On-Demand Monitoring: a Monitoring Paradigm for Traffic Flows in Multi-Service Self-Managing Networks

The Dissertation Committee for Ranganai Chaparadza Certifies that this is the approved version of the following dissertation:

The On-Demand Monitoring: a Monitoring Paradigm for Traffic Flows in Multi-Service Self-Managing Networks

Committee:

Prof. Dr.-Ing. Ina Schieferdecker,
Supervisor (TUB, Germany)

Prof. Dr.-Eng. Symeon Papavassiliou
Second Advisor (NTUA, Greece)

Prof. Shiduan Cheng
Third Advisor (BUPT, China)

On-Demand Monitoring: a Monitoring Paradigm for Traffic Flows in Multi-Service Self-Managing Networks

By
Ranganai Chaparadza

Dissertation

Presented to the Faculty of Electrical Engineering and Computer Science (German: *Fakultät Elektrotechnik und Informatik: Fakultät IV*) of
The Technical University of Berlin (TUB)
in Partial Fulfilment
of the Requirements
for the Degree of

Doctor of Engineering (Dr.-Ing)

Technical University of Berlin (TUB), Berlin, Germany
April, 2011

Dedication

To my father, the late Ezekiel, and my mother Tabeth, my brothers and sisters, my whole family and friends.

Acknowledgements

To be Done……

On-Demand Monitoring Paradigm: a Monitoring Paradigm for Traffic Flows in Multi-Service Self-Managing Networks

Publication No._____

Ranganai Chaparadza, degree sought: Dr.-Ing.
Technical University of Berlin, Germany, 2011

Advisor: Prof. Dr.-Ing. Ina Schieferdecker (Technical University of Berlin, Germany)

Second Advisor:

Prof. Dr.-Eng. Symeon Papavassiliou (National Technical University of Athens, Greece)

Third Advisor:

Prof. Shiduan Cheng (The State Key Lab of Networking & Switching Technology of Beijing University of Posts and Telecommunications (BUPT), China)

Abstract:

The field of Traffic Monitoring in IT and Telecommunication Networks continues to experience a lot of research as traffic monitoring increasingly plays a very significant role in implementing emerging and future networks that self-configure and self-adapt to changes in their usage context as well as to challenging adverse conditions experienced by the network elements. Such networks fall under the umbrella of what are called self-managing networks. With the advent of the emerging self-managing networks and their evolution in the future, there is a need for researching the key design and operational principles for traffic monitoring components and platforms that are suitable for the emerging, multi-service self-managing networks, given the characteristics of such networks discused below.

A self-managing network is a network whose network elements (see definition of *network element*$_{re\text{-}defined}$) and Network Management Systems (NMSs) work co-operatively to perform what has come to be the so-called self-* operations such as self-configuration, self-diagnosing, self-healing/self-repairing, self-optimization, etc., with the aim of eliminating or drastically reducing human intervention in some of the complex aspects of manual and error prone, or daunting tasks of network management. In the past, "self-management" as it is called, was achieved through some automation techniques such as scripting, yet more advanced self-management is difficult to achieve through only scripting techniques. *Autonomicity*—realized through control-loops and feedback mechanisms and processes operating within individual network elements and the network as a whole, as well as the information or knowledge flow used to drive individual control-loops, is now well understood to be an enabler for advanced and enriched self-manageability of network elements and networks.

In contrast to today's non self-managing networks, self-managing networks are expected to require more computing resources for the computation of decisions taken by an individual network element and by the network as a whole under some situations such as adverse conditions or context changes, based on huge and diverse data or information sets collected by monitoring components. Also, in self-managing networks, depending on the dynamics of the network, a lot more information (or knowledge) exchange than in the case of non self-managing networks may be required to flow between the network elements e.g. routers, hosts, switches. Also, information (or knowledge) must flow between the elements and special types of components of the network that aggregate global network state information, compute and perform more sophisticated decisions for the network and, communicate control information to the fundamental network elements in order to force the network to achieve some desired goal such as global self-adaptation behaviour of the whole network. Also, in self-managing networks, more information (or knowledge) storage resources are required than in non self-managing networks, for the storage of the network state information including historical network state data. No doubt, these networks will require intelligent and opportunistic use and sharing of resources available at individual network element as well as in the network as a whole. Another important characteristic of self-managing networks is that of requirement for flexibilibity to support context-driven (re)-configuration and self-adaptation of the network tasks. One way of guaranteeing intelligent and opportunistc use and sharing of resources of the network, as well as context-driven (re)-configuration and self-adaptation of network tasks, is to design functions and components of the network elements and the network such that their operational principles allow functions, including monitoring functions, to be invoked on-demand by automated tasks that drive a self-managing network.

The network monitoring paradigms of today such as the ones supported by frameworks such as SNMP, Netflow, IPFIX, and other types of active or passive monitoring paradigms were not designed for the emerging, highly complex, dynamically "resource and flexibility demanding" multi-service self-managing networks. This is because, the set of monitoring functions, their invocations and who invokes them (whether an automated task or a human) on a monitoring component or system e.g. a router supporting the above mentioned paradigms, are normally pre-defined at design stage or when the monitoring component is installed. This means that with the current approach to designing and operating monitoring components and their associated functions, there is virtually no support for purpose driven on-demand creation and termination of monitoring tasks by distributed automated tasks of the network. In order to support intelligent and opportunistic use and sharing of resources of the network, distributed automated tasks that drive a self-managing network aught to be able to flexibly locate monitoring components of desired capabilities and request the monitoring components (or systems) in the network for dynamically (re)-configurable and programmable monitoring services that can be tuned, paused and terminated depending on the monitoring needs of the requesting automated tasks. Intelligent use of resources involves checks by a monitoring component on whether resources are available to satisfy the requirements expressed in an arriving monitoring-request and perform admission

control on a monitoring request and associated requested monitoring-behavior accordingly. Opportunistic use of network resources requires that a monitoring component should free resources of the system whenever monitoring is temporarily or no longer required, such that resources freed can be used by tasks requiring resources other than the monitoring tasks.

In this dissertation, we present a traffic monitoring paradigm we are calling the ODM-Paradigm, a monitoring paradigm we consider and demonstrate its suitablilty for automated tasks of multi-service self-managing networks. The ODM-Paradigm takes into account the need to use network resources intelligently, and opportunistically throughout the network, as well as the need for monitoring components to self-describe their monitoring capabilities to the network to enable automated tasks in a self-managing network to locate and select a monitoring component of desired capabilities when a need arises, trigger monitoring functions and manage the execution of those functions, and free resources on the targeted component whenever monitoring is temporarily not required or no longer required by an automated task(s). The ODM-Paradigm is based on our research on design and operational principles of traffic monitoring components, which we call ODM-Principles, and are meant to address the problems mentioned above, regarding traffic monitoring in the emerging, complex and highly dynamic self-managing networks characterized by high resource demands and the requirement for context-driven (re)-configuration and self-adaptation of network tasks. We also provide a functional specification and prototype implementation and evaluation of an ODM-Capable Probe that serves as proof of concept for the ODM-Paradigm. Although the main focus of this dissertation is about design and operational principles for traffic monitoring components, the research work also introduced an architectural Reference Model for Autonomic Networking and Self-Management called the GANA (Generic Autonomic Network Architecture) Reference Model. The GANA Model is a *Hierarchical Autonomic Management and Control Architectural Framework*. The GANA Model is a blueprint that prescribes design principles for autonomic management and control of resources that are "generic" to be applicable in designing autonomic components for diverse implementation-oriented architectures and network environments. It is a hybrid model that enables combining *centralized and distributed decision-making based management and control of resources (which include monitoring functions and associated resources),* while pointing out the main limitations of decentralization and distributed decision-making in network management and control, and providing techniques for addressing *stability of control-loops* in such a framework. The GANA principles have been validated in diverse cases of the instantiation of the model, as discussed in this dissertation and supported by pointing to relevant publications.

Table of Contents

List of Figures

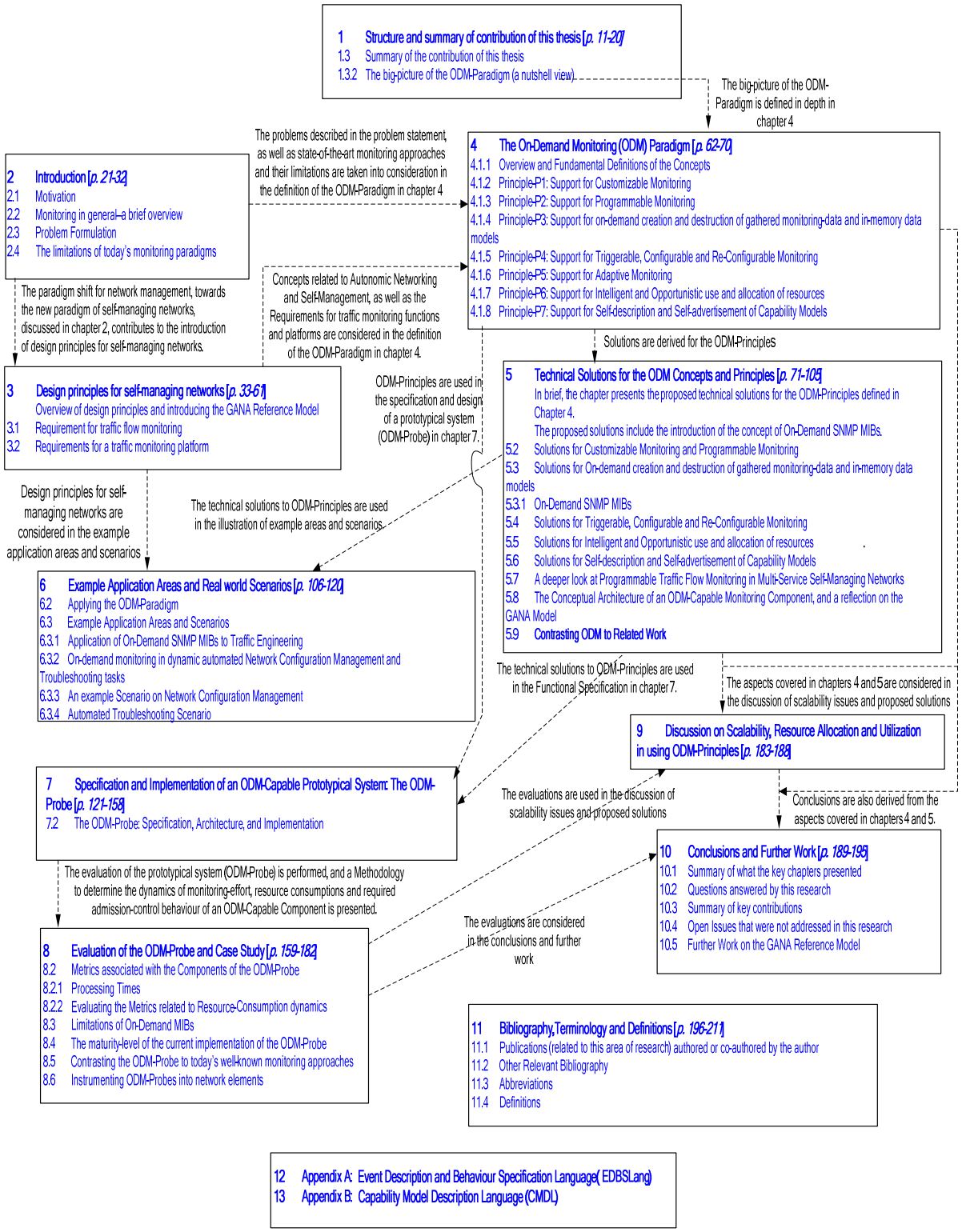
1 Structure and summary of contribution of this thesis

1.1 Overview

This chapter consists of two parts: **(1)** the *structure of this thesis* and, **(2)** *summary of the contribution of this thesis,* in which we present in brief, the concepts and technical solutions of the On-Demand Monitoring Paradigm, including what has been achived and validated by this research and the insight into further research.

1.2 Structure of this thesis

The diagram below summarizes the chapters and some key relationships between the core chapters. The diagram is augmented by more detailed textual descriptions of each chapter and its scope, given just after the diagram.


1 Structure and summary of contribution of this thesis [p. 11-20]
1.3 Summary of the contribution of this thesis
1.3.2 The big-picture of the ODM-Paradigm (a nutshell view)
The big-picture of the ODM-Paradigm is defined in depth in chapter 4
2 Introduction [p. 21-32]
2.1 Motivation
2.2 Monitoring in general–a brief overview
2.3 Problem Formulation
2.4 The limitations of today's monitoring paradigms
The problems described in the problem statement, as well as state-of-the-art monitoring approaches and their limitations are taken into consideration in the definition of the ODM-Paradigm in chapter 4
4 The On-Demand Monitoring (ODM) Paradigm [p. 62-70]
4.1.1 Overview and Fundamental Definitions of the Concepts
4.1.2 Principle-P1: Support for Customizable Monitoring
4.1.3 Principle-P2: Support for Programmable Monitoring
4.1.4 Principle-P3: Support for on-demand creation and destruction of gathered monitoring-data and in-memory data models
4.1.5 Principle-P4: Support for Triggerable, Configurable and Re-Configurable Monitoring
4.1.6 Principle-P5: Support for Adaptive Monitoring
4.1.7 Principle-P6: Support for Intelligent and Opportunistic use and allocation of resources
4.1.8 Principle-P7: Support for Self-description and Self-advertisement of Capability Models
The paradigm shift for network management, towards the new paradigm of self-managing networks, discussed in chapter 2, contributes to the introduction of design principles for self-managing networks.
Concepts related to Autonomic Networking and Self-Management, as well as the Requirements for traffic monitoring functions and platforms are considered in the definition of the ODM-Paradigm in chapter 4.
Solutions are derived for the ODM-Principles
3 Design principles for self-managing networks [p. 33-61]
Overview of design principles and introducing the GANA Reference Model
3.1 Requirement for traffic flow monitoring
3.2 Requirements for a traffic monitoring platform
ODM-Principles are used in the specification and design of a prototypical system (ODM-Probe) in chapter 7.
5 Technical Solutions for the ODM Concepts and Principles [p. 71-105]
In brief, the chapter presents the proposed technical solutions for the ODM-Principles defined in Chapter 4.
The proposed solutions include the introduction of the concept of On-Demand SNMP MIBs.
5.2 Solutions for Customizable Monitoring and Programmable Monitoring
5.3 Solutions for On-demand creation and destruction of gathered monitoring-data and in-memory data models
5.3.1 On-Demand SNMP MIBs
5.4 Solutions for Triggerable, Configurable and Re-Configurable Monitoring
5.5 Solutions for Intelligent and Opportunistic use and allocation of resources
5.6 Solutions for Self-description and Self-advertisement of Capability Models
5.7 A deeper look at Programmable Traffic Flow Monitoring in Multi-Service Self-Managing Networks
5.8 The Conceptual Architecture of an ODM-Capable Monitoring Component, and a reflection on the GANA Model
5.9 Contrasting ODM to Related Work
Design principles for self-managing networks are considered in the example application areas and scenarios
The technical solutions to ODM-Principles are used in the illustration of example areas and scenarios.
6 Example Application Areas and Real world Scenarios [p. 106-120]
6.2 Applying the ODM-Paradigm
6.3 Example Application Areas and Scenarios
6.3.1 Application of On-Demand SNMP MIBs to Traffic Engineering
6.3.2 On-demand monitoring in dynamic automated Network Configuration Management and Troubleshooting tasks
6.3.3 An example Scenario on Network Configuration Management
6.3.4 Automated Troubleshooting Scenario
The technical solutions to ODM-Principles are used in the Functional Specification in chapter 7.
The aspects covered in chapters 4 and 5 are considered in the discussion of scalability issues and proposed solutions
9 Discussion on Scalability, Resource Allocation and Utilization in using ODM-Principles [p. 183-188]
7 Specification and Implementation of an ODM-Capable Prototypical System: The ODM-Probe [p. 121-158]
7.2 The ODM-Probe: Specification, Architecture, and Implementation
The evaluations are used in the discussion of scalability issues and proposed solutions
Conclusions are also derived from the aspects covered in chapters 4 and 5.
10 Conclusions and Further Work [p. 189-195]
10.1 Summary of what the key chapters presented
10.2 Questions answered by this research
10.3 Summary of key contributions
10.4 Open Issues that were not addressed in this research
10.5 Further Work on the GANA Reference Model
The evaluation of the prototypical system (ODM-Probe) is performed, and a Methodology to determine the dynamics of monitoring-effort, resource consumptions and required admission-control behaviour of an ODM-Capable Component is presented.
8 Evaluation of the ODM-Probe and Case Study [p. 159-182]
8.2 Metrics associated with the Components of the ODM-Probe
8.2.1 Processing Times
8.2.2 Evaluating the Metrics related to Resource-Consumption dynamics
8.3 Limitations of On-Demand MIBs
8.4 The maturity-level of the current implementation of the ODM-Probe
8.5 Contrasting the ODM-Probe to today's well-known monitoring approaches
8.6 Instrumenting ODM-Probes into network elements
The evaluations are considered in the conclusions and further work
11 Bibliography, Terminology and Definitions [p. 196-211]
11.1 Publications (related to this area of research) authored or co-authored by the author
11.2 Other Relevant Bibliography
11.3 Abbreviations
11.4 Definitions
12 Appendix A: Event Description and Behaviour Specification Language (EDBSLang)
13 Appendix B: Capability Model Description Language (CMDL)

This dissertation is organized as follows:

Chapter 2 presents the ***motivation of this research***. The chapter discusses the shift in current network management paradigms and practices defined by the FCAPS [ITU-T Rec. M.3400] management framework, towards the definition, design and implementation of "self-managing" functions and processes that drive a network. It also presents the advent of the emerging self-managing networks and their evolution in the future, the characteristics of such networks, and a contrast between non self-managing networks and multi-service self-managing networks. The subject of monitoring is presented in brief and a tone is set, on the pursuit for key design and operational principles for traffic monitoring components and platforms that are suitable for the emerging, multi-service self-managing networks. In section 2.3 of the same chapter we present the ***Problem Formulation***, and the arguments as to why a traffic flow monitoring paradigm (i.e. what we are calling the ODM-Paradigm) for the emerging, multi-service self-managing networks is required. In section 2.4 of the same chapter we present ***the known limitations of today's monitoring paradigms*** known in state-of-the-art and explain why the paradigms are not suitable for multi-service self-managing networks and why monitoring paradigms suitable for multi-service self-managing networks are required.

Chapter 3 presents ***Design principles for self-managing networks***. The subject of ***Task Automation*** being at the very heart of self-managing nodes, devices and networks, is discussed. The subject of *Autonomicity*—realized through control-loops, feedback mechanisms and processes operating within individual network elements (see definition of *network element*$_{re\text{-}defined}$) and the network as a whole, as well as the information (or knowledge) flow used to drive individual control-loops is discussed. Also discussed, is how *autonomicity* is becoming well understood to be an enabler for advanced and enriched self-manageability of network elements and networks, beyond what can be achieved through automation techniques like scripting. Section 3.1 of the same chapter presents the ***requirement for traffic flow monitoring***. The role of monitoring in self-managing networks and the requirement for monitoring paradigms that are suitable for multi-service self-managing is discussed. Also presented is the subject of "how *Monitoring* plays a significant role in enabling the autonomicity of a single self-managing system or a network". The concept of an Automated Task and how it relies on a monitoring service or behaviour, to drive its behaviour and goal, is presented. Essentially, the definition of the relation between an Automated Task and Monitoring is established. This is useful towards capturing the design and operational principles sought for a traffic monitoring paradigm suitable for self-managing networks. Also presented in the same chapter, in section 3.2, is the ***requirement for a traffic monitoring platform***. As we analyze the requirement, we introduce the basic concepts and elementary building blocks of a traffic monitoring platform suitable for *traffic flow(s)* monitoring in multi-service self-managing networks. Concepts such as a *monitoring-request, monitoring-session, monitoring-behavior*, etc, are introduced and discussed in relation to design and operational principles sought for a traffic monitoring paradigm suitable for self-managing networks. Essentially, the key concepts of the ODM-Paradigm and their relations are defined.

Chapter 4 introduces the ***Fundamental theory, Concepts and Principles of the On-Demand Monitoring (ODM) Paradigm***. A definition of the ODM-Paradigm is provided along with the foundation upon which the definition is based. Seven design and operational principles for ODM-capable traffic flow monitoring components, simply called *ODM-Principles* (***Principle-P1 to Principle-P7***) are defined and their roles in addressing the outlined problem statement of this thesis are described. The technical solutions proposed for each of the ODM-Principles are then presented in Chapter 5.

Chapter 5 presents our ***Technical Solutions to realizing the ODM Concepts and Principles, and examination of Related Work*.** In this chapter we describe the technical solutions we introduced for the ODM-Concepts and ODM-Principles defined in Chapter 4, which are meant to address the issues outlined in the problem statement. The Conceptual Architecture of an ODM-Capable Monitoring Component or Subsystem is presented and is used later in Chapter 7 in the Functional Specification, Prototype Implementation and Evaluation of an ODM-Capable Probe that serves as proof of concept for the ODM-Paradigm and its associated ODM-Principles**.** We also discuss **Related Work** by contrasting our technical solutions proposed for the realization of the ODM-Paradigm as presented in the same chapter, in section 5.9, to other types of monitoring approaches that support some limited forms of On-Demand Monitoring. In the same chapter, a discussion on the summary of problems inherent in today's state of the art monitoring paradigms and frameworks, with respect to supporting the ODM concepts and principles, is provided.

Chapter 6 presents **example application areas and real-world scenarios on the application of the ODM-Paradigm**. We present selected application areas and scenarios. Three application areas are presented namely: *application of On-Demand MIBs to traffic engineering, dynamic network configuration, and automated troubleshooting scenarios*.

Chapter 7 provides ***Functional Specification and Implementation of an ODM-Capable Prototypical System—the ODM-Probe***, which serves as proof of concept for the ODM-Paradigm. The functional specification and implementation of the ODM-Capable Probe is based on the exploitation of some strengths exhibited by some well known traffic monitoring methods and tools of today, in particular, SNMP, protocol analyzers, and flow and packet trace analyzers. The ODM-Probe (a prototype probe) is contributed to the open-source research community. Therefore, the approach taken in designing and implementing the ODM-Capable probe is meant to help in creating a picture on how to make today's tools evolve towards supporting the ODM-Principles in full or partially.

Chapter 8 presents an ***Evaluation of the ODM-Probe and Case Study*.** An illustration of the interaction of the components of On-Demand Monitoring in a distributed environment is given and the ***metrics*** i.e. the ***processing times*** and ***resource consumption dynamics metrics*** associated with the Components of the ODM-Probe are derived and discussed. We also provide a contrast of the ODM-Probe to other types of closely related well-known monitoring approaches, as well as a brief discussion on the design principles employed by some of the well-known monitoring

systems or tools that we consider relevant to the further development of the ODM-Probe by the open-source community. The chapter also provides an insight on ***how to instrument ODM-Probes into network elements (see definition of network element$_{re\text{-}defined}$) and migrate today's traffic monitoring frameworks*** to supporting the ODM-Paradigm i.e. answers to how ODM concepts can be integrated into existing traffic monitoring methods and tools are provided.

Chapter 9 provides a discussion on ***Scalability Issues, Resource Allocation and Utilization Efficiency in using ODM-Principles***. In the chapter, we propose some solutions to the optimal setups that consist of example configurations which can be considered for allowing certain Automated Tasks in the network, such as management tasks, to configure and manage monitoring tasks on targeted points in the network, on behalf of some other Automated Tasks which then simply use the services offered by the monitoring tasks and need not manage the monitoring tasks.

Chapter 10 gives a ***Conclusion on the main contributions of this thesis,*** as well as some insights into ***Further Research Work*** necessary.

Bibiliography, Terminology and Definitions, and Appendicies are provided at the end.

1.3 Summary of the contribution of this thesis

1.3.1 Overview

The research work presented in this dissertation is based on finding the key design and operational principles for traffic monitoring components and platforms that are suitable for the emerging multi-service self-managing networks—expected to be resource demanding, namely the ODM-Paradigm principles defined and described in Chapter 4, and whose technical solutions are presented in Chapter 5. The research work presented here, is not about a call to the design of new traffic monitoring functions that are tailored to solving a particular traffic monitoring problem(s) and are complementary to the existing diverse traffic monitoring functions and architectures already covered by other research work, such as, say, packet capturing and processing functions on high speed links. Rather, we provide an answer to the question of how the traffic monitoring functions of monitoring tools and components or devices can be made to fit into the ODM-Paradigm by making them evolve towards supporting the ODM-Principles which address the problems described in the problem statement presented in section 2.3.

We introduce a traffic monitoring paradigm suitable for automated tasks that are meant to drive multi-service self-managing networks. Therefore, this dissertation presents a traffic monitoring paradigm we are calling the On-Demand Monitoring (ODM)-Paradigm, which we consider

suitable for automated tasks of multi-service self-managing networks. The ODM-Paradigm addresses the following issues:

- The key design and operational principles, which we call ODM-Principles for traffic monitoring components that enable the components to self-describe their traffic monitoring capabilities and their points of attachment to the network topology, so that distributed automated tasks can locate components of desired capabilities, request for some monitoring services or behaviours of interest and manage the execution and termination of the monitoring services (behaviours), thereby freeing resources on the target whenever monitoring is temporarily not required or no longer required by an automated task(s).

- How to support intelligent, and opportunistic use of resources such as processing power and memory in any system hosting an ODM-capable monitoring component as well as bandwidth consumption in the transfer of monitoring data. The ODM-Paradigm attempts to the answer the question: *How can traffic monitoring components e.g. probes be designed in such a way that the monitoring effort, once activated or triggered by monitoring requests, can either grow or shrink depending on the needs for monitoring (i.e. the required monitoring data, the required monitoring-behaviours) expressed in monitoring requests issued by distributed automated tasks?* This means that additional monitoring effort should be activated only when there is a need to do so (as driven by the diverse monitoring needs expressed in arriving monitoring requests). Intelligent and opportunistic use of resources is achievable when the monitoring functions of traffic monitoring components are designed following the ODM-Principles described in detail in **chapters 4 and 5**. This is because, the ODM-Principles include the freeing of resources of the system by a monitoring component whenever monitoring services are temporarily or no longer required—when indicated by automated tasks that triggered the monitoring-behaviour(s) in the first place, such that the freed system resources can be dynamically used for some tasks other than monitoring tasks.

In **Chapter 4** of this dissertation, we provide the extensible foundation upon which we define the ODM-Paradigm. Because monitoring is a broad area, this research focused on traffic monitoring because, a lot can be learnt i.e. deduced by some automated tasks of the network, from relevant knowledge gathered about the characteristics of some traffic flow(s) of interest flowing at a *selected point(s)* and *time* in the network, and such knowledge plays a significant role in the automation of the tasks meant to drive self-managing networks. What has been considered out of focus in our research on designing the ODM-Paradigm is the interaction between ODM-capable components. Rather, the main focus we present is the relation between an ODM-capable component as a platform for traffic monitoring, and the users—namely the automated tasks of the network that create monitoring-sessions on the component(s), thereby triggering monitoring functions and managing the execution and termination of those functions.

Therefore, in this dissertation, we provide answers to the following key questions:

- *What sort of design and operational requirements i.e. principles does the ODM-Paradigm impose on traffic monitoring components in order to support the concepts such as customizable and programmable monitoring described in Chapter 4, all of which facilitate intelligent and opportunistic use of resources in the network?* In **Chapter 4**, what we call ODM-Principles are defined, which form the basis for designing ODM-Capable (ODM-supporting) traffic monitoring components. *When talking about a traffic monitoring platform whose individual traffic monitoring components operate on ODM-Principles i.e. an ODM-Platform that is suitable for automated tasks of multi-service self-managing networks, how should the nature of such a platform look like?* In **Chapter 3-section 3.2** the basic concepts and elementary building blocks of such a platform are defined.

- *How should the conceptual architecture of an ODM-capable component i.e. a monitoring component that supports ODM-Principles and is an elementary building block of an ODM platform look like?; and How can ODM-capable components embedded in network systems be viewed as forming an ODM-platform?* A conceptual architecture of an **ODM-Capable Component (i.e. ODM-Supporting Component)** i.e. a traffic monitoring component that supports the ODM-Principles is presented in **chapter 5—section 5.8**.

- Customizability and management of monitoring services on a monitoring component requires concepts such as *a monitoring-session, session-ownership, creation and management of multiple monitoring-sessions.* This calls fort scalability issues be examined and addressed. *Scalability* issues are addressed in **Chapter 9**. In Chapter 3-section 3.2, two types of monitoring sessions are defined, namely *a "managed monitoring-session" and a "query-driven non-managed monitoring-session",* and the chapter discusses the sort of situations in which the types of monitoring-sessions are required.

- *What sort of a language(s) is required to enable automated tasks to specify the monitoring-behaviours that are meant to be executed by monitoring components i.e. the behaviours of managed monitoring-sessions dynamically requested for creation on a particular monitoring component by automated tasks of the network?* We introduce a **Composition Language for specifying monitoring-behaviours** that includes the possibility to *specify the reactions* of a monitoring-behaviour that can be *instantiated* on an ODM-capable component. The composition language, called the *Event Description and Behaviour Specification Language* (EDBSLang*)* is described in **chapter 5—section 5.2.1**.

- *What sort of a data modeling framework can be used for on-demand creation of in-memory data models of certain types of monitoring data gathered by a monitoring component which do not need to be stored or exported as packet*

traces or flow traces?; And the destruction of the data models when no longer required by automated tasks? The concept of *On-Demand SNMP MIBs* is introduced in answer to this question and was implemented and evaluated in the implementation of on ODM-capable prototypical system specified in **Chapter 7**.

- *How does the "Big-Picture" i.e. the full perspective of the ODM-Paradigm look like? What roles do components like network management components, special diagnostic traffic flow generator and sinker components, ODM-capable components, play in on-demand monitoring in a multi-service self-managing network?* In **Chapter 6**, we present the full perspective on the application of the ODM-Paradigm.

- *How about the creation of Capability Models of monitoring components that describe traffic monitoring capabilities of components and their points of attachment to the network topology, the dissemination and update of the Capability Models to the network and their use by automated tasks*? Automated tasks running the network should be able to locate components of desired capabilities, request for some monitoring services or behaviours of interest and manage the execution and termination of the monitoring services (behaviours). The subject of self-description and self-advertisement of traffic monitoring capability models of ODM-Capable components is addressed in **chapter 5—section 5.6,** and in **Appendix B**, which presents a *Capability Model Description Language* (CMDL) that can be used by ODM-capable monitoring components to self-describe their *Monitoring Capability Models* and self-publish them to the network.

- *How evolvable is the ODM-Paradigm, since a lot more work than addressed by this dissertation needs to be tackled as further research?*The conclusions and further work chapter (Chapter 10) provides answers to this question.

1.3.2 The big-picture of the ODM-Paradigm (a nutshell view)

In the full perspective of the ODM-Paradigm, the availability of ODM-capable components embedded in network elements (see definition of *network element*$_{re\text{-}defined}$) creates the notion of a network wide ODM platform required to develop or improve task automation for multi-service self-managing networks. Each ODM-capable is meant to support the key ODM-Principles that are listed below (detailed definitions and descriptions are provided in chapters 4 and 5):

- Support for Customizable Monitoring.
- Support for Triggerable, Configurable and Re-Configurable Monitoring.
- Support for Programmable Monitoring.
- Support for On-Demand Monitoring Data and their associated Data Models.

- Support for Adaptive Monitoring.
- Support for intelligent and opportunistic use and allocation of resources.
- Support for Self-description and Self-advertisement of a traffic monitoring Capability Model to the network by the monitoring component in order to allow automated tasks to locate, select a monitoring component of desired capabilities and point of attachment to the network, trigger monitoring functions and manage the execution and termination of those functions. The monitoring component must update the network of any changes to its monitoring Capability Model.

As part of the "Big-Picture" of the ODM-Paradigm, we introduce and present the following key achievements in the research carried out:

- A *Composition Language for specifying monitoring-behaviours* called EDBSLang (*Event Description and Behaviour Specification Language*) that provides the possibility to *specify the reactions* of a monitoring-behaviour that can be *instantiated* on an ODM-capable component. The specification of a monitoring-behaviour can be created programmatically by an automated task and uploaded into a target for execution or can be created by network monitoring engineers and uploaded (instrumented) into ODM-capable components of the network so that automated tasks of the network can simply select and request for the execution of a selected monitoring-behaviour-specification without creating and uploading the specifications by themselves. The EDBSLang language also allows for the specification of *actions* to be performed on the target by an associated monitoring-session created on a monitoring component by a session-owner; the specification of *events to be detected and the corresponding actions (i.e. "monitoring-policies") to be executed by the monitoring-session-behaviour*; the specification of *event-notification propagations* and *designation of notification recipients;* the specification of the *actions to be performed by notification recipients;* as well as other types of specifiable attributes of a monitoring-behaviour-specification defined by the EDBSLang language presented in Chapter 5.

