Using CIM to Realize Policy Validation within the Ponder Framework

Leonidas Lymberopoulos, Emil Lupu and Morris Sloman

Imperial College, Department of Computing, 180 Queen's Gate, SW7 2BZ, London, UK
{llymber, e.c.lupu, m.sloman}@doc.ic.ac.uk

Abstract

The validation of policy is necessary to ensure that it will lead to a feasible implementation for the environment. This requires checking that the policy is consistent with the functional or resource constraints within the target environment. For example, do the policies assume functionality or specific operations, which do not exist in the target routers, or bandwidth in excess of the capacity of the data links? The objective is to support static checking, where possible, prior to deployment in order to detect invalid policies at design time. However there are some policies related to resource allocation that depend on the current state of the system, and require policy constraints that must be checked at execution time. In this paper, we will discuss how CIM can be used within the Ponder Policy Framework to validate network policies that apply to a Differentiated Services (DiffServ) domain against the capabilities of the individual network elements that comprise the DiffServ domain.

1. Motivation

There is considerable interest in the Internet community in policy-based networks as a means of implementing adaptive QoS (Quality of Service) management, caching, persistence and security to support modern multimedia applications, mobility and ubiquitous computing. Policies are rules governing the choices in behaviour of a system. Authorisation policies are used to define what services or resources a subject (management agent, user or role) can access. Obligation policies are event triggered condition-action rules which can be used to define the conditions for reserving network resources, changing queuing strategy, or loading code onto a router. In a real networking scenario, multiple policies will apply to the network elements in order to support the requirements of different applications, different users and cooperating but distinct administrative domains. Furthermore, the shared resource that the network represents is itself composed of different elements with varying capabilities and interfaces. A prime concern is to ensure that the network elements have the capability to implement the policy.

Policy validation ensure that only the policies that satisfy constraints, such as those relating to device functionality or resource availability, will actually be enforced within the managed environment. This will prevent the management overhead and the potential problems that may arise when trying to enforce policies that are not feasible in the given network environment. In this paper, we will discuss how network policy that applies to a DiffServ domain can be validated against the capabilities of the individual network elements that comprise the DiffServ domain. Validation of policies that relate to end-to-end flows or service level agreements (SLAs) is not discussed in this paper.

2. The PONDER Policy Framework

PONDER is an object-oriented, declarative language developed at Imperial College for specifying management and security policies [1]. For example, the following authorisation policy with the name bwalloc permits the Agroup to perform the action of setting up a videoconference with bandwidth of 4 Mb/s and priority of 3 to the BGroup in New York or the Dgroup in Boston between 16.00 and 18.00 daily.

```
inst auth+ bwalloc {
  subject Agroup;
  target BGroupNY + DGroupBoston;
  action videoconf (bw=4, priority=3);
  when time.between (1600, 1800); }
```

The following obligation policy type named videoSetUp, takes two parameters – a subject which evaluates the policy and a target on which the action to reserve bandwidth is performed, when an event is received for a videoRequest with the requested bandwidth bw as a parameter. A constraint defines that the reservation will only take place if the allocated

bandwidth plus the request is less than a maximum allowed bandwidth. It is assumed that the allocated and maximum bandwidth are variables held within the subject. (The subject would either be a single object in the CIM Schema, or information found by traversing the information model starting from the subject s). Two instances of the policy are then created for different gateway subjects and router targets.

```
type oblig videoSetUp (subject s; target t;) {
  on videoRequest (bw);
  do t.bwreserve (bw);
  when ((s.allocatedbw + bw) < s.maxbw); }

inst USvideoSetUP = videoSetUp
(gateways/USgateway, routers/USedgeRouter);
inst UKvideoSetUP = videoSetUp
(gateways/UKgateway, routers/UKedgeRouter);
```

PONDER also supports grouping of policies into roles related to positions in organisations or the set of policies applying to a particular network component [2]. Management structures can be defined as configurations of roles with policies applying to relationships between roles for organisational units such as departments or buildings. Inheritance permits specialisation of existing policy specifications for different environments. PONDER also allows complex actions to be implemented by dynamically loaded scripts within the subject policy interpreter. Details of the PONDER language are described in [1], [3].

It should be noted that PONDER realizes the basic constructs of the IETF's and DMTF's Policy Core Information Model (PCIM) in its approach to obligation policies.

