are some others that should be taken into account.

The monitoring of an Individual Statement might require the monitoring of many resources. Based on all the information obtained from the different resources monitored the MM should decide the value of the Individual Statement being monitored and in case the value has changed warn the DmMs about it.

Moreover, the Monitoring Meter component must be capable of stopping the monitoring of an Individual Statement when requested (as part of the policy removal process). The finalisation of an IS monitoring task will cause, the removal of any configuration that might have been carried out in the underlying device to obtain the needed information.

A peculiarity of Monitoring Meters when working over other MANBoP instances is that, to obtain the value of the monitored ISs they need to create monitoring policies that will be processed by specialised PCs and MMs components. The enforcement of these monitoring policies causes that event reports are sent (either periodically or when the IS value changes). The upper MM receives such events and re-acts accordingly. Obviously, when the IS is obliterated, the MM requests the removal of the corresponding monitoring policies in the underlying instances using the already described methodology for doing so (i.e. introducing the same policy with the *'act'* policy field with the 'Remove' value).

In order to detail and sort out the functional concepts already stated for the Monitoring Meter components, we provide below two activity diagrams. They show the tasks that the different classes of this component should carry out when a request is received through any of the two public methods.

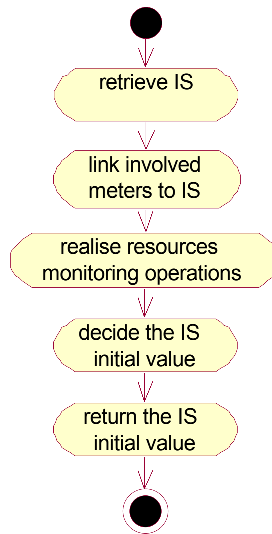

Figure 4 - 61. Monitoring Meter: Monitor Individual Statement activity diagram

The figure above shows the activity diagram of any Monitoring Meter component any time it receives a request for an Individual Statement monitoring. The first task that the component does is analysing the IS to be monitored to decide what resources in the managed network should be monitored to calculate the IS value.

The meter classes are linked with the ISs assigned to them before realising the corresponding monitoring operations. Based on the first result of these operations the component calculates and returns the initial value of the IS.

The component should also be capable of stopping the monitoring of an Individual Statement when requested. The activity diagram below shows the main tasks carried out by the component when that happens.

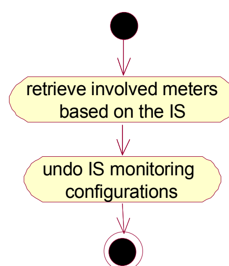

Figure 4 - 62. Monitoring Meter: Stop Individual Statement (IS) monitoring activity diagram

As it can be seen in figure 4 – 62, the tasks undertaken by the component are quite straightforward. With the Individual Statement identifier received in the request, the MM obtains the meter classes involved in the IS monitoring. Then, it requests the removal of all configurations done in the underlying devices to monitor that IS if any.

B Component Design

All tasks described in the component behaviour section can be easily summarised in three points:

- Analysis of the Individual Statement and how to monitor it: The MM must decide what resources should be monitored to calculate the IS value and monitor these resources with the appropriate meter classes. Based on the results obtained from the meter classes, the component establishes the value of the IS.
- Monitoring operations: To obtain the required resource values, the component sends one or more commands to the underlying device. If these commands cause a permanent configuration in the device, this configuration should be removed when the monitoring is finished.
- Finishing the monitoring of an Individual Statement: The component must be capable of stopping an IS monitoring assuring that no configuration related with this monitoring is kept in the monitored device.

In order to cope with the three points shown above, we have designed a Monitoring Meter component composed of two classes: the MFact class (Meter Factory) and the Meter class. The MFact realises all tasks except the monitoring operations, which are realised by the Meter class. The class diagram that follows shows this structure:

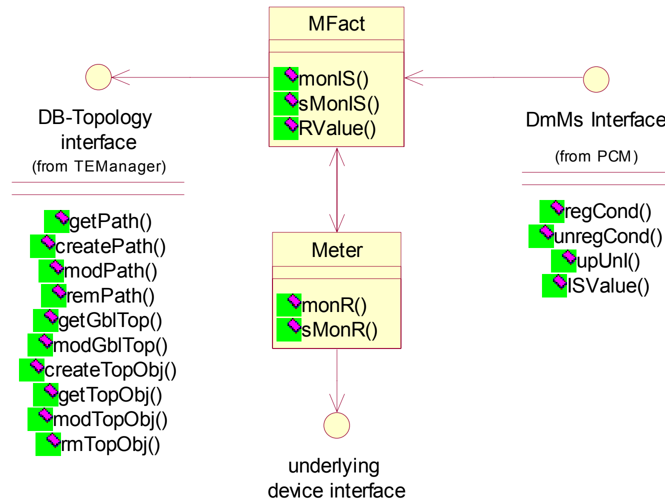

Figure 4 - 63. Monitoring Meter class diagram

In the next sub-sections, we are going to detail how these classes carry out the mentioned tasks.

a *MFact class*

The MFact class is the core of the Monitoring Meter component. It interacts with the Decision-making Monitoring system component and receives the monitoring requests. It also analyses the Individual Statement and decides what resources should be monitored. Then, it demultiplexes to the corresponding Meters the resource monitoring requests. Finally, it is also in charge of finishing smoothly the monitoring tasks related with an IS.