- We capture the requirements for on-target run-able components (or functional entities) such as *Scripts* that can be invoked on a system hosting an ODM-capable component when indicated through monitoring-behaviour-specifications specified using the EDBSLang language. The targeted components (functional entities) are the kind that requires exporting monitoring data to a remote ODM-session-owner or designated remote entities. We define the appropriate interface of an ODM-capable component that enables data registration and export.

- We introduce the concept of *On-Demand SNMP MIBs* that enables an ODM-capable monitoring component to *support the creation of in-memory monitoring-data models required for the duration for which the monitoring-data is required by automated tasks,*

as well destroying those dynamic monitoring-data models when no longer required. This enables the intelligent use of system resources.

- We introduce a *Capability Model Description Language* (CMDL) that can be used by ODM-capable monitoring components to self-describe their *Monitoring Capability Models* and self-publish i.e. advertise them to the network. CMDL is a meta-language for describing capability models of ODM-capable components and devices i.e. *ODM supporting devices (routers, switches, hosts, signalling gateways, special monitoring probes that can be instrumented in a network, etc), diagnostic/test traffic flow generators and sinkers.* All these types of components and network devices may be required to participate in a monitoring objective spanning diverse components and devices i.e. monitoring points, as may be required by an automated task(s).

- We discuss the role of *active* and *passive* monitoring and the need for *Capability Models* self-described and published by monitoring components and devices, in on-demand monitoring. We also discuss the *dissemination* (export) of capability models by monitoring components and devices of the network into Capability Models Databases accessible to automated tasks of the self-managing network.

- We also present an idea of the building of *trees of monitoring-behaviour-specifications instantiable* or allowable on an ODM-capable component by the component itself through learning from the history of traffic monitoring on the component itself. The ODM-capable component learns from its own history about the previously executed monitoring-behaviour-specifications and builds a tree of monitoring-behaviour-specifications it already knows. The tree description can then be disseminated to the network so that ODM-triggering automated tasks can simply indicate from the tree, the assigned *identifier* of a monitoring-behaviour-specification that is instantiable on the component, without the need to upload specifications of monitoring-behaviour into the targeted ODM-capable componet, in order to conserve network bandwidth usage and avoid this overhead when initiating the monitoring behaviour. More information on this subject can be found in [Chaparadza07a] and in **chapters 5 and 7**.

- We provide a *Functional Specification and Prototype Implementation and Evaluation of an ODM-Capable Probe* that serves as proof of concept for the ODM-Paradigm and the associated ODM-Principles.

- We present a set of *example task automation applications* that demonstrate how the ODM-Paradigm can be used by automated tasks of multi-service self-managing networks.

- We provide a discussion on *Scalability Issues, Resource Allocation and Utilization Intelligence in using ODM-Principles*.

2 Introduction

2.1 Motivation and Overview

The field of network management is seeing a lot of new management paradigms and functions being introduced in both evolutionary and revolutionary ways. This is mainly due to the advent of Autonomic Computing [Autonomic_Computing][MAPE][Abbas03] [AT&T04] and Autonomic Networking [Strassner06][Mortier06][Autonomic_Networking]. In short, *autonomic systems engineering or autonomics* involves designing systems that exhibit so-called *self-** properties e.g. *self-configuring, self-healing etc*, in short, *self-managing* properties, in order to address systems and network management complexities that are otherwise difficult or daunting to be handled by humans i.e. systems and network operations personnel. network management paradigms and practices as currently defined by the FCAPS [ITU-T Rec. M.3400] management framework, are moving towards the definition, design and implementation of "self-managing" functions and processes that drive a network, with the aim of eliminating or drastically reducing human intervention in some of the complex aspects, manual and error prone or daunting tasks of network management. Self-management as it is called—is meant to primarily reduce operational costs for the increasingly complex IT and Telecommunication networks [AT&T04] [MAPE] [Intel].

Traditionally, network management is divided into five so called management functional planes that are considered as sub-planes of the general management plane of the network, namely: **F**ault-management functional plane, **C**onfiguration-management functional plane, **A**ccounting-management functional plane, **P**erformance-management functional plane, and **S**ecurity-management functional plane. This is known as the FCAPS management Framework [ITU-T Rec. M.3400]. For example, the Fault-management plane of the FCAPS framework deals with the following functions: *fault-detection, fault-diagnosis (localization or isolation), and fault-removal* [Steinder04][ANA_delivD3.5][Avizienis04].

In contrast to today's network management practices applied to non self-managing networks, self-management of networks requires that the so-called traditional network management functions, defined by the FCAPS management framework for Fault-management, Configuration-management, Accounting-management, Performance-management and Security-management, as well as the fundamental network functions such as routing, forwarding, discovery, monitoring, together with automated management functions such as fault-detection and fault-removal, are all made to automatically feed each other with information (knowledge) such as goals and events, in order to effect feedback processes among the diverse functions. These feedback processes enable

reactions of various functions in the network and/or individual network elements (see definition of *network element*$_{re\text{-}defined}$) in order to achieve and maintain well defined network goals. Therefore, *autonomicity* [MAPE][Strassner06] realized through control-loops and feed-back mechanisms and processes operating within individual network elements and the network as a whole, as well as the information (or knowledge) flow used to drive individual control-loops, becomes an enabler for advanced and enriched self-manageability of network elements and networks. As such, even the FCAPS functions become diffused within *network element*$_{re\text{-}defined}$ architectures, apart from being part of an overall network architecture—whereby traditionally, a distinct management plane is engineered separately from the other functional planes of the network. The diffusion of the FCAPS functions into the *network element*$_{re\text{-}defined}$ architectures enables us to reason and talk of "*network element*$_{re\text{-}defined}$ *-intrinsic and network-intrinsic management and decision-making-processes*".

All the FCAPS related management functions as well as emerging management functions, together with the components implementing the functions thereof will continue to see a shift in design principles even in the far future, in order to meet *design for autonomicity* requirements. In **Chapter 3**, we describe the concepts associated with autonomic systems design and paint a picture on design for autonomicity and self-management requirements.

With the advent of the emerging self-managing networks and their evolution in the future, there is a need to capture and define the key design and operational principles for traffic monitoring components and platforms that are suitable for the emerging, multi-service self-managing networks. The principles sought by this research, for traffic monitoring components and platforms are defined by what we are calling the ODM-Paradigm presented and validated by this dissertation.

Therefore, in this dissertation we aim at introducing a traffic flow monitoring paradigm (ODM-Paradigm) that is suitable for the multi-service self-managing networks. Such networks are characterized by requirement for context-driven (re)-configuration and self-adaptation of network tasks, and are resource demanding networks—in terms of computation resources for state-analysis and self-adaptation, as well as resources for information (or knowledge) sharing and storage. Therefore, we say a few words on the characteristics that define multi-service self-managing networks in the next paragraph, starting with the notion of multi-service nature of a network as understood in literature.

A ***multi-service network*** is a network consisting of a number of connected networks that use different protocols on the same layer of either the TCP/IP model or the OSI model, and requiring translation (gateways) in order to seamlessly provide end-to-end services across the differing networks [Day08]. A multi-service network can also be characterized by multiple services in terms of the type of user information or user data transferred end-to-end e.g. voice, data or video

transport services i.e. the case of converged networks [IEC]. Refer to sources such as [Day08] that describe the TCP/IP model and the OSI model, including the difference between the two models. A ***multi-service network*** may also be a network characterized by a number of schemes (e.g. configuration profiles or dynamically enforceable algorithmic behaviours) that can be employed by a single protocol or by a combination of multiple protocols such as diverse routing or transport/forwarding schemes that can be used in different situations (contexts) in which it pays to use a particular scheme. The common denominator to all the above characteristics that can be used to define the "multi-service" aspects of a network is the fact that different protocols of a multi-service network are identifiable by applying traffic or protocol analysis via appropriate **traffic filtering** at appropriate vantage monitoring points in the network**.** The schemes employed may also be inferred through monitoring the behaviours of protocols of interest (again traffic filtering plays a role). Refer to sources like: (chapter 8 of [Varghese05]) [Ethereal] [tcpdump] [CoralReef] [libpcap] [Watson03] [Bos04] for traffic filter types and languages. In this dissertation, we talk of a **multi-service self-managing network** when the multi-service network implements self-managing functions across all the FCAPS management functional planes for **F**ault-management, **C**onfiguration-management, **A**ccounting-management, **P**erformance-management and **S**ecurity-management [ITU-T Rec. M.3400]. In Chapter 3 we discuss design principles for self-managing networks along with the desired characteristics for such networks.

2.2 Monitoring in general—a brief overview

Monitoring related concepts and aspects are considered widely ranging, and include the following:

- Functions for internal monitoring of node-local entities, such as CPU and memory utilization. Refer to tools like *nmon* [nmon], SNMP based approaches [SNMP] or to sources such as [Patarin00].
- Functions for traffic monitoring i.e. capturing and filtering. We refer the reader to monitoring libraries or information available from tools and sources such as: [Ethereal] [CoralReef] [libpcap] [PassiveActiveMonit] [MirrorAndSpan] [Arlos05] [Hruby05].
- Functions for creating, managing, disseminating and making use of monitoring data for diverse purposes such as accounting, capacity planning, traffic matrix computations, etc. Refer to sources such as [NetFlow] [SNMP].
- Functions for detecting events related to traffic or protocol-behaviour occurring internally and externally to the system via traffic capturing and analysis techniques (refer to sources such as [Varghese05]). Protocol analysers such as Ethereal [Ethereal] and tcpdump [tcpdump] are widely used for traffic analysis and protocol-decoding for diagnostic, testing, or troubleshooting purposes.
- Design-related aspects of concern for monitoring functions. This includes aspects such as modular design of monitoring-functions i.e. functions-splitting in general.

- Assignment of monitoring functions (by design) to individual monitoring components, thereby making the components *service-offering* components (i.e. the notion of monitoring services).
- Placement and distribution of monitoring components such as sensors i.e. probes in a network. The placement may be done on the basis of capabilities of individual monitoring components or the need to place some monitoring components in points considered strategic for the observation of network or traffic behaviour.

Literature shows that a lot of research work has been done, and still continues, in addressing all these monitoring related topics above. Yet, it is commonly understood that there are still a number of limitations suffered by current monitoring paradigms and frameworks implemented by standalone tools and/or system-embedded monitoring components. Such limitations include: *Limited monitoring data-sets; Limited flexibilities in traffic monitoring components or subsystems; Limited customizability of monitoring-behaviors; Limited support for programmability of monitoring services at run-time; Limited possibilities in deriving some data-sets from raw monitoring-data made available by monitoring components and tools*. These limitations, which are described in more detail in Chapter2-section 2.4, emanate from the fact that often, research on monitoring in general, has always targeted a specific problem at a time and for a specific requirement and context for monitoring.

2.3 Problem Formulation

In contrast to today's non self-managing networks, self-managing networks are expected to require more computing resources for the computation of decisions taken by an individual node and by the network as a whole under some situations such as adverse conditions or context changes, from huge and diverse data or information sets collected by monitoring components. Also, in self-managing networks, depending on the dynamics of the network, a lot more information (or knowledge) exchange than in the case of non self-managing networks may be required among the network elements (see definition of *network element*$_{re\text{-}defined}$) e.g. routers, hosts, switches, as well as between the nodes and special components of the network that aggregate global network state information, compute and perform more sophisticated decisions for the network and, communicate control information to the normal network nodes in order to force the network to achieve some desired goal such as global self-adaptation behaviour of the whole network. Also, in self-managing networks, more information (or knowledge) storage resources are required than in non self-managing networks, for the storage of the network state information including historical network state data. No doubt, these networks will require intelligent and opportunistic use and sharing of resources available at individual nodes as well as in the network as a whole. Intelligent use of resources involves checks by a monitoring component on whether resources are available to satisfy the requirements expressed in arriving monitoring request and perform admission control on monitoring requests and requested

monitoring-behavior accordingly. Supporting opportunistic use of network resources requires that a monitoring component should free resources of the system whenever monitoring is temporarily or no longer required, such that resources freed can be used for tasks requiring resources other than monitoring tasks.

Another important characteristic of self-managing networks is that of requirement for flexibilibity to support context-driven (re)-configuration and self-adaptation of network tasks. One way of guaranteeing intelligent and opportunistc use and sharing of resources of the network, as well as context-driven (re)-configuration and self-adaptation of network tasks, is to design functions and components of the network elements (see definition of *network element*$_{re\text{-}defined}$) and the network as a whole, such that their operational principles allow functions, including monitoring functions, to be invoked on-demand by automated tasks that drive a self-managing network.

The network monitoring paradigms of today such as those based on frameworks like SNMP [SNMP], Netflow [NetFlow], IPFIX [IPFIX], and other active or passive monitoring paradigms were not designed for the emerging, highly complex, dynamically resource demanding multi-service self-managing networks—which require supporting context-driven (re)-configuration and self-adaptation of the network tasks. This is because, the set of monitoring functions, their invocations and who invokes them (whether an automated task or a human) on a monitoring component or system e.g. a router supporting the above mentioned paradigms, are normally pre-defined at design stage or when the monitoring component is installed. This means that with the current approach to designing and operating monitoring components and their associated functions, there is virtually no support for purpose driven on-demand creation and termination of monitoring tasks by distributed automated tasks of the network. In order to support intelligent and opportunistic use and sharing of resources of the network, distributed automated tasks that drive a self-managing network aught to flexibly be able to locate monitoring components of desired capabilities and request the monitoring components or systems in the network for dynamically (re)-configurable and programmable monitoring services that can be tuned, paused and terminated depending on the monitoring needs of the requesting automated tasks.

In this dissertation, we present a traffic monitoring paradigm we are calling the ODM-Paradigm, a traffic flow monitoring paradigm we demonstrate as suitable for automated tasks of multi-service self-managing networks. The ODM-Paradigm takes into account the need to use network resources intelligently and opportunistically throughout the network, as well as the need for monitoring components to self-describe their monitoring capabilities to the network to enable automated tasks in a self-managing network to locate and select a monitoring component of desired capabilities when a need arises, trigger monitoring functions and manage the execution of those functions, and free resources on the targeted component whenever monitoring is temporarily not required or no longer required by an automated task(s). The ODM-Paradigm is based on our research on design and operational principles of traffic monitoring components,

which we call ODM-Principles, and are meant to address the problems mentioned above, regarding traffic monitoring in the emerging, complex and highly dynamic self-managing networks, which are characterized by high resource demands and the requirement for context-driven (re)-configuration and self-adaptation of network tasks. A functional specification and prototype implementation and evaluation of an ODM-Capable Probe that serves as proof of concept for the ODM-Paradigm is presented in chapters 7 and 8.

A summary of the contribution of this dissertation is given in detail in Chapter 1-section 1.3.

2.4 The limitations of today's monitoring paradigms

In this Chapter we present the limitations of today's monitoring paradigms and explain why the paradigms are not suitable for multi-service self-managing networks and why monitoring paradigms suitable for multi-service self-managing networks are required. After first presenting the design principles for self-managing network elements (see definition of a *network element*$_{re-defined}$) and networks in Chapter 3, in Chapter 3—section 3.1 and Chapter 3—section 3.2 we further present the arguments on why the requirement for an appropriate traffic monitoring paradigm and platform suitable for multi-service self-managing networks as we examine the characteristics of self-managing networks in relation to monitoring.

What is hampering advancements in task automation or the evolution of task automation applications meant to drive multi-service self-managing networks are the limitations of today's traffic monitoring paradigms and tools, described below. This issue of limitations of today's monitoring paradigms, frameworks and tools is revisited in chapter 5—**section 5.9**, after the definition of the ODM-Paradigm, where we then closely examine some well known traffic monitoring paradigms of today against the requirement for supporting the on-demand monitoring principles defined by the ODM-Paradigm. Therefore, in chapter 5—**section 5.9**, we further examine the limitations in more detail. The following sub-sections describe the main limitations as known in state-of-the-art.

2.4.1 Limited monitoring data-sets

The ***limited monitoring data sets*** made available to distributed automated tasks by today's network monitoring frameworks and tools. The monitoring data available from a monitoring point such as a router, a switch or a probe e.g. a RMON [RFC 3577] (*Remote Monitoring*) probe, is *limited* to the implemented capabilities of the monitoring function(s). A router, a switch or a RMON probe collects limited monitoring data (mostly coarse-grained) and makes the data available to other systems through methods such as SNMP [SNMP], NetFlow [NetFlow] or IPFIX [IPFIX]. The limited *monitoring data* collected by a monitoring component or device that operates at a given point in the network either as a normal host or as a intermediary *network*

element$_{re\text{-}defined}$ such as a router or a switch, is carefully defined prior to the implementation of the monitoring function(s) of the component or device, in order to focus on collecting only the kind of monitoring data that is believed to be "*useful i.e. necessary*" or may be required by a Network Management System (*NMS*). The argument behind this practice has always been that, this approach helps mitigate the problem of resources i.e. processing power requirements, storage capacity requirement and bandwidth consumption during the transfer of monitoring data. Yet, the developers of automated tasks such as protocols, special customized task automation applications such as automated network or service troubleshooting or debugging applications require that monitoring components and devices offer rich monitoring data sets, including fine grained monitoring such as captured test/diagnostic packets, packet traces and flow traces, etc, useful for driving the execution logic of the automated task. The usual case in today's monitoring paradigms, frameworks and tools, is that the set of monitoring tasks and their invocations on a monitoring component or device is pre-defined, with virtually no support for on-demand creation and termination of monitoring tasks by diverse automated tasks running in the network. **Why is the need for more diverse monitoring data sets important to consider for self-managing networks?** In self-managing networks we are faced with the following problem: the need for having more diverse monitoring data sets as necessitated by the need to improve task automation—resulting in enabling the development of more and more automated tasks for driving the self-manageability aspects of the network. For self-managing networks, there is a need to break today's practice of: **(1)** limiting monitoring data sets made available by a monitoring device; **(2)** inflexibly fixing the data generation-behaviour of the component with little or virtually no external control by automated tasks of the network. Developing monitoring components, tools, frameworks and paradigms according to specific monitoring needs or problems *i.e. specificity* of monitoring tools, platforms, paradigms and frameworks, has always been the practice. In [Hruby05], the authors discuss the issue of specificity of some of today's well established monitoring solutions and why the practice has always been about developing tailor-made solutions that address specific problems. [Bos04] also provides some few examples of monitoring applications and their associated monitoring needs. A different practice and understanding should be adopted—that monitoring components can be designed in such a way that the monitoring data sets and the generation of the data sets should be driven and controlled by the automated tasks of the network. This call for flexibility in design of monitoring components is discussed further in the next section on the problem of limited flexibilities of today's monitoring components.

2.4.2 Limited flexibilities in monitoring components or subsystems

The ***limited flexibilities in monitoring components or subsystems*** meant to operate in a host or in special devices such as routers, probes, switches, gateways, etc. The state of the art in traffic monitoring shows that each monitoring paradigm, framework or tool is meant to address a set of problems and is tailored to some specific purposes of monitoring. Some tools and traffic

monitoring paradigms employed by today's well known frameworks and protocols such as SNMP [SNMP], NetFlow [NetFlow], IPFIX [IPFIX] etc are designed for long-term monitoring of the network, for the needs of reporting on resource utilization, capacity planning, billing, studies of network behaviour through gathering and analyzing traffic traces (post data collection analysis), and network intrusion detection, etc, with virtually no support for on-demand creation and termination of monitoring tasks on monitoring components by diverse distributed automated tasks—one of the key principles required for self-managing networks that is being put forward by the ODM-Paradigm. Likewise, some tools and platforms like NIMI [Paxson98], Pandora [Patarin00], NetraMet [Brownlee97], etc, are simply meant for measurements (e.g. delay, round-trip time, jitter, throughput etc). Some tools such as protocol analyzers e.g. Ethereal [Ethereal], tcpdump [tcpdump], etc, are meant for non-automated troubleshooting carried out by a human. Some of the monitoring devices, probes or components such as SNMP agents are designed in such a way that they operate on the assumption that the monitoring effort or depth offered by the implemented monitoring functions must always be active, whether needed or not, without offering flexibility that allows automated tasks to actually trigger and control the monitoring effort or depth on the monitoring device or component. **Why is the issue of flexibility in monitoring services important to consider for self-managing networks?** In self-managing networks we are faced with the following problem: the need to have monitoring components designed in such a way that the dynamics of automated tasks have influence on the monitoring data sets, the diversity, the time and space (points in the network) at which monitoring data sets are generated by monitoring components—meaning that a monitoring component should be designed in such a way that it has the capacity to generate as much of diverse monitoring data sets as is required by automated tasks of the network.

2.4.3 Limited customizability of monitoring-behaviours

The ***limited customizability of monitoring-behaviours*** *(services)* at a monitoring component meant to operate in a host or in special devices such as routers, probes, switches, gateways, etc. All of the above mentioned monitoring tools and platforms have never been designed in such a way that distributed automated tasks can identify the capabilities of the monitoring tool (component) and create the notion of *monitoring-sessions* of desired behaviours and manage the sessions at run-time on a target tool or component, thereby freeing resources on the monitoring component or device when some monitoring function is no longer or temporarily not required—another key principle being put forward by the ODM-Paradigm. **Why is the issue of customizability of monitoring services important to consider for self-managing networks?** In self-managing networks we are faced with the following problems: Automated tasks should be given the ability to create and manage monitoring-sessions as described later in Chapter 3-section 3.2; the ability to identify, via the use of identifiers, the binding of the automated task to a particular monitoring-session created on a monitoring component—thereby enabling automated tasks to trigger and control monitoring services on-demand.

2.4.4 Limited support for programmability of monitoring services at run-time

Here we discuss the ***limited support for programmability of monitoring services at run-time*** on a monitoring component(s) according to the needs of a specific automated task. In sources like [Cranor02] some discussions on some limitations regarding programmability for network monitoring tasks are provided. **Why is the issue of programmability of monitoring services important to consider for self-managing networks?** In self-managing networks we are faced with the following problem: automated tasks should be able to specify the monitoring-behaviour-specification for the monitoring-behaviour required on a targeted monitoring component via the use of an appropriate monitoring-behaviour-specification language. If the monitoring-behaviour-specifications are authored by humans, still the automated tasks of the network should be able to select and transfer or indicate from the monitoring component's cached specifications repository a desired monitoring-behaviour-specification for execution on the target. This also includes the possibility for monitoring-query triggered monitoring on a targeted monitoring component via the use of an appropriate query language as described later in Chapter 3-section 3.2.

The problem of programmable monitoring has been researched and is still undergoing research. Today, programmable monitoring is achieved through a number of approaches. One approach is to write scripts that are then uploaded and executed on a monitoring target. The ScriptMIB [RFC 3165], for example, allows scripts to be uploaded by a remote system and executed on the monitoring point e.g. a router that implements the MIB and has an appropriate execution engine that understands the scripting language used. One of the problems of the scripting approach is that each script execution usually starts a separate process and this may pose a problem of resource demands, context switching overhead, and scalability on the execution target. It may also imply that a number of script execution engines may be required for supporting a number of scripting languages. In some cases, the scripting approach may be vendor specific, which may mean that a lot of effort is required in order to engineer programmable monitoring in a multi-vendor environment.

Another approach to implementing programmable monitoring is to implement the monitoring functions of a monitoring component in such a way that a desirable monitoring-behaviour can be triggered through the use of parameters composed from the set of parameters understood by the monitoring functions or a by monitoring tool(s) implemented on the target. DiMAPI [Trimintzios06] and MAPI [Polychronakis04], CAIDA tools [CAIDA], and similar approaches are examples that are based on this approach. The more we have such parameters, the greater the call for a composition language. The advantage of this approach is that it comes closer to the creation of a unified universal composition language for specifying monitoring-behaviours that can be instantiated on monitoring points, provided that the elements of the language, their syntax and semantics, are standardized across a number of monitoring devices or components from different vendors. The other advantage is that a unified composition language, instead of the scripting approach, calls for certain design principles of the monitoring components and their

functions which ensure that resources are used intelligently on the monitoring target. This is done by ensuring that the design allows for processing different monitoring requests having varying behaviour compositions, and making the requests to partially or wholly share an already triggered monitoring-behaviour(s) on the target, without the need to start unnecessary processes. Suppose that a standardized unified universal composition language for specifying monitoring-behaviours for traffic flow exists, and more importantly for programmable monitoring in multi-service networks, then the designers of monitoring components and functions would only need to focus on ensuring that the implemented monitoring functions of the device or software component in question, support the triggering of a monitoring-behaviour specified using the composition language. Later, in chapter 5—section 5.2.1, we present a composition language we developed for use in specifying monitoring-behaviour.

2.4.5 Limited possibilities in deriving data-sets from raw monitoring data

The *limited possibilities in deriving some necessary data e.g. fine-grained monitoring data from the raw monitoring data sets* gathered or captured by some monitoring component or software tool, which may also be limited as explained above, in the first bullet point. In sources like [Trimintzios06], some discussions on some limitations regarding the gathering of fine-grained monitoring data, are provided. Not only is the monitoring data gathered by a monitoring component or device of today limited but, the monitoring function(s) of a monitoring component or device are in most cases not flexible to allow for the derivation and computation of some data (e.g. fine-grained data) from the raw data the function(s) gather, as may be required by a remote entity intending to describe to the monitoring component what needs to be computed from the raw data and communicated to the remote entity or to some recipients(s) designated by the entity (remote or local). An automated troubleshooting application running in a Network Operations Centre (NOC) for example, may require triggering the capture of flow-specific traffic (or packet) at some selected monitoring component(s) or devices running at some point(s) in the network, and demand that each targeted monitoring component or device running at point in the network, exports fine-grained data such as a single captured diagnostic packet(or the value of a field in the packet) of the specified traffic flow to the remote application for analysis and reasoning. In troubleshooting, a consolidated single view of say a VoIP call that traverses different interfaces and protocols is a quick way to isolate faults. Also, in automated troubleshooting, the automated trouble-shooter may want to query selected monitoring points in the network for the presence or absence of some traffic of interest and demand notifications (or actions). For example, it may want to formulate such a request(s) to a set of selected monitoring components or systems: “if you capture RSVP traffic/packet destined for *system Q* anytime from now, notify me”, a phenomenon we may call *traffic or packet presence detection*. The traffic requested for presence detection at a particular point i.e. traffic under tracking, may be some diagnostic traffic generated by the automated tasks. The presence or absence of some traffic of interest at some point and interfaces can indicate mis-configuration problems or malfunctioning of a network device(s).

The picture about task automation in self-managing networks is that an automated task such as a special task automation application e.g. an automated troubleshooting application may require triggering the generation of test/diagnostic traffic having known characteristics from a certain point(s) in the in the network and destined for a certain point(s) in the network and *demand* that some systems e.g. probes in strategic monitoring points in the network along the path perform the following: capture the diagnostic traffic, perform some computations on the traffic(e.g. rate of flow, packet arrival-rate), detect some changes in the characteristics of the diagnostic traffic, detect events(derived from the captured diagnostic traffic) and send notifications to the task automation application or to some other recipients(s) designated by the task application. This enables the automated task to learn the behaviour or state of the network, the behaviour of running application(s) or service(s) etc and use such knowledge in influencing its execution decisions such as, say re-assigning the bandwidth utilized by a traffic class on a DiffServ router, carrying say traffic to and from some mobile clients accessing some service or a server during a certain period of time. **Why are all these issues and aspects important to consider for self-managing networks?** In self-managing networks we are faced with the following problems: (**1**) a monitoring component may be required by an automated task to compute or derive fine grained monitoring from the raw data gathered by the monitoring component, and export the derived data to the automated task or to some recipients designated by the automated task—all needing to be indicated via a monitoring-behaviour-specification; (**2**) Also, as described later in **Chapter 3-section 3.2**, monitoring components should provide support for on-demand creation of in-memory data models (On-Demand Data Models) of certain types of monitoring data gathered by the component that do not need to be stored or exported as packet traces or flow traces, and the destruction of the data models when no longer required by automated tasks. All this is required for the purposes of using resources intelligently i.e. optimally.

2.4.6 Some remarks

The problem of *resources*, *limited data sets* made available by today's monitoring components, devices and tools, and the problem of *inflexibilities and lack of support for customization and programmability* of monitoring services of today's monitoring devices such as routers and switches, has lead to the development of the so-called non-router based traffic monitoring, powered by passive traffic monitoring techniques (refer to sources such as [PassiveActiveMonit] and [Arlos05] for passive monitoring techniques), which nevertheless suffer the same problems as far as fulfilling the requirements i.e. the design and operational principles imposed on a traffic monitoring component by some automated tasks meant to drive a self-managing network. The monitoring capabilities and efforts on today's monitoring devices such as routers and switches are limited i.e. constrained and fixed by design, to allow for the fundamental functions they are designed for, such as routing or switching to have the required share of resources and run smoothly.

Another development in monitoring paradigms worth noting, according to state-of-the-art, is that there have been a lot of research breakthroughs in the techniques and tools for active and passive traffic monitoring, including efficient traffic capturing and real-time processing on high-speed links [Hruby05] [Sourdis03]. But, as of today, to the knowledge of the author, little attention has been invested into finding exploitable strengths exhibited by some selected tools and techniques in order to design traffic monitoring platforms meant for the kind of automated tasks that aught to drive the behaviours of a multi-service managing network. More breakthroughs in this respect can only be achieved if the monitoring needs of diverse automated tasks for a self-managing network are defined along with the definition of the design and operational principles of the monitoring components that form the elementary building blocks for a network wide traffic monitoring platform. The design and operational principles are defined by the ODM-Paradigm we are presenting in this dissertation.

More *limitations* inherent in today's monitoring paradigms, frameworks and tools are exposed by the example task automation scenarios provided later in **Chapter 6**. Apart from examining the limitations of state-of-the-art we also identify the requirements imposed on a traffic monitoring component or platform by task automation applications such as automated troubleshooting applications, if they were to be developed. Such *requirements* i.e. the design and operational principles for monitoring components defined by the ODM-Paradigm, have been the main subject of our research presented by this dissertation. The design and operational principles of traffic monitoring components supporting the ODM-Paradigm, are meant to enable distributed automated tasks driving the behaviours of a multi-service self-managing network to locate monitoring components of desired capabilities and request for monitoring services, with the ability to manage and control the monitoring services.

3 Design principles for self-managing networks

Overview of design principles and introducing the GANA Reference Model

The views expressed in this chapter regarding design principles for self-managing networks are according to the opinion and understanding of the subject by the author as published in [Chaparadza08b] [Chaparadza08c], since there are still ongoing debates (likely to be endless debates) on whether it is really necessary to draw lines between autonomicity, cognition and self-manageability of systems and networks. As such, it is very common to read some literature that does not separate the two, as both are understood to mean one and the same thing [Abbas03]. The concepts and an associated Generic Autonomic Networking Architecture (GANA), dubbed the **GANA Reference Model**, which we introduced in [Chaparadza08b], [Chaparadza08c], and [Chaparadza09] were contributed to the EU-funded FP7-IP research project: EFIPSANS [EFIPSANS], and are being implemented and further researched in the project. In GANA, *autonomicity*—realized through control-loop structures operating within network devices and the network as a whole, is viewed as an enabler for advanced and enriched self-manageability of network devices and networks. The **GANA Reference Model** is an evolving model that defines "autonomic-manager components" referred to as Decision Elements (DEs) at four basic levels of abstraction of functionality within device architectures and up into the overall network architecture, which are capable of performing autonomic management and control of their associated Managed-Entities (MEs) e.g. protocols, protocol stacks and mechanisms, as well as co-operating with each other in driving the self-management and control features of the devices and the network(s). MEs are started, configured, constantly monitored and dynamically regulated by the DEs towards achieving optimal and reliable network services. A central concept of GANA is that of an autonomic *Decision-Making-Element ("DE" in short—for Decision Element)* that is context-aware and can exhibit *Cognitive properties*. A DE implements the logic that drives a control-loop over the "management interfaces" of its assigned (by design) Managed Entities (MEs). The GANA Model defines a *framework of Hierarchical Control-Loops* and their associated Decision Elements (DEs), at *four basic levels of abstraction of functionality, namely* ***protocol-level*** *[level-1],* ***Function-Level*** *[Level-2],* ***Node-Level*** *[Level-2] and* ***Network-Level*** *[Level-4]* (types of DEs include: Node-Main-DE, Routing-Management-DEs, Security-Management-DEs, etc, at the different levels of the GANA Decision Plane). A DE(s) detects context, then, start, configure, constantly monitor and dynamically regulate the behavior of their specifically assigned (by design) Managed Entities (MEs) i.e. managed resources such as

protocols, protocols stacks and mechanisms. DEs apply configuration profiles (which encapsulate policies and config-data) on their MEs, and then react to incidents, state changes and context changes by communicating with other DEs to enforce changes on the behaviour of various types of MEs of the devices, in order to ensure optimal settings of network operation, reliable and secure services or behaviours. Therefore, what determines "autonomicity" for a "functionality" e.g. routing functionality are two mechanisms: (1) *the auto-discovery of items required by the functionality to perform an auto-configuration/self-configuration process*; (2) *the predictions or forecasting and listening for some events and reactions by the Decision Element (DE) that controls and adapts the behaviour of the functionality towards some goal, based on the events.*

As such, the GANA Reference Model is a *Hierarchical Autonomic Management and Control Architectural Framework.* The GANA is a blueprint that prescribes design principles for autonomic management and control of resources that are "generic" to be applicable in designing autonomic components for diverse implementation architectures and network environments. It is a hybrid model that enables combining *Centralized and Distributed Decision-Making based Management and Control of Resources (which include monitoring functions and associated resources),* while pointing out the main limitations of decentralization and distributed decision-making in network management and control.