3. Policy Validation within the Ponder Framework

Enforcement of a network-level policy requires the creation and/or configuration of the functional elements which implement the policy within the routers to which the policy applies. Consider a network element that implements various functional elements to support DiffServ e.g. classifiers, meters, schedulers, queues, etc. Several information models, such as the CIM Network sub-model [4], the IETF DiffServ router model [5], etc. represent the individual functional elements which a DiffServ-enabled device supports.

In this context, a policy is valid with respect to the target device capabilities when the target device is able to either create and configure a new functional element with the requested values or configure an existing one. In the following, we will provide details of each case of validation and we will outline how policy validation can be implemented within the Ponder policy framework using CIM as the model for representing the policy's target mechanisms and capabilities.

3.1 Validation of the ability of target devices to create DiffServ functional elements

When creating new DiffServ functional elements it is necessary to check whether the target device is capable of performing the "add" operation, i.e.

i) Does the device support the requested DiffServ functional element type?
ii) Does the device have enough resources to create (add) the new functional element?

For the first case, consider the following policy that the administrator specifies for providing traffic conditioning on a set of edge routers.

Example 1: Network policy rule for creating DiffServ functional elements within the target devices

```
inst oblig /Policies/TrafficConditioningOnEdge {
subject /PolicyAgents/NetworkPMA;
target t = /Routers/EdgeRouters;
on   TrafficConditioningRequest ( meter_rate, dscp);
do   AverageRateMeter avg_meter =
       new AverageRateMeter (meter_rate) ->
     t.addDiffServElement (avg_meter) ->
     DSCPMarker dscp_marker =
       new DCSPMarker (dscp) ->
     t.addDiffServElement (dscp_marker); }
```

The policy TrafficConditioningOnEdge instructs the target edge routers to create two functional elements: a meter of type AverageRateMeter and a marker of type DSCPMarker. This policy is valid only if the target devices support the meter type AverageRateMeter and the marker type DSCPMarker. In order to perform this type of validation, we must know the different types of functional elements a DiffServ-enabled device supports. A solution to this would be to query the CIM representation of the device which would indicate which types of functional elements a DiffServ-enabled device implements – the CIM network sub-model provides this information.

This type of policy validation can be implemented within the Ponder framework by using meta-policies as a means to specify the constraints that network policies must satisfy with respect to the DiffServ mechanisms which the policies' targets support. Meta-policies defined in Ponder specify constraints over a set of policies, on the permitted types of policies or their policy elements. These constraints apply to policies within a specific scope and limit the permitted policies in the system. The syntax of a meta-policy is based on

the syntax of the object constraint language (OCL). The body of the meta-policy specifies the constraint as a series of OCL expressions separated by semicolons. The expressions can be boolean or navigation expressions. If any of the boolean expressions evaluate to *true*, execution stops and the action following the *raises*-clause is executed.

Example 2 shows a meta-policy for specifying constraints over a set of network policies with respect to the DiffServ mechanisms supported by the policy targets (i.e. routers). It is assumed that the policies to be validated would be similar to the one shown in Example. Note that meta-policies could also specify constraints with respect to network protocols other than DiffServ. This provides a flexible framework for validating policies independently of the underlying technology.

Example 2: Meta-policy for specifying constraints over the permitted DiffServ mechanisms

```
inst meta notSupportedElement raises
  notSupportedElementException(invalid_policies) {

[invalid_policies] = this.policies ->
  select  (p |  p.action ->
   exists (a | a.name = "addDiffServElement"
        and p.target ->
        exists(t | t.notSupports(a.parameter.oclType))));

invalid_policies -> notEmpty; }
```

The body of the meta-policy contains two OCL expressions. The first one selects all policies (p) with the following characteristics: the action set of p contains at least one action named "addDiffServElement", whose parameter's type (i.e. the type of the DiffServ element that p wants to add to each of the target devices) is not supported by at least one of the policy's target devices. Note that we use the OCL method oclType to obtain the type of the "addDiffServElement action's parameter. The action notSupports on the target device is a look-up operation on the CIM representation of the device. It returns true if the device does not support the specific DiffServ element type, which is carried as parameter in the action notSupports.

The second OCL expression returns true if the variable invalid_policies, which is returned from the first OCL exception is not empty. If the result of the last expression is true, the exception notSupportedElementException specified in the raises-clause executes. It receives the invalid_policies set as a parameter.

An obligation policy could be triggered by the exception raised by the meta-policy rule in order to performing corrective actions as a means to resolve policies that are invalid with respect to the mechanisms of the target devices. An example would be to install the missing DiffServ mechanisms in a programmable router.