The class offers three public methods, two of them are used by the DmMs to request the beginning or the ending of an IS monitoring, while the third one is used by the meter classes to indicate the values of the resources being monitored. These three methods are, respectively: monIS(), sMonIS() and RValue(). The concrete description of the goal, functionality and parameters of these methods is given in the table below:

Interface	Input parameters	Output parameters	Functionality
monIS()	string ISId, string ISSer, credential User	boolean Result, boolean initV	The DmMs makes a call to this method to request the monitoring of an Individual Statement. The method specifies three input parameters and two output ones. The input parameters are a string identifying the Individual Statement to be processed, another string with the IS serialised and finally, the user's credentials, which might be needed by the corresponding MM in case the resources requested are allocated to a user. The output parameters are a boolean indicating the operation result and a boolean with the IS initial value. This initial value is obtained as result of processing the initial value of the corresponding resources monitored by meter classes. When the MFact receives such a request, it carries out the tasks described in Figure 4 - 61, except the task 'realise resources monitoring operations', which is realised by the meter classes.
sMonIS()	string ISId	boolean Result	When a policy is removed, the DmMs component uses this method to request the finalisation of monitoring tasks related with an Individual Statement. The method specifies a string identifying the IS that should be no longer processed as input parameter and a boolean returning the result as output parameter. The MFact class, when receiving a call to this method, realises the tasks drawn in Figure 4 - 62, except the 'undo IS monitoring configurations' task, which is realised by meter classes.
RValue()	string ISId, int Value		Meter classes use this method to inform to the MFact class about the monitored resource values related with an Individual Statement. The method only specifies two input parameters: a string identifying the IS that has changed, and an integer that indicates the resource value. When a request is received through this interface, the MFact class recalculates the overall value of the IS taking into account this new resource value. In case the IS value changes, the MFact class notifies this change to the DmMs component through the ISValue() method.

Table 4 - 25. The MFact class interface description table

b Meter class

The Meter class receives requests to monitor resources from the MFact class, which it must 'translate' into monitoring commands on the monitored underlying device interface. There can be a number of different Meter classes each being able to monitor one or more concrete types of resources.

The Meter class must be capable also of removing any kind of monitoring-related configurations when requested.

The class offers an interface with two methods, namely the monR() and the sMonR(). A detailed description of the goal, functionality and parameters of these methods is given in the table below:

Interface	Input parameters	Output parameters	Functionality
monR()	string ISId, string nodeInt, boolean type, int period, int upperTh, int lowerTh, string ResourceId, credential User	boolean Result, int initV	The MFact makes a call to this method to request a resource monitoring. The method specifies eight input and two output parameters. Input parameters are first, a string identifying the Individual Statement linked to this resource. Second, a string that specifies the interface of the node where the resource to be monitored is located. Third, a boolean that indicates whether the resource value should be provided periodically (0) or when crossing the specified thresholds (1). Then, three integers that specify the period and the upper and lower thresholds respectively. A string that specifies the resource to be monitored and finally, the user's credentials. The output parameters are: a boolean that indicates the operation result and another boolean with the resource initial value. When the Meter receives such a request it monitors the indicated resource through the node interface given as parameter and returns the resource value, either periodically or when crossing a threshold. The resource monitoring is linked to the IS identifier to easily remove the monitoring configurations when requested.
sMonR()	string ISId	boolean Result	When a policy is removed, the MFact class uses this method to request the finalisation of monitoring tasks related with a resource. The method specifies a string identifying the IS that must no longer be processed as input parameter and a boolean returning the result as output parameter. The Meter class, when receiving a call to this method, removes all monitoring configurations that might have been realised on the underlying device due to the resource monitoring linked with that IS.

Table 4 - 26. The Meter class interface description table

c *Sequence diagrams*

As a complement of the Monitoring Meter component description above, we provide below three sequence diagrams with the main interactions that might occur between the classes of the component:

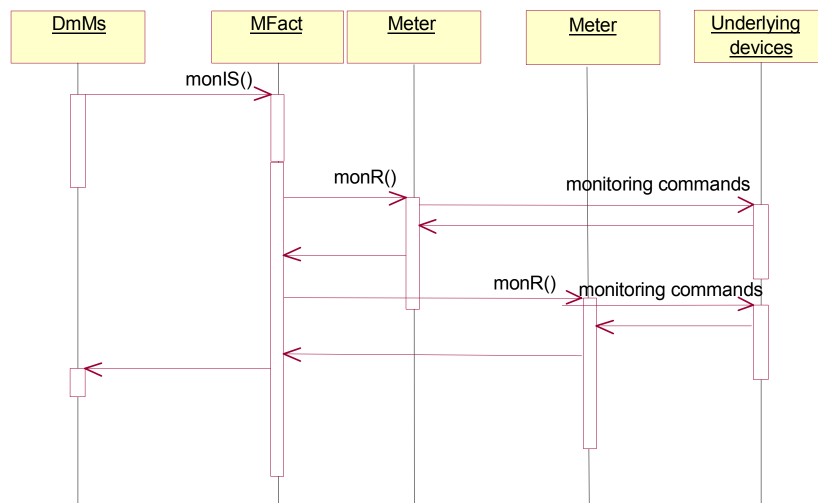

Figure 4 - 64. Monitoring Meter: monIS sequence diagram

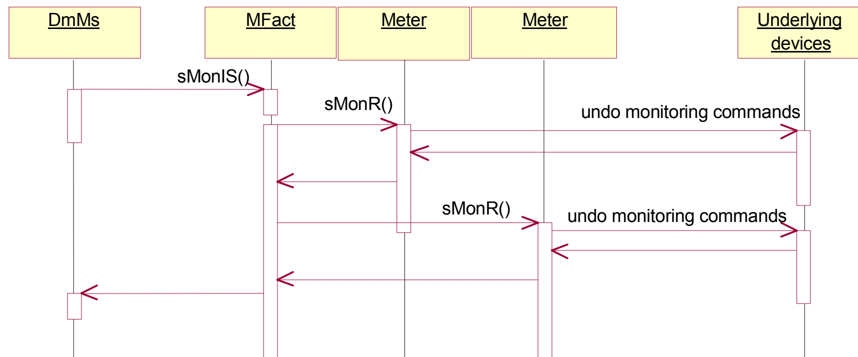

Figure 4 - 65. Monitoring Meter: sMonIS sequence diagram

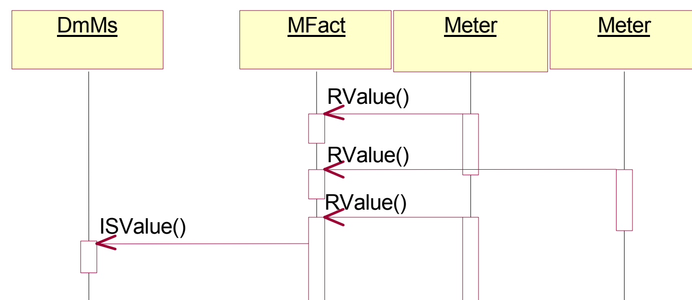

Figure 4 - 66. Monitoring Meter: RValue sequence diagram

8th Policy Consumer

A Component Behaviour

Policy Consumer components (PC) are mainly responsible of interacting with underlying network devices; both mapping policy actions into understandable commands, and receiving signalling requests from the managed devices. Requests are then forwarded in the appropriate format to the Policy Editor component to make a decision.

At the end of the use cases section, in Table 4 - 1, we summarised the main tasks expected from the PC. Hereafter, we enumerate and describe those tasks:

- *Enforce Policy*: This is probably the most important task of the component. When a request for enforcing a policy is received through the enforceP() method, the component maps the action specified in the policy into commands understandable by the underlying device. The device can be a passive, active or programmable router or a lower-level MANBoP

instance. When the latter, the action will be mapped into the appropriate lower-level policies. Moreover, when a Policy Consumer is responsible of several devices, the component has to demultiplex the commands onto the devices specified in the request.

- *Forward request to Policy Consumer*: Policy Consumer components might be also capable (depending on the type of Policy Consumer component) of processing certain types of signalling requests. Calls for processing signalling requests will come from the SigDemux component through the sigRequest() method, which will be described later on. The Policy Consumer component assigns a unique identifier to the signalling request to link the decision (taken in the future) with the request.

- *Format signalling info*: In addition to the task described above, when receiving a signalling request, Policy Consumers must format the information within the request to the format needed by the Policy Editor component, where it will be forwarded.

- *Enforce decision to signalling request*: The enforceP() method previously mentioned for the enforcement of policies is overloaded to support also the enforcement of signalling requests. The concrete description of these methods will be given in the component design section. Nonetheless, the only difference in the enforcement procedure is that the component must link decisions with signalling requests and notify this decision, via the corresponding commands, only to the device that raised the request.

In addition to the above, when enforcing a signalling request, the Policy Consumer component must check if it is the first enforcement request (both from policies or signalling) it processes. In that case, it means that it has been installed by the SigDemux component[19]. Therefore, and only in this particular case, if the signalling request has been refused, the Policy Consumer component must forward the refusal to the device that raised the signalling request and uninstall itself. The explanation to this behaviour is given in the SigDemux section (see pag.198). The framework might also implemented so that only those Policy Consumers that can be installed by the SigDemux component contain this functionality (to avoid an unnecessary degradation in the performance of the other ones).

These functionalities represent the main logic of the component. However, there are two additional functional requirements that the component must fulfil. Both requirements must be carried out during the component instantiation.

The first one is the registration of the Policy Consumer instance, as event consumer, in the Notification Service of the framework. This is particularly

[19] The first 'job' establishes somehow the necessity that caused its installation and thus, the component that installed it.

important when the MANBoP instance where the component is running works over other MANBoP instances, since the PC will receive the enforcement results in the form of events.

The second one is the registration in the corresponding SigDemux component, which will be described in detail in the next section. The Policy Consumer should only register in the SigDemux component running within the same nodeSet. This registration is needed only when the Policy Consumer being instantiated is capable of processing one, or more, types of signalling requests and it is working directly over managed resources. In the registration process, the component must indicate the signalling types that it is capable of process.

In some of the previous component behaviour sections, we have included activity diagrams for clarifying, and sorting out, the functionality of the component being described. In this case, we feel that such activity diagrams will not provide any extra information, since there are only a few tasks.

B Component Design

All the above-described logic for Policy Consumer components can be easily summarised in a few groups of tasks:

- Map the received policy or signalling request into commands: The functionality in both cases, policy or signalling request, is almost the same, the only difference is that the signalling decision should be linked with the request.
- Demultiplex commands to device: The component, once the action specified in the policy has been mapped into the underlying device commands, must forward these commands to the specified devices.
- Process signalling requests: Certain PC might be capable of processing some types of signalling requests. Those that are capable of doing so must extract the signalling request information and format it to be forwarded to the Policy Editor component together with a signalling request identifier.
- Registration processes at component instantiation: During the component instantiation, the PC must register in the Notification Service as event consumer. Additionally, if they are capable of processing signalling requests, they must also register in the SigDemux component within its nodeSet. Note that the registration to the SigDemux component is only realised when the instance is working directly over managed devices.

The concrete Policy Consumer logic depends on the management level at which they are acting, the underlying devices and the functionality they support. In this section, we only aim to design a "generic" Policy Consumer component capable of handling at least the mandatory, generic tasks expected from this component. Such design is represented in the class diagram below.

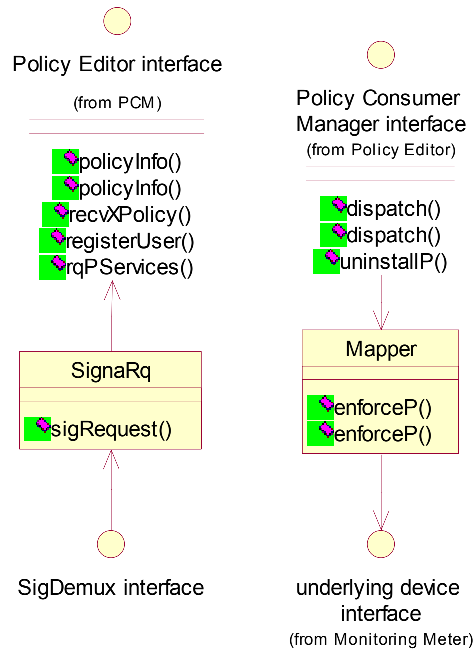

Figure 4 - 67. Policy Consumer components class diagram

A detailed description of these classes, their functionality, interfaces and how do they fulfil the expected functionality is given in the following sub-sections.

a *Mapper class*

The Mapper class is mainly responsible of processing all enforcement requests. That is, mapping the policy action into underlying device commands and demultiplexing these commands into the appropriate devices. It is also in charge of registering in the Notification Service as event consumer.