The concepts and principles of the still evolving GANA Reference Model, and how it was derived for the domain of autonomic networking and self-management, are incrementally developed and described in the later paragraphs of this section.

Task automation is at the very heart of self-managing nodes and networks. We talk of the network as being a self-managing network when processes such as troubleshooting, debugging, configuration, which normally require human intervention, are automatically implemented and executed by the network and its elements co-operatively, without the involvement or with minimal involvement of human intervention. Today's network management paradigms, both those based on centralized Network Management Systems (NMSs) and distributed network management systems, already have some limited network automation capability for the above mentioned tasks and other types of tasks including performance optimization, fault-detection and recovery, etc. The trend is that there is an ever growing need for improving or enabling the automation of *dynamic, adaptive or context-driven network configuration and performance management tasks*, *network debugging and troubleshooting tasks*, *network service auditing or scanning and diagnosis etc*. The automation of such tasks is a pre-requisite for the operation of a *self-managing network.*

The key concepts: *Manager and Managed Element, Managed Object, Managed Resource* are defined in the network management frameworks and paradigms of today—be it the general

FCAPS [ITU-T Rec. M.3400] and TMN [TMN] management frameworks, Policy-based Network management (PBNM) [PBNM] or say SNMP-based network management. The term "managed element" is normally associated with a physical network element (device) while the terms "managed object" and "managed resource" are interchangeable and may be used to refer to entities that are device-oriented or software-oriented in nature i.e. peripheral or internally embedded in devices. We shall use the term *"Managed Entity" (ME)* to refer to any of the terms *managed element or object or resource.* We simply call The *Manager* is a decision-making component that performs operations on the *Managed Entity (ME)* based on objectives defined in most cases by the network management personnel. Implementing self-managing aspects or features in networks requires enriching the decision-making capabilities of Network Manager Components by enabling them to capture wider sets of the data and network views (input information) the manager component(s) should operate on in order to perform management operations on the Managed Entity (ME). Also required are: distribution of manager components where the network is partitioned into regions, as well as hierarchical structuring of the manager components. Capturing and widening the sets of the data and network views (input information) the manager component(s) should operate on in order to perform management operations on the Managed Entity (ME), requires first studying and understanding the dynamics of the network intended to be self-managing. For more information about dynamics of a self-managing network, we refer the reader to our work in [Chaparadza08a]. This means that some of the data and network views are dynamic and need to be supplied by diverse information suppliers to the *Manager* component(s). In today's management paradigms, the Manager and its associated *Managed Entity (ME)* are normally in different physical boxes. For a self-managing network(s) with different levels of self-managing properties, the Manager and its associated *Managed Entity (ME)* may be inside the same physical box or network node, meaning that the management aspects may be targeting software modules, in which case the manager and the Managed Entity (ME) are actually software components or modules interacting with each other. Regarding the information suppliers that drive the behaviour of a Manager component, the main information supplier is the associated *Managed Entity (ME)*, and other types of information suppliers either inside the box (or node) or outside, such as specialized dedicated traffic monitoring components or the substrate on top of which the *Managed Entity (ME)* is running, can all be considered the information suppliers to the decision making process of the Manager component. Therefore, *feedback mechanisms* are required to supply information (knowledge) that drive the Manager component(s) in managing the associated *Managed Entity (ME)* thereby forming a *control-loop* whose lifetime is determined by the lifetime of the Manager component i.e. the decision making element performing operations on the Managed Entity (ME). A control loop binds the Manager and the *Managed Entity (ME)* as well the objectives (goals) that are enforced on the *Managed Entity (ME)* based on information supplied to the Manager by the relevant information suppliers, which includes the "views" such as state changes and events such as fault, error or failure related incident information exposed to the Manager by the *Managed Entity (ME)*. The concept of a control loop is also known to be associated with the operation of the human autonomic nervous system [ANS], which involves the *brain*(*Manager component*), the *spinal cord* i.e. an information relay structure that relays sensory and motor information to and from the brain

towards the *effectors* (muscles, glands, etc) i.e. the *managed objects*. It is from the autonomic nervous system of a human that the field of autonomic computing [MAPE] and autonomic networking [Chaparadza08a] [Strassner06] was inspired and born. Therefore, an autonomic system or functionality is a system or functionality that is characterized by a control loop(s) governing the behaviour of the system or functionality throughout its lifetime. Therefore, we can say that **autonomicity enables the implementation of advanced self-managing properties of a system(s) or a network(s) beyond what can be achieved through automation by scripting techniques.**

It is now widely understood that the complete view of self-managing systems and networks includes autonomic computing and networking [Abbas03] [Chaparadza07a] [Smirnov03] [AT&T04] [Strassner06], whereby a *system* (macro-level view) or even a single *system-function* (micro-level view) is implemented with learning capabilities and the ability to autonomously, co-operatively or collaboratively with some other entities, operate on variable input sets such as monitoring data, data or information(knowledge) received from other entities, and strive to achieve and maintain some defined goals even in the face of challenging conditions. This requires a feedback control loop(s) that drives autonomic behaviours like self-adaptation and other self-* functions [Chaparadza07a] [Strassner06]. One expects that the space defined by self-managing functions of a system or network can actually grow very large depending on system or network functions that are meant to be automated. As such, *task automation*, as mentioned earlier, is at the very heart of self-managing systems and networks [Mortier06] [Chaparadza07a]. Central to task automation or co-operative and collaborative task automation among network entities, is the aspect of *knowledge or information acquisition and sharing* among the co-operating or collaborating entities. To mention a few examples of required task automations: dynamic, adaptive or context-driven network or system configuration and performance management tasks, automated network debugging and troubleshooting tasks, automated network service auditing or scanning, and automated fault-diagnosis, etc. Decision-making processes in automated tasks are driven by knowledge or information flow among entities. Refer to the Teminology and Definitions part of this document for the definition of an "*Automated Task*". The richer the knowledge i.e. information acquired and shared through monitoring, the better the decision making capability of a functional entity such as an automated task, a self-managing (autonomic) system or the network as a whole.

In the figures below, we illustrate how control loops can be implemented by diverse functions of a node (micro-level), as well as by a particular system (node) or the network as a whole (macro-level).

We start with the abstract model of an autonomic system defined by IBM (see [MAPE] or [Strassner06]) from which we derive control loops adapted to autonomic networking, whereby the functionality implementing the control loop i.e. the functionality considered autonomic is related to networking functions such as routing, forwarding, mobility management, QoS

management, etc. This means we can reason about autonomic networking functions such as *autonomic routing*, *autonomic forwarding*, *autonomic fault-management*, and *autonomic configuration management*, in the sense that the Manager component(i.e. the Decision-making Element) driving the control loop uses information learnt from its required information suppliers (possibly diverse) to control the behaviour of the functionality considered autonomic.

Figure 1 presents an abstract model of an autonomic system defined by IBM [MAPE] [Strassner06]. The model shows the separation between an *Autonomic Manager* (assumes similar functionality to what we called the Manager component earlier) and the *Managed Element* (assumes the same role as what we called the *Managed Entity (ME)*). The model also shows the constituent parts of the two concepts. It also shows the aspect of "knowledge" being the central driver to the control loop. The "*knowledge*" relates to all the pieces connected by the general "knowledge" scope indicated on the figure. For example, monitoring information supplied by the sensor part of the Managed Element constitute "knowledge". Similarly, the aspects of how to *analyse* the monitoring information, the *plan* of actions to be executed, how and when to *execute* the actions on the Managed-Element, all amount to "knowledge".

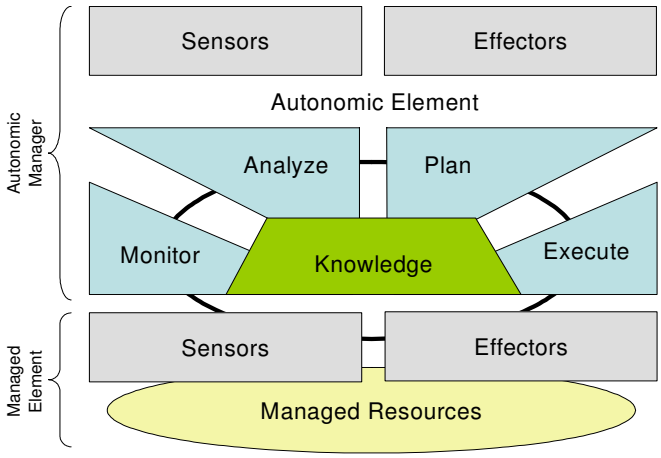


Figure 1: MAPE generic model of an autonomic system [MAPE] [Strassner06]

Figure 2 shows a model of a control-loop we derived for the domain of autonomic networking, and is the basis in a model we introduced called "the GANA Architectural Reference Model" (refer to [Chaparadza08b] for more information). The model is still considered as a generic model but illustrates(in contrast to the IBM MAPE model [MAPE]) the possible distributed nature of the information suppliers that supply a Decision-Making Element(DME) or simply, in short, Decision-Element (DE) with information(knowledge) that can be used to manage the associated Managed Resource(s) or element(s). It also illustrates the fact that the behaviours or actions taken by the Decision-Making Element (DME) do not all necessarily have to do with

triggering some behaviour or enforcing a policy on the Managed Resource(s) — i.e. changing the behaviour thereof) but, that some of the behaviours executed by the Decision-Making Element (DME) may have to do with communication between the Decision-Making Element and other entities (i.e. other Decision-Making Elements (DMEs) in the system or network). This is indicated by the extended span of the arrow "*Downward Information or Communication flow*" and the "*Horizontal Information/Communication flow*" to other DEs, as well as the fact that a DME also exposes *views* such as *events* to its "upper" DE and receives *policies, goals and command statements* from its "upper" DME. For example, the Decision-Making Element (DME) may need to self-describe and self-advertise to other Decision-Making Elements (DME) in order to be discovered or to discover other Decision-Making Elements (DMEs) in order to communicate or get "views"—knowledge known by the Decision-Making Elements (DMEs). What is also illustrated on the figure is the fact that the nature of the managed entity (or entities) may be of a *physical nature* e.g. a device, memory, or may have the nature of a an *Automated Task* (in an abstract sense) implemented by either a system function(s) or say a networking function(s) that is implemented by a single protocol or diverse protocols that employ diverse algorithmic schemes or policies. In network management, a *managed resource* is normally associated with a managed Network Element (NE) [MAPE] [Strassner06], hence we use the terms *Managed Resource* and *Managed Automated Task* in order to cover physical network resources as well software modules that can be represented as *Managed Automated Tasks*. The generic model of an autonomic networked system presented in **Figure 2**, can be applied to different types of abstraction of a system or functionality. In the figures that follow we illustrate this with different types of a "system" and the associated types of control-loops that would apply in the various cases we present. What is also special to note in the figures is the fact that a Decision-Making-Element (DME) requires to be supplied with *monitoring data or information* by relevant *monitoring components* (information suppliers) in order to drive its control-loop over the *Management Interface* of its assigned *Managed Resource(s)* or *Managed Automated Task(s)*— both of which we call *Managed Entities (MEs)* in general.

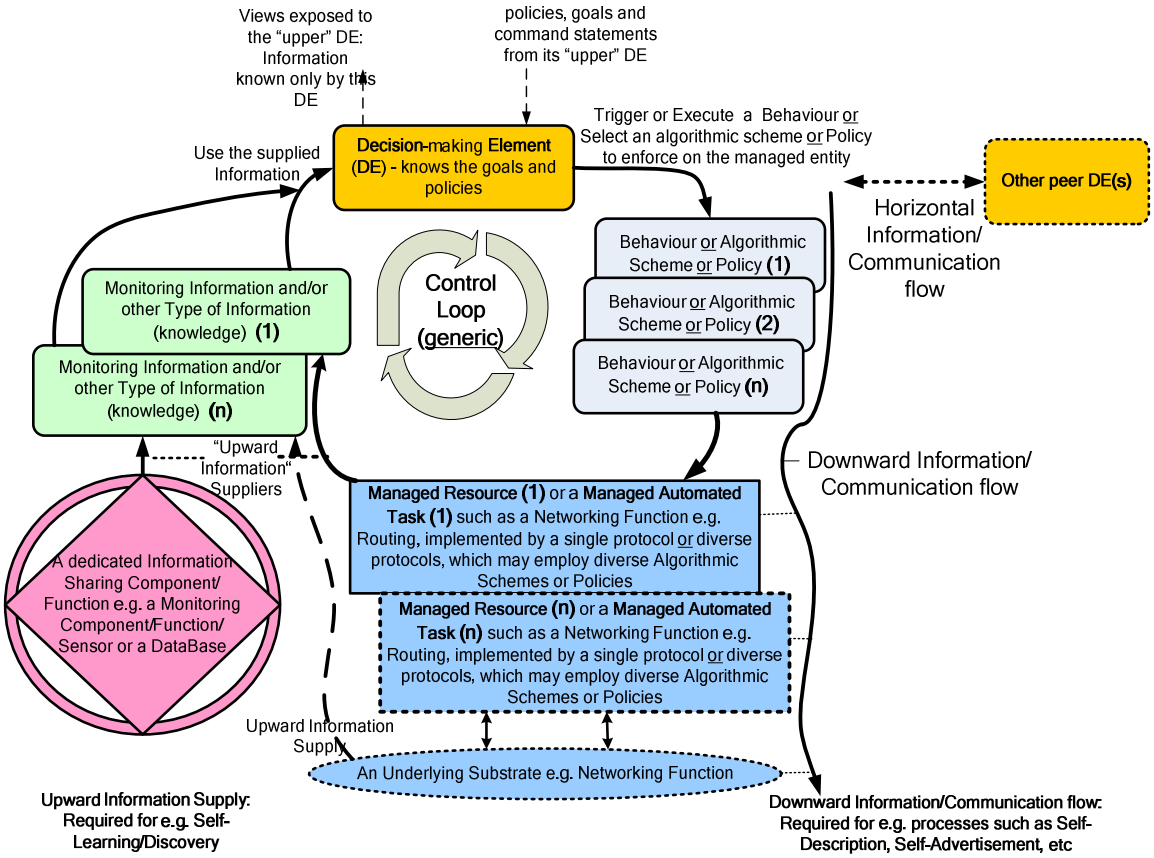


Figure 2: A "generic model" of abstract autonomic networked system

The generic model can be applied at *"four" different Levels* of "*abstractions*" of functionality for which Decision-Making-Elements (DMEs), Managed Entities (MEs) and Control-Loops can be designed in GANA, with all DMEs inter-working with each and forming the Decision-Plane of the network. In the next paragraphs we illustrate how the generic model can be applied at different levels of abstraction of functionality—thereby designing hierachical control-loop structures that must interwork to achieve the overall goals, while employing techniques such as synchronization of behaviours of the DMEs that help to achieve stability for the network as a whole. GANA incorporates the following key design principles: **(1)** Lower level DMEs expose "views" up the Decision Plane hierarchy, allowing the upper (slower) control loops to control the lower level (faster) control-loops driven by lower level DMEs); **(2)** Changes computed in the upper DMEs implementing slower (i.e. within larger time frames) Control-Loops are propagated down the DME hierarchy to the Functions-Level DME(s) implementing the faster control-loops that then arbitrate and enforce the changes to the lowest level Managed Entities (protocols and mechanisms).

The generic model of an autonomic networked system presented in **Figure 2** can be applied at a level of a single protocol, i.e. the concepts of a *Control-Loop*, *Decision-Making Element*, *Managed Resource* or *Managed Automated Task*, as well as the related (associated) self-manageability aspects may be associated with some implementation of a single network protocol (whether monolithic or modular). For example, *self-adaptive protocols* like OSPF and TCP are known to have the above concepts intrinsically implemented within the holistic behaviour of the protocol. For more information on this subject we refer the reader to [Mortier06]. **Table 1** describes the case of a protocol-intrinsic control-loop. This allows us to reason about a protocol being autonomic or having some degree of autonomicity. Normally, as of today, protocols do not access other types of information suppliers such as the external information suppliers depicted on the generic model presented in **Figure 2**.

Table 1: A protocol-intrinsic control-loop

Type of *Decision-Making-Element(DME)*	Type of *Control-Loop*	Type of the *Managed Entity or Entities (i.e. the Managed Resource(s) or Managed Automated Task(s))*	*Mapping of the DME to GANA levels* of functionality abstractions
Protocol-intrinsic DME *(knows and applies the goals and policies required for the "protocol behaviour").* A Protocol-Level-DME may peer with other Protocol-Level-DME(s)	The **Control-Loop controls the overall behavior of a single protocol**, resulting in an autonomic i.e. self-adaptive protocol like OSPF or TCP. The decision logic implemented by the DME that drives the control loop may require that certain types of information be exposed (i.e. communicated) to it by the protocol or stack directly below the protocol whose overall behaviour is regulated by the intrinsic DME.	The "**managed functions" of a Protocol Driver** that are bundled together with the protocol module as a single functional entity whose behavior can be (re)-configured and controlled by the internal DME.	**Protocol-Level**

There is growing opinion, however, that future protocols need to be simpler (i.e. with no decision logic and cognition embedded) than today's protocols which have become too hard to manage.

This means, there is a need to rather implement decision logic at a level higher (i.e. outside the individual protocols) [Greenberg05].

The generic model of an autonomic networked system can be applied at a level higher than a single protocol i.e. the concepts of a *Control Loop*, *Decision-Making Element*, *Managed Resource* or *Managed Automated Task*, as well as the related (associated) self-manageability aspects may be associated with a higher level of abstraction than a single protocol or mechanism. This means the aspect of autonomicity may be addressed on the level of abstracted networking functions (or network functions) such as routing, forwarding, mobility management, QoS management, etc. At such a level of abstraction (see **Table 2**), referred to as "Function-Level", what is managed are a group (lets call it a Function Block i.e Functional Block) of protocols and mechanisms that are considered to belong to the functionality of the abstracted networking functions e.g. all routing protocols and mechanisms of a node become managed by a Decision-Making Element assigned and designed to autonomically manage only those protocols and mechanisms. From an implementation point of view, the managed Functional Block can be considered as a wrapper around all the related mechanisms and protocols of the abstracted networking function of the node, exposing their "views" to the Decision-Making Element(DME) or otherwise the DME may be considered to have direct access to the managed mechanisms and protocols and manages the directly. The managed Functional Block may be the one that has direct access to the mechanisms and protocols and orchestrates them based on the decisions enforced by the DME. This level of abstraction allows us to reason about autonomicity of self-managing properties at this particular level of abstraction e.g. *autonomic (self-managing) routing* in the node or in the network as a whole. DMEs at this level are called "Function-Level-DMEs". Protocols and mechanisms, including protocol-intrinsic DMEs (i.e. autonomic protocols) become Managed Entities (MEs) of the DMEs at the Function-Level (i.e. abstracted-functions level).

Table 2: A *Control Loop* specific to an abstracted "*Networking Function*"

Type of *Decision-Making-Element(DME)*	Type of *Control-Loop*	Type of the *Managed Entity or Entities(i.e. the Managed Resource(s) or Managed Automated Task(s))*	*Mapping of the DME to GANA levels* of functionality abstractions
NetworkingFunction-specific Decision Making Element *(knows and applies the goals and policies required for the "protocols and mechanisms that implement the particular*	The **Control-Loop controls the overall behavior of all the protocols (including protocols with intrinsic DMEs) and mechanisms that implement the "network function"**	**The managed "Function Block" for a particular "Networking Function" e.g. the "Routing Function Block" of a node**, whose behavior can be (re)-configured and	**Function-Level**

function"). A Function-Level-DME may peer with other Function-Level-DME(s)	e.g. the "routing function", "forwarding function", "mobility-management function", etc. The decision logic implemented by the DME that drives the control loop may require that certain types of information be exposed i.e communicated to it by the protocol or stack directly below the protocols and mechanisms whose overall behaviour is regulated by the Function-Level DME.	controlled by the DME.	

On a higher level of autonomic networking functionality than the level of abstracted networking functions of a node and the network, such as routing function or forwarding function, the concepts of a *Control Loop*, *Decision-Making Element*, *Managed Resource* or *Managed Automated Task*, as well as the related (associated) self-manageability aspects may be associated with a system (node) as a whole. **Table 3** illustrates that on this level of self-managing (autonomic) properties, the lower level Decision-Making Elements (DMEs or DE in short) operating on the level of abstracted networking functions (as described earlier) become some of the *Managed Automated Tasks (entities)* of the main Decision-Making Element (DME) of the system(node). This means the node's main DME has access to the "*views*" exposed by the lower level DMEs and use its knowledge of the higher level (system and network level objectives or goals) to influence (enforce) the lower level DMEs to take certain desired decisions, which may in turn inductively further influence or enforce desired behaviours on their associated *Managed Resources* or *Managed Automated Tasks* at the "protocol and mechanisms"-level.

Table 3: The main DME and *Control-Loop* of an autonomic node

Type of *Decision-Making-Element(DME)*	Type of *Control-Loop*	Type of the *Managed Entity or Entities(i.e. the Managed Resource(s) or Managed Automated Task(s))*	*Mapping of the DME to GANA levels* of functionality abstractions

Node's Decision Making Element *(knows and applies the goals and policies required for the overall behavioiur of the node as a whole).* A Node-Level-DME may peer with other Node-Level-DME(s)	The **Control-Loop controls the overall behavior of the "*network element*$_{re\text{-}defined}$".** The decision logic implemented by the Node-DME that drives the control loop requires that Function-Level DMEs expose "views" to it, such as events that the lower level DMEs can not handle. The "views" exposed by the Function-Level DMEs may include "views" specific to protocol-level related events that can not be handled at the protocol level and are propagated up the Decision Elements hierarchy to the node level if they turn to be the kind of events that can not be handled at Function-Level.	**The overall "*Managed Resource(s)*" of the node and the *Function-Level Decision-Making-Elements* that manage specific Networking Functions**, i.e. the Managed Entities that must be (re)-configured and controlled by the node's main DME.	**Node-Level**

The next higher level of self-manageability (autonomicity) after the node level described above is the network level. **Table 4** illustrates that there may exist a logically centralised Decision Making Element or plane such as in the 4D network architecture [Greenberg05] that knows (through some means) the objectives, goals or policies to be enforced by the whole network. The objectives, goals or policies may actually require that the main DMEs of the network covered by the centralized DME or overlay cloud of network-level DMEs export "views" such as events and state information to the centralized DME or cloud in order to influence or enforce the DMEs of the nodes to take certain desired decisions that may in turn have an effect of inductive decision changes on the lower level DMEs of individual nodes i.e. down to protocol level decisions. **Table 4** describes how a distributed Network–Level Control Loop may be designed. We call this case **Case-B**. **Case-A** would involve the main Decision Making Elements of nodes working co-

operatively to manage the network without the presence of a logically centralized DME(s) or overlay cloud of DMEs.

Table 4: Managed main Decision-Making Elements of Nodes

Type of *Decision-Making-Element(DME)*	Type of *Control-Loop*	Type of the *Managed Entity or Entities(i.e. the Managed Resource(s) or Managed Automated Task(s))*	*Mapping of the DME to GANA levels* of functionality abstractions
Network-level Decision Making Element *(knows and applies the goals and policies required for the network as a whole).* A Network-Level-DME may peer with other Network-Level-DME(s)	The **Control-Loop controls the overall behavior of the "network"**. The decision logic implemented by a Network-Level DME that drives the control loop requires that Node-Level DMEs expose "views" to it, such as events that the lower level DMEs can not handle. The "views" exposed by the Node-Level DMEs may include "views" specific to protocol-level related events that can not be handled at the protocol level and are propagated up the Decision Elements hierarchy to the network level if they turn to be the kind of events that can not be handled at Function-Level and can not be handled at Node-Level.	**The Nodes' DMEs** i.e. the Managed Entities that must be (re)-configured and controlled by the specific Network-Level DME.	**Network-Level**

Network-Level-DMEs are characterized by the following attributes: (1) they are the ones with wider network-wide view to perform sophisticated decisions e.g. network optimization; (2) they

are logically centralized to avoid processing overhead in managed nodes with having distributed decision logic in network elements; (3) they are the ones that provide an interface for a human to define goals and objectives or policies e.g. Business Goals, to the autonomic network.

Figure 3 shows an example of architecture of "Decision Making Elements"—"Decision Elements" in short, that takes into account the possibility for the support of both: Case A and Case B, as well as the *hierarchical, peering and sibling* [Chaparadza08b] relations and communications that can be allowed by such an architecture. We refer to the GANA architecture [Chaparadza08b] for more information on this subject. A *peer* relationship is about the facilitation of some communication between DMEs for exchanging *views*, *negotiations* or *requesting* each other for some service(s). A *sibling* relation simply means that the entities are created or managed by the same upper level Decision Making Element ("DME" or simply "DE" in short for "Decision Element"). This means that the entities having a sibling relation can form a peer relationship within the autonomic node or with other entities hosted by other nodes in the network according to the protocol defined for their means to communicate. For more information on this subject and further related perspectives we refer the reader to [Chaparadza08b].

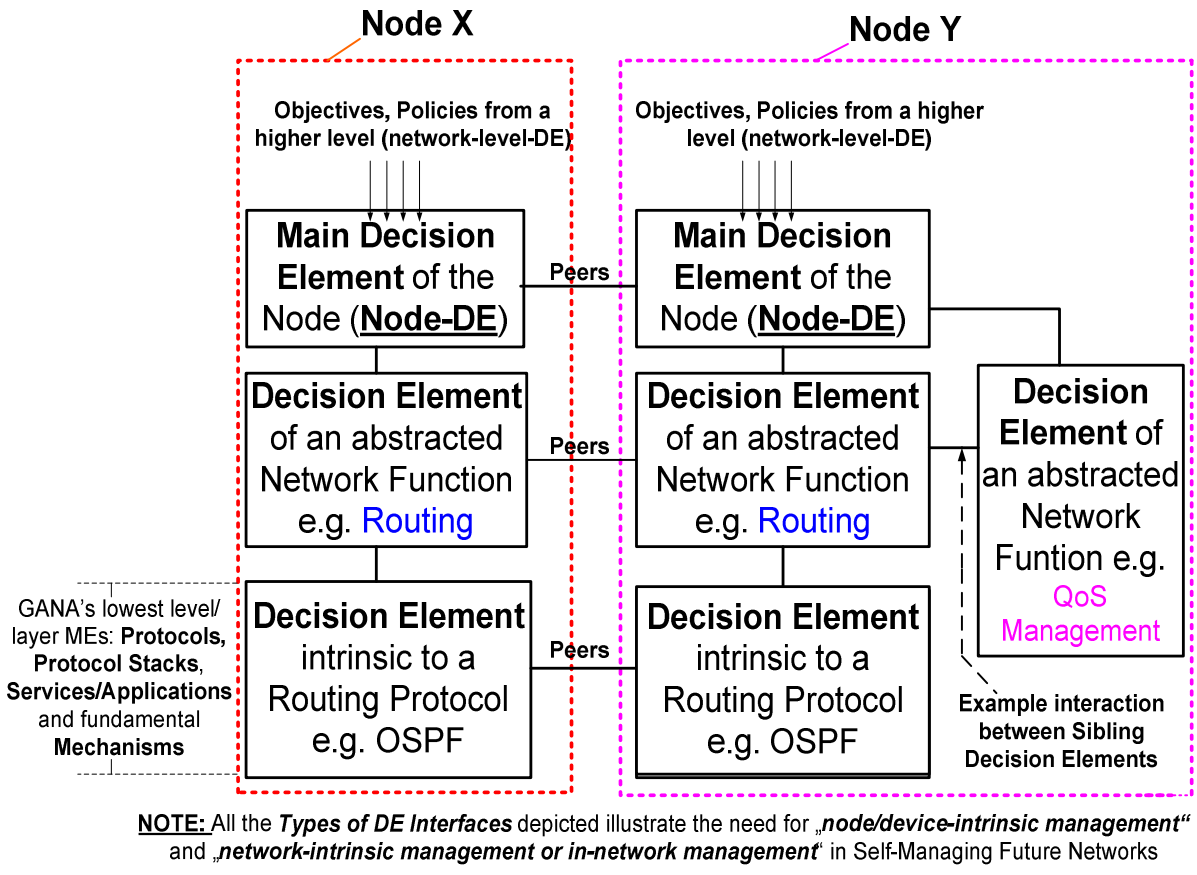


Figure 3: Hierarchical, Peering and Sibling Relations, and interfaces of DMEs

In a self-managing network, not every network device e.g. a system, network node, switch, Network Management System (NMS), a specially instrumented passive traffic monitoring probe, etc, need to implement autonomic functionality. That is to say, not every device needs to necessarily implement control-loops, since some of the devices may be dedicated to the task of capturing traffic and supplying monitoring data to other systems in the network whose Decision-Making Elements(DMEs) then use the supplied information or monitoring data to trigger self-* behaviours such as self-regulating or self-adaptation behaviours, etc. For robustness, a self-managing network should have a special *Information-dissemination functional plane* that is separated from the *data plane*, that is used to convey information such as control and management related messages and information, monitoring data, as well as time-critical information that is not user data, delivered to the distributed DMEs of the network to effect self-management. In our work in [Chaparadza08a] we introduced the concept of ***S**pinal **C**ord of an **A**utonomic **N**etwork (SCAN)* that defines such a kind of an information-dissemination functional plane. The SCAN [Chaparadza08a] is defined as a resilient communication structure consisting of network-wide information channels—*spinal channels* through which specialized types of messages and information that are especially sensitive to delay or loss can be conveyed efficiently. We refer the reader to [Chaparadza08a] for more information on the subject.

What is meant by being "Generic" in the GANA Reference Model

The main properties of GANA Reference Model being a "*Generic Model*" are summarized as follows:

1. The Reference Model enables to produce *Specification and Description Models* (e.g. formal models such as SDL Models (see ITU-T Z.100 language [ITU-T Z.100]) of the fundamental building blocks i.e. "autonomic manager components i.e. elements" referred to as "Decision-making Elements" ("DEs" in short) and their *Interfaces,* that leave out the implementation-oriented details. For details on formal approaches to producing specification and description models for GANA-oriented design specifications of autonomic components see [Prakash10b];
2. *Fundamental Interfaces* and associated *Primitives/Operations of the DEs* i.e. basic structures of messages or pocedures and as defined by the Reference Model must be generic to support different types of Data Models of data exchanged(communicated) on interfaces that are used later in the actual implementation;
3. The Decision Elements (DEs) that can be instantiated by design, for a particular *network element*$_{re\text{-}defined}$, are decided upon by the *context and role* the *network element*$_{re\text{-}defined}$ can play in the target network.

Addressing *Stability* in *Hierarchical Autonomic Management and Control Architectural Frameworks* such as the GANA Reference Model

The research work published in [Kastrinogiannis10][Prakash10c][Tcholtchev09] presents methods and techniques for addressing *Stability of Control-Loops in Hierarchical Autonomic Management and Control Architectural Frameworks* such as GANA in particular i.e. *Stability in GANA*.