Example 3 gives an outline of the idea presented above. The exception notSupportedElementException that the meta-policy in Example 2 raises triggers the obligation policy PolicyForInvalidDiffServPolicies. This obligation policy tries to install the non-supported DiffServ mechanisms to the relevant programmable devices.

Example 3: Policy rule for installing the non supported DiffServ mechanisms to the appropriate programmable routers.

```
inst oblig PolicyForInvalidDiffServPolicies {
subject /PolicyAgents/NetworkPMA;
on notSupportedElementException (Policy p[])
/* p is the set of policy objects that the meta policy in
Example 2 has returned as not valid*/
do
foreach pol in [p] {
  foreach action in [pol.actions]
     foreach t in [pol.target]
 { if (action.name = "addDiffServElement" and
t.notSupports( (DiffServ_Class) action.parameter))
        then  t.installMechanism
              ((DiffServ_Class) action.parameter)); }}}
```

The policy rule PolicyForInvalidDiffServPolicies receives the set of invalid network policies with the event notSupportedElementException, which the meta-policy in Example 2 raises. The pseudocode following the do statement could be implemented as a script policy action. This action finds the pairs < target_device, non-supported mechanism > and installs the non-supported mechanisms to the appropriate target devices with the execution of the method installMechanism .

As we mentioned in the beginning of this subsection, a policy is also not valid when the device does not have the resources to create the requested functional element. As an example, consider a DiffServ device that can only support a limited number of traffic classes. A policy that aims at implementing a new traffic class within the device will fail if the limit is exceeded. This case of policy validation can be implemented using CIM as a means to specify the necessary information about the **capabilities** and the **current state** of the device: the maximum number of traffic classes and the number of implemented classes respectively. However, unlike the previous case, we can not decide offline whether the policy is valid by checking the CIM representation of the device, since the decision depends on the state of the device. This means that the decision must be made at

the time the policy is to be enforced on the device. This in turn means that the conditions under which the policy is valid must be specified as constraints as part of the policy rule itself rather than as a meta-policy. The following is an example policy for specifying constraints on the current state of the device.

Example 4: Policy rule for creating a new traffic class when the target device supports a limited number of traffic classes

```
inst oblig /Policies/PolicyToAddTrafficClass {
subject /PolicyAgents/NetworkPMA;
target t = /Routers/CoreRouters;
on addTrafficClassRequest (classOfTrafficParams[]);
do t.createTrafficClass (( classOfTrafficParams [] );
when t.CIM_GetCurrentClassesOfTraffic() <
t.CIM_GetMaximumClassesOfTraffic(); }
```

The policy rule PolicyToAddTrafficClass will only add a new traffic class when current number of classes less than the maximum.

3.2 Validation of permitted values for configuration variables within the DiffServ device

Policies can also be used to set values for configuration variables for DiffServ elements. It this case it is necessary to check that requested variable values fall within permitted value ranges with respect to the device capabilities. We can distinguish between variables with static bounds and variables with dynamic bounds.

Variables with static bounds. As an example, the length of a DiffServ queue is a variable that has a specific upper bound. A policy rule that involves setting variable is only valid if it requests a value less than or equal to the upper bound. We can decide offline if the policy is valid, if we check the CIM representation of the device, which indicates the maximum length of DiffServ queues within the **capability** information of the device.

Variables with dynamic bounds. In this case, the bounds of the variable to be set are not static, but they depend on the state of the device. As an example, consider the bandwidth that the administrator wants to allocate to a class of traffic. The maximum bandwidth that can be allocated to this class of traffic cannot exceed the available free bandwidth that the device can offer, which is calculated as:

available_free_bw = total_output_bw - current_allocated_bw (for all traffic classes)

Assume that CIM can provide the necessary information about the **capabilities** and the **current state** of the device: the total output bandwidth and current bandwidth allocated to each traffic class. This cannot be checked offline as it depends on dynamic state so the conditions under which the policy is valid must be specified as constraints in the policy rules.