To fulfil this functionality, the Mapper class offers an interface with two methods, both named enforceP() method. The method has been overloaded to also support enforcement of signalling requests decisions, in addition to the policy enforcement support.

A concrete description of goal, functionality and parameters of these methods is given in the table below:

Interface	Input parameters	Output parameters	Functionality
enforceP()	string policySer, string[] nodeInt	int result, string error	The PCM component requests, through this method, the enforcement of a policy in one or more underlying devices. The method specifies two input and two output parameters. The policy that needs to be enforced serialised (policySer) in a string, and the list of node interfaces where this policy should be enforced, are the input parameters. The output ones are: an integer specifying the result and a string providing more details when an error occurs. The possible integer values are: (0) enforced, (1) enforcement removed, (2) policy removed, (3) enforcement error, (4) undefined result. Enforcement removed applies when, due to the policy conditions, the configurations in the managed device related to this policy are removed. Policy removed applies when the policy is uninstalled due to its expiration, a conflict or any other reason. When receiving a call to this method, the Mapper class realises the tasks described at the section introduction necessary for enforcing the policy.
enforceP()	string policySer, string[] nodeInt, string sigRqId	int result, string error	The PCM component requests, through this method, the enforcement of a signalling request decision. The parameters specified in the method are those described for the previous method plus the signalling request identifier. This identifier is used by the PC component to link the decision made by the system with the signalling request raised. Again, the tasks that the Mapper class realises have been described in the previous paragraphs. In brief, the class links the decision to the request and enforces it in the corresponding underlying device interface.

Table 4 - 27. The Mapper class interface description table

b *SignaRq class*

The SignaRq class deals with the processing of signalling requests (i.e. formatting of request information) and the registration to the SigDemux component.

To fulfil this functionality the SignaRq class offers an interface with the sigRequest() method. The goal, functionality and parameters of this method are described in detail in the table below.

Interface	Input parameters	Output parameters	Functionality
sigRequest()	string request, string nodeAddr		The SigDemux component requests, through this method, a decision for a signalling request. The method specifies two input parameters: a string with the request to be processed and another string with the IP address of the node from where the request is raised. When receiving a call to this method, the SignaRq class realises the tasks described before necessary for processing the signalling request.

Table 4 - 28. The SignaRq class interface description table

c *Sequence diagrams*

To complement the component description, we provide below sequence diagrams reflecting the interactions occurring when public methods of the component are accessed.

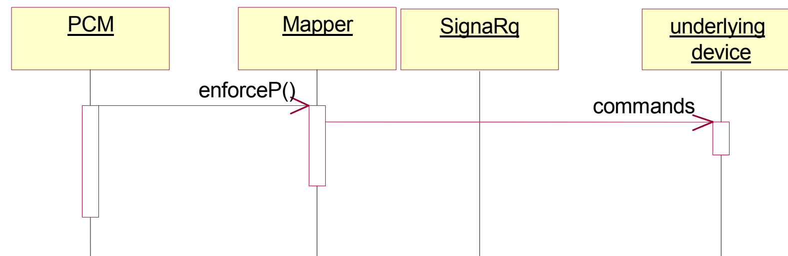

Figure 4 - 68. Policy Consumer: enforceP sequence diagram

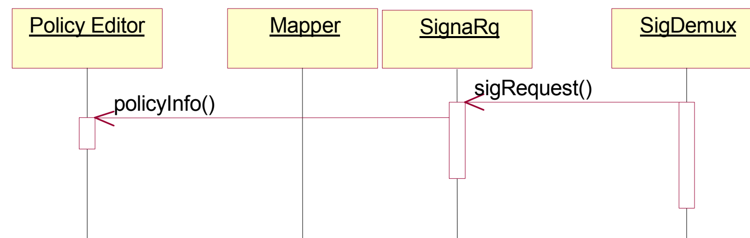

Figure 4 - 69. Policy Consumer: sigRequest sequence diagram

9th SigDemux

A Component Behaviour

The SigDemux component is exclusively involved in the processing of signalling requests. In particular, it is in charge of correctly receiving signalling requests supported by the framework (i.e. those supported by PCs already installed in the framework) and demultiplexing them to the appropriate Policy Consumer component. Additionally, the SigDemux might be able to detect the most common signalling request types and force the installation of a Policy Consumer to process them if none is available yet.

When we analysed the functionality expected from the framework and assigned functionalities to be covered to the different components (see Table 4 - 1), the ones assigned to the SigDemux component were:

- *Wait for signalling request:* The component must be always listening for signalling requests from underlying devices within its nodeSet. When a request is received, the SigDemux must detect the request type and act accordingly.

- *Find appropriate Policy Consumer:* When a known signalling request is detected the component looks for a Policy Consumer component associated with the signalling request type received.

- *Is it installed?:* Finally, it verifies if the corresponding Policy Consumer is installed. If so, it simply forwards to it the signalling request together with

the underlying device from where the request has been raised. In case it is not installed, the component might even request to the Code Installing Application (CIA) the installation of the corresponding Policy Consumer component to process that request within that nodeSet.