Instantiations and validations of the GANA Model in diverse architectures and networking environments

The GANA Model has been "***instantiated***" and "**validated"** in the EFIPSANS research project [EFIPSANS] (refer to the project's deliverables), for *autonomic management and control* of different types of Managed Entities (*protocols, protocol stacks and mechanisms at GANA' lowest level/layer*") for diverse device and network architectures and network environments (*fixed, mobile/wireless networks*). The **GANA instantiations** mean *specification and design of GANA Decision Elements (DEs) for a specific autonomic functionality and level of operation in the GANA Decision Plane Hierarchy*)**.** Example instantiations and types of networks:

- *GANA instantiation for 3GPP/LTE/SAE*[Aristomenopoulos10] [Chaparadza09],
- *GANA for autonomic management and control of IPv6 and the supporting lower layer Transport Protocols and Mechanisms in Wired Networks* [Chaparadza10a] [Chaparadza10c] [Prakash10a][Retvari09],
- *GANA for other types of networks and devices such as ad-hoc networks*[Simon10] [Kaldanis10][Chaparadza09],

Examples of DEs for specific "autonomic functionality" and "level of operation in the GANA Decision Plane Hierarchy"):

- *DEs for Auto-Discovery, Auto-Configuration i.e Self-Configuration (refer to* [Prakash10a] [Chaparadza10c]*),*
- *DEs for Autonomic Mobility Management (refer to* [Aristomenopoulos10] [Chaparadza09]*),*
- *DEs for Autonomic QoS Management (refer to* [Aristomenopoulos10] [Zhang10] [Chaparadza09]*),*
- *DEs for Autonomic Routing (refer to* [Retvari09] [Chaparadza09]*),*
- *DEs for Autonomic DataPlane_and_Forwarding_Management (refer to* [Chaparadza10c]*),*
- *DEs for Autonomic Resilience and Survivability (refer to* [Tcholtchev10a] [Tcholtchev10b] [Chaparadza10b] *),*
- *DEs for Autonomic Fault-Management (refer to* [Tcholtchev10a] [Chaparadza10b] [Tcholtchev10b]*),*
- *DEs for Autonomic Monitoring (refer to* [Liakopoulos10] [Zafeiropoulos09] [Liakopoulos08] *),*
- *DEs for Autonomic Security Management (refer to* [Rebahi10]*).*

3.1 Requirement for traffic flow monitoring

In this section we discuss the role of monitoring in self-managing networks and the requirement for monitoring paradigms that are suitable for multi-service self-managing.

Monitoring plays a significant role in enabling the autonomicity of a single self-managing system or a network as a whole by supplying information (monitoring data) to the Decision-Making-Element(s)—DME(s) that drives a control-loop(s) that determines the autonomicity of the system or the network. This has been discussed in this chapter. In this section, we give an overview on the fundamental functions of traffic monitoring, and provide a discussion on the characteristics of self-managing networks, especially in relation to monitoring. In this section, we also look into the role of monitoring in the self-manageability of networks, and the design principles we seek for traffic monitoring components and their functions, that collectively define a traffic flow monitoring paradigm suitable for self-managing networks. The paradigm dubbed On-Demand Monitoring (ODM) is presented later in Chapter 4 and subsequent chapters of this dissertation.

An autonomic network or system (node) can be dynamically flexible in its normal operation. A self-managing or an autonomic system may, for example, require triggering some self-protecting behaviours that at the same time switch off certain of its behaviours in order to self-degrade (temporarily reduce service(s) availability) and then re-initiate the bootstrapping process. The dynamics may also be characterized by the need for *dynamic, adaptive or context-driven configuration* management or even context-driven accounting management, etc. As such, self-manageability hinges on complex system and network dynamics that enable a networked system to strive to achieve and maintain some goals even in the face of challenging conditions. Each system or network behaviour considered to be autonomic has some specific monitoring requirements in terms of specific monitoring services that must be in place at some specific points and interfaces where monitoring is required and the time and duration for which monitoring is required.

Because self-managing networks are complex and highly dynamic networks (in terms of computations and sensory information flow and analysis) that may be characterized by variable resource demands according to the network functions executing at a given time, their realisation require that resources in the network should be used by network functions on an “on-demand” basis, in order to intelligently use scarce resources and keep maintaining a picture on resource utilization. This means that, network functions, including monitoring functions of monitoring components must be designed in such a way that they can be invoked on-demand. Later in this chapter, we begin narrowing our focus to the issues pertaining to our search for design principles that allow for monitoring functions to be invoked on-demand, taking into account the need to use resources intelligently and opportunistically throughout the network. The design principles, as they are well defined later, also take into account the need for monitoring components to self-

describe their monitoring capabilities to the network to allow automated tasks i.e. network or system functions to locate, select a monitoring component, trigger monitoring functions and manage the execution of those functions and free resources on the targeted component whenever monitoring is temporarily not required or no longer required by an automated task(s).

In a self-managing network, decisions to trigger monitoring at a particular point(s) in the network, aught to be taken by some decision making entities e.g. functional entities such as protocols, applications, etc, which may be distributed in nature, with each decision maker having is own specific monitoring needs (i.e. the required monitoring-behaviour(s)). The purposes for triggering monitoring range from context-driven configuration management or self-adaptation, self-optimization, maintaining the health and performance of the node or the network, etc. The evolution of today's network management paradigms and practices to self-managing paradigms hinges on the notion of *task automation* (as explained earlier), meaning that those tasks that are normally carried out by a human, such as troubleshooting, fault-diagnosis/isolation etc, which are expected to become even more complex as systems become complex, are expected to be automatically carried out by a system or collaboratively by network systems. In the following paragraphs the role of monitoring in self-managing networks is explained in detail.

As explained in chapter 3: **task automation** for some of the complex, error-prone, time-consuming and daunting manual tasks or operations that are normally done by network operations(management) personal is what keeps driving the quest for self-managing nodes and networks. Therefore, there is an ever growing need for improving or enabling the automation of *dynamic, adaptive or context-driven network configuration and performance management tasks*, *network debugging and troubleshooting tasks*, *network service auditing or scanning and diagnosis etc*. The automation of such tasks is a pre-requisite for the operation of a *self-managing network*. For some examples on the subject of task automation we refer to [Chaparadza05b] and [Chaparadza06b]. Functional diversity and dynamic composition styles on some functions including systems functions and networking functions, as well on schemes are at the heart of self-adaptive systems, which need to compose functional behaviours based on context or situation. Such flexibilities call for design principles that allow the functions of a system(s) including monitoring functions, to be triggered on-demand to either use resources intelligently or due to context or situation changes such as in the case of mobile services and community based communications. The need to use resources intelligently is due to the fact that self-managing systems and networks are expected to require more resources (storage, computation, bandwidth, etc) than today's non self-managing networks.

Monitoring plays a very significant role in enabling the autonomicity of a standalone system i.e. a node, protocol, application or a network as a whole [Chaparadza05d]. An automated task may require that monitoring supply it with information e.g. detected events in order to make advanced decisions such as triggering some functions or adapting to changing or challenging conditions. Monitoring plays a significant role in driving the so-called control loop(s) described in

autonomic communications or autonomic computing related literature such as the **IBM-MAPE-model** [MAPE] and can also be found in [Strassner06] [Chaparadza07a]. A control-loop can be specific to an automated task or the system as a whole. A number of control loops may exist at different granularity levels as described in **Chapter 3**. For instance a control loop may be specific to a particular automated task such as a troubleshooting task (locally scoped to a node or distributed). A self-managing system may require the ability to self-degrade (reduce or stop offering services) in the face of security threats, meaning that it may need to turn off some functions (running tasks), including some running monitoring functions or may actually require triggering more monitoring functions while other kind of functions(tasks) of the system become temporarily turned off. On the other hand, monitoring functions (tasks) for such a system may be resource demanding and so due to the need to use resources intelligently, it is desirable that any running monitoring functions (tasks) must have been invoked by specific automated tasks and must be always tailored to the needs of users of the monitoring functions i.e. the automated tasks of the system or distributed (remote) automated tasks. A monitoring function is a piece of code or even a thread or process that can be invoked at run-time, serving a particular monitoring purpose e.g. a packet capturing function.

In the next paragraphs we discuss some of the key design and operational requirements or principles imposed on monitoring facilities (components and platforms) by automated tasks meant for driving a multi-service self-managing network. It is from these requirements and the need for intelligent and opportunistic use or allocation of resources in self-managing networks, that we introduce a monitoring paradigm suitable for multi-service self-managing networks, which has been the subject of this research and is presented in the subsequent sections and chapters of this dissertation. The key requirements on monitoring, in particular, traffic flow monitoring, we outline in this section, call for an answer to requirement for design and operational principles of traffic monitoring facilities (components and platforms) suitable for multi-service self-managing networks, which are expected to be resource demanding.

In order to design a traffic flow monitoring paradigm suitable for multi-service self-managing networks, thereby defining the *users* of monitoring services offered by the traffic monitoring facilities (components and platforms) as well as the design and operational principles of the traffic flow monitoring components, we introduce a concept of what we call an ***Automated Task*** that we consider to be generic. Therefore, we consider an *Automated Task* as a *user* of monitoring services offered by "*traffic monitoring components and platforms of the network*", whose design and operational principles are determined by *diverse monitoring requirements (needs)* of *diverse Automated Tasks* designed to run in devices of the self-managing network. Later in this dissertation, we define a set of design and operational principles for traffic flow monitoring components that define the On-Demand Monitoring (ODM) Paradigm—a traffic monitoring paradigm we consider suitable for multi-service self-managing networks (presented by this dissertation).

First, the notion of an "automated task" in a self-managing network comes from the automation of the techniques that are traditionally executed by humans such as troubleshooting techniques, fault-diagnosis techniques, etc [Mortier06]. The notion also comes from functions introduced into protocols and applications that are specific to aspects such as adaptive and self-optimization behaviours specific to the protocol or application.

In this dissertation we adopt a more generic concept (notion) of an ***Automated Task*** than the one presented in approaches in [Mortier06] and similar approaches, which we define as an executable sequence of steps defined using a *machine-executable language* in order to address a well-defined problem. Therefore, an *Automated Task* has a start-time and stop-time, some logic (a set of ordered steps and decisions executed to achieve its intended goal(s)) and may require some monitoring at some point in *space* (location) and *time* during its execution lifetime in order to use monitoring information in taking some decisions during its execution lifetime. Later, in **Figure 6**, we define the relation between an ***Automated Task*** and ***Monitoring***. **Figure 4** illustrates the abstract concept of an *Automated Task* and specializations of the abstract concept.

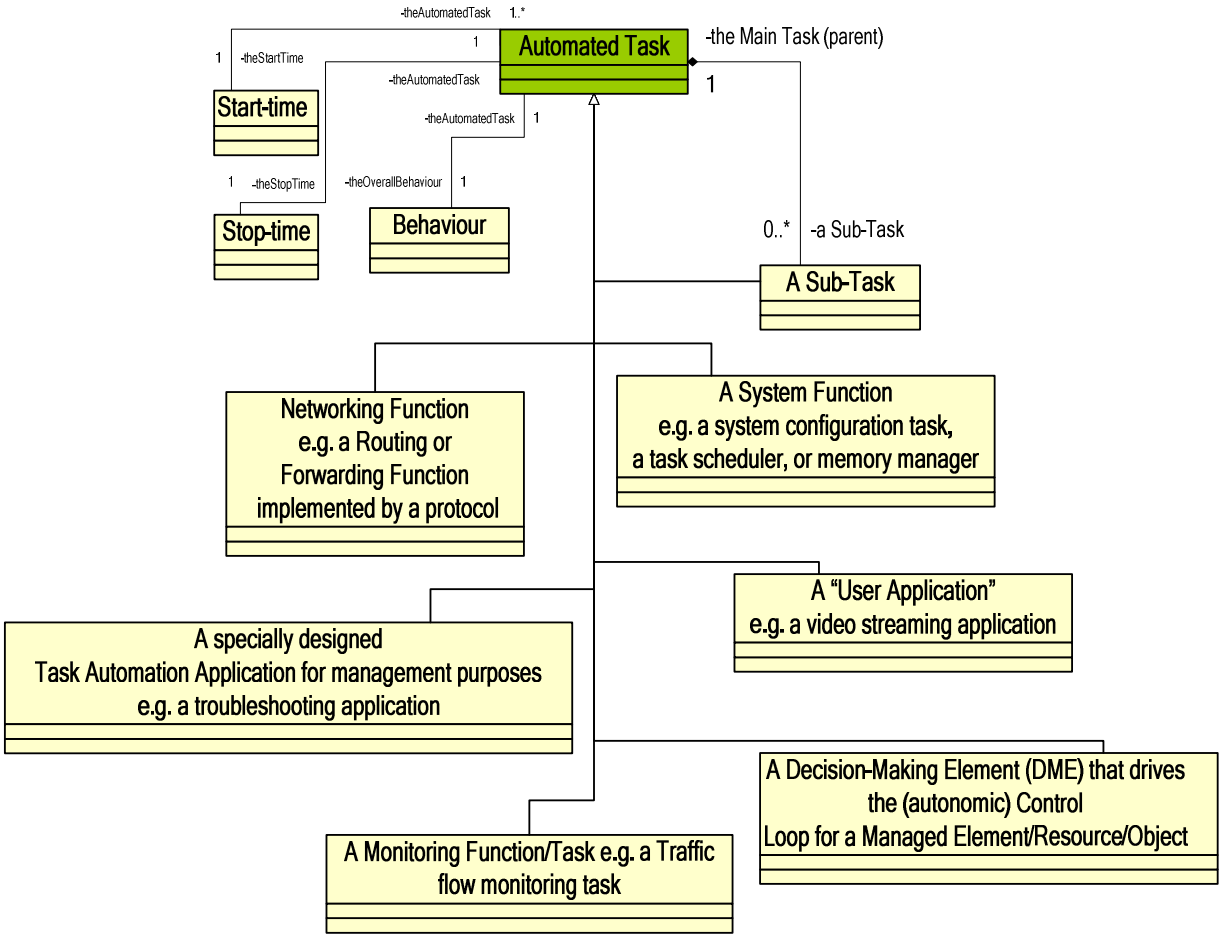


Figure 4: UML Diagram illustrating the concept of Automated Task, and a few examples of an Automated Task

The invocation time, the execution lifetime and the monitoring requirements (needs) of diverse automated tasks vary. The monitoring needs vary in terms of the required monitoring points and components, the monitoring interfaces of network elements (see definition of *network element*$_{re\text{-}defined}$) and the monitoring-behaviour(s), monitoring services or functions to be in place or to be triggered on those points, the invocation time of the monitoring-behaviour and its execution

lifetime. Some automated task such as an automated troubleshooting or fault-diagnosis task may be triggered by events such as failures detected at some point and time in the network. Such an automated task may therefore require that some monitoring services or behaviours be active at some point(s) and interface(s) in the network or that if not active, be activated. The automated task and the monitoring-behaviour required by the task may even have an on-off behaviour. Later, in **Figure 6**, we define the relation between an ***Automated Task*** and ***Monitoring***. The picture about task automation in self-managing networks is that an automated task may, for example, require triggering the generation of test/diagnostic traffic having known characteristics from a selected point(s) in the network and destined for a selected point(s) in the network and demand that some systems e.g. probes in strategic points in the network along the path perform the following: capture the diagnostic traffic, perform some computations on the traffic(e.g. rate of flow, arrival-rate), detect some changes in the characteristics, or the presence or absence of such traffic, detect events(derived from the captured traffic) and send notifications to the automated task and/or to other recipient(s) designated by the automated task. **Figure 5** illustrates the steps of this simple "requested monitoring-behaviour". The subject of composition of more complex monitoring-behaviours is covered in more detail in chapters 5 and 6. This enables the automated task and any notification recipients designated by the automated task to learn the behaviour or state of the network, the behaviour of running application(s) or service(s), etc, and use such knowledge in influencing its execution decisions such as re-assigning the bandwidth utilized by say a traffic class on a DiffServ [RFC 2474] router, carrying say traffic to and from some mobile clients accessing some service or a server during a certain period of time.

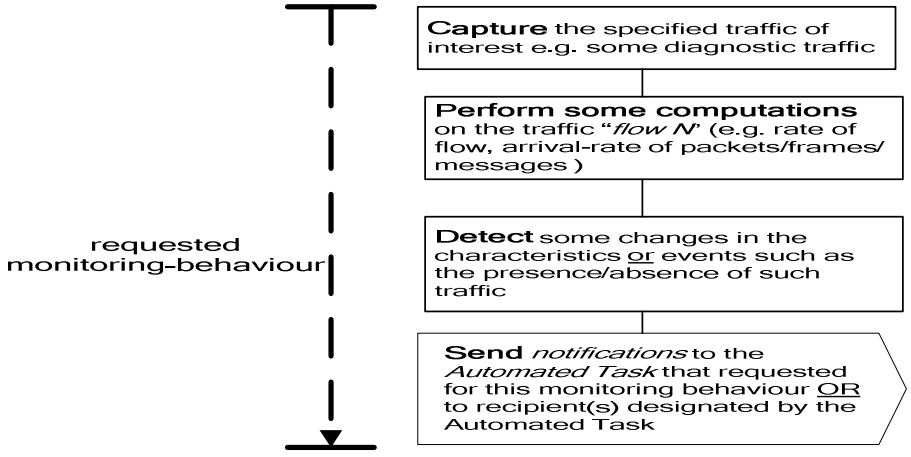


Figure 5: An example of a "*monitoring-behaviour*" requested by an *Automated Task*

The monitoring needs of some automated task(s) may even vary according to network usage context, say on a given day. For example, intelligent *Network Manager or Supervisor components* in a self-managing network, having the responsibility of maintaining the health and performance issues of the whole network(s) may have the ability to forecast and anticipate

diverse network usage demands even per day of the week or month. As such, these components may require the ability to autonomically install monitoring-behaviour(s) at selected points and time in the network and allow themselves or other automated tasks running on other systems in the network, to gather knowledge about traffic flow at those monitoring points, so that the tasks may co-operatively attempt to address traffic engineering aspects like congestion avoidance, state or event dependent routing, QoS routing, etc [Chaparadza06a].

The other view of an automated task running in a self-managing system or network, concerning monitoring, is that of *priorities* on the *purposes of monitoring*. Every monitoring activity has some purpose(s) we may call the purpose(s) of monitoring, which range from accounting management, context-driven configuration management or self-adaptation, self-optimization, maintaining the health and performance of the node or the network, etc. On the global level of a system or network, the automated tasks running in the system or network may have priorities. Likewise, on the global level of a system or network, the *purposes* of monitoring may actually have *priorities* associated with them, along with the set of corresponding monitoring functions that should participate in a particular monitoring objective for a given purpose. The reason for the priorities may have to do with limited resources dedicated for monitoring processes on a particular monitoring point and device or it may have to do with the requirements or needs for monitoring (i.e. *why is monitoring required at a particular moment, and at a particular point in the network*?). For example, at some point in time, a security monitoring function i.e. a *threat detection function* running on a node may be considered by the node's node-manager component responsible of overseeing the performance and health of the whole system, to have more priority over a monitoring function that is responsible of regularly creating and disseminating monitoring data e.g. packet/flow traces to other nodes of the self-managing network. Self-managing networks are networks that require a lot of resources in terms of computing power, memory and bandwidth consumption in communicating network state information and control messages among entities. Therefore, we expect that resource sharing and opportunistic resource usage and allocation will be prevalent in self-managing networks. These networks can be designed and engineered in such a way that systems can trigger monitoring on one another provided that they have the ability to locate the monitoring points (systems) and monitoring components of interest in order to request for a monitoring service(s) i.e. the triggering of some monitoring-behaviours of interest. This also means that some automated tasks running and/or belonging to a number of system(s) may actually require triggering a number of monitoring-behaviours on the same targeted monitoring component. On this same targeted monitoring component, some local automated tasks may have triggered some monitoring functions or behaviours that may have more priority over behaviours triggered by other systems according to some policies of the targeted monitoring component's node-manager. Designing monitoring components that are able to adapt to monitoring demands, co-operatively share monitoring burdens in such situations is clearly a challenge and is not a trivial issue.

On the other hand, from the view of a monitoring component running on a system that can serve multiple and possibly distributed independent automated tasks, indicating along with a

monitoring request, the *timeliness* as well as *priority levels* on the component's requested monitoring services, behaviours or functions, would help the component to know which services, behaviours or functions have to have higher priority for execution and which monitoring-behaviours can be terminated in the event of critical problems and resource starvation on the node hosting the monitoring component.

In order to mitigate the problem of resource demands (e.g. storage capacity for monitoring data and processing power) on a monitoring component and a particular point e.g. a monitoring probe, it is necessary to have the necessary monitoring function(s), and only those functions, *invoked only when monitoring, as defined by the monitoring functions, is required*, otherwise resources on the node may be used opportunistically for something else other than monitoring functions on the system or even shared among systems. This requirement and the need to support the notion of monitoring-requests, monitoring-behaviour-specifications issued by automated tasks (local and remote) targeting some selected monitoring component, and the execution of requested monitoring-behaviours by a monitoring component(s), as well as the ability of a monitoring component to self-describe its location and monitoring capabilities, all entail that certain design principles must be followed when designing monitoring functions of a component meant for serving the needs and dynamics of a self-managing network(s).

3.2 Requirements for a traffic monitoring platform

While in the previous section we introduced the requirement for a traffic monitoring paradigm suitable for multi-service networks, in this section, we move on to introduce the basic concepts and elementary building blocks of traffic monitoring platform suitable for *traffic flow(s)* monitoring in multi-service self-managing networks. We define a *traffic flow* as a stream of transferable units of information e.g. packets that share some properties. Without talking about whether the units of information are in the process of being transferred from one point to another, we can allow some space to think of a flow in an imaginary sense and be able to reason about the absence or presence of a flow at some point of observation.

In order to enable the development of diverse *Automated Tasks* such as the ones mentioned earlier, one of the key assets required is a traffic monitoring platform consisting of *monitoring components*, distributed throughout the network. Each of the monitoring components should implement *flexible, customizable* and *programmable monitoring functions* that can be requested to capture traffic and can be queried for relevant traffic monitoring-data i.e. *knowledge about the traffic flow of interest, traffic characteristics and presence or absence of traffic of interest at the selected monitoring point.* Such monitoring componets should also support the possibility to be requested to *perform computations* or *detect some events concerning traffic flow* and *send notifications to designated notification recipients*. Each of the monitoring components of the

traffic monitoring platform should potentially support the notions (concepts) that we describe below, and are associated with the principles of the ODM-Paradigm—referred to as ODM-Principles (defined later in **Chapter 4**). Later, in **Figure 6**, we define the relations among the key notions (concepts) associated with the principles of the ODM-Paradigm, as well as their relations to the concept of an ***Automated Task***. The key notions (concepts):

- The notion of a *monitoring-request*. A monitoring request is issued by an *Automated Task* to a monitoring component and conveys the monitoring-behaviour requested for execution or to be used as a new binding monitoring-behaviour-specification to an already running monitoring service. A monitoring-request also can be used for conveying a *monitoring-query* (described later in the last bullet points). More detailed information on the subject comes in **Chapter 4** and the subsequent chapters thereafter.
- The notion of *monitoring contexts,* due to the different needs of the diverse automated tasks (the users of the platform) and the need to support customization of the offered monitoring service to the aggregate monitoring needs expressed by requesting automated tasks. A *monitoring context* is the instantaneous aggregation of all the monitoring needs of diverse automated tasks being served by a monitoring component in terms of the executing monitoring functions or services (including all running threads and processes). More detailed information on the subject comes in **Chapter 4** and the subsequent chapters.
- The notion of *monitoring-sessions, session-requester(s), session-owner(s)*, whereby, a *monitoring-session* a monitoring service offered to an *automated task* by a monitoring component. A monitoring component provides the means to identify multiple sessions running on the component using *session-identifiers*, and also associates individual sessions with their session-owners i.e. the *automated tasks* (*session-requesters*) whose monitoring-requests resulted in the creation of monitoring-sessions on the targeted monitoring component. A *session-owner* is an *automated task* e.g. a protocol, a task automation application such as an automated trouble-shooter or an entity acting on behalf of other entities e.g. a Network Management System (NMS) that invokes a monitoring-behaviour (session-behaviour) on a monitoring component at a point in the network that is specific to the needs and control of the session-owner and can be terminated by the session-owner. Session-owners receive their assigned *session-identifiers* for use in subsequent session-management on a target component. An *automated-task* requesting for a *monitoring-session* on a targeted monitoring component transitions from being a *session-requester* to a *session-owner* if the request has been accepted by the component and a monitoring-session has been successfully created. The session-owner can be remote or local to the system hosting the monitoring component. As such, a *monitoring-session* is viewed by the associated session-owner as a monitoring-behaviour belonging the *session-owner* and has a lifetime i.e. Time-To-Live (TTL) determined by the session-owner. The other concept that can be associated with a *monitoring-session* that we introduce is that of a *session-view,* which has two perspectives: **(1)** The *session-view,* according to an automated task that is considered the session-owner by a monitoring component, is the view that the monitoring component

provides a *session-identifier* that distinguishes the session from other sessions running on the component, and allows the session-owner to use the *session-identifier* in subsequent session-management (see definition of session-management) requests issued by the session-owner. **(2)** The *session-view,* according to the monitoring component on which the monitoring-session has been created is that, it has a session-identifier and the monitoring-session may be sharing resources on the system with other sessions, such as thread-specific monitoring functions e.g. packet capturing function (a thread or process). This means a monitoring component keeps mappings of monitoring-session identifiers to identifiers of the runtime entities (threads and processes) serving the session, as well as mappings of session-identifiers to appropriate monitoring-behaviour-specifications. Since the monitoring component views a monitoring-session as an orchestration of monitoring functions by the monitoring component, and associating them with an identifier, which is then mapped to identifiers of the run-time entities (threads and processes) it invokes, the monitoring component views a monitoring-session as an association it can create and also manage by itself without necessarily having a session-owner that is not the monitoring component itself. By orchestration of monitoring functions, we mean starting the functions, some of which may be implemented to run as separate processes and threads (i.e. sub-functions), managing the execution and termination of the functions by main monitoring component that provides an interface for accepting monitoring requests from automated tasks (local to the system hosting the monitoring component or remote). This means a monitoring component also views itself as having the ability to create and manage sessions apart from offering distributed automated tasks the ability to create and manage monitoring-sessions on the monitoring component whenever deemed necessary. More detailed information on the subject comes in **Chapter 4** and the subsequent chapters thereafter.

- The notion of *monitoring-behaviour-specification(s),* whereby, a behaviour-specification is a specification of the behaviour of the requested monitoring service, specified using an appropriate language and passed to the monitoring component along with a monitoring-request. In this research we developed an extensible language that can be used for specifying a monitoring-behaviour since no composition language for specifying monitoring-behaviour-specifications existed that fulfils the requirements described later along with the language. The language, called the **E**vents **D**escription and **B**ehaviour **S**pecification **Lang**uage (EDBSLang) is presented in full detail in **chapter 5**.
- The notion of *monitoring-session management* by *session-requesters/owners* (i.e. automated tasks) on the targeted component. In chapters 4 and 5 we present the use of the following *primitives* that can be used for monitoring-session management, which are further specified and implemented by an ODM-capable probe presented in Chapter 7*:* **`create-session`**, **`pause-session`**, **`resume-session`** and **`stop/terminate-session`**. The support for such session-management primitives by a monitoring component is useful for allowing an automated task to create and manage its monitoring-session on a target monitoring component. As said in the previous

bullet points, a monitoring component should also view itself as having the ability to create and manage sessions apart from offering distributed automated tasks ability to create and manage sessions on the component whenever deemed necessary. This means for those types of automated tasks that do not require creating and managing monitoring-sessions on a monitoring component, a query like language can be developed that allows the automated tasks to simply issue *monitoring-queries* to a monitoring component, thereby causing the monitoring component to trigger the monitoring functions that can provide responses to the *monitoring-query* and are bound to a monitoring-session(s) whose session-identifier is not communicated outside the monitoring component but is rather used by the monitoring component itself to manage such a type of session(s). *Monitoring-Queries* such as "*What is the rate of flow of traffic flow X now?*" or "*What is the rate of flow of traffic flow Y at time T in the future i.e. from the time the query is received?*" or "W*hich aggregate flow consumes the highest bandwidth share on a particular interface (physical or logical)?*" can be introduced by the query language. Therefore, we can distinguish between two types of monitoring-sessions: namely a *managed monitoring-session* (managed by an automated task) and a *query-driven non-managed session*—that is not managed by an automated task (i.e. from outside the monitoring component) but is rather managed internally by the monitoring component. Manually controlled monitoring sessions can be realized through say a Graphical User Interface (GUI) or Command Line Interface (CLI) that can be used by a human to create a monitoring-session on a targeted monitoring component. In such cases the GUI or the CLI can be seen as a session-requester or session-owner.

- The need for *Capability Model* description describing the traffic monitoring capabilities of the monitoring component. The monitoring component must self-describe its traffic monitoring capabilities by creating a model that it publishes and updates to the network whenever there are changes to the capabilities of the component. Automated tasks, having learnt about capability models of different monitoring components, which include information about monitoring components' points of attachment to the network, use the capability models to select components on which to trigger monitoring-behaviours of interests. The concept of capability models of monitoring components is described in full detail in **chapters 4 and 5 and**.**Appendix A**
- The notion of *traffic generator* and *traffic sinker* roles for components that can be requested using some specially defined primitives, to generate test or diagnostic traffic flows to specified destinations and/or can be requested using some specially defined primitives, to sink traffic flows. A traffic sinker sinks a specified traffic flow(s), and does not allow (i.e. blocks) the traffic to be forwarded (pass) further into the network even if the traffic in question is not destined for the sinker component or its hosting node. Traffic generator and sinker components may be required to take part in active monitoring or measurements (refer to sources such as [Chaparadza05a] for more information on this subject). Components of such capabilities also need to self-describe these capabilities, publish and update the network to enable automated tasks to locate

and select them to play their roles when a need arises. The concept of capability models of traffic flow generator and sinker components is described in full detail in **chapters 4 and 5 and Appendix B**.

Such properties of monitoring components and other properties are captured and defined by the principles of the On-Demand Monitoring (ODM)-Paradigm, the subject of **Chapter 4**. A number of monitoring components or probes supporting these ODM-Principles may exist inside a single network node.

Figure 6 is a UML diagram illustrating the underlying key concepts of the ODM-Paradigm and their relations. Here, we focus on a high level structure that defines the key concepts of the ODM-Paradigm and their relations, leaving out their internal structures, data and interaction flows, a subject addressed later in the functional specification and implementation of a prototype ODM-capable Probe presented in Chapter 7.

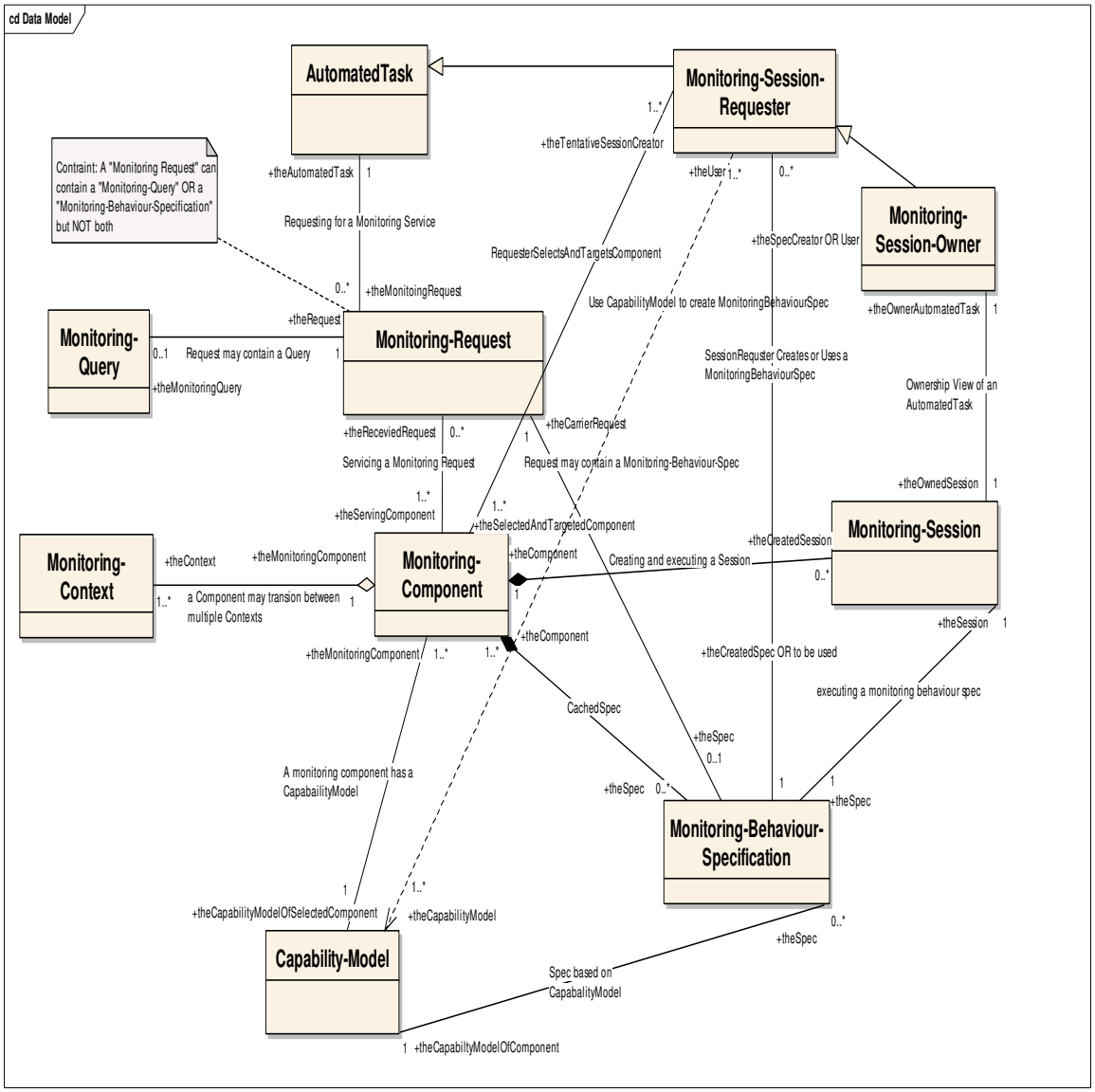


Figure 6: UML diagram illustrating the key concepts of the ODM-Paradigm and their relations

As already mentioned earlier, in order to mitigate the problem of resource requirements (e.g. storage capacity for monitoring data and processing power) on a monitoring component, it is necessary to have the necessary monitoring function(s) and those functions *invoked only when monitoring is required* i.e. during the execution lifetime of an automated task e.g. a troubleshooting application(s) that triggered the monitoring. One way of guaranteeing this is to design monitoring functions and components of the network such that their operational principles allow functions to be invoked on-demand by automated tasks via the specification of the desired monitoring-behaviour. When considering that some automated tasks require some monitoring data at some network point(s) and time and, the ability to *install, modify and terminate some*

monitoring-behaviour on the supporting network traffic monitoring platform, we see that such automated tasks and their varying needs dictate that the required network monitoring platform supports *flexible, customizable and programmable monitoring*, as well as the other principles of On-Demand Monitoring (ODM)-Paradigm defined in detail in **chapters 4 and 5**.