3.3 Conclusion

Using CIM as the model for representing network devices can providtwo types of management information:

i) Information about device's capabilities (e.g. type of mechanisms it supports, bounds on resources, bounds on specific objects' attributes)

ii) Information about the current state of the device (e.g. current resource allocation, current values of specific objects' attributes)

We can use this information to provide two main types of policy validation:

i) Validation with respect to device capabilities, which can be performed offline using meta-policies as a means to specify the constraints on the permitted DiffServ mechanisms, resources and bounds on objects' attributes. Only information about the capabilities of the device is needed and this can be extracted from CIM.

ii) Validation with respect to both device capabilities and current state of the device can only be performed at the time the policy is to be enforced. Information extracted from CIM is used to evaluate constraints within the policy rules which indicate the conditions which must be true for the policy to trigger and be enforced on the device.

4. Implementation

As we have discussed, for the purpose of validating policy against device and network capabilities, it is necessary to extract several parameters such as the number of QoS classes, the bandwidth allocated to each class, the QoS mechanisms the managed routers support, etc. CIM provides a generic model for representing DiffServ-enabled managed devices. In particular, the CIM Network sub-model (v2.7) defines the class QoSService, which is a subclass of the generic class NetworkService [4]. The QoSService is an aggregation of instances of the ConditioningService class, whose subclasses define the DiffServ mechanisms in a DiffServ router. These mechanisms are represented by the following CIM classes: ClassifierService, MeterService, MarkerService, DropperService, DropperThresholdService and PacketSchedulingService.

Our implementation uses the CIM network sub-model for representing the DiffServ elements that a router supports. In addition, since we need statistics related to DiffServ, e.g. number of implemented traffic classes, bandwidth allocated to each class of traffic, etc. we have designed a "DiffServ metrics" CIM sub-model which provides this information for the management system. The proposed "DiffServ metrics" CIM sub-model is presented in Figure 1.

The CIM class DiffServRouter is the abstraction of a DiffServ-enabled router. It derives from the ManagedElement class in the Core Model. The classes DSCPStatistics, NetworkInterfaceStatistics and GroupDSCPStatistics derive from the CIM class LogicalElement in the Core Model (this inheritance is not shown in Figure 1). The NetworkInterfaceStatistics class provides traffic statistics for every network interface card that belongs to the router. The DSCPStatistics class caters for statistics per implemented DSCP. A DSCPStatistics instance is associated to one or more NetworkInterfaceStatistics instances, as a particular DSCP may be implemented to more that one network interfaces on a single router. Finally, the GroupDSCPStatistics class is an agrregation of DSCPStatistics classes and provides statistics for a group of DSCPs which together define a class of service offered to the corresponding traffic aggregates. For example, the "Gold Group" in a router may be constructed from the DSCP offering the EF PHB [6], while the "Silver Group" may be constructed from the DSCPs offering the AF PHBs [7]. A DSCPStatistics instance may belong to more than one GroupDSCPStatistics instances.