The SigDemux might be hardcoded with the knowledge of the most common signalling requests types, so that if one request of any of these types arrives, the corresponding Policy Consumer can be installed to process such request. Nevertheless, the SigDemux component is capable of "learning" new types of signalling requests to be able to detect them later. This happens when a Policy Consumer capable of processing a new type of signalling request registers in the SigDemux component. The Policy Consumer will register to the SigDemux component indicating not only a handle to receive processing requests of this signalling request type, but also indicating to the SigDemux component how to detect this signalling request type. Indeed, the Policy Consumer component provides a filter matching requests of the new type. Obviously, a new SigDemux component with new types of signalling requests can also be developed, either for a new nodeSet managed by the MANBoP instance, or for an existing one (thus, replacing the old SigDemux component).

As we have seen in previous sections, the Policy Consumer components are usually installed by the Policy Consumer Manager component, and more concretely by its PCCnt class. When a PC component is installed by the SigDemux component the "registration" of the new PC within the PCM component will be done progressively (during the signalling request processing). When the signalling request arrives to the PCCnt class, the new Policy Consumer-related information is stored (i.e. its PCId and component interface) in the list kept by this component (see page 115) and the "registration" process is finished. By "registration" process we meant, in this case, the process of letting know to the PCM that a new Policy Consumer is being installed, registering its expiration date and updating the list of installed Policy Consumers kept by the Policy Consumer Manager component.

Nonetheless, this "registration" process is complete only when the signalling request is accepted. That is, the request is authorised and there is no dynamic conflict. For that reason, when the signalling request is refused the Policy Consumer component will not have been registered in the PCM (always assuming that it is the first request it processes). Hence, to avoid having not-registered, and therefore, unused Policy Consumers within the MANBoP instance, the Policy Consumer forwards the refusal to the device that raised the signalling request and uninstalls itself.

The activity diagrams below sort out the above-described functionality for this component, when receiving signalling requests or when a new Policy Consumer registers in the component.

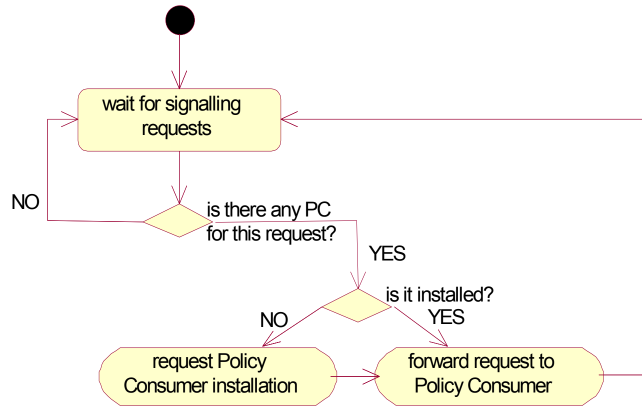

Figure 4 - 70. SigDemux: Signalling request processing activity diagram

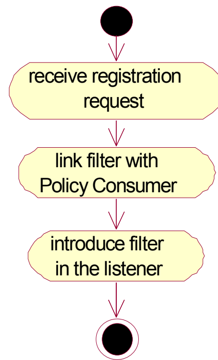

Figure 4 - 71. SigDemux: Policy Consumer registration activity diagram

B *Component Design*

All the above tasks can be summarised in two groups: tasks for detecting signalling requests and finding the appropriate Policy Consumer, and tasks for registering a new Policy Consumer. This component behaviour has leaded us to a component design composed of two classes. The class diagram of the SigDemux component is shown in the figure below:

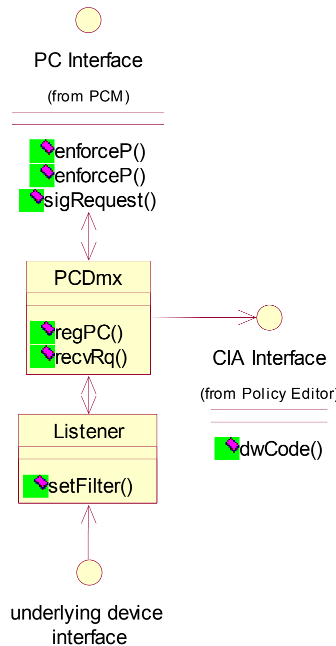

Figure 4 - 72. SigDemux class diagram

A detailed explanation of how these classes cope with the expected functionality of this component is given in the following sub-sections.

a PCDmx class

The PCDmx (Policy Consumer Demultiplexer) class is in charge of finding the appropriate Policy Consumer for a signalling request received. Moreover, in case the corresponding Policy Consumer component is not installed, the PCDmx contacts the Code Installing Application (CIA) component to request the PC installation within the nodeSet.

Additionally, the PCDmx class is responsible of receiving registration requests from PCs, updating the list (kept within the component) of registered and installed PC components and contacting the Listener class to include the signalling request filter specification of the new PC.

To handle with this functionality the class offers an interface with two methods, namely the regPC() and the recvRq() methods. The concrete goal, functionality and parameters of these methods is described in the interface description table below:

Interface	Input parameters	Output parameters	Functionality
regPC()	string PCId, string PCInt, string[] filterId, string[] filter	boolean result	The PC components use this method to register within the SigDemux as entities capable of processing a particular type of signalling requests. The method specifies four input parameters and a boolean indicating the operation result as output. The input parameters are two strings and two arrays of strings that indicate respectively: the identifier of the PC being registered, its interface, the identifier of the filters associated to the supported signalling types and the filters themselves. When receiving a call to this method, the PCDmx class develops the tasks described in the previous paragraphs and shown in Figure 4 - 71.
recvRq()	string filterId, string request, string nodeAddr	-	The Listener class uses this method to inform about the detection of a signalling request coming from a managed device. The parameters specified in the method are three strings that include the following information, respectively: the identifier of the filter met by the signalling request, the request itself and the address of the node raising the signalling request. These three parameters are input parameters; no output parameter is included in the method. The PCDmx class realises mainly the tasks described before and reflected in Figure 4 - 70.