Therefore, the research work presented by this dissertation also sought to identify the nature and *requirements* for a *traffic monitoring platform*, which ought to consist of distributed components running in the network systems (nodes) that are designed to operate on the principles of the ODM-Paradigm, and having some monitoring capabilities and interfaces that may be required in a network wide traffic monitoring objective. In **chapters 5 and 6**, we identify the nature of those components, the roles they play, their distribution and locations with respect to topological aspects of a network. In **Chapter 4**, we define the design and operational principles i.e. the ODM-Principles, which are imposed on those components in order to make the components suitable for building a traffic monitoring platform that is suitable for automated tasks that drive multi-service self-managing networks. In a self-managing network, systems e.g. nodes may need to trigger (i.e. request) monitoring on one another or on systems dedicated to running only traffic monitoring tasks i.e. special monitoring probes that can be instrumented to monitor vantage points in the network, such as in the case of passive monitoring probes like RMON probes [RFC 3577], provided the automated tasks know how to locate a monitoring component of interest at a given point on the network topology, including its monitoring capabilities, in order to request for a monitoring service of some specified behaviour. As already mentioned earlier, the automated tasks that require selecting and triggering the monitoring of traffic at a particular point(s) may be distributed in nature, having varying monitoring needs and imposing varying requirements on the design of the traffic monitoring platform or a single monitoring component e.g. a probe, thereby necessitating the principles defined by the ODM-Paradigm for monitoring components. As such, a monitoring component e.g. a probe itself, built and operating on ODM-Principles should be viewed as both a platform in its own capacity and as a fundamental building block for a network wide traffic monitoring platform for self-managing networks.

4 The On-Demand Monitoring (ODM) Paradigm

4.1.1 Overview and Fundamental Definitions

The focus of this chapter is to provide a theoretical foundation of the ODM-Paradigm that allows architectures for monitoring components for multi-service self-managing networks to be developed and implemented following the principles and the foundation of the ODM-Paradigm laid in this chapter.

What is On-Demand Monitoring (ODM)?

We seek to define ODM, with a focus on the subject of monitoring to traffic monitoring, the notion of monitoring-needs, the dimension and scope of monitoring e.g. flow(s)-specific traffic monitoring in multi-service self-managing networks.

We start with the definition of the following notions: monitoring demand and on-demand and then associate them with the relevant concepts we defined for task automation and monitoring in multi-service self-managing networks in the previous chapters.

- A ***monitoring demand*** *is a need for some monitoring service to be offered, i.e. a monitoring-behaviour or monitoring function to be executed by a monitoring component as a result of receiving a monitoring-request from an automated task, whereby a monitoring component is a deployable run-time software or hardware entity that implements some selectable and triggerable monitoring functions.*

This definition reflects the connection between the following concepts: an "*automated task"*, *monitoring-request*, a "*monitoring component" and "monitoring function"*. Examples of *monitoring functions* are: a *packet capturing function*, *diagnostic-flow(s) generator function, monitoring data management function, event detection and computation functions, etc.* The concept of a *monitoring-request* issued by an *automated task* indicates that the monitoring service must be provided only when requested for (needed), otherwise resources (processing power, memory, etc) can be used intelligently and opportunistically by other automated tasks of the network i.e. the resources can be used for other purposes other than monitoring on the point in question. A *monitoring-request* does not necessarily need to be a request for monitoring needs for the "now" but may be a request for monitoring needs at some point in the future, starting

from the time a monitoring request was received by a monitoring component from some automated task (remote or local to the node hosting the monitoring component).

At a given point in time, the *monitoring services running on a monitoring component* may have some associated priorities. A Monitoring service is provided by the *executing monitoring functions* upon which the monitoring service is composed.

The idiomatic term i.e. adverb "**on-demand**" means *when needed or required* [Wiktionary English Dictionary]. This notion is relevant to addressing the problem of the need for a monitoring component to use resources on the hosting system for any monitoring services only when needed i.e. when a new monitoring request with different demands has arrived, otherwise resources on the system can be allocated for tasks that opportunistically need them within the self-managing network.

We now take our two definitions of a "**monitoring demand**" and the idiom "**on-demand**", together with the outlined problem statements regarding task automation and monitoring in multi-service self-managing networks, and try to identify *design and operational principles* of monitoring facilities (components, platforms) for multi-service self-managing networks which are required in order to address the outlined problem statements. This then becomes the extensible foundation of the definition of the On-Demand Monitoring (ODM) paradigm.

- "**Design and Operational Principles** *of a monitoring component that enable a monitoring demand to be expressed and communicated to the monitoring component by an Automated Task (local or remote) such that the monitoring service requested transpires on the monitoring component only for the duration for which it is needed, and that resources are freed whenever monitoring is temporarily not required or no longer required".*

The main design principles for traffic monitoring components that support the ODM-Paradigm is that the paradigm calls for an answer to the question: *How can a traffic monitoring component be designed in such a way that the "monitoring-effort" of a component, once the component has been initialized to start accepting monitoring requests, can either grow or shrink depending on the diverse monitoring demands (needs) for monitoring (the required information or monitoring-behaviours, etc), meaning that additional monitoring effort is activated only when there is a need to do so (driven by the different needs expressed in received monitoring requests)?.*

By associating the notion of "monitoring-effort" with the number and diversity of the monitoring functions invoked and executed by a monitoring component to provide monitoring services at a particular time in order to satisfy monitoring demands, we can express and intuitively define the ODM as a function:

- $\boldsymbol{M_{effC} = f\,(M_{DT})}$
- *Where:* $\boldsymbol{M_{effC}}$ **= *MonitoringEffort*** *on a Monitoring Component*

  ***MonitoringEffort* =** a measure of the *resources consumed* by the *executing monitoring functions* at *time T*.

  $\boldsymbol{M_{DT}}$ **= *MonitoringDemands at time T***

  MonitoringDemands = {F1, F2,….., Fn} i.e. a set of *monitoring-functions* ***F1, F2,…,Fn*** which must be executed i.e. invoked by a Monitoring Component as a result of receiving a ***monitoring- request(s)***. The monitoring-functions are atomic functions such as *packet capturing function*, *diagnostic-flow(s) generator function, monitoring data management function, event detection and computation functions, etc,* that can be used for composing a composite monitoring service.

The dynamics of the $\boldsymbol{M_{effC}}$ can be determined in the following manner:

Let:

$[T_0 - T_n]$ be a window of experiment for determining the dynamics of the $\boldsymbol{M_{effC}}$ on a system capable of executing monitoring functions but not yet executing any monitoring function.

Time ***t=T*** is "sampled" time that falls within the time window **$[T_0 - T_n]$** i.e. $t \in \{T_0 ..... T_n\}$

R represents overall resources such as {memory, CPU cycles and bandwidth for data transfer required by monitoring-functions running on the system}.

$R\_demanded_{it}$ = Resources demanded by the *monitoring-functions* that must be executed to satisfy the requirements of a *monitoring-request* ***i*** at time ***t.***

R_freed_{pt} = Resources freed by a "pausing" or "terminating" *monitoring-session* ***p*** at time ***t.***

Refer to chapter 7 and the Terminology and Definitions Appendix where the concept of a *monitoring-session* as a *monitoring-service* offered to a *monitoring-request* is defined.

$R\_shared\_Funcs\_executing_{it}$ = a share of resources already seized by some *monitoring-functions* that are already executing and were triggered by some early *monitoring-requests*, whereby the functions are only a 'subset' of the collective *monitoring-functions* required for fully satisfying the requirements of a *monitoring-request* ***i*** at time ***t.*** **$R\_shared\_Funcs\_executing_{it} = 0$** if no such *monitoring-functions* are already executing.

The already executing subset functions can be represented by the "*Set"***:**

$Mon\_Request\_shared\_Funcs\_executing_{it} = \{Fexec_{i_1}, ..., Fexec_{i_z}\}$**;** and the "*Set*" may contain functions common to more than one *monitoring-request* under processing at time **t.**

$\Delta R_{it} = ("R\_demanded_{it}" - "R\_shared\_Funcs\_executing_{it}")$ = the additional resources required for satisfying in full the requirements of a *monitoring-request* ***i*** at time ***t*** by invoking the additional *monitoring-functions* required to satify the requirements of the monitoring-request, without taking into account those functions that are not yet executing and yet are shared by at least two ("2") *monitoring-requests* i.e. are common to some *monitoring-requests*.

Total_R_shared_Funcs_Not_yet_executing = a total share of resources that would be required by a number of some *monitoring-functions* that are "not" yet executed and are shared by at least two ("2") *monitoring-requests, where "Y" is the total number of such functions:*

$$"Total\ \_R\_required\ \_for\_shared\ \_Funcs\ \_Not\_yet\_executing\ _t"$$

$$=\sum_{s=1}^{Y}"\text{Re}\ sources\ \_required\ \_by\_shared\ \_Function\ \_not\_yet\_executing\ _s"$$

Total_R_for_Funcs_unique_to_each_monitoring-request = a total share of resources that would be required by a number of some *monitoring-functions* "X" that are "not" yet executed and are "unique" to each of the *monitoring-requests "**i**"* being processed at time *"**t**", where N is number of monitoring-requests:*

$$"Total\ \_R\_required\ \_for\_Funcs\ \_unique\ \_to\_each\ \_mon\_request\ _t"$$

$$=\sum_{i=1}^{N}(\sum_{q=1}^{X}"\text{Re}\ sources\ \_required\ \_by\_unique\ \_Function\ \_not\_yet\_executing\ _q")$$

Total additional Resources demanded at time ***t*** on the system**:**

$$"Total\ \_R_{additional}\ \_demanded\ _t"$$

$$=SUM\ ("Total\ \_R\_required\ \_for\_shared\ \_Funcs\ \_Not\_yet\_executing\ _t",$$

$$"Total\ \_R\_required\ \_for\_Funcs\ \_unique\ \_to\_each\ \_mon\_request\ _t")$$

R_{00t} = Resources seized by *monitoring-functions* that are not considered "shared-functions" with any *monitoring-requests* under processing at time "t" but are serving the already admitted *monitoring-requests*.

The "set": $Actual\_shared\_Funcs\_executing_t=\{F_{x1},....,F_{xw}\}$ **=** a collection of functions from all the sets $\{Fexec_{i1}, ..., Fexec_{iz}\}$ corresponding to each *monitoring-request* "***i***" under processing at time "***t***".

$RshdF_t$ = Resources seized by the functions in the set "$Actual\_shared\_Funcs\_executing_t$":

$$RshdF_t=\sum_{k=1}^{W}\text{Re}\ sources\_consumed\_by\_Function\_F_{xk}$$

Aggregate Resources already seized by the executing *monitoring-functions* serving the already admitted *monitoring-requests* at time ***t:***

$$"Total\ \_R\_already\ \_seized\ _t"=R_{00\,t}+RshdF_t$$

Total Resources "being freed" by *monitoring-sessions* "pausing" or "terminating" at time ***t:***

$$"Total\ \_R\_freed\ _t"=\sum_{p=1}^{P}"R\_freed\ _{pt}"$$

Where P is the number of monitoring-sessions pausing or terminating at time "t"

Total Resources in actual use at time ***t*** after the admission of the newly arrived *monitoring-requests:*

$$"Total\_R\_in\_use_t"=$$

$$("Total\_R_{additional}\_demanded_t"+"Total\_R\_already\_seized_t")-"Total\_R\_freed_t"$$

At time "**t**" = "$\mathbf{T_0}$" and after the admission of the *monitoring-requests*:

"$Total\_R\_in\_use_{T0}$" = "$Total\_R_{\text{additional}}\_demanded_{\mathbf{T0}}$", since the other two components "$Total\_R\_already\_seized_t$" and "$Total\_R\_freed_t$" are equal to "Zero" at the beginning of the measurement experiment.

Then:

A plot of the following function at "time samples" in the time window **[T_0 –T_n],** *for a specific resources such as memory, CPU cycles or bandwidth consumed by the monitoring-functions in the transfer of data***,** estimates the dynamics of $\boldsymbol{M}_{\mathbf{effC}}$ = ***MonitoringEffort*** *on a Monitoring Component* that processes *monitoring-requests* and executes the required *monitoring-functions:*

$$"Total\ \_R\_in\_use_t" = \ f(t)$$

The dynamics of this $\boldsymbol{M}_{\mathbf{effC}}$ as well as pre-determined measurements of resources required by individual monitoring-functions a Monitoring Components can invoke, help in two things:

(1): *Determining the way an "actual deployment" of such a Monitoring Component should queue and process monitoring-requests*;

(2) *Using the following measurements of resource-demand variables or estimations thereof in the design of the admission control algorithm, as well as tuning the admission control policies employed by the Monitoring Component on monitoring-requests*: (a) The pre-determined measurements of resources required by each individual monitoring-functions under varying load on the system hosting the monitoring-component; (b) The estimations of the following resource measurement variables at a particular time "t" of processing monitoring-requests:

$$\{$$
$$Total\_R\_already\_seized_t,$$
$$Total\_R\_shared\_Funcs\_Not\_yet\_executing_t,$$
$$Total\_R\_for\_Funcs\_unique\_to\_each\_monitoring\text{-}request_t$$
$$\}$$

In chapter 8—section 8.2.2, where a prototype of such a Monitoring Component, called the ODM-Probe, is evaluated, we present a methodology on how to measure and make use of knowledge about the dynamics of such resource measurements and estimations in monitoring-requests queueing and processing strategy and in the admission control mechanisms and policies of the ODM-Probe.

In the course of our research on such design and operational principles for traffic monitoring components, suitable for multi-service self-managing networks in particular, we have identified and define the following principles we call the key principles of the ODM-Paradigm as described below (please note: this is a list of only what we consider as the key principles). Along the principles we call ODM-Principles we introduce some concepts and explain their importance when designing monitoring components and functions. After defining the principles of the ODM-Paradigm we provide a conceptual architecture of what we then define as an ODM-Capable (ODM-Supporting) component i.e. a traffic monitoring component that is designed to support the ODM-Principles (**Principle-P1** to **Principle-P7**) described in the subsequent sections. The conceptual architeure of an ODM-Capable component becomes the basis for the design and specification of an ODM-Capable prototypical system as proof of concept in **Chapter 7**.

4.1.2 Principle-P1: Support for Customizable Monitoring

This principle, **Customizable Monitoring**, for potential diverse needs of diverse automated tasks of multi-service self-managing networks, means that a monitoring component must support the concepts of a *monitoring-session (referred to as an ODM-session or simply session),* a *session-requester,* a *session-owner* and, allows for the specification of what is to be monitored e.g. the traffic flow(s), statistical data, event notifications, etc, i.e. "monitoring attributes" and not some desired monitoring-behaviour (an aspect covered by the principle of Programmable Monitoring described below) i.e. not how to perform monitoring. The *session-requester/owner* can be remote or local to the *network element*$_{re\text{-}defined}$ hosting the monitoring component. A monitoring-session is a perception of a monitoring-behaviour tailored to the monitoring needs of the *session-requester/owner*. Such a session-owner is typically an ***Automated Task*** such as an application, a protocol or a Decision-Making-Element (DME) that implements a control-loop and drives autonomic behaviour within a *network element*$_{re\text{-}defined}$ or the network. Customizability for session-requesters/owners also requires that the ODM-capable component provides security guarantees to remote automated tasks (i.e. remote session-requesters/owners). Therefore, *Customizability* of the monitoring functions of a monitoring component means the support for the following notions by the monitoring component, which have been described earlier: *monitoring-session*, *session-requester/owner* and, *allowing for the specification of what is to be monitored e.g. the traffic flow(s), required statistics, event notifications, etc*. Therefore, customizability refers to the "monitoring attributes" and associations to the monitoring needs of a particular Automated Task leaving out the aspect of specification of some desired monitoring-behavior (i.e. programmability).

4.1.3 Principle-P2: Support for Programmable Monitoring

In contrast to the previous principle (Customizable Monitoring), which needs to be complemeted with the aspect of specification of some desired monitoring-behavior (i.e. programmability), this principle, **Programmable Monitoring**, means that a monitoring component e.g. a probe must allow for the specification of the *behaviour of a monitoring-session (i.e. a monitoring-behaviour-specification)*, *the installation of the session-behaviour on a selected monitoring point (component), the modification and termination of the behaviour by the session-owner* at any time during the operation of the monitoring component. As mentioned earlier, a *session-requester/owner* can be remote or local to the *network element*$_{re\text{-}defined}$ hosting the monitoring component. A *session-requester/owner* should use a specification language for specifying the intended monitoring-behaviour to be executed. Therefore, a Composition Language for specifying a monitoring-behaviour is required. Such a language should include the possibility to specify actions to be performed on the target by the required behaviour, event-descriptions and event-notification propagations to designated parties, including actions to be performed by notification recipients etc.

4.1.4 Principle-P3: Support for on-demand creation and destruction of gathered monitoring-data and in-memory data models

This principle, **on-demand creation and destruction of gathered monitoring-data and any associated in-memory data models (on-demand data models) of monitoring-data**, means that a monitoring component must support on-demand creation and destruction of monitoring-data and any associated in-memory data models for some of the monitoring-data derived and computed from the captured traffic such as the rate of traffic flow, arrival-rate, or statistics that require to be stored in memory. Data models of in-memory data enable a session-owner that specified the need for the creation of a data model on the target i.e. an automated task and any other tasks authorized by the session-owner to retrieve the gathered data in a systematic and secure way. The "in-memory data models" should be considered as part of On-Demand Data Models in general—including data models that need not be created in-memory e.g. trace files, created as representation of gathered monitoring data. On-Demand Data Models can be created subject to availability of resources to satisfy the needs of a monitoring-request. This is because the creation of an in memory data model may affect the performance of the other monitoring functions such as packet-analysis and event detection functions triggered as a result of the monitoring request i.e. serving the monitoring-request. Once an On-Demand Data Model has been created by the monitoring component, the definition of the data model should be provided to the session-owner that requested the creation of the data model or to other designated system or tasks designated by the session-owner, to allow them to access it. The monitoring component should then destroy the data model (remove from memory) when the binding monitoring-session-lifetime has expired, thereby freeing resources. The concept of On-Demand Data Models in the larger picture includes not only in-memory data models of some monitoring data, but also data models such as packet traces or flow traces that can be created and stored directly on storage devices by a monitoring component and are not supposed to be stored and queried in-memory on the monitoring component or device.

4.1.5 Principle-P4: Support for Triggerable, Configurable and Re-Configurable Monitoring

This principle, **Trigger-able, Configurable and Re-Configurable monitoring**, implies that the monitoring functions of a monitoring component e.g. a probe are designed in such a way that each function or a group of functions can be triggered or invoked on-demand and that, the monitoring component supports *parameterized primitives* for managing a monitoring-session in order to ensure that *resources are freed* whenever monitoring is temporarily not required or no longer required and that, values of *parameters to monitoring functions* can be modified on-the-fly. It means a *monitoring-request* issued by an automated task, triggers only that monitoring function(s) considered necessary (required) and that the monitoring component supports *parameterized primitives* (similar to primitives we introduced in [May07]) that enable an

automated task to create the notion of a *monitoring-session* on the component and manage the *monitoring-session*.

4.1.6 Principle-P5: Support for Adaptive Monitoring

This principle, **Adaptive Monitoring**, means that the monitoring-behaviour bound to a particular monitoring-session can adapt to unexpected events occurring on the monitoring point and can be adaptive to changing resource availability. The monitoring component itself may prohibit or switch-off some monitoring-behaviours belonging to different session-owners in an attempt to adapt to changes in resource availability or other factors such as priorities associated with monitoring-behaviours.

4.1.7 Principle-P6: Support for Intelligent and Opportunistic use and allocation of resources

This principle, **Intelligent and Opportunistic use and allocation of resources**, means that a monitoring component must check whether resources are available to satisfy the monitoring requirements expressed in the monitoring request (monitoring-session-behaviour spec) and perform admission control on monitoring requests. As such ODM proposes to apply *admission control* to monitoring requests and requested behaviours based on say, resource availability to satisfy the demands of the monitoring-behaviour requested for execution. Together with its implemented monitoring functions the component must ensure that a monitoring-behaviour can be shared by a number of monitoring-requests (from multiple automated tasks i.e. session-owners) if possible and avoid starting additional behaviours or function threads unnecessarily. We call this "*session-merging"*. Upon the reception of *pause-session* and *terminate-session* commands (primitives) from an automated task, resources seized by the monitoring component from the overall system resources for the monitoring behaviour(s) being paused or terminated must be freed.

The idea behind the session-merging done by the monitoring component, is to let session-owners have the notion that they are managing their sessions without them having to know that the sessions are sharing some threads or processes of execution on the target or that a set of session-behaviours are merged into an aggregate behaviour that covers the aggregate needs of the associated set of monitoring-behaviour-specifications, according to some aggregation (merging) policy (criterion). The aggregation (merging) policy (criterion) may have to do with say, all monitoring-behaviour-specifications specifying the same traffic capture filter result in a dedicated traffic filter specific thread being created and serving a number of sessions belonging to multiple session-owners. This would make the monitoring component intelligently use resources by also ensuring that whenever a session-owner issues any of the following primitives: *pause-session, refresh-session resume-session* and *terminate-session*, the monitoring component checks to see if there are any sessions still running on the shared resources e.g. threads, and

ignore the primitives if the shared resources are still in use by other sessions. If there is no monitoring-session still associated with the shared resources, except the session for which any of the above session-management primitives has been received, then monitoring component respects the desired effect of the primitive(s).

4.1.8 Principle-P7: Support for Self-description and Self-advertisement of Capability Models

This principle, **Self-description and self-publishing of Capability Models**, means a monitoring component should self-describe its monitoring capabilities by creating what we call a *capability model* that is self-published (self-advertised) and updated to the network whenever there are changes to the capabilities of the component. Automated tasks, having learnt about capability models of different monitoring components, which include information about their points of attachment to the network, use the capability models to select components on which to trigger monitoring-behaviours of interests.

Therefore, apart from the principles already mentioned, the ODM-Paradigm requires that an ODM-Capable monitoring component operates on the following principles:

- *Self-describes its monitoring capabilities using a Capability Model*;
- *Disseminates the model to the network and keeps the network updated of changes to the Capability Model;*
- *Supports discovery of presence and point of attachment to the network;*
- *Supports solicitation for its Capability Model.*

5 Technical Solutions for the ODM Concepts and Principles

5.1 Overview

In this chapter we describe the technical solutions we introduced to the ODM-Principles proposed and defined in Chapter 4, which are meant to address the issues outlined in the problem statement.

5.2 Customizable Monitoring and Programmable Monitoring

Since the principles Customizable Monitoring and Programmable Monitoring are closely connected, we present the solutions to realizing them jointly in this section, starting with customizable monitoring and moving to programmable monitoring, which is rather more complex.

In order to realize this principle (**Principle-P1**), we introduce the concepts of *monitoring-session* and *session-requester/owner* in order to allow for the design of monitoring components that can assign unique *session-keys* (*identifiers*) to sessions created on the monitoring component and allow the session-creator/owner i.e. an automated task, whether remote or local to the system hosting the monitoring component, to use the assigned session-key (identifier) as an argument supplied to the monitoring component whenever the automated task seeks to manage its session by either pausing, resuming or terminating the execution of the monitoring-session-behaviour. This kind of flexibility required by session-owner in managing a monitoring-session, is described along **Principle-P4** whereby the focus is on the actual primitives for managing a monitoring-session. The ODM-capable component should allow multiple sessions of diverse behaviours to run in parallel and performs admission control on monitoring requests as described later in **Principle-P6**. This principle is the one that results in a monitoring-behaviour being admitted (or not) for execution by a targeted monitoring component. This raises the question of scalability, because it is impossible to allow all kinds of Automated Tasks in the network to target a monitoring component and create Monotoring-Sessions. Later, in **Chapter 9**, which discusses scalability issues, we address the question of whether all kinds of **Automated Tasks** in the network should be allowed to locate a

monitoring component of some given capabilities and create a **Monitoring-Session** (i.e. an ODM-Session) on the component.

In order to realize this principle (**Principle-P2**), we developed a composition language for the specification of the *behaviour of a monitoring-session (i.e. a monitoring-behaviour-specification)*, called EDBSLang (*Event Description and Behaviour Specification Language*) [Chaparadza07a]. The language is described below, including example scenarios on how it can be used for specifying *monitoring-behaviour-specifications* that can be conveyed by monitoring-requests issued by diverse Automated Tasks. EDBSLang is based on the XML language [XML] and can be used by humans to engineer i.e. author monitoring-behaviours that are meant to be selected and executed at some points (components) by Automated Tasks in a self-managing network. Automated tasks can also use the EDBSLang to programmatically specify *monitoring-behaviour-specifications* meant to be parsed and executed on a selected target. A number of monitoring-sessions and associated session-behaviours (monitoring-behaviour-specifications) may run on a target and may belong to different session-owners (local or remote). A monitoring-session created as a result of a monitoring request should detect the events specified in the associated monitoring-behaviour-specification and propagates notifications to the recipients designated by the session-owner; executes the session-owner specified actions upon the detection of an event(s). The EDBSLang allows for the specifications of the following items in a monitoring-behaviour-specification that can then be requested for execution by an ODM-capable component:

a) The *Traffic-Flow* to be monitored i.e. the packet stream to be captured based on a specified *Traffic-Filter* (also referred to as *ODM-Traffic-Filter* in this document).
b) *Event-Descriptions* derivable from the traffic-flow characteristics of interest i.e. from the traffic flow to be monitored (observed).
c) Event-associated *Actions* to be performed by the monitoring component i.e. by the monitoring-session running the monitoring-behaviour-specification.
d) *Event-Notification descriptions*, *notifications recipients*, *Actions* to be performed by notification recipients.
e) *Finite-State-Machine (FSM)* based monitoring-behaviour-specification.
f) *For* Loops in the monitoring-behaviour-specification.
g) *Labels* and *goto* statements in the monitoring-behaviour-specification.
h) *Assertions* on a packet(s) injected into the network for diagnostic purposes and requiring the monitoring-session to observe and notify of the asserted events.
i) *On-Demand SNMP MIB Data Models* to be created and destroyed upon the expiration of a binding session(s).
j) *Trace creation and dissemination* method for requesting for packet trace and or flow trace creation and dissemination.
k) *Decisions* and *Evaluations* based on the characteristics of traffic flow(s) specified for monitoring.

Programmable traffic flow monitoring plays a very significant role in driving the logic of control loops for autonomicity in multi-service self-managing networks. The art of engineering programmable traffic flow monitoring in such networks is a question that continues to require some research in order to come up with frameworks that include languages for programmable monitoring across multi-vendor environments. In this section we present the EDBSLang language. We also provide an insight into the art of engineering programmable traffic flow monitoring in multi-service self-managing networks. We provide and discuss a comparison between the approach followed by the EDBSLang and different known achievements and approaches to programmable traffic monitoring in general. The rest of this section is organized as follows: **sub-section 5.2.1** of this section presents the EDBSLang composition language. We also discuss a *Validator* for the language and how it must implement some "constraints" on the combination of some elements used in the composing monitoring-behaviour. **Sub-section 5.2.2** gives some remarks, including the open problems to be addressed. **Section 5.7** discusses how the language can be applied for programmable traffic flow monitoring in multi-service self-managing networks. **Sub-section 5.7.1** provides an example scenario on the specification of a monitoring-behaviour using EDBSLang.

In [Chaparadza07a], we discussed what has been achieved in programmable monitoring: a script-based approach and an approach based on implementing the monitoring functions of a monitoring component in such a way that a desirable monitoring-behaviour can be triggered through the use of parameters composed from the set of parameters understood by the monitoring functions or a by monitoring tool(s) implemented on the target. As described in [Chaparadza07a], the second approach calls for the creation of composition language for use in specifying monitoring-behaviours that can be requested for execution by a monitoring component. This is why we introduce such a language i.e. the EDBSLang.

However, in some cases, both, the script-based approach and the specification-based approach for monitoring-behaviours (the approach upon which EDBSLang is based), may be used in combination depending on resource availability and flexibility requirements on a target system providing the monitoring services. The EDBSLang is a specification based approach rather than a script-based approach in the sense that the monitoring component supporting the EDBSLang is meant to support translating an EDBSLang based monitoring-behaviour-specification on-the-fly by self-configuring and triggering the required internal monitoring functions on-demand. In addition, the EDBSLang generically supports specifying "script-names" as a special type of "actions" rather than script-behaviours (the case for the script-based approach), with the assumption that a repository of scripts is maintained by the monitoring components. This means the "script-names" i.e. special types of "actions" and all other types of actions could actually be standardized across monitoring components.

EDBSLang language is based on the XML language. Therefore, we present the XML-Schema of the language and discuss the significance of its elements and constructs, syntax and semantics. The EDBSLang Schema itself can be found in **Appendix A**. The development of the language is inspired and motivated by the ODM-Paradigm. The foundation of the composition language is based on our research on monitoring needs (requirements) and scenarios for task automation in

self-managing networks, such as the automation of *dynamic, adaptive or context-driven node or network configuration and performance management tasks*, *network debugging and troubleshooting tasks*, *network service auditing or scanning and diagnosis etc*. The automation of such tasks is a pre-requisite for the operation of a *self-managing network* [Chaparadza06a]. As mentioned in Chapter 3-section 3.1, an *automated task* has a *start-time* and *stop-time*, some *logic* (a set of ordered steps executed to achieve its intended goal(s)) and may require gathering some monitoring data at some point in *space* (location) and *time* during its execution lifetime i.e. to program monitoring-behaviour(s) at selected points in the network. The invocation time, the execution lifetime and the monitoring requirements of diverse automated tasks vary.

5.2.1 The EDBSLang Language and its XML-Schema

As mentioned earlier, the EDBSLang language was developed on the basis of researched needs and design requirements for programmable monitoring in multi-service self-managing networks. The language is evolvable in the sense that more constructs can be added. We first present the structure of the XML-Schema i.e. the *meta-model (meta-language)*, which is presented in **Appendix A**. The **figures: Figure 7, Figure 9, and Figure 10** show the tree of the *TAGs* i.e. the *Elements* of the language that form the concepts from which instances can be created and composed into a monitoring-behaviour. The root *Element* is called the `ODM_Behaviour_Specification` and has a number of child elements, which may also have child elements as seen on the tree. We describe the elements and indicate whether some elements are optional or must be specified during the instantiation of the meta-model. Such constraints and other constraints such as the combinations of tags allowed in a behaviour composition, the number of times a tag may appear in the composition etc. can be enforced by a *Validator* we implemented in Perl.

<ODM_Traffic_Filter>: (see **Figure 7**) - This tag is of type "`string`" and is used to specify the initial traffic filter to be bound to the monitoring-behaviour. The specification follows the syntax defined by the language to be used for filtering, which could be the BPF [BPF] filter syntax used by well known pcap [libpcap] based traffic capturing tools like *tcpdump* [tcpdump], ethereal [Ethereal] etc. The traffic filter may be modified during the execution lifetime of a monitoring behavour as indicated in subsequent specifications issued or special commands i.e. primitives for managing a monitoring-session such as the *modify-session* and *set-filter* commands i.e. primitives described in [Chaparadza06a] [Chaparadza05d] and in more detail in **Chapter 7**.