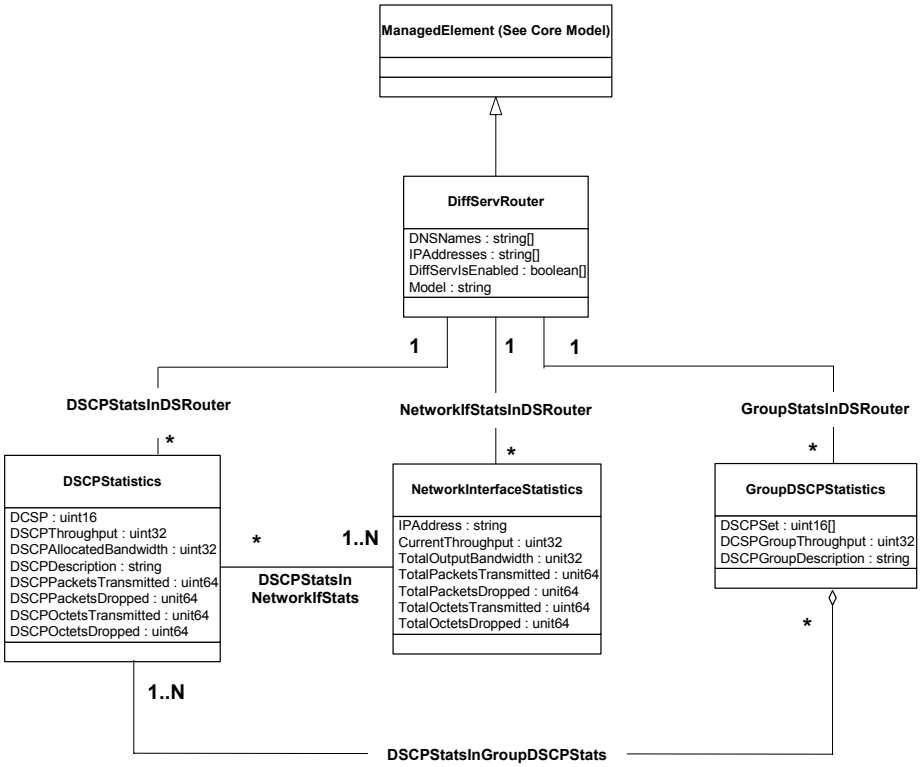

Figure 1. UML Diagram of the DiffServ-metrics CIM sub-model

The DiffServ part of the CIM network sub-model and the DiffServ metrics sub-model will both be implemented within the CIM Object Manager (CIMOM)provided by the WBEM Services project [8]. This project provides a Java implementation of a CIMOM and two interfaces for communicating with the CIMOM which are shown in Figure 2. These are:

- The javax.wbem.client interface, which clients use to communicate CIM classes to and from the CIMOM. This communication can be realised either through HTTP (where CIM operations are defined in XML) or through Java RMI.
- The javax.wbem.provider interface, which providers attached to the CIMOM use to communicate with the CIMOM. Providers are implemented as Java classes. Each provider is responsible for handling a number of CIM classes.

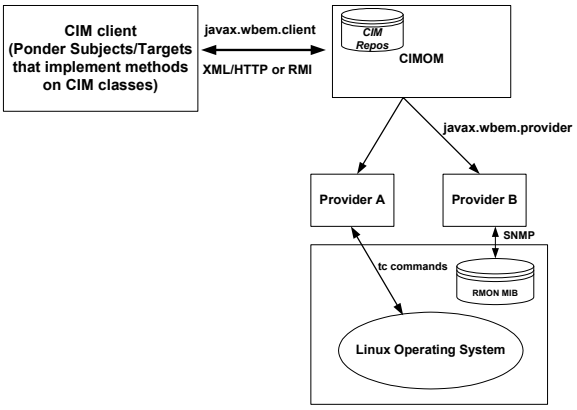

Figure 2. Architecture of the Policy Validation System

In our current implementation we have deployed:

- The DiffServ metrics sub-model within the CIMOM and a Provider that handles the DiffServ metrics sub-model's classes. Linux traffic control commands ("tc commands") are issued by the Provider to get variables from the Linux operating system. An alternative could be to use SNMP to get variables from a DiffServ RMON probe. However, there is no implementation of an RMON probe available for DiffServ on Linux, so we have to use "tc" commands to get DiffServ variables from the Linux kernel. Figure 3 shows the DiffServ metrics classes and instances that we have deployed within the CIMOM, using the Graphical User Interface CimNavigator [9].

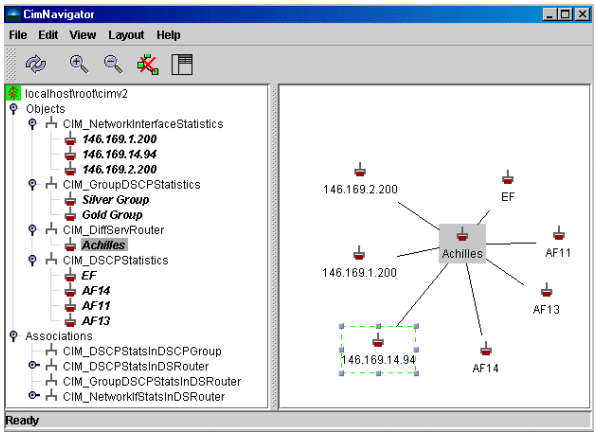

Figure 3. Graphical User Interface to the CIMOM

- Ponder PMAs and Target Policy Objects within the Ponder Toolkit [10] as CIM clients. This allows the Ponder components to retrieve variables that belong to CIM classes as a means to evaluate policy constraints or metapolicies that contain such variables, as shown in the example bellow.