Table 4 - 29. The PCDmx class interface description table

b *Listener class*

The Listener class continuously waits for signalling requests and compares whether the information received matches with any of the specified filters or not[20]. In case it does, the class must inform about it to the PCDmx class. In addition, the class offers an interface that allows the introduction of new filters.

A detailed description of the goal, functionality and parameters of the methods within the Listener interface is given in the table below:

Interface	Input parameters	Output parameters	Functionality
setFilter()	string filterId, string filter	boolean result	The PCDmx class uses this method to introduce a new signalling request filter. The method specifies two input parameters and just one boolean indicating the operation result as output. The input parameters are a couple of strings that represent, respectively, the identifier of the filter being introduced and the filter itself. The Listener class, when receiving a request through this method, introduces the requested filter within its list.

Table 4 - 30. The Listener class interface description table

c *Sequence diagrams*

Although the sequence diagrams for this component are quite straightforward, we include them to give a clearer view of the interactions

[20] At component instantiation, the class might need to register within the managed devices as receiver of signalling requests.

occurring within the component when receiving a registration request from a PC or when detecting a signalling request.

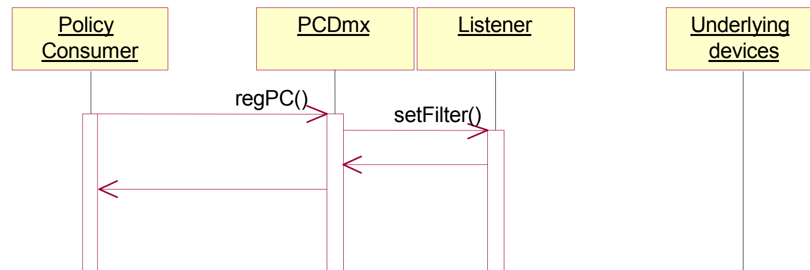

Figure 4 - 73. SigDemux: regPC sequence diagram

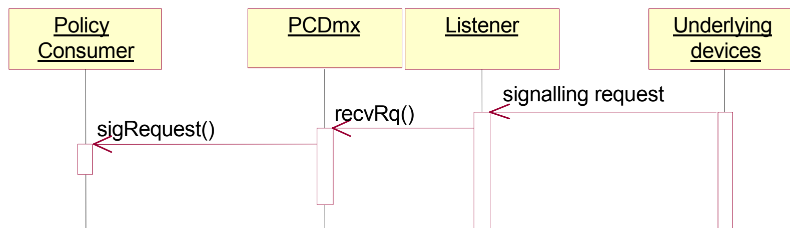

Figure 4 - 74. SigDemux: Signalling request detection sequence diagram

10th Database

A Component Behaviour

No database functionality appears in Table 4 - 1. There are two reasons for this. The first one is that the Database functionality is masked behind the other tasks. The second reason is that the Database is mainly a passive component. In this case, with passive we refer to the fact that the Database component implements no algorithm for realising any task other than storing and allowing the retrieval of objects. Additionally, as the Database component offers methods for each particular Information Model Object (IMO) that can be stored, the logic for finding objects is very simple. The advantage of using these fine-grained methods in the Database is that the logic needed by other framework components for introducing and retrieving objects from the Database is simpler. Another advantage of this approach is that in this way the Database interface is independent of the database technology used. The only drawback for this approach is that the Database interface offers a higher number of methods. We have tried to minimise this drawback by grouping all these methods into several interfaces.

The methods offered by the Database component are grouped under different interfaces. These groups have been created taking into account the functionality represented by the IMOs handled by the group methods. For

example, all methods handling IMOs related with topological aspects of the framework are grouped under the Topological interface. The goal of using this type of grouping is that on the one hand, components need to access the minimum number of interfaces to retrieve and store the objects they are dealing with. On the other hand, that the size of the Database interfaces is not too big.

B Component Design

As we have extensively commented in the previous sub-section, the Database is a passive component. Thereby, it has no other classes than the ones implementing its interfaces. Hence, we will proceed directly for describing the interface methods.

Database methods realise the same functionality depending on whether they are used for create, get, set, modify or remove IMOs. The first characters of the method name help to identify which of these functionalities is the method implementing.

Those methods used for creating objects in the Database, receive information to create the IMO. Once they have created the object, they send the corresponding Database technology commands to store the new object.

The set methods realise the same functionality of the create methods except for the fact that they already receive the IMO.

Get methods introduce the corresponding Database technology commands to look for an IMO based on the parameters they have received. If they find the IMO, they return it.

The modify methods implement the same functionality as the get methods except for the fact that they do not return the found IMO. Instead, they modify it and store it again following the same behaviour as the set methods.

Finally, remove methods introduce the corresponding Database technology commands to look for an IMO and, if it exists, it requests the removal from the Database of that IMO again introducing the appropriate commands.

The class interface is described in the table below. As we have already described the functionality of the database methods, we just include in the table a brief description of the input and output parameters for each method.