<Nth_packet>: (see **Figure 7**) - This tag is of type "`complexType`" and is used to specify an assertion concerning a specific packet to be detected by a monitoring-session. This is especially useful when the session-owner (an automated task) is running some diagnosis, debugging or troubleshooting task that injects test/diagnostic packets into the network and demands that some activated i.e. triggered monitoring-behaviours at selected points be able to detect and notify of the assertion. The <number> tag is used to indicate the specific packet relative to the start time of the monitoring-session. The <layer> tag is used to indicate the layer of interest relative to the

physical layer of the stack on the intended monitoring target. The <Label> tag is used to give an identifier to this statement to allow conditional jumps to this statement from another statement in the behaviour composition.

The <field> tag is used to specify the protocol field in the packet and the <value> to be used in finding a match. The <field> must include one of the following: an <event_notification> tag indicating if the monitoring-behaviour must notify the session-owner; an <event_description> tag providing the event description that the monitoring-behaviour must include in its notification message issued to the session-owner(i.e. session-owner designated notification recipients); an <event_name> tag naming the event; a <dest> tag indicating the designated parties to which the monitoring-behaviour should send the event_notification; a <notification_sink_action> tag indicating the action to be taken by every notification recipient specified in the <dest> tag; a <pause_session> tag – used to indicate that the monitoring-behaviour must temporarily stop executing, thereby freeing some resources, and wait for commands from the session-owner for resumption or termination of execution; a <terminate_session> tag indicating that the monitoring-behaviour must stop execution (terminate) and free resources completely; a <goto_statement> tag indicating a jump to some *labeled* statement.

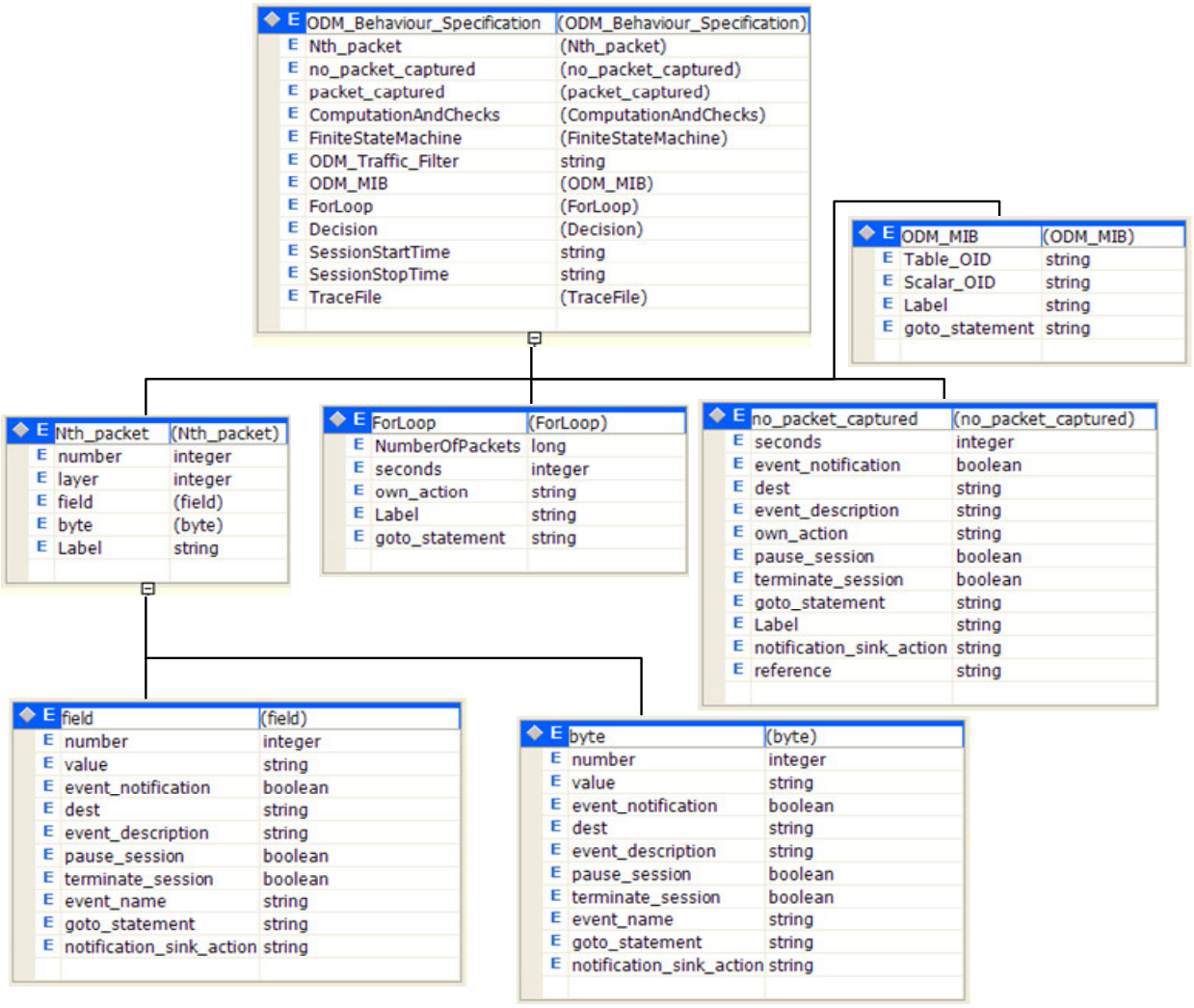


Figure 7: EDBSLang Schema tree (1)

Regarding constraints, the <terminate_session> implies that a <goto_statement> can not be included in the composition within the enclosing <field> tag. The issue of constraints is a subject for further research due to the fact that constraints have a lot do with capabilities and resources that can be supported by monitoring components.

The <byte> may be used instead of the <field> or together with the <field>, in order to make an assertion on a byte inside a packet e.g. a diagnostic packet injected into the network by a troubleshooter i.e. a session-owner for some monitoring-session running on a monitoring component. When the two are used together, there are also some constraints on the use of their child elements, to avoid conflicting or ambiguous monitoring-behaviour-specification.

<packet_captured>: (see **Figure 9**) - This tag is of type "complexType" and is used to specify what the monitoring-behaviour must do (<own_action>) when a packet matching the <ODM_Traffic_Filter> has been captured. The <reference> tag indicates the reference-point of interest at which the evaluation must take place relative to the start time of the monitoring-behaviour or together with the <seconds> tag, can be used to indicate the time allowed to elapse before an evaluation can be performed on whether a packet has been captured or not. The other constituent tags allowed in the <packet_captuted> tag have already been described above. **<no_packet_captured>:** (see **Figure 7**) - This tag is of type "complexType" and is used to specify what the monitoring-behaviour must do (<own_action>) when a packet matching the <ODM_Traffic_Filter> has NOT been captured. It is the converse of the <packet_captured> tag and has similar semantics.

<ComputationAndChecks>: (see **Figure 9**) - This tag is of type "complexType" and is used to specify a computation to be performed and the checks to be performed on the computation. The <computation> tag is used to indicate what is to be computed e.g. bandwidth utilization, packet arrival rate, rate of flow, etc, over some <duration> of observation. <computationStep> is used to specify the steps at which the computation is repeated and either values stored (as indicated by the <storeComputation> flag) and/or propagated (as indicated by the <propagateComputedValue> flag). <computed_event_check> is used to specify a desired computed event that is based on the base <computation>. The <computed_event_check> must include the computation being referred to, which may actually be a derived computation specified in addition to the base computation, and a <booleanCheck> that is used to specify how the check should be performed e.g. whether the <computation> is *greater than*, *smaller than* or *equal* to a specified <threshold>.

<TraceFile>: (see **Figure 10**) - This tag is of type "complexType" and is used to specify that captured traffic should be stored into a trace file; the <StartTime> for the trace creation; the <CaptureDuration> for the trace; the <CreateInterval> to indicate the interval at which new traces are created; the <dest> to specify the destination(s) where a trace(s) is disseminated to; the <DisseminationMethod>; including a <Label> and <goto_statement> to enable jumps between statements if required.

<FiniteStateMachine>:(see **Figure 10** and **Figure 8**) - This tag is of type "complexType" and is used to specify a finite-state-machine based monitoring-behaviour, which involves a performing some operations on a stream of packets directly observed from the network or from packets captured and stored in trace file. This involves specifying so-called m_states (match states), which involve matching packets against a given filter. The <m_state> tag has a number of child tags that can be used in the specification. These are: <TrafficStreamSource>; <Filter> for specifying the filter to be applied; <StateName> for use in jump statements (transitions); <Relative_time> for specifying the point i.e. position in time, relative to the start time, when an attempt to match packets and/or make a computation around that reference point e.g. bandwidth at that point in time is performed; <PacketCount> can be used for the same goal as the <Relative_time> tag, in which case, packet count is used for specifying the point in time at which a match is attempted; <NextStateMatch> is used for specifying the next state for transition

in the event that a match occurred i.e. some packets were captured that match the specified filter; <NextStateNoMatch> is the next state to go to in the event of no match; the other constituent tags possible such as <event_notification> etc. have already been described earlier.

```
<m_state State="matchICMP" Filter="ip proto icmp" Rtime ="<300"
    NextStateNoMatch ="matchUDP" action ="ScriptM.pl" NextStateMatch
    ="matchTCP">
</m_state>
<m_state State="matchUDP" Filter="ip proto UDP" PCount ="=217"
    NextStateMatch ="matchSCTP" action ="ScriptN.pl" NextStateNoMatch
    ="matchSIP" >
</m_state>
<m_state State="matchSCTP" Filter=" ip proto SCTP" PCount =">217"
    NextState ="matchRSTP" action ="ScriptN.pl" >
     Computation ="Total Bytes">
</m_state>
<m_state State="Final">
</m_state>
```

Figure 8: Example fragment of an FSM-based part of a monitoring-behaviour-specification

<ODM_MIB>: (see **Figure 7**) - This tag is of type "`complexType`" and is used to specify that an On-Demand MIB (Management Information Base) be created by the monitoring-behaviour. As defined in [Chaparadza06a] and later along **Principle-P3,** an On-Demand MIB is a data model that is created on-demand on a data collection point and has a *lifetime* i.e. TTL (Time- To-Live). It is a MIB that is created to store information that is temporarily required and is destroyed when the information is no longer required. Some *Table type of Object-Identifiers (OIDs)* and *Scalar OIDs* can be specified. Examples are: the Protocol_Hierachy_Statistics Table that stores the statistics of the captured traffic, based on the ODM_Traffic_Filter, and reflecting statistics per layer according to the protocol stack reflected in a decoded captured packet; a DetectedEvents_Table that records all the events detected by the monitoring-behaviour, which can then be read by the owner of the monitoring-behaviour (session-owner); a Generic_Packet_HexDump scalar OID for storing a hexadecimal string dump of a packet that can be read and further analyzed by a remote session-owner, a case useful in automated network diagnostics whereby diagnostic packets are injected into the network and selected monitoring components are requested to capture the diagnostic traffic in order for the automated diagnosing component to inspect the traffic characteristics in order to infer, observe or verify the behaviour of the network or services.

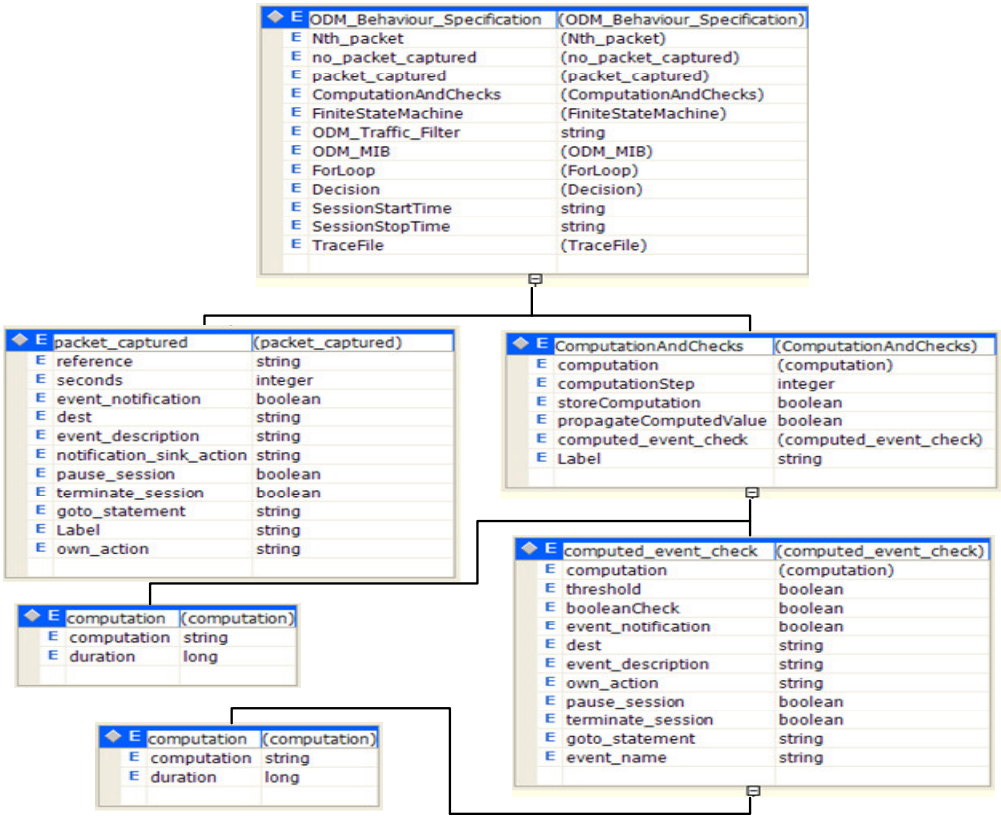


Figure 9: EDBSLang Schema tree (2)

<ForLoop>: (see **Figure 7**) - This tag is of type "`complexType`" and is used to specify a behaviour that iterates over the <NumberOfPackets> and performs some action (<own_action>). The "ForLoop" can also be specified based on a time duration parameter instead, by using the <seconds> tag. For example, a behaviour may be specified as follows: *for "300 seconds" perform action "X" on the captured traffic*.

<Decision>: (see **Figure 10**) - This tag is of type "`complexType`" and is used to specify a decision-based monitoring-behaviour that evaluates a specified <expression>. The behaviour to be executed if the expression evaluates to TRUE is specified using the <EvaluationTrue> tag. The behaviour to be executed if the expression evaluates to FALSE is specified using the <EvaluationFalse> tag. The other tags that can be included in the composition of such behaviour have already been described earlier.

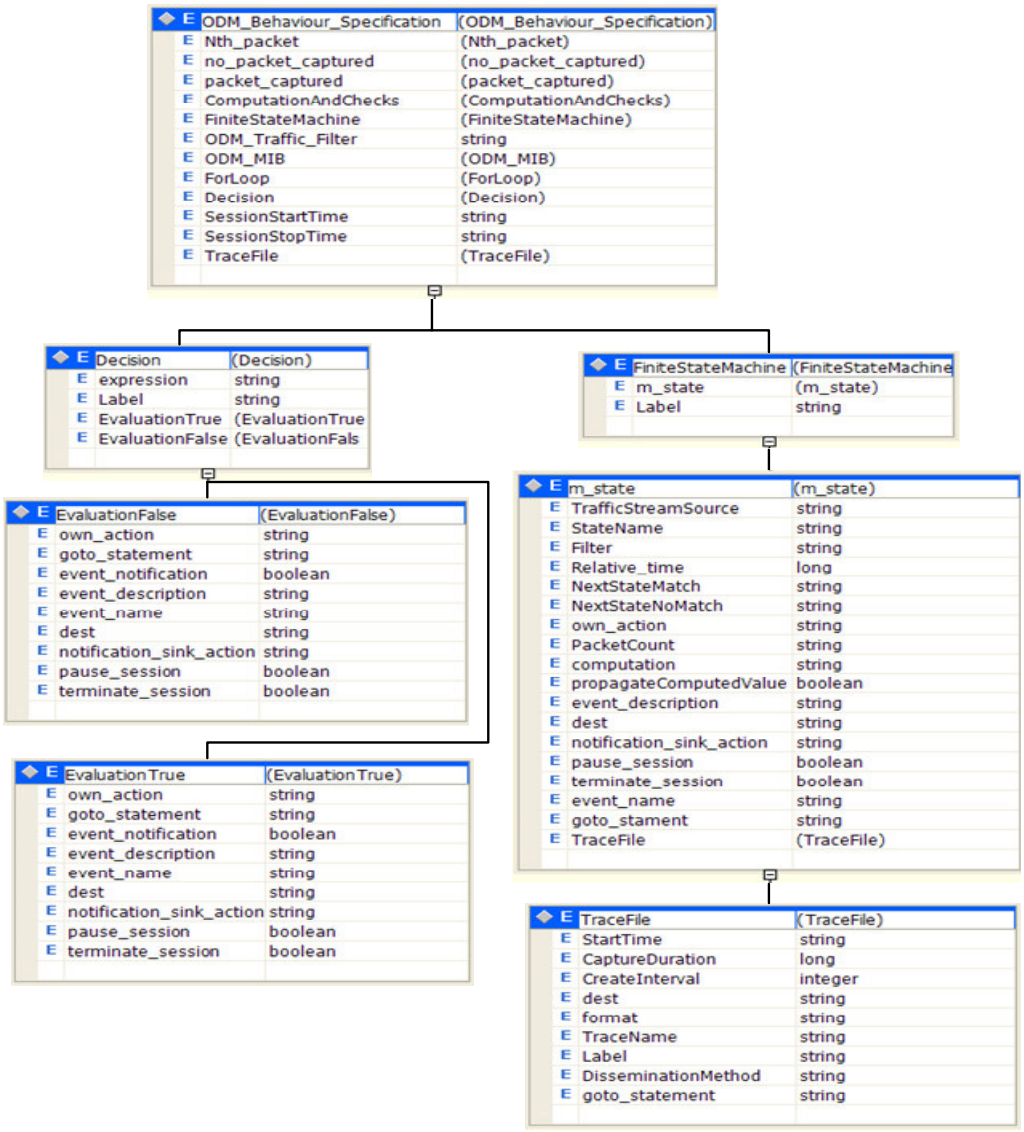


Figure 10: EDBSLang Schema tree (3)

All the above tags i.e. the child elements of the `ODM_Behaviour_Specification` tag can appear a number of times in a single behaviour composition. However, all constraints can easily be implemented and/or changed in the validator. Constraints in behaviour compositions may also have to do with the fact that some monitoring components or devices in some points in the network may be constrained by policies and/or capabilities or limitations by design e.g. a router may not allow i.e. accept certain monitoring-behaviour compositions due to potential processor overhead while a specially instrumented passive monitoring probe in the network may allow such composed behaviours. Therefore, instead of trying to have a common EDBSLang Validator applicable for all monitoring components, we propose that every monitoring component should

support the concept of a *dynamic EDBSLang Validator* that is specific to enforcing constrains of the monitoring component and can be generated by the component at run-time and exported to any environment or automated tasks where monitoring-behaviour compositions targeting the component can be produced and validated before sending them in a monitoring request to the monitoring component.

5.2.2 Some remarks

In this section, we presented a solution to the requirement for a composition language for programmable traffic flow monitoring for multi-service self-managing networks. We presented a composition language for specifying monitoring-behaviours, called the EDBSLang language. We provided the foundation and the motivation behind this language. In section 5.7 we present the application of the EDBSLang to programmable traffic flow monitoring in multi-service self-managing networks, as well as some examples of its use. The language will continue to evolve since the needs for programmable traffic flow monitoring can not be exhausted within a single step of research. We have also provided some answers to some of the questions that arise from specification based programmable monitoring as an alternative to the script based approach. The issues of enforcing constraints on the EDBSLang language constructs, enforcing constraints through a *validator*, the concept of *dynamic validators* specific to monitoring components, as well as a validator we implemented in Perl have also been discussed. One of the open issues to be still addressed as part of future work is to answer the question on the resource requirements imposed by certain monitoring-behaviour compositions on a monitoring component. This issue would require some analytical approach and some experiments where necessary. The other thing to be further investigated is how the always increasing network interface speeds interplay with the dynamics that may be involved in programmable monitoring. The future work, a subject discussed further in the **Chapter 7** where we present the ODM-Probe as a prototypical monitoring component that supports the ODM-Paradigm, include the algorithms for building trees of monitoring-behaviours uploaded and cached on a target and creating unique identifiers that can be used to indicate desired monitoring-behaviour to be triggered on the target, thereby avoiding pushing bandwidth consuming monitoring-behaviour-specifications, even though one can not expect significant bandwidth consumption in XML based communications.

5.3 On-demand creation and destruction of gathered monitoring-data and in-memory data models

In order to realize this principle (**Principle-P3**)**,** we developed the concept of On-Demand MIBs for the purposes of support for on-demand creation and destruction of in-memory data models of gathered monitoring data by a monitoring component.

In pursuit of an appropriate data modeling framework for the creation of dynamic in-memory data models, we seek data modeling frameworks which integrate well with existing network management frameworks such as SNMP are desirable. An on-demand data model may be required by an automated task requesting for the creation of a monitoring session on a monitoring component. The on-demand data model(s) should be destroyed when it is no longer required by the session-owner i.e. when the monitoring-session has expired. In [Chaparadza06a] we introduced the concept of On-Demand SNMP MIBs, which qualify very well as data modeling framework for On-Demand Data Models and is described in more detail later in this section. The requirement for data models should be expressed in the monitoring-behaviour-specification(s) uploaded into the targeted ODM-capable monitoring component by a *session-creator(s)*. The language used for specifying a *monitoring-behaviour-specification* must support the possibility of specifying the elements of the data model required to be created by a monitoring-session. In [Chaparadza07a] and in section 5.3.1.1 we illustrate how the EDBSLang supports the specification of the required On-Demand MIB. In [Chaparadza06a] [Chaparadza07a] and in **section 5.3.1.1**, examples of how On-Demand SNMP MIBs (described in more detail in **section 5.3.1**) can be specified in a monitoring-behaviour-specification are given. An example has also been included in section 5.3.1.1 (see **Figure 12**).

As described earlier in the definition of **Principle-P3**, in the ODM-Paradigm some in-memory data model (knowledge derived and computed from the captured traffic e.g. *rate of flow, arrival-rate, or some other types of statistics*) may need to be created on-demand and destroyed later by an ODM-capable monitoring component requested to run a monitoring-behaviour. Such a requirement would depend on the monitoring needs of the automated task(s) triggering the monitoring on the ODM-capable component of interest. The monitoring needs of the automated task(s) with regards to on-demand data models are about the nature of the data model and its lifetime i.e. the duration from the time the data model must be created by the targeted monitoring component and the time when the automated task(s) no longer require the data model to be still stored in memory by the monitoring component. The automated task(s) would then query the data model during the duration the data model is required and would require a systematic and secure way to do that, with the possibility to integrate well with existing network management frameworks such as SNMP.

A Network Management System (NMS) may require triggering the monitoring of some traffic at some ODM-supporting probes placed in strategic points in the network and demand the creation of a data model(s) reflecting some derived traffic characteristics e.g. statistics, so that the NMS or other systems may query the data model during the period in which monitoring is required for say traffic engineering. In the sub-sections of this section we introduce the concept of On-Demand MIBs we developed as a solution to the requirement of dynamic data model(s) creation and deletion in On-Demand Monitoring.

In **section 5.3.1.3** we give some remarks and an insight into the necessary further research work.

5.3.1 On-Demand SNMP MIBs

An On-Demand MIB (Management Information Base) is a data model that is created on-demand on a data collection point and has a *lifetime* i.e. TTL (Time-To-Live). It is a MIB that is created to store information that is temporarily required and is destroyed when the information is no longer required, such as in the case of On-Demand Monitoring (ODM). On-Demand MIBs are different from today's MIBs that are implemented on routers, switches etc, in the sense that On-Demand MIBs are not static. The prefixies e.g. *1.3.6.1.3.X.y.z* of the Object Identifiers (OIDs) in the MIB are assigned dynamically by a component that assigns the root identifier (i.e the value of *X*) of a subtree hook to the universal MIB and manages the assignments and freeing of root identifiers of different subtree hooks that are instances of sub-MIBs attached to the universal MIB. The OIDs themselves may have a *lifetime* that is less than or equal to the *lifetime* of the On-Demand MIB itself. For example, an OID may be destroyed once it has been read by a remote entity via an *snmp-get* operation, in order to optimize the use of resources on the target. **Figure 11** shows how subtree hook assignments can be done, as shown by the *dotted* branches. The AgentX protocol [RFC 2741] is a protocol that allows runtime components called *SNMP sub-agents* to register MIB subtrees with a so-called *SNMP Master-agent* and receive requests targeting OIDs of which the sub-agent implements and can respond to the requests. An additional component that assigns root identifiers for subtree hooks to runtime components wishing to act the role of a sub-agent, would ensure that the root identifiers of all the sub-tree hooks are unique. Because Object Identifiers (OIDs) in an On-Demand MIB are assigned dynamically as opposed to the case in static MIBs, the monitoring component should therefore implement *an algorithm* that takes care of defining the On-Demand MIB in SMI (Structure of Management Information) [RFC 2578] and advertising the MIB definition to the parties interested in querying this MIB to enable entities to access the On-Demand MIB.

However, On-Demand SNMP MIBs can co-exist with the "traditional MIBs" of a system, whose Object-Identifiers (OIDs) are known and defined "*a priori*", and are statically instantiated (not on-demand) by the system, as known in today's current practices in MIB engineering.

The other characteristic of On-Demand MIBs is that they also include the creation of some OIDs meant to covey event notifications via the use of *SNMP-Inform* and *SNMP-Trap* messages that include the description of the event(s) without the need to explicitly define and advertise a MIB that would then include the definition of such OIDs.

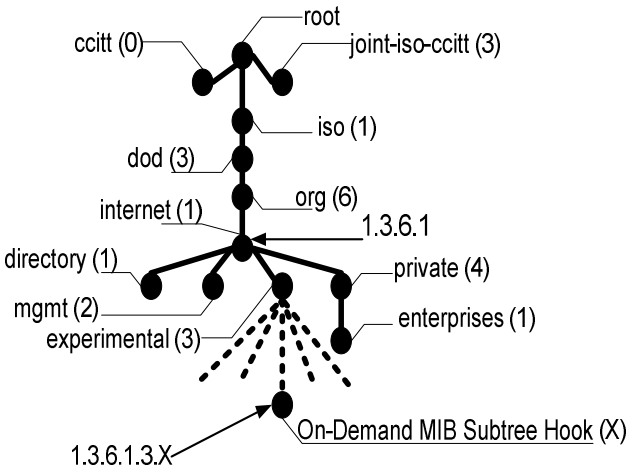


Figure 11: On-Demand MIB subtree hook assignments

The other interesting characteristic about On-Demand MIBs is that an On-Demand MIB may continue to add OIDs during its lifetime and the OIDs can be advertised after the initial advertisement of some OIDs which were already known at some point. To illustrate this: (1) imagine that a data collection point supporting the creation of On-Demand MIBs has created some OIDs that a remote entity requested(via the use of *monitoring-behaviour-specification*) to be created e.g. some OIDs implementing counters for some statistics, and the data collection point has already advertised the OIDs to the remote entity that requested the creation of the MIB; (2) then a local application that the data collection point invoked due to the occurrence i.e. detection of an event specified in the *monitoring-behaviour-specification*, has finished execution and would like the results(data) to be also added to the MIB; (3) then some additional OIDs would be created and advertised to the remote entity. Such a data collection point can be a traffic capture and analysis probe (for example) that can create certain OIDs e.g *Traffic Statistics Table* OIDs and provides an interface that allows local applications that it might need to invoke as specified by a remote entity in a *monitoring-behaviour-specification*, to register additional OIDs if they wish. In this research, we have identified the need to extend the API (Application Programming Interface) used by AgentX sub-agents to register OIDs with the *master-agent*, by providing an additional *AgentX-Wrapper* API that allows applications (other types of runtime components) that do not need to run as sub-agents, to register with a sub-agent some data that other entities can then query via SNMP. This will allow so many already existing monitoring applications that operate on traffic traces produced via the use of libpcap [libpcap] or similar libraries, to make their computations available to other systems via SNMP, by simply having them modified to register whatever type of OIDs (*scalars or tables*) they may wish to make available via the use of this *AgentX-Wrapper* API we also introduce. The *MIB advertisement algorithm* should support advertisements of MIBs as an SMI (Structure of Management Information) file(s) and advertisement of additional OIDs via the use of SNMP Inform messages or Trap type of messages that encode the textual representation of an OID.

What sort of data collection (monitoring) devices can we envision to support On-Demand MIBs? These can be traffic capture and analysis probes that can gather rich monitoring data and/or produce some statistics that can not be gathered from routers and switches. Passive monitoring techniques [PassiveActiveMonit] [CoralReef] have demonstrated the possibility to gather richer fine-grained monitoring data, detect traffic-related events and compute some statistics beyond the SNMP coarse-grained statistics available from routers, switches etc. So-called non-router based monitoring techniques supported by the use port mirroring or spanning [MirrorAndSpan] and link-taping [CoralReef] provide the possibility of both programmable and flexible monitoring such as is the requirement for On-Demand Monitoring using even standard PCs. Because the main idea behind On-Demand MIBs is about the creation of in-memory (dynamic) data models that have a lifetime, driven by a request and its requirements, we are very likely to see routers in the future also supporting the concept of On-Demand MIBs in addition to their static MIBs. On-Demand MIBs can co-exist with traditional MIBs of a box, whose Object-Identifiers (OIDs) are "known" (defined) "*a priori*" and statically instantiated (not on-demand).

5.3.1.1 Types of Object Identitifiers (OIDs) i.e. elements of On-Demand MIBs

In the following paragraphs we describe the three types of Object Identifiers (OIDs) that pertain to On-Demand SNMP MIBs i.e. elements of On-Demand MIBs. The concept of On-Demand SNMP MIBs was implemented and evaluated in the implementation of an ODM-supporting monitoring component called the ODM-Probe—presented later in Chapter 7. The types of OIDs currently supported by the current implementation of the ODM-Probe are all described in more detail in Chapter 7, including details of how they are implemented by the ODM-Probe.

I. EDBSLang-specifiable OIDs i.e. those that can be specified by session-creators/owners (see Table 5)

These are OIDs (**see** Table 5) that can be specified in the monitoring-behaviour-specification using special EDBSLang tags as described in **section 5.2.1** as well as in [Chaparadza07a]. The kind of OIDs presented here are simply example OIDs we considered only for prototyping purposes, meaning that other types of OIDs can be defined in the further development of both, the concept of On-Demand SNMP MIBs as well as the ODM-Probe. The OIDs presented here apply to a monitored traffic flow specified by an "ODM-Traffic Filter" in a monitoring-behaviour-specification.

Table 5

Node(Hook) of attachment to the On-Demand MIB tree	Name of the OID
1.3.6.1.3.**X.2**	*ProtocolHierachyStatisticsTable:* [stores statistics on each layer of a fully decoded packet, for each of the packets captured and analyzed in the monitoring duration for which the Table is required]. *OID_Type → Table*.
1.3.6.1.3.**X.3**	*CurrentRateOfFlow:* [Stores the current Rate of traffic flow requested for monitoring in a monitoring-behaviour-specification]. *OID_Type → Scalar*.
1.3.6.1.3.**X.4**	*GenericEventDescription:* [Stores an "event description" of an "Event" requested for detection in monitoring-behaviour-specification]. *OID_Type → Scalar*.
1.3.6.1.3.**X.5**	*GenericPacketHexdump:* [Stores a hexadecimal representation of a dumped packet—useful for a diagnostic type of packet captured for export and offline analysis]. *OID_Type → Scalar*.
1.3.6.1.3.**X.6**	*PacketSummaryTable:* [Stores a summary of a number of sampled packets decoded and described in a textual form. This is useful in the case of diagnostic packets under observation]. *OID_Type → Table*.
1.3.6.1.3.**X.12**	*PacketCounter:* [Stores the number of packets belonging to a traffic flow under monitoring, captured from the time the flow observation started]. *OID_Type → Scalar*.
1.3.6.1.3.**X.13**	*PacketArrivalRate:* [Stores packet arrival rate of a flow in packets per second]. *OID_Type → Scalar*.
1.3.6.1.3.**X.1**	*DetectedEventsTable:* [Stores the descriptions of Events detected, from the Events requiring detection, as specified in a

	monitoring-behaviour-specification]. *OID_Type → Table*.