Example 5: Policy rule for creating a new traffic class when the target device can provide the requested bandwidth for the new class

```
inst oblig /Policies/PolicyToAddTrafficClass {
subject /PolicyAgents/NetworkPMA;
target t = /Routers/Achilles;
on addTrafficClassRequest ( interface, dscp,
                            bandwidth );
do t.addTrafficClass ( interface, dscp, bandwidth);
when t.CIM_GetThroughput(interface) + bandwidth
      < t.CIM_GetTotalBandwidth(interface); }
```

The policy rule PolicyToAddTrafficClass is evaluated at run-time by the management agent NetworkPMA. When the constraint following the when clause evaluates to true, the target router is instructed to perform locally the addTrafficClassRequest operation. Figure 4 presents the result of the enforcement of the rule PolicyToAddTrafficClass on the target router /Routers/Achilles. The Network PMA receives the obligation event addTrafficClassRequest with parameters for the network interface on the target where the new class is to be added, the dscp associated with the traffic class and the bandwidth that will be assigned to this class. As we can see in Figure 4, the policy rule retrieves the relevant CIM variables from the target router and the policy constraint evaluates to true upon the request addTrafficClassRequest ("146.169.14.94", 63, 4096). This in turn means that the policy rule is

valid for the current network state and therefore is enforced on the network.

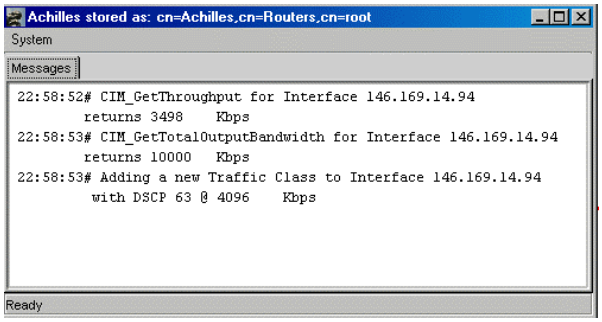

Figure 4. Result of enforcement of the policy validation rule PolicyToAddTrafficClass

Further implementation work is needed to deploy a Provider that can support the DiffServ classes with the Network CIM sub-model and to support the enforcement of metapolicies within the Ponder toolkit. We also aim to integrate our policy validation framework with the CISCO Information Model (CIM-CX).

5. Conclusions and Future Work

The above approach applies to individual network devices within a domain in which the DiffServ based policies apply. The situation becomes much more complex when you consider interactions between the policies related to end-to-end flows or different service level agreements (SLAs). Here you are trying to determine whether the introduction of a new SLA could potentially violate the policies relating to existing SLAs. In the simplest situation this could require determining whether the current network topology has the resources to support the new SLA which many current traffic Engineering systems can do. However, when the SLAs cater for dynamic allocation of resources based on time or application requests, this becomes more complex to do.

Another issue is that the end-to-end path may not be within the administrative domain of a single service provider. This requires interaction between service providers to exchange policy information, and current state of the network topology.

Acknowledgements

We gratefully acknowledge the support of the EPSRC for PolyNet research grant GR/R31409/01 as well as Cisco Systems for support on the Polyander project.

References

[1] Damianou, N., N. Dulay, E. Lupu and M. Sloman. *Ponder: A Language for Specifying Security and Management Policies for Distributed Systems. The Language Specification - Version 2.2*. Research Report DoC 2000/1, Imperial College of Science Technology and Medicine, Department of Computing, London, 3 April, 2000.

[2] Sloman, M. and E. Lupu. *Policy Specification for Programmable Networks.* Proc. of First International Working Conference on Active Networks (IWAN'99), Berlin, June 1999, ed. S. Covaci, LNCS, Springer Verlag, Berlin, June 1999, pp. 73-84.

[3] Damianou, N., N. Dulay, E. Lupu and M. Sloman. *The Ponder Policy Specification Language.* Workshop on Policies for Distributed Systems and Networks (Policy2001), HP Labs Bristol, 29-31 Jan 2001.

[4] Common Information Model (CIM) Version 2.7 Specification, Distributed Management Task Force.

[5] Bernet, Y., Smith, A., Blake, S. & Grossman, D., An Informal Management Model for DiffServ Routers, RFC 3290, May 2002

[6] Heinanen, J., Baker, F., Weiss, W. & Wroclawski, J., Assured Forwarding PHB Group, RFC 2597, September 1999.

[7] Jacobson, V., Nichols, K. & Poduri, K., An Expedited Forwarding PHB, RFC2598, September 1999.

[8] WBEM Services Project, http://wbemservices.sourceforge.net

[9] CimNavigator, Available from http://www.cimnavigator.com

[10] Damianou, N., Dulay, N., Lupu, E., Sloman, M., Tonouchi, T. *Tools for Domain-based Policy Management of Distributed Systems*. Proc. NOMS 2002: 8th Network Operations and Management Symposium, Florence, Italy, 15-19 Apr. 2002.