Interface	Input parameters	Output parameters	Functionality
Topology interface			
getPath()	string pathId	Path object	The input parameter is a string with the identifier of the path that must be obtained from the Database. The output parameter is the Path object, if any is found.
createPath()	string pathId string[] nodes, string[] links, string costId, boolean looping, string parentPathId.	boolean	The method receives seven input parameters. These input parameters represent the path identifier, the nodes crossed by the path, the links crossed by the path, the identifier of the cost object containing the path costs, a boolean used for path calculation purposes and the identifier of the parent path to this one used for calculation purposes. For more information about these parameters see page 154. The result is a boolean that indicates whether the operation could be realised successfully or not.
setPath()	Path object	boolean	The input parameter is the Path object that must be stored in the Database. The result is a boolean that indicates whether the operation could be realised successfully or not.
modPath()	string pathId string[] nodes, string[] links, string costId, boolean looping, string parentPathId.	boolean	The input and output parameters are those described for the createPath method.
remPath()	string pathId	boolean	The input parameter is the a string with the identifier of the path that must be removed from the Database. The output parameter is a boolean that indicates the result of the operation.
getGblTop()	-	GblTop IMO	No input parameter is needed since there is only one GblTop IMO in the Database. The output parameter is the GblTop IMO.
modGblTop()	string[] nodes, string[] aps, string[] links	boolean	The input parameters are three arrays of strings. These arrays contain the identifiers of nodes, access points and links forming the managed topology. As output parameter just a boolean is returned indicating if the modification was successful.
createTopObj( )	String nodeId, int type, boolean edge, String[] outL, String[] inL, String nResoId, String nUResoId	boolean	Seven input parameters are introduced in the method. The first one is the identifier of the node represented by the IMO that has to be created. The second parameter is an integer that indicates the type of device (active, programmable or passive). A boolean indicating whether the node acts as access point is included next. The fourth and fifth parameters are two arrays of strings containing the identifiers of outgoing and incoming node links, respectively. The last two input parameters are two strings that identify the resources IMO and used resources IMO. The output parameter is a boolean that indicates the result of the operation.
getTopObj()	String nodeId	Node IMO	The input parameter is a string with the identifier of the node that must be obtained from the Database. The output parameter is the Node IMO, if any is found.
modTopObj()	String nodeId, int type, boolean edge, String[] outL, String[] inL, String nResoId, String nUResoId	boolean	The input and output parameters are those described for the createTopObj.
rmTopObj()	String nodeId	boolean	The input parameter is a string with the identifier of the node that must be removed from the Database. The output parameter is a boolean that indicates the result of the operation.
Resource interface			

setPRI()	PRI IMO	boolean	The only input parameter defined in the method is the PRI IMO that must be stored in the Database. The output parameter is a boolean that indicates if the operation has been developed successfully or not.
getPRI()	string policyId	PRI IMO	To identify the requested PRI IMO, which is the output parameter of the method, we only need to introduce the identifier of the policy linked with that PRI IMO.
rmPRI()	string policyId	boolean	The input parameter is the same as for the previous method while the output parameter just tells if the removal was done correctly.
getRI()	string resoId, string nodeId	NResources IMO	The input parameters used to identify the NResources IMO to be obtained from the Database are two strings. The first one contains the identifier of the NResources IMO while the second one contains the identifier of the Node IMO linked with the previous one. The only information returned is the NResources IMO if any.
getUsedRI()	string resoId, string nodeId	UNResources IMO	The only difference with the parameters defined for the previous method is that the returned object is a UNResources IMO.
createRI()	string resoId, int cpu, int memory, int disk, int EEs, string[] EEIds	boolean	The method defines six input and one output parameter. The input parameters are first, a string with the identifier of the NResources IMO to be created. Then, four integers are included representing respectively, the available cpu, memory, disk and number of EEs. Finally, an array of strings with the identifiers of the available EEs is also included. The output parameter is a boolean showing the operation result.
modUsedRI()	string uresoId, int cpu, int memory, int disk, int EEs, string[] EEIds	boolean	Both the input and the output parameters defined in this method are equal to those in the previous method.
modRI()	string resoId, int cpu, int memory, int disk, int EEs, string[] EEIds	boolean	Both the input and the output parameters defined in this method are equal to those in the previous method.
getRouteI()	string routeId, string pathId, string[] flow	Route object	The input parameters introduced in the method to obtain the Route object stored in the Database, which is the output parameter, are two strings and an array of strings. The two strings represent respectively the route object identifier and the identifier of the path object linked with this route. The array of strings contains five strings used to specify a flow: source and destination IP addresses, source and destination ports and protocol.
setRouteI()	Route object	boolean	To store a Route object in the Database the only input parameter defined is the Route object itself. The output parameter is a boolean showing the result of the operation.
modRouteI()	string routeId, string pathId, string[] flow, string[] uResoId, strin[] linkIds	boolean	The input parameters specified for modifying a Route object in the Database are two strings and three arrays of strings. The two strings contain respectively, the identifier of the route object to be modified and the identifier of the path object linked with this route object. The arrays of strings contain respectively, information to identify the flow linked with this route, the identifiers of the UNResources IMOs used in that route and, finally, the identifiers of the Link IMO used in that route.
Policy interface			
setPolicy()	Policy IMO	boolean	The only input parameter defined in this method is the Policy IMO to be stored in the Database. The output parameter is a boolean that shows the operation result.
getPolicies()	string[] policyIds	Policy[]	The input parameter used to identify the requested Policy IMOs is an array of strings with the identifiers of these policies. The method returns an array with all Policy IMOs found.
getPolicies()	string conds	Policy[]	The input parameter for this overloaded method is a string with a concatenation of conditions in alphabetic order. This string will be used to obtain from the Database all those Policy IMOs that contain such conditions. The output parameter is an array with the Policy IMOs found.