II. Automatically created OIDs—created by an ODM-supporting Monitoring Component—mainly for storing ODM-Session related Information

Except for the "*Time*" OID, these OIDs (see Table 6) should be created for each monitoring-session running on an ODM-supporting monitoring component such as the ODM-Probe.

Table 6

Node(Hook) of attachment to the On-Demand MIB tree	Name of the OID
1.3.6.1.3.**X.7**	*Time:* [Stores the current Time on the monitoring component]. *OID_Type → Scalar*.
1.3.6.1.3.**X.8**	*ErrorsTable:* [Stores the Errors pertaining to a monitoring-session detected by the monitoring component]. *OID_Type → Table*.
1.3.6.1.3.**X.9**	*FailuresTable:* [Stores the Failures pertaining to a monitoring-session experienced by the monitoring component]. *OID_Type → Table*.
1.3.6.1.3.**X.10**	*SessionDescriptionTable:* [Stores Information about a particular monitoring-session belonging to a session-owner]. *OID_Type → Table*.
1.3.6.1.3.**X.11**	*SessionStatus* [Stores status information of a so-called dynamic OID registration session created by an entity running on the monitoring system and intending to add OIDs to the MIB tree (e.g. a monitoring application or "Action-Script" invoked by the monitoring component)]. *OID_Type → Scalar*.

III. Dynamically registered OIDs (see Table 7)

These are OIDs for which an ODM-supporting monitoring component such as the ODM-Probe must provide a special API(Application Programming Interface) that allows entities running on the box, in particular Action-Scripts invoked by monitoring-session-behaviours to register new Object Identifiers (OIDs) into the MIB tree.

Table 7

Node(Hook) of attachment to the On-Demand MIB tree	Name of the OID
1.3.6.1.3.**X.20**	*DynamicOIDregistrations* [A branch reserved for dynamic OID registrations]
1.3.6.1.3.**X.20.1**	*RegistrationSessionNode* [A node assigned for a particular dynamic OID registration session among multiple sessions allowable]
1.3.6.1.3.**X.20.1.1**	*Generic"Scalar1" or Generic"Table1"* [For use in registering any Scalar or Table OID being dynamically registered]
1.3.6.1.3.**X.20.1.2** . . . 1.3.6.1.3.**X.20.1.N**	*Generic"Table2" or Generic"Scalar2"* [For use in registering any sub-sequent Scalar or Table OID being dynamically registered i.e. for a number of OIDs being sequentially registered into the MIB tree]

```
<ODM_Traffic_Filter>“ip proto udp” </ODM_Traffic_Filter>
< ODM_MIB>
<table_OID> Protocol_Hierachy_Statistics
traffic_capture_window_size=”900s”
        </table_OID >
        < table_OID > EventsDetected_Table
        </ table_OID >
        <scalar_OID> Current_RateOfFlow </scalar_OID>
        <scalar_OID> PacketCounter </scalar_OID>
        <scalar_OID>Generic_Event_Description
        </scalar_OID>
        <scalar_OID>Generic_Packet_HexDump</scalar_OID>
< /ODM_MIB>
```

Figure 12: Example of a fragment of a monitoring-behaviour-spec in EDBSLang that includes the specification of a required On-Demand MIB

In **Chapter 7**, we provide a functional specification of the ODM-Probe, which supports what we call *known (*EDBSLang-*specifiable) ODM_MIBs* or so to say, *known (specifiable) OIDs* (see example OIDs in the Tables and **Figure 12** above) that can be specified using EDBSLang language by a ODM-Session-owner (session-requester) and are then instantiated by the probe. These types of OIDs have been described earlier in this section. **Figure 13** shows how an On-Demand MIB definition in the SMI language looks like. The ODM-Probe assigns the identifiers of the specified OIDs and then advertises the OIDs (i.e. the On-Demand MIB) to the ODM-session-owner and to other entities (automated tasks) designated in the monitoring-behaviour-specification by the session-owner, in order to enable them to read (access the On-Demand MIB). The other types of dynamic OIDs required for support by an ODM-capable component such as the ODM-Probe, are what we call *generic OIDs*. These are OIDs that an “*Action*” e.g. an “*Action-Script*” invoked by a monitoring-behaviour on the targeted monitoring component can register as say additional OIDs via an *AgentX-Wrapper* API that we developed to support the creation and registration of OIDs by scripts triggered by monitoring-behaviours on a monitoring component. These types of OIDs have also been described earlier in this section. More detailed information on this subject is found in **Chapter 7**.

```
------------  Current Rate of flow of ODM specified traffic  ---------------
rateOfFlow OBJECT-TYPE
        SYNTAX  Unsigned32
        MAX-ACCESS  read-only
        STATUS  current
        DESCRIPTION
                "The computed Rate of Flow of the specified traffic."
        ::= { ODMSessionSubTreeHook 1 }
------------------- Event Notification ---------------------------------
odmDescribedGenericEvent NOTIFICATION-TYPE
        OBJECTS {EventTimeStamp, EventDescription}
        STATUS  current
        DESCRIPTION
                "An event described by the ODM session-owner"
        ::= { ODMSessionSubTreeHook 2 }
---- The Protocol Hierarchy Statistics Table --------------------------
ProtocolHierarchyStatisticsTable OBJECT-TYPE
        SYNTAX SEQUENCE OF protocolStatisticsEntry
        DESCRIPTION   "This table contains entries for statistics e.g. packet/byte counts,
                        bandwidth consumption etc.  of each protocol within the stack,
                over a capture duration(window) specified by the ODM session-owner."
        ::= { ODMSessionSubTreeHook 3 }
---- The Detected Events Table, including the corresponding Timestamps ----
EventsDetectedTable  OBJECT-TYPE
```

Figure 13: An example fragment of an example SMI definition of an On-Demand MIB

5.3.1.2 Limitations of On-Demand MIBs

On-Demand MIBs inherit the same types of limitations as today's staticically defined MIBs—the differences of which have been described in this section. However, there are some key limitations specific to On-Demand MIBs we highlight here. Just like for static MIBs, once an OID has been assigned (a leaf in the MIB tree i.e. a number in the dot-notation), the mapping between an "OID name" and the OID can not be changed. With the exception of a case described later in this section, the need for the mappings requires that for On-Demand MIBs, a SMI MIB definition and advertisement algorithm is required on the monitoring component in order to communicate the mappings between an "OID name" and the actual OID (in the dot-notation) defined on the fly, to interested entities such as monitoring-session-owners (i.e. Automated Tasks of the network). This is because, as was discussed earlier in the sub-sections above, On-

Demand MIBs are not pre-defined. MIB tree hook assignments (e.g. `1.3.6.1.3.`**`"X".2`**) are done by the monitoring component on the fly. However, there is one exceptional case for which OIDs of an On-Demand MIB do not require an explicit SMI definition and advertisement to interested entities by use of an SMI definition file. This exceptional case applies to OIDs whose values only need to be communicated using "Notification type of messages" such as SNMP-INFORM or SNMP-Trap type of messages, since the definition of the OID itself can simply be embedded in the notification message as sub-string part of a "notification-description", apart from the actual value of the OID being conveyed by the message. In chapter 6—section 6.3.1, we provide examples of such cases.

In SNMP, the use of "OID names" instead of the OIDs themselves enables programmability, and this is especially very important for On-Demand MIBs than for today's static MIBs. The MIB tree hooks assignments created by the monitoring component (e.g. `1.3.6.1.3.`**`"X".2)`** are of type **`integer64`** meaning each of the allocated slot e.g. **`"X"`** is limited by the **`maximum value`** of the **`integer64 type.`** In the case of the dynamically registed OIDs, the time between the successful registration of an OID or set of OIDs and the time the "OID to Name mappings" are received by an interested entity, determines the time an OID is ready for reading its actual value by the interested entity. Factors affecting the scalability of static MIBs would also apply to On-Demand MIBs.

5.3.1.3 Some remarks

In this section we described how On-Demand MIBs are very useful in the context of On-Demand Monitoring (ODM). Though the focus of this dissertation is on On-Demand Monitoring of traffic flows, nothing stops the concept of On-Demand MIBs from being applied to other areas where dynamic data models can be needed. In **Chapter 7**, we present the functional specification of the ODM-Probe in which we implemented the concepts behind On-Demand MIBs. As such the ODM-Probe is the only system that supports the concept of On-Demand MIBs. In **chapter 6—section 6.3.1** we present an example application scenario for the application of On-Demand MIBs to traffic engineering. Regarding further research on the subject of On-Demand MIBs, a number of issues still need to be further researched such as performance and scalability evaluations, especially at data collection devices and points in the network that need to support a number of ODM-sessions running in parallel, with each session creating its own unique On-Demand MIB. All such issues are the subject of our further research in this regard.

5.4 Triggerable, Configurable and Re-Configurable Monitoring

In order to realize this principle (**Principle-P4)**, we introduce the concept of *parameterized primitives* illustrated below, (similar to primitives we introduced in [May07]), that must be supported by a monitoring component to enable an *automated task* to create the notion of a *monitoring-session* on the component and manage the *monitoring-session:*

```
//This IDL Specification serves only for illustrative purposes—implementation specific
details are presented later in chapter 7.

interface OnDemandMonitoringPrimitives_for_Montoring_Component {
        struct monitoring-behaviour-specification{ //See COMMENT_1 below
                string file_name;
                attribute_type_1 attribute_1;
                attribute_type_2 attribute_2;
                attribute_type_N attribute_N; //The attribute types is determined by the
                                //Composition Language used for specifying monitoring-behaviour

       /*------COMMENT_1:---------------*/
                //A monitoring-behaviour-specification can be based on the EDBSLang Composition
Language, and must include specification of the “Traffic-Flow-Filter”,
                //and optional specifications of any “Events” to be detected,
                //any “Actions” to be performed on some specified “conditions”,
                //and the “data models of monitoring data” required]; Time-To-Live[TTL]
        }
        struct  parameters_of_the_monitoring-session {
                param_type_1 param1;
                param_type_1 param2;
                param_type_N paramN;
        };

  // The attributes.
  readonly attribute string theAssigned-monitoring-session-Identifier;
  readwrite attribute string Traffic-Flow-Filter;

  // The Primitives/Operations
  int start-session (in monitoring-behaviour-specification monitoring-behaviour-spec);

  int pause-session(in monitoring-session-ID);
  int resume-session(in monitoring-session-ID);

  int modify-session((in monitoring-session-ID, in parameters_of_the_monitoring-session
New_session_parameters);

  int modify-session((in monitoring-session-ID, in monitoring-behaviour-specification
New_monitoring-behaviour-spec);

  int refresh-session (in monitoring-session-ID; unsigned long long new_TTL). //refreshes the
session with a new value for Time-To-Live

  int terminate-session(in monitoring-session-ID);
  int set-filter(in monitoring-session-ID, in/out Traffic-Flow-Filter);
  int monitoring-Query (in string Query_formulation); //see COMMENT_2 below
  /*------COMMENT_2:---------------*/
  //This is to enable automated tasks to issue queries to a monitoring component,
  //such as: “what is the rate of UDP traffic flow at time_x, from the current time”.
};
```

Regarding the primitive *“modify-session*(*session-id, new parameters* [or *a new monitoring-behaviour-specification*])”, the question of how monitoring-behaviour modification by an automated task plays with traffic speeds and link speeds is an issue that needs to be considered

when designing the automated task in question. As such, some automated tasks may require modification of a monitoring-behaviour installed at a given monitoring point, regardless of how the modification plays with the traffic speed or link speed. For example, some automated troubleshooting tasks, automated network-debugging tasks or automated Fault-diagnosing tasks, may require injecting some test or diagnostic packets into the network, and then modify the monitoring-behaviour according to the events and monitoring data communicated by the monitoring components requested to first observe the test or diagnostic traffic and detect traffic presence or some other derived events.

These primitives, referred to as *ODM-primitives*, allow a session-owner to ensure that resources are freed whenever monitoring is temporarily not required or no longer required by issuing appropriate primitives to the monitoring component running the requested behaviour. Also, values of parameters of monitoring functions must be modifiable on-the-fly whenever a *monitoring-session-behaviour* requires modification by the session-owner. **Principle-P4** and the *ODM-primitives* presented here, as well as all the principles defined in this chapter, are incorporated into the functional specification of the ODM-Probe—a prototypical monitoring component that supports the ODM-Paradigm that is presented in detail in **Chapter 7**.

5.5 Intelligent and Opportunistic use and allocation of resources

As a solution to realizing this principle (**Principle-P6**) we propose to design a monitoring component such that it checks whether resources are available to satisfy the requirements expressed in the monitoring request (monitoring-session-behaviour spec) and perform admission control on monitoring requests. We propose to apply *admission control* to monitoring requests and requested behaviours based on, say, resource availability to satisfy the demands of the monitoring-behaviour requested for execution. Together with its implemented monitoring functions the component must ensure that a monitoring-behaviour can be shared by a number of monitoring-requests (from multiple automated tasks i.e. session-owners) if possible and should avoid starting additional behaviours or function threads unnecessarily. We call this "*session-merging"*. Upon the reception of *pause-session* and *terminate-session* commands (primitives) from an automated task, resources seized by the monitoring component from the overall system resources for the monitoring behaviour(s) being paused or terminated must be freed.

The idea behind the "session-merging", carried out by the monitoring component, is to let session-owners have the notion that they are managing their sessions without them having to know that the sessions are sharing some threads or processes of execution on the target or that a set of session-behaviours are merged into an aggregate behaviour that covers the aggregate needs of the associated set of monitoring-behaviour-specifications, according to some aggregation i.e. merging policy i.e. criterion. The aggregation/merging policy i.e. criterion may have to do with

say, all monitoring-behaviour-specifications specifying the same traffic capture filter result in a dedicated traffic filter specific thread being created to serve a number of sessions belonging to multiple session-owners. This would make the monitoring component intelligently use resources by also ensuring that whenever a session-owner issues any of the following primitives: *pause-session, refresh-session resume-session* and *terminate-session*, the monitoring component checks to see if there are any sessions still running on the shared resources e.g. threads, and ignore the issued primitives if the shared resources are still in use by other sessions. If there is no monitoring-session still associated with the shared resources, except the session for which any of the above session-management primitives has been received, then the monitoring component respects the desired effect of the primitive(s).

5.6 Self-description and Self-advertisement of Capability Models

As a solution to realize this principle (**Principle-P7**): we introduce the concept of *Capability-Model Description,* and its *self-description and self-publishing* by the monitoring component, to the network. We developed the *Capability Model Description Language* (CMDL) that can be used by ODM-capable monitoring components to self-describe their *Capability Models* and publish them to the network. CMDL is a meta-language (an XML Schema) for describing capability models of ODM-capable components or devices i.e. *ODM supporting devices (routers, switches, hosts, signalling gateways, special probes instrumented in a network, etc), diagnostic/test traffic flow generators and sinkers.* All these types of components and network devices may be required in some combination to participate in a monitoring objective spanning diverse components and devices i.e. monitoring points, as may be required by an automated task(s). Capability Models enable automated tasks to select monitoring components of desired capabilities and create monitoring-sessions whose behaviour specifications can be specified using the EDBSLang language and uploaded along with a monitoring request to the target monitoring component for execution. Capability models enable automated tasks of the network to view a collection of capability-model exporting monitoring components and devices as a monitoring platform, so that appropriate *request-primitives* can be issued to individual components or devices during an on-demand monitoring need.

CMDL can still be further evolved as research in this subject continues. The *self-description* and *self-publishing* of capability models by a system or component should be subject to security policies. More information regarding self-description of capability models by monitoring components, the kind of information that must be self-described, the kind of changes that may occur in capabilities, and the need to update the network when capabilities change, is given in the next chapter.

The issue of security in the descriptions and publishing of capability models by monitoring components was not covered in this research and would amount to a subject for further research. **Appendix B** presents the CMDL meta-language (an XML Schema).

5.7 Programmable Traffic Flow Monitoring in Multi-Service Self-Managing Networks

In this section we present how EDBSLang can be applied to programmable traffic flow monitoring in multi-service self-managing networks, as well as some examples of its use.

First we discuss on what programmable traffic flow monitoring in multi-service self-managing networks entails. To enable programmable traffic monitoring in such networks, it is important to have all network elements (see definition of *network element*$_{re\text{-}defined}$) in the network e.g. hosts, routers, switches and gateways, that have monitoring capabilities expose information about such capabilities to other systems so that appropriate monitoring requests can be issued. This means that the systems i.e. devices posing as potential monitoring points in the network should provide some interfaces through which *monitoring requests* can be injected and through which monitoring data can be retrieved. This also means that systems i.e. automated tasks anywhere in the network, intending to trigger monitoring on some monitoring points should first have the possibility of locating a monitoring point on the network topology in order to select a monitoring point i.e. a monitoring component of interest. **Principle-P7** talks about systems self-describing their monitoring capabilities to the network via the use of *capability models* that are then stored in specially instrumented databases across a self-managing network. A capability model can be solicited for and should be automatically updated by the system depending on factors such as changes to the configuration, topology, and policies at the monitoring point. The other thing required is a way to formulate the monitoring requests, either as *commands i.e. primitives* to be issued by an automated task to a selected monitoring point(s) or as *monitoring-behaviour-specifications* or *scripts* to be pushed into the monitoring point, which then parses and executes them. In case a monitoring component at a particular point in the network stores monitoring-behaviour-specifications it already knows from the history of monitoring-behaviour-specs previously uploaded into its local repository or cache, then, assigning unique identifiers to such possible cached monitoring-behaviour-specs would give a way for a potential session-owner(creator) to indicate the desired monitoring-behaviour from the list of already known monitoring-behaviour-specifications without having to upload a behaviour-spec to a target component for execution. This would help in ensuring that bandwidth is not consumed unnecessarily by a remote system (session-owner) intending to send a monitoring-behaviour-specification into the monitoring point for execution. Instead, a desired monitoring-behaviour is simply indicated by using behaviour identifiers.

As already known in this section, EDBSLang would play a role in the specification of monitoring-behaviours to be executed on some monitoring points. The question is: Who can use EDBSLang to create specifications of monitoring-behaviours for multi-service self-managing

networks? The answer is: The specifications can be engineered and authored by humans and can also be produced programmatically by systems (automated tasks) intending to trigger i.e. request monitoring on some targets.

Humans i.e. network management experts can decide to study the monitoring needs in their multi-service network that require programmability, and author appropriate monitoring-behaviour-specifications, which are then uploaded into specific monitoring points in the network. They may also develop some automated tasks that run in some network elements (see definion of *network element*$_{re\text{-}defined}$), whose behaviour may also require selecting some monitoring points in the network and triggering a monitoring-behaviour(s) that is based on a concrete pre-uploaded authored specification. The systems i.e. automated tasks triggering specific monitoring-behaviours at selected points could be driven by some contexts such as adaptive configuration management functions, network-usage demands per time of day or per day of the week, etc, thereby selectively triggering some monitoring-behaviours in order to achieve some goals in the self-managing operations of the network. To enable humans to engineer and author some monitoring-behaviour-specifications using the EDBSLang language, an appropriate editor, most appropriately a graphical one, that understands the language can be developed and integrated with the EDBSLang *Validator* that checks constraints to allow well-formed specifications to be produced. Now, how do monitoring capabilities of a monitoring point on which a monitoring-behaviour-specification is to be uploaded interplay with the uploading and acceptance process? The monitoring capability of a target monitoring point (device) can also be determined by some policies that may actually not allow certain monitoring-behaviours to be permissible. Therefore an algorithm that verifies permissibility of a monitoring-behaviour-specification against the monitoring capability model of the monitoring component and/or policies on the target needs to be developed.

Compositions of monitoring-behaviours i.e. monitoring-behaviour-specifications for some target points can also be produced programmatically by automated tasks themselves, either embedded into some protocols or in applications. This would require that the automated task understands the EDBSLang schema in order to instantiate it i.e. compose a monitoring-behaviour-spec(s), and also needs to understand the monitoring capability (expressed by a capability model) of a target monitoring point (a device or software component) for which the monitoring-behaviour is intended.

One of the other problems to be solved in composition language based programmable monitoring is the need to conserve bandwidth by ensuring that a monitoring-behaviour-specification is only sent to a target by a remote session-owner (automated task) when the target monitoring component does not already have this behaviour in its repository or cache, otherwise the desired behaviour should be indicated through some behaviour identifier. For this problem, we propose to have an algorithm that runs on every potential monitoring point, goes through all the uploaded monitoring-behaviour-specifications in the repository or cache and creates a tree of monitoring-behaviours, whereby a node is a position (a sequential indexing similar to line numbers in a textual file) in a specification where an EDBSLang tag (and its elements) appears, and a leaf is determined by the last tag in a composition branch. The algorithm must then create a unique

identifier for *every whole (complete) path from the root node to a leaf node* in the tree. These identifiers of monitoring-behaviours can then be included in the monitoring capability model disseminated or advertised by the monitoring point to the network. Remote systems (automated tasks) can then use the identifiers to indicate desired monitoring-behaviour to be triggered on the selected target, without having to send bandwidth consuming specifications. Such an algorithm is part of our future work. This would also include cases whereby a remote entity i.e. an automated task sees an identifier of a monitoring-behaviour-specification on some selected target and intends to flexibly change some of the settings or add a new tag(s) to a selected behaviour, which should then be triggered on the selected target. This subject is discussed further in **Chapter 7**, in which we present the ODM-Probe as a prototypical monitoring component that supports the ODM-Paradigm. The ODM-Probe, presented in detail in **Chapter 7** and briefly in [Chaparadza06a] [Chaparadza05b] [Chaparadza05d], and will continue to evolve, is the first monitoring component architecture to support the EDBSLang.

5.7.1 An example scenario on the use of the EDBSLang Language

The example below (see **Figure 14**) shows a fragment of an example EDBSLang-based monitoring-behaviour-specification that specifies that: UDP traffic be monitored; if some UDP packet/traffic has been seen after 30 seconds then send notifications to two designated recipients indicating the action they need to execute; a *Protocol-Hierarchy-Statistics MIB Table* be created for later reading by the session-owner; some computations on the *Rate of flow* be performed every 600 seconds and the value be propagated to the session-owner, computed values must also be stored for later reading , and if the Rate of flow is greater than 1Mbit/s threshold then an Event Notification must be issued and the specified “Action” must be taken by the installed monitoring-session-behaviour.

```
<ODM_Traffic_Filter>“ip proto udp” </ODM_Traffic_Filter>
<packet_captured reference=”from_capture_start_time” seconds=”30s” event_notification=“yes”
        notification_dest =”ipaddr1, ipaddr2” event_description=”30 seconds elapsed from start
        capture and a packet has been captured” notification_sink_action=”ScriptY.pl”>
</packet_captured>
< ODM_MIB>
        <table_OID> Protocol_Hierachy_Statistics </table_OID >
< /ODM_MIB>
<ComputationAndChecks computation=”Rate” computationStep=”600s”
        storeComputations=”Yes” propagateComputedValue=”yes”>
        <ComputedEventCheck computation=”Rate”  threshold=”1Mbit/s”
                        booleanCheck=”threshold < Rate”
                        event_nofitification=”yes”
                        notification_dest=”ipaddr1, ipaddr5” event_description=”Rate exceeds
        threshold” own_action=”ScriptG.pl”>
        </ComputedEventCheck>
</ComputationAndChecks>
```

Figure 14: Example of a fragment of an EDBSLang based monitoring-behaviour-specification

5.8 The Conceptual Architecture of an ODM-Capable Monitoring Component, and a reflection on the GANA Model

Regarding design principles, what ODM also entails is that *monitoring functions* (threads or processes of execution, code-level functions) are designed following the *functional composition* approach. This is an approach that allows some functions to be composed via a composition language such that the functions can invoke additionally required functions on-demand, creating new threads or processes if necessary and where necessary. This also requires the sharing of services provided by threads or processes created by early service-requests (invocations). The *monitoring functions* we are considering here can be a *packet capturing function, diagnostic-flow(s) generator function, monitoring-data management function, event detection and computation functions, flow sinker function, performance and resource optimization function, security handling function, etc* invoked on-demand.

Figure 15 shows the conceptual architecture of a component that we consider as fundamental architecture for supporting the ODM-Principles. It shows the interfaces between components (*dotted arrows*) and the information flow between components (*solid arrows*). We call a monitoring component that supports the ODM-Principles (**Principle-P1 to Principle-P7**) described in the previous sections, an *ODM-Capable Component (i.e. an ODM-Supporting*

Component). Such an ODM-Capable Component may be implemented as forming the **traffic monitoring subsystem** of system or device. A system embedding such an ODM-Capable component is an ODM-Capable system. It may be a router, a host, a switch, a signalling gateway *or* a special traffic monitoring probe or a component embedded in such kind of systems. In **Chapter 7**, we present the functional specification of the ODM-Probe that derives from this conceptual architecture.

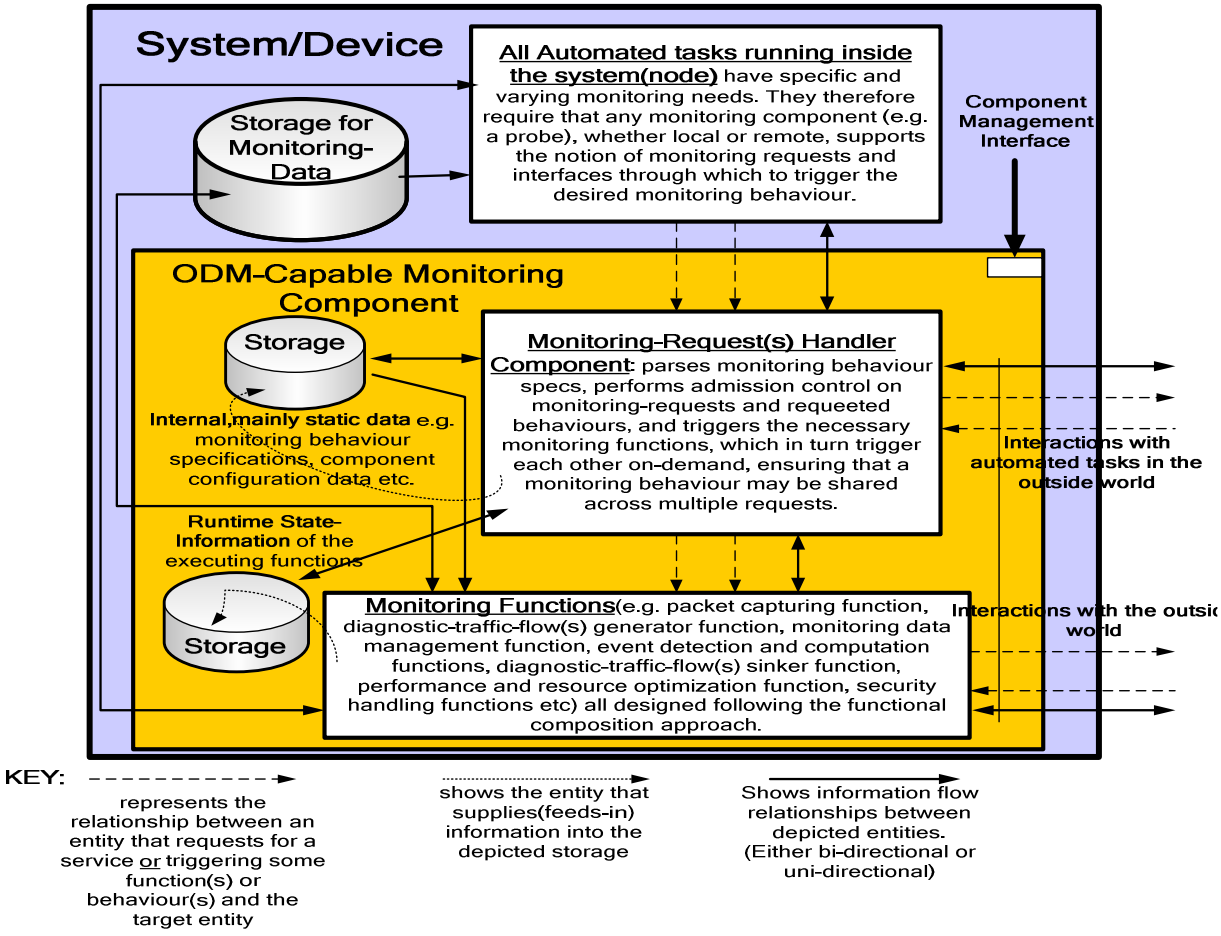


Figure 15: The Conceptual Architecture of an ODM-Capable-Component

Applying some ODM-Principles into the GANA Reference Model

The GANA Reference Model [Chaparadza08b] [Chaparadza09] [Chaparadza10a] defines a "Function-Level" Decision Element (DE) that must be embedded into a *network element*$_{re\text{-}defined}$ architecture called the Monitoring_Decision Element (Mon_DE) [Liakopoulos10] [Zafeiropoulos09] [Liakopoulos08] , which must implement a control-loop over the "management interfaces" of monitoring protocols and mechanisms as its associated Managed Entities (MEs). The DE must apply configuration profiles (which include policies) on this type of MEs, and then react to incidents, state changes and context changes by communicating with

other DEs to enforce changes on the behaviour of various types of MEs of the network elements (see definition of *network element*$_{re\text{-}defined}$) to ensure optimal conditions of network operation. The MEs are orchestrated and parameters of the MEs are dynamically adjusted by the Monitoring-DE(s). A "Network-Level" Monitoring Decision Element may also be designed to work with "Function Level" Monitoring DEs in network elements (see definition of *network element*$_{re\text{-}defined}$). From the perspective of traffic monitoring, the ODM-Principles and the ODM-Probe described in chapter 7 can be incorporated into the design of Monitoring-DEs.

5.9 Contrasting ODM to Related Work

In this section, we provide a contrast of the technical solutions proposed for the realization of the ODM-Paradigm as presented in this chapter, to other types of monitoring approaches that support some limited forms of On-Demand Monitoring.

It is worth mentioning that the principles of the On-Demand Monitoring paradigm defined in this research work are meant to set a path for the evolution of monitoring systems and components to suit the needs of self-managing networks. This means, the key issue is how to make today's monitoring paradigms; approaches and systems evolve towards fully supporting the ODM-Principles defined in this dissertation. Therefore, in this section, our review on related work is mainly focused on comparing the ODM-Paradigm in its holistic picture, to some approaches that inherently have some limited forms of On-Demand Monitoring or should be considered for the creation of evolved monitoring components that conform to the ODM-Principles (**Principle-1** to **Principle-P7**). In the paragraphs that follow, we select the well-known approaches that maximally capture some characteristics common to most of today's approaches to traffic monitoring. The subject of the limitations of today's approaches to traffic flow monitoring, with respect to supporting the demands imposed on traffic monitoring components and platforms by automated tasks meant to drive multi-service self-managing networks, has been addressed in **Chapter2-section 2.4**.

DiMAPI: An Application Programming Interface for Distributed Network Monitoring [Trimintzios06] does offer some limited support for on-demand monitoring in the sense that it presents a unified interface for applications to interact with distributed monitoring sensors and express their monitoring needs. DiMAPI [Trimintzios06] and MAPI [Polychronakis04] operate on the concept of a "flow" (defined as a sequence of packets that satisfy a given set of conditions) that can be created (because a flow is like a socket) and destroyed on-demand by applications. In DiMAPI and MAPI, a "flow" offers an application the possibility to receive packets or computations e.g. statistics from a monitoring sensor or distributed monitoring sensors. The support for **Principle-P2**, which talks about programmable monitoring, is limited in approaches like DiMAPI and MAPI in the sense that such approaches and similar approaches do not support the use of a monitoring-behaviour-specification via the use of a composition

language like EDBSLang, rather than simply the use of function arguments, to trigger and drive the monitoring functions required. The use of a composition language, rather than the use of function arguments, offers more widened expressive power for monitoring needs, programmability and extensibility power for monitoring platforms, including the possibility for standardization of monitoring interfaces of monitoring components and platforms for multi-vendor support. DiMAPI, MAPI and similar approaches such as the ones covered by CAIDA [CAIDA] do not fully support the notion of "*monitoring-session management primitives*" defined by **Principle-P4**, which are meant to include *pausing*, *resuming* and *modifying* monitoring-behaviours, to ensure that resources can be temporarily freed when monitoring-sessions are not required to be executing on a monitoring component. The other ODM-Principle that is not fully supported by all these approaches is **Principle-P6**, which talks about the need to introduce admission control for monitoring-requests, because in general, today's approaches focus on access control as some form of admission control for monitoring requests but not admission control based on resource-availability at a monitoring component. The ODM-Principle **Principle-P7** is not supported by today's monitoring paradigms, and this means platforms and tools like DiMAPI, MAPI, CAIDA based approaches, NetFlow-based systems [NetFlow], IPFIX [IPFIX], etc, should evolve to support **Principle-P7** (if they are to be fully usable for monitoring in self-managing networks), in order to enable the monitoring components to self-describe their monitoring capabilities to the network, so that automated tasks in the network can locate, select a monitoring component of desired capabilities, trigger monitoring functions and manage the execution of those functions. Though the concept of on-demand data models (covered by **Principle-P3**) is supported in the category of the systems or approaches described above, the concept of On-Demand SNMP MIBs we introduced along **Principle-P3** is not supported in those approaches.