modPSts()	string policyId, int value	boolean	The input parameters defined for this method are a string with the identifier of the policy to be modified and an integer with the new status value. The output parameter indicates if the operation was successful or not.
removeP()	string policyId	boolean	This method defines just one input parameter. This parameter is a string with the identifier of the policy to be removed from the Database. A boolean indicating if the operation was successful is the output parameter.
getPSts()	string policyId	int	The input parameter is the identifier of the policy from which we want to obtain its status. The result is the policy status of that policy, if any is found.
Group interface			
setGroup()	Group IMO, string username	boolean	The input parameters defined in this method are the Group IMO that must be stored in the Database and a string with the name of the user introducing that group. The output parameter is a boolean that shows the operation result.
setGroupP()	Policy IMO, string XPolicy, credential user, string policyId	boolean	There are four input parameters defined for this method. The first one is the Policy IMO that must be stored in the Database. Second, a string with the policy in XML. The credential of the user introducing the group is given in third place. Finally, the identifier of the policy that must be stored is the last parameter. The operation result is shown as a boolean returned by the method.
getGroupSt()	int groupnum, string username	int	The input parameters defined in this method are an integer with the group number and a string with the name of the user introducing the group. The output is an integer showing the group status information if any is found.
getGroupP()	int groupnum, string position, string username	Policy IMO	The method receives three input parameters. The first one is an integer with the group number. Second, a string with the position of the policy searched within the group. The last parameter is the name of the user introducing the group. The output parameter is the Policy IMO obtained from the Database.
modGroupSt()	int groupnum, string username, int value	boolean	The input parameters defined in this method are an integer with the group number, a string with the name of the user introducing the group and an integer with the new group status value. The output parameter is a boolean that shows the operation result.
rmGroupP()	int groupnum, string position, string username	boolean	The method receives three input parameters. The first one is an integer with the group number. Second, a string with the position of the policy searched within the group. The last parameter is the name of the user introducing the group. The operation result is shown as a boolean returned by the method.
rmGroup()	int groupnum, string username	boolean	The input parameters defined in this method are an integer with the group number and a string with the name of the user introducing the group. The output parameter is a boolean that shows the operation result.
Schema interface			
setSchema()	Schema IMO, string username, string domainId	boolean	The input parameters defined for this method are the Schema IMO that must be stored in the Database, a string with the name of the user to which that schema applies and a string with the identifier of the functional domain represented by the schema. The output parameter is a boolean that shows the operation result.
getSchema()	string username, string domainId	Schema IMO	The method is defined with two input parameters. These are a string with the name of the user to which that schema applies and a string with the identifier of the functional domain represented by the schema. The method output is the Schema IMO obtained from the Database, if any.
remSchema()	string username, string domainId	boolean	The method defines two input parameters: a string with the name of the user to which that schema applies and a string with the identifier of the functional domain represented by the schema. The output parameter is a boolean that shows the operation result.

Table 4 - 31. The Database interfaces description table

Section IV.4 – Conclusions

Along this chapter we have described in detail the design of the proposed solution. We have first explored all functionality that the solution offers, to explain, afterwards, how this functionality is achieved by all framework components.

The management framework proposed consists of two main sets of components: fixed components and dynamically installable components. Fixed components, i.e. the Policy Editor (PE), the Policy Consumer Manager (PCM), the Decision-making Monitoring system (DmMs), the Authorisation Check Component (ACC), the Database (DB), the TEManager and the SigDemux, provide the policy logic independent of both the functionality and the managed device. The other components are dynamically installed when needed, either based on the functionality, as the Policy Conflict Check (PCC), or based on both the functionality and the managed devices, as the Policy Consumers (PCs) and Monitoring Meters (MMs).

The goal of this division of components is to make the framework generic enough to be instantiated at different management levels and over heterogeneous managed devices; and clever enough to automatically extend itself with the needed functionality at each particular time. Moreover, when instantiated at network and subnetwork levels, the system can either work directly over the managed resources or over element or subnetwork level managers.

The logic behind such an approach (i.e. to have a core set of components that are dynamically extended with the needed functionality) is to apply active networking philosophy to the management plane. In this way, policies act as active packets at the management plane, which carry pointers to the components that should process them. When the policy arrives to the management station, it will automatically download the components that process the policy before forwarding it to them. The processing of the policy will finally derive in the appropriate configuration actions over the managed devices.

This approach greatly simplifies the creation of a management infrastructure to network operators since they only need to instantiate the required MANBoP instances at the appropriate locations within the management infrastructure. For example, the network operator can manage specific geographic areas with independent subnetwork managers and keep the management of the whole network with a common network manager. Also, it can choose between a more cost-effective solution and a more efficient, scalable and distributed one.

Apart from this flexible and dynamic extensibility property, the framework includes the possibility of adding or removing network nodes to the managed topology. This capability together with the support of heterogeneous

technologies permits the progressive introduction of active routers on the managed network without stopping or modifying the management infrastructure.

The delegation capabilities supported by the framework are mainly represented by the Authorisation Check Component (ACC). This component is in charge of checking all policies arriving to the framework against the access rights information stored in the database for that user. This information is introduced in the system by means of delegation policies sent by the principal who is delegating the functionality. The enforcement of these policies results in the storage of access rights data in the Database (DB) in a format used by the ACC.

The most important output from the chapter is that, as we have briefly summarised, the design presented in this chapter supports all the requirements set to the Thesis in Chapter Two. Most of the requirements are achieved through the division of the framework functionality in two sets of components and the extensibility mechanism designed. Nevertheless, there are others, like the delegation mechanism, concentred in one component, in this case the Authorisation Check Component (ACC).

All the framework functionality, even the one that is considered as out of the scope of this thesis, has been designed and explained along the chapter. The reasons for including also the design of out-of-scope functionality are mainly two. The first one is to have a complete document describing all functionality needed by the framework to work. The second one is to show the feasibility of the solution proposed in all its aspects, including those considered as out of the scope of this thesis.

The design description is heavily supported on UML diagrams with the objective of easing significantly the comprehension of the framework and its behaviour. More specifically, activity and sequence diagrams have been used systematically for describing the behaviour of every component within the framework and its relations with other framework components. Additionally, a class diagram has been included for each of the framework components. These class diagrams show the component classes designed and their methods. Indeed, the whole framework could have been described using UML tools, however this would have required detailed knowledge of these tools to follow the description. We have opted for a combined approach, mixing text with UML diagrams to facilitate the comprehension of the proposed framework.

The following chapter provides a detailed description of the proof-of-concepts implementation carried out to afterwards evaluate the solution proposed. The chapter includes an extensive description of all implementation aspects as the naming convention followed, implemented code and technologies used. Furthermore, within the next chapter we also describe the Information Model designed and implemented. The reason for including the Information Model description in the implementation chapter is

that a substantial part of the Information Model depends on the functional domains used in the proof-of-concepts. Thereby, to keep the entire description of the Information Model in a unique section, we have included it inside the proof-of-concepts implementation chapter. At the end of the chapter, the scenarios that will be used to evaluate the proposed framework are also detailed.