Because the ODM-Principles (**Principle-P1** to **Principle-P7**) defined in this dissertation are not the only principles required for the design of traffic monitoring components and platforms, it is important to note that the ODM-Principles may be viewed as high-level principles that must be augmented by some low level design principles that must be followed when designing ODM-Capable components. The low-level (core) design principles for traffic monitoring components include the following: flexible and efficient packet processing at high speeds in order to cope with the growing speeds of communication links e.g. (multi-)gigabit interfaces; engineering for high throughput; efficient and/or diverse packet filters; engineering for the exploitation of hardware heterogeneity and the need to accommodate software diversity as far as the users of the services of a monitoring component is concerned. [Bos04] presents FFPF: Fairly Fast Packet Filters. Both [Bos04] and [Hruby05] provide a summary of today's well known traffic filtering techniques and filter languages. In [Hruby05], the authors present a framework for efficient, flexible packet processing that supports a diverse set of concurrently active applications (i.e. users of the services of a monitoring component). In [Hruby05], a number of design principles along with the framework are presented, that should augment the ODM-Principles for the design of ODM-Capable components. [Hruby05] presents the following design principles:

- *Generalise flows*. The generalized concept of a "flow", as any subset of network packets that is of interest to a user i.e. an automated task, is similar to the one adopted in defining the ODM-Paradigm. The authors in [Hruby05] discuss a number of vital issues for consideration in relation to this principle and introduce the concept of a "flowgraph" which is a chain of interconnected flow processing functions such as filters, counters and queues, through which a "flow" streams through.
- *Support heterogeneous processing hierarchies*. The authors in [Hruby05] discuss the need to process packets at any level that allows for the possibility to do so, e.g. userspace, kernel, stream processors and programmable network cards, and even on remote devices such as Juniper routers [Juniper]. This effectively introduces the need for heterogeneous hierarchical tree-like processing structures for flow packets, and allows for automated tasks (users) that need monitoring data to be hooked to different processing levels as may be necessary.
- *Engineer for barrier diversity*. The authors in [Hruby05] present the concept of "flowspace" (userspace, kernelspace, etc.) that is introduced in FFPF terminology [Bos04] and is an abstract interface through which a node in the processing tree can be programmed. Flowspaces, as described in [Hruby05], are separated by barriers, such as network links, PCI buses, and kernel-userspace boundaries.
- *Avoid needless copying and context switching*. The authors in [Hruby05] discuss the issues of efficiency in packet processing brought by this design principle.
- *Maintain language and platform neutrality*. The authors in [Hruby05] discuss why it is hard to combine different packet processing platforms and languages and point out why it is desirable to mix and match diverse approaches.
- *Support complex processing graphs*. The authors in [Hruby05] discuss why there is a need for complex processing graphs.
- *Separate control and data planes*. The authors in [Hruby05] discuss why there is a need to separate control and data planes in high speed stream processing.
- *Support different levels of abstraction*. The issue of abstractions and support for composability for monitoring services or behaviour, a key requirement for on-demand monitoring, are also discussed in [Hruby05].

Therefore, by examining some monitoring approaches and systems which fall into the category of "monitoring systems and approaches that specifically address the low-level (core) design principles for traffic flow monitoring components", we see that such low-level (core) are needed for augmenting the ODM-Principles (**Principle-P1** to **Principle-P7**) for the design of ODM-Capable components. What has to be said however, is the fact that, the ODM-Paradigm is meant to enable automated tasks of a self-managing network to express monitoring needs and trigger monitoring-behaviours on ODM-Capable components, as opposed to having monitoring tasks of monitoring system or component configured to start gathering data at boot-up or installation time

of the system, and not being invoked on-demand and further controlled i.e. managed by some automated tasks.

The other category of monitoring systems worth to consider for possible evolution towards partial or full support for ODM-Principles is the category of systems that support the concept of "Triggers". [Jain04] and [Huang06] describe examples of monitoring systems that support "Triggers". Triggers can be local to a single *network element*$_{re\text{-}defined}$ or can be distributed (i.e. "Distributed Triggers"). The key issue in such systems is about: maintaining some defined constraint or invariant. [Jain04] distinguishes "aggregate triggers" from "local triggers" where the constraint or invariant to be maintained can be defined entirely in terms of the state at, or behaviour of, an individual node rather than multiple nodes. In distributed triggers, participant nodes must work together to maintain invariants and constraints over the network-wide behaviour [Jain04]. For more information about how to achieve this (whether by employing some centralised coordinator or a fully distributed approach), we refer the reader to [Jain04] and [Huang06]. As pointed out earlier, the ODM-Paradigm is not only about automated tasks requesting for monitoring services in the "*now*" but includes a flexibility in requesting for monitoring services to be triggered in preparation for the reception of monitoring data or event notifications at some point in the "*future*". Therefore, the design of monitoring components that support "Triggers" should incorporate all the ODM-Principles (**Principle-P1** to **Principle-P7**). However, the ODM-Principle we can emphasize in relation to these types of monitoring systems is **Principle-P2**, which covers the aspect of programmable monitoring. The design of such systems can evolve in such a way that automated tasks of the network can express the "invariants" and "constraints" in a *monitoring-behaviour-specification* meant to be requested for execution on a target monitoring component, and specified using a composition language like the EDBSLang presented along **Principle-P2**. This means that a single automated task running in the network e.g. a context-driven network configuration manager could specify in an EDBSLang-based *monitoring-behaviour-specification,* the "invariants" and "constraints", as well as desires for "event-notifications" in the event of constraint-violations detection by the ODM-Capable components. The *monitoring-behaviour-specification* is then uploaded by the automated task into a number of ODM-Capable components at multiple vantage points of the network for execution. In such a case, the automated task plays the role of the coordinator and may need to do the *aggregation* of events or data communicated by the ODM-Capable components executing the *monitoring-behaviour-specification.* In a different context, an automated task such as a routing protocol, running on a device hosting an ODM-Capable component, may create a *monitoring-behaviour-specification* that includes the "invariants" and "constraints" as well as desires for "event-notifications" and/or "data propagations" to itself and to other remote automated tasks (e.g. all the routing protocol instances in the network). The event-notifications or data propagations from the local ODM-Capable component would need to be done in the event of violations of constraints detected by the local ODM-Capable component. In order to enable reactions to violation of constraints, either it is an automated task or multiple automated tasks co-operatively react to "event-notifications" and/or "data" communicated by the ODM-Capable component(s), and/or the ODM-Capable component(s) is the one that reacts

according to the "*Action*" specified using the EDBSLang *Tag:* "*own_action*", in the *monitoring-behaviour-specification* requested for execution. Therefore, the definition of different types of "invariants" and "constraints" on some network behaviour can be seen as a necessary step towards enriching the EDBSLang language with special *Tags* that enable the specification of "invariants" and "constraints" in monitoring-*behaviour-specifications*. Also in this regard, we also conclude that further work on the extension of the ODM-Paradigm should include the support for what we may call ***Flow-based "Triggers"*** such as SUM, MIN, MAX, etc, to be expressed in *monitoring-behaviour-specifications,* whereby a trigger is a condition leading to an event. Also for further research is the issue of enabling automated tasks to specify (in *monitoring-behaviour-specifications)* ***monitoring policies*** such as inter-flow related events and their correlation, expressions on flow characteristics of different flows such as the following:

```
if(flow X consumes more bandwidth than flow Y by N-percent)
      THEN "ACTION";
if(flow G is routed or policied differently compared to
flow P)
     THEN "ACTION";
```

The other category of monitoring systems worth to consider for possible evolution towards partial or full support for ODM-Principles is the category of "Network Traffic Measurement Systems" such as RTFM based systems [RFC 2721], IPFIX based systems [IPFIX], NetFlow based systems [NetFlow]. While network traffic monitoring is a broad subject, these systems are dedicated to measurements of traffic flow metrics such as packet loss, packet delay, packet counts, etc. Clearly, all of the ODM-Principles (**Principle-P1** to **Principle-P7**) are applicable to the evolution of traffic measurement components towards ODM-Capable components that support such traffic measurement approaches in their diversity. What has to be noted when it comes to the concept of a "*traffic flow*" embraced by these systems is that, it is somewhat different from the concept of a "*traffic flow*" embraced in the definition of the ODM-Paradigm, which is as follows: "a *traffic flow* is a stream of transferable units of information e.g. packets that share some properties. Without talking about whether the units of information are in the process of being transferred from one point to another, we can allow some space to think of a flow in an imaginary sense and be able to reason about the absence or presence of a flow at some point of observation." To mention a few evolution perspectives for network traffic measurement systems: the need to support self-description and dissemination of capability models by traffic monitoring and measurement components as described in **Principle-P7**; the need to support other types of On-Demand Data Models such as On-Demand SNMP MIBs described by **Principle-P3**; the need to support **Principle-P1**, **Principle-P4**, and programmability through *monitoring-behaviour-specifications* as defined by **Principle-P2.** Also, in order to have traffic flow measurement components that operate on the basis of ODM-Principles and can be driven by *monitoring-behaviour-specifications* from automated tasks of the network, the concepts behind programmability of today's network traffic measurement systems

can contribute significantly to further extensions to the EDBSLang composition language presented along **Principle-P2**.

ANEMOS: An Autonomous Network Monitoring System [Danalis03] is a monitoring system that supports "scheduling" of measurement tasks, an issue covered by **Principle-P4**. However, the user of the system is meant to be a human, who can interact with the system via a Graphical User Interface (GUI) for performing scheduling measurements, defining rules, and observing or visualising the results of the measurements. It falls into the same categories of monitoring systems such as HP Open-view [Openview], NIMI [Paxson98], SCNM [Agarwal03], IBM's Netview [Netview] and SUN's Solstice [SUN]. The ODM-Paradigm is meant for the design of monitoring components whose users are automated tasks of a self-managing network(s), meaning that the monitoring needs of automated tasks is at the heart of the ODM-Paradigm rather than the monitoring needs of human users (network operators or end-users). Obviously, it is expected that both types of monitoring systems are expected to be instrumented in self-managing networks. Therefore, some of the ODM-Principles such as **Principle-P1**, **Principle-P4**, and **Principle-P7** may not apply to the design of systems in the category of ANEMOS, HP Open-view, etc. However, there would be a need to evolve those kinds of tools which can be integrated into some monitoring components that are meant to support direct interaction with automated tasks of a self-managing network according to the principles of the ODM-Paradigm.

The other class of monitoring approaches and paradigms is that of monitoring systems that support *Queries* on monitoring data. Among such systems we find systems like Gigascope: High Performance Network Monitoring with an SQL Interface [Cranor02], which provides an SQL interface to the network monitoring system, with the aim of greatly simplifying the task of managing and interpreting a stream of data. In such systems, the intended user is a human user, not the automated tasks that are meant to drive the network's behaviour. It is therefore clear that some of the ODM-Principles such as **Principle-P1**, **Principle-P4**, and **Principle-P7** may not apply to the design of systems in this kind of category. However, in the overall picture of trying to incorporate ODM-principles in such systems, the application offering the interface to the human user (instead of only a GUI) could be considered as the automated task that would need to use the *ODM-primitives* defined by **Principle-P4** to create and manage monitoring-sessions on targeted monitoring components. Regarding the design of the monitoring components themselves, if the design is such that a query issued by a user triggers the monitoring on-demand or that the *query-support* also covers the scheduling of monitoring tasks for providing answers to non real-time queries (e.g. queries targeting some time in the "future"), then such systems could be considered as systems supporting some aspects of On-Demand Monitoring. Without having a lot of detail on the design of systems like Gigascope, one can assume that such monitoring systems are designed in such a way that monitoring tasks are configured to start gathering data at boot-up or installation time of the system, and not necessarily being meant to be invoked on-demand.

6 Example Application Areas and Real world Scenarios

6.1 Overview

In this chapter we present the overall more complete view of the ODM-Paradigm from its application i.e. usage point of view. We present selected application areas and scenarios. Three application areas are presented namely: **application of On-Demand MIBs to traffic engineering**, **dynamic network configuration**, **and automated troubleshooting scenarios**. As explained in chapter 3 - sections 3.1 and 3.2, the ODM-Paradigm is meant to cover diverse automated tasks of a self-managing network other than the example scenarios presented in this chapter.

As already described in chapter 3 - sections 3.1 and 3.2, the vision of ODM-Paradigm is to enable automated tasks in some systems to select a point(s) i.e. an ODM-Capable component(s) in the network when a need arises and describe monitoring requirements to a point (target component) using a specification language such as the EDBSLang described in **Chapter 5** as well as in [Chaparadza07a]. Examples of such monitoring requirements are -: **1.** *what sort of traffic needs to be monitored*, **2.** *the event(s) to be watched for (derived from the captured traffic) and/or invariants and constraints*, **3.** *the recipients of event notifications*, **4.** *the action(s) to be taken by the target monitoring component or system itself and or by the designated recipients of event notifications*, **5.** *the behaviour of a monitoring-session*, etc.

6.2 Applying the ODM-Paradigm

The full perspective concerning the application of the ODM-Paradigm is that, components: namely *ODM-Capable components or systems*, *traffic flow(s) generator components or systems*, *traffic flow(s) sinker systems*, self-describe, disseminate and update their capabilities by the use of capability models that are pushed into a *Capability Models Database*(s) and/or can be solicited for. Systems (i.e. automated tasks) in the network e.g. Network Management Systems (NMSs) or any distributed tasks can access the capability models database, select components of certain capabilities at some points of interest on the network topology, and request selected systems or components on-demand using appropriate *primitives*, to perform some roles i.e. in running monitoring related behaviours such as analyzing traffic, generating, sinking or dumping

some diagnostic/test traffic on-request. As such, systems exporting their capability models can then be controlled by automated tasks for some monitoring objective. Therefore, an *ODM-Capability-Model* of an ODM-Capable component or system in larger context should include descriptive information such as:

1. *The description of monitorable(observable) traffic flows that includes attributes such as: protocol or service descriptor, traffic filter language(s) supported, from where to where i.e. the previous and next hop for the monitorable(observable) flow, the network prefixies of the monitoring interface and the outgoing interface for the monitorable flow, actions that can be performed on the flow e.g. re-marking, blocking or shapping;*
2. *The permissible monitoring-behaviours that the component can apply to this flow,*
3. *The kind of monitoring data that can be collected and whether the data can be stored locally or distribute,*
4. *On-demand data models that the component can support for creation and destruction e.g.* On-Demand SNMP MIBs [Chaparadza06a],
5. *The protocols and mechanisms supported for disseminating monitoring data and event notifications, etc*.

The *Flow(s) Generator Capability Model* of a traffic flow(s) generator should include such information as: *flow properties e.g. source and destination address information of a diagnostic/test flow that can be generated on request, next-hop for this flow, traffic characteristics e.g. allowable bit-rates, shape, etc, and traffic measurements that can be performed on the flow, etc*. An example of a programmable traffic flow generator is presented in [Chaparadza05a]

The *Flow(s)-Sinker Capability Model* of a traffic *flow(s) sinker* (i.e. supports sink i.e. block or dump on request behaviour such that the traffic does not flow further past the point), should include such information i.e. attributes such as: *flow properties e.g. source and destination address information, previous and next-hop for this flow, traffic measurements that can be performed on the flow*, etc. Some ODM-Capable systems may play the role of both a traffic flow generator and sinker. The ODM-Capability Model automatically generated and published by such systems (i.e. ODM-Capable components) would then include Flow(s)-Generator and Sinker Capability models. **Appendix B** presents the CMDL meta-language (an XML Schema), which is meant to be used by an ODM-Capable component for describing its capabilities by creating and publishing its Capability Model, be it an *ODM-Capability Model*, *Flow(s) Generator Capability Model* or a *Flow(s)-Sinker Capability Model*.

Figure 16 shows ODM-Capable components i.e. systems (routers, probes, switches, etc), dedicated flow(s) generator and sinker systems, in different network segments, exporting their self-described capability models into the Capability Models Database. The diagram also shows an example of a domain with sub networks (segments) and components for On-Demand

Monitoring. ODM-Capable components, traffic generators and traffic sinkers push and update their models to the Capabality Models Database. They can also be queried of their Capabilities. Their *presence* and location can be discovered. An ODM-Capable component may have Traffic Flow(s) Generator (TG) and/or Traffic Flow(s) Sinker (TS) Capabilities as part of its overall Capabilities. On the diagram, we can see a potential remote session-owner i.e. a Network Manager component (automated task) that queries the Capability Models Database in order to select ODM-capable systems on which to install and manage monitoring-behaviours via the appropriate control channel and primitives. The same session-owner queries the Capability Models Database in order to select flow(s) generator and sinker systems on which to request the generation and sinking of a diagnostic/test flow(s) of specific properties and/or possibly the participation of those systems in active measurements of traffic characteristics, again using the appropriate control channels and primitives. An example of a programmable traffic flow generator is presented in [Chaparadza05a] Capability Models may change over some time, for example an ODM-Capable system may be unwilling to accept monitoring requests at a certain time due to limited resources or because it can longer be in a position to observe i.e. monitor traffic on say, failed interfaces or links etc, forcing the system to update the network concerning its capabilities. Flow(s) generators and sinkers may also experience changes and failures that may require them to update the network concerning their capabilities. Each Capability Model also contains information about the point to which the system (capability model exporter) is attached to the network. Therefore, the topology of capability model exporters can be constructed and known to other systems in the self-managing network(s). The *self-description* and *self-publishing* of capability models by a system or component should be subject to security policies. The issue of security in the descriptions and publishing of capability models by monitoring components was not covered in this research and would amount to a subject for further research.

Therefore, the big picture i.e. full perspective about the ODM-Paradigm includes the role of *active* and *passive* monitoring and the need for *Capability Models* of all components that may be required to participate in a monitoring objective by automated tasks. These components include: *ODM capable systems (e.g. routers, switches, hosts, signalling gateways, special probes etc), traffic flow generators, traffic flow sinkers. Meta-languages for expressing such Capability Models* are required. **Appendix B** presents the CMDL meta-language (an XML Schema), which is meant to be used by an ODM-Capable component for describing its capabilities by creating and publishing its Capability Model. The Capability Models can be stored in a logically centralized database that can be replicated in the network for redundancy and resilience. Capability Models allow other systems (automated tasks) to view a collection of such components as a monitoring platform so that appropriate *request-primitives* can be issued to components of some capabilities during the operation of a self-managing network. The CMDL language i.e. an XML schema for expressing capability models, which can be used by vendors for implementing monitoring components i.e. systems that can export and update capability models, has been described briefly in **chapter 5—section 5.6** and is fully described in **Appendix B**.

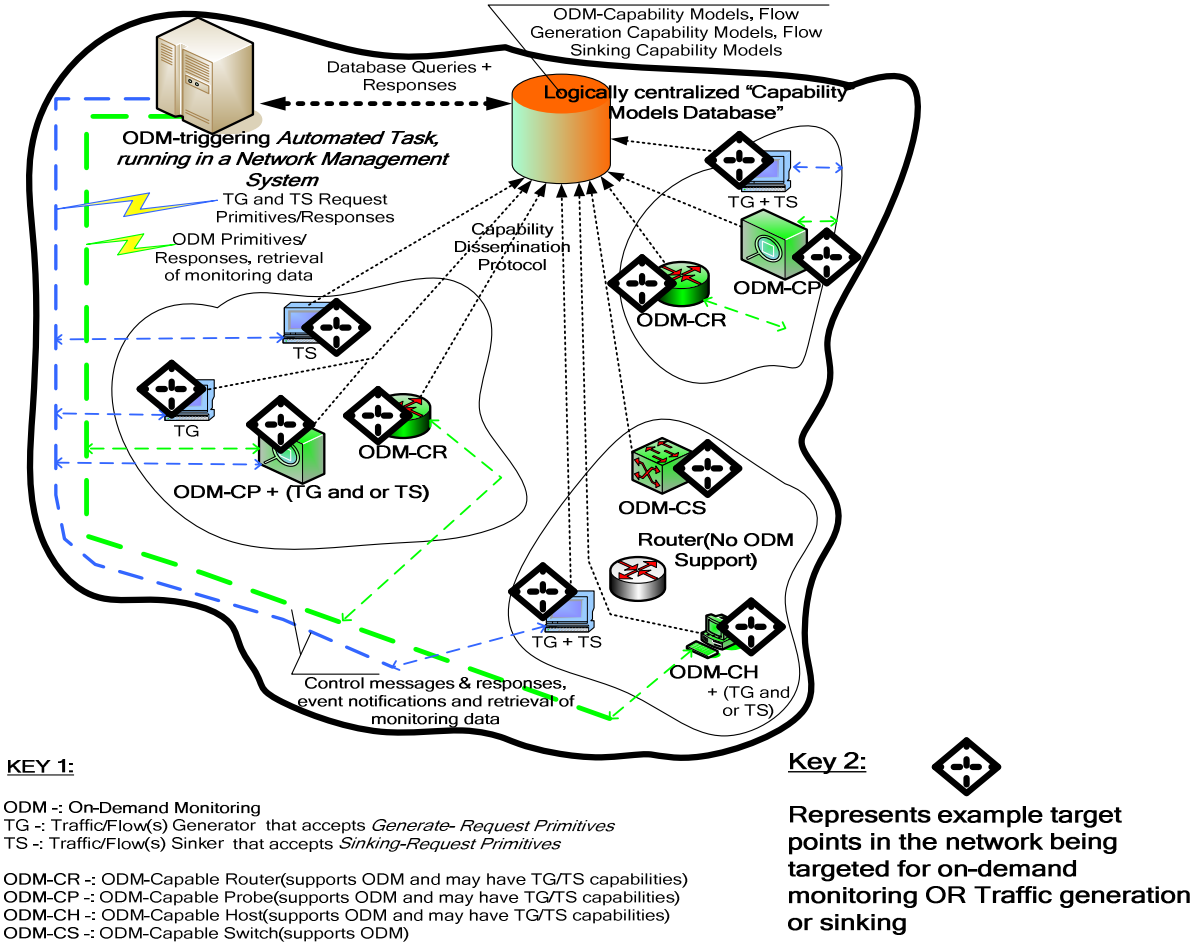


Figure 16: Distributed interactions between *Automated Tasks* and *Monitoring Components* in an on-demand monitoring scenario

In **Chapter2-section 2.4**, we described the limitations of today's monitoring paradigms, frameworks and tools with regards to their applicability to resource-demanding multi-service self-managing networks. Self-managing networks are networks enriched with autonomic behaviour such as self-configuration, self-diagnosing and self-troubleshooting. Evolving management systems for such networks will require intelligent and rich monitoring solutions for monitoring components that will still attempt to limit resource consumption by offering the possibility to provide rich and on-demand monitoring data as may be required by automated management and troubleshooting tasks. Flow-specific traffic monitoring will play a key role in these automated applications and so, these automated tasks will require the ability to trigger the monitoring of flow-specific traffic at some selected target point(s) and time in the network, specify flow(s)-specific or protocol-specific behaviour that needs tracking on the target(s) and demand rich event-notification propagation from the targeted system i.e. monitoring component. Task automation for self-managing networks can be improved by enabling automated tasks to

trigger the generation and tracking of test or diagnostic traffic at selected points in the network in order to gather detailed knowledge about what is really happening in the network and on particular systems so that systems such as management systems can use this knowledge to dynamically configure certain network devices or hosts, co-operatively troubleshoot and diagnose services. The evolving automated management or troubleshooting applications will require the ability to remotely look for some flow-specific or protocol-specific behaviour that is an indication of a failure or an indication of something suspicious on a system or some systems exchanging packets. Such monitoring requirements can be met by an On-Demand Monitoring (ODM) platform as described in **chapter 3—sections 3.1 and 3.2**. **Sections 6.3.1 to 6.3.4** give real-world example scenarios on the application of the ODM-Paradigm, presented earlier in [Chaparadza05b] [Chaparadza06a].

As already described in Chapter 3-section 3.1, monitoring needs vary from short term to long-term needs. For example, in troubleshooting, there are cases whereby the trouble-shooter (human or software) needs to trigger monitoring of certain incoming and outgoing protocol messages or packets on any of the OSI or TCP/IP model layers on a particular system or on a set of systems for a short period of time i.e. only during the troubleshooting process. Another example could be a Network Management System (NMS) requiring short-term triggered monitoring of incoming and outgoing protocol messages (or packets) on a particular system in order to gather some data e.g. some statistics on any OSI layer and furthermore, intends to request the monitored system(s) to send notifications to the NMS or to some other designated recipient(s) when certain events derived from the captured traffic have been detected on the monitored system(s). Often, it is desirable to have an NMS that can gather information from the network, create a picture of what is happening in the network at a certain point(s) and time (snapshot) and dynamically re-configure certain devices depending on the created picture. Such type of monitoring requires a platform or tool(s), available within the network or on the monitored system(s) that supports the concepts and principles of On-Demand Monitoring as described in **chapter 3-section 3.2 and chapter 4**, to ensure that resources are freed whenever some monitoring functions are no longer needed at points and time in the network. The monitoring components that constitute the elementary building blocks of such a platform should support the following properties:

1. The ability to receive and process monitoring requests and, perform admission control on monitoring-requests at any time. A request triggers the monitoring and, selective traffic monitoring is achieved via rich *traffic filters*.
2. Support for the notion of *monitoring contexts, session, session-owner and session-behaviour (behaviour-specification, instantiation, modification, removal)*
3. The ability to *suspend*, *resume* and *stop* a monitoring-session (*ODM-Session*) upon the reception of an authorized request.
4. The ability to receive and process *data retrieval requests* and to send responses (requested monitoring data) to the requester and the ability to export data. This implies that the monitoring data is stored by the target system in a data model well understood by the requester e.g. an SNMP MIB data model [SNMP] [Chaparadza06a] [Chaparadza05c].

5. The ability to detect user defined events and send notifications to some user defined destination system(s). The ODM-session-owner defines the event(s) to be watched for and specifies where notifications should be sent. The events are derived from the captured traffic e.g. "no RSVP packet [RFC 2205] captured within 20 seconds from the start of capturing" can be specified as an event, resulting in a notification.
6. The ability to execute special task programs (actions) upon the reception of an authorized request or due to the occurrence of a user specified event. The ODM-session-owner specifies the action to be taken by the target system when an event has been detected.
7. If required, provide timestamps associated with the departure or arrival of each captured packet of interest, subject to the capabilities of the monitoring component.
8. Security guarantees for remote tentative ODM-session-owners i.e. encryption of ODM requests and responses to and from the target system i.e. monitoring component.
9. The ability to create dynamic in-memory on-demand data models for storing the monitoring data derived from the captured traffic apart from the ability to create packet or flow traces as already supported by today's monitoring devices. For example, dynamic on-demand SNMP MIBs for storing data derived from the captured traffic e.g. the ***computed rate of flow*** or **message** ***arrival-rate or traffic-shape*** of the captured traffic. The good thing about the SNMP MIB data model is that, it allows a remote automated task to retrieve only the information that is required at any time via the use of OIDs, without having to read a lot of data using file transfer and then scan the data for the necessary details as is the case with flow and packet trace files.
10. Support for "*query-based monitoring requests"* and "*EDBSLang based, monitoring-behaviour-specification driven monitoring requests"* and the creation of the appropriate monitoring-sessions accordingly (as described later in **Chapter 7**, where we describe the ODM-Probe as a prototypical ODM-Capable component).

A Network Management System (NMS) might need to automatically scan the network and create an end-to-end picture of consolidated messages or packets including some timestamps associated with the departure or arrival of a message or packet i.e. a test/diagnostic packet(s) at some point (e.g. a system) in the network, by creating a monitoring-session on the target and releasing i.e. terminating when no longer required. Quoting Cisco [Cisco] and Tekelec Inc [Tekelec], "a single view of a call's signalling messages is a quick way to isolate faults". By triggering some monitoring of protocol-specific or flow-specific traffic at different points in the network(i.e. creating monitoring-sessions on the selected targets) and collecting the decoded packets from the target points, a consolidated view of a call's signalling messages i.e a VoIP call that traverses different interfaces and protocols can be created by an automated troubleshooting task.

Generally speaking, self-managing (or autonomic) networks are the kind of networks that know on their own, how to gather and act upon rich sets of traffic characteristics derived at selected points in the network and at some points in time in order to maintain the health of the network, to

intelligently use resources, and strive to achieve the goals of automated tasks running in the network. Self-managing networks are expected to be complex than non self-managing networks, due to the need for more automation across multiple management and network functions. They require a lot of resources in terms of computing resources, data storages, and knowledge-sharing among systems, implying that resources throughout the network must be used intelligently and opportunistically throughout the network by ensuring that all network functions including monitoring functions should be engineering in such a way that they can be invoked on-demand. The ODM-Paradigm ensures that resources are freed whenever some monitoring functions are no longer needed at points and time in the network, all within the control of automated tasks of the network and their dynamics. Flow(s)-specific traffic monitoring plays a key role in driving decisions in automated tasks of the self-managing network such as dynamic *self-adaptive routing*, etc.

6.3 Example Application Areas and Scenarios

In this section, we provide three application areas namely: **application of On-Demand MIBs to traffic engineering**, **dynamic network configuration**, **and automated troubleshooting scenarios**.

6.3.1 Application of On-Demand SNMP MIBs to Traffic Engineering

The picture about task automation in self-managing networks is that a task automation application or simply an automated task e.g. a special purpose dynamic network-configuration management-task may require triggering the generation of test/diagnostic (i.e. tagged-packets) traffic having known characteristics, from a certain point(s) in the network and destined for a certain point(s) in the network, and demand some probes in selected strategic points in the network to perform the following operations: *capture the diagnostic traffic, perform some computations on the traffic(e.g. rate of flow, message arrival-rate), detect some changes in the characteristics, detect events(derived from the captured traffic), and send notifications to the automated task or to some other designated recipients.* This enables the automated task to learn the behaviour or state of the network, the behaviour of running application(s) or service(s) and use such knowledge in influencing its execution decisions such as re-assigning the bandwidth utilized by a traffic class on a DiffServ [RFC 2474] router, carrying say, traffic to and from some mobile clients accessing some service or a server during a certain period. Other example scenarios involve cases such as state-dependent routing and event-dependent routing based on on-demand knowledge about network state and events gathered from an ODM platform of the network.

Traffic engineering in self-managing networks calls for more intelligent management systems that can gather deep knowledge about the state of the network and services by activating monitoring-behaviour(s) at selected points and time in the network to allow other systems in the network to gather such knowledge in order to co-operatively address traffic engineering aspects like congestion avoidance, state or event dependent routing, QoS routing, etc. For example, in **Figure 17,** assuming that on each targeted monitoring point (device) there is an ODM-capable component embedded (such as the ODM-Probe presented in **Chapter 7**), a Network Management System (NMS) i.e. acting as the ODM-session-requester/owner, triggers the monitoring (by creating an ODM-Session) of *T1* traffic e.g. UDP traffic on *protocol analyzer "P"* and demands that an *On-Demand Statistics MIB* be created and advertised to the NMS itself and to *System X* so that the On-Demand MIB can be continuously queried over a certain duration (ODM-Session lifetime) by the two systems. For an example of an On-Demand MIB refer to **Figure 13**, and **Figure 14** and the description provided in section 5.7.1 —whereby the Object Identifiers (OIDs) are dynamically instantiated as described earlier in section 5.3.1. On *protocol analyzer "V",* requested to analyze *T2* traffic, the created ODM-Session advertises the MIB only to the NMS as indicated by the session-owner in the monitoring-behaviour specification. On *protocol analyzer "Z"*, the NMS triggers the monitoring of *T3* traffic and demands that the ODM-session announces the *current rate of flow* of the traffic at a specified heartbeat (specified by the session-owner) to the NMS itself and to *System Y*. *System X* and *System Y* can use the knowledge about traffic flow they get from the ODM target points, to influence their decisions in selecting routing paths that avoid points through which say, too much UDP traffic is flowing through. The NMS may need to execute such monitoring-behaviours on say, a day that the NMS knows or perceives that a lot of traffic of a certain type will be flowing through certain points. This could be on the day large volumes of say electronic pay-roll documents are transferred between company departments, that is, between network segments. In a self-managing network, other systems, not only the NMS may need to execute and terminate such ODM monitoring-behaviours by creating and terminating monitoring-sessions on selected points in the network. In **Chapter 9**, which discusses scalability issues, we present how a Network Management System (NMS) can act as an *Automated Task* that locates a monitoring component of some given capabilities and create and manage a *Monitoring-Session* (i.e. an *ODM-Session*) on the component on behalf of some other kinds of *Automated Tasks* in the network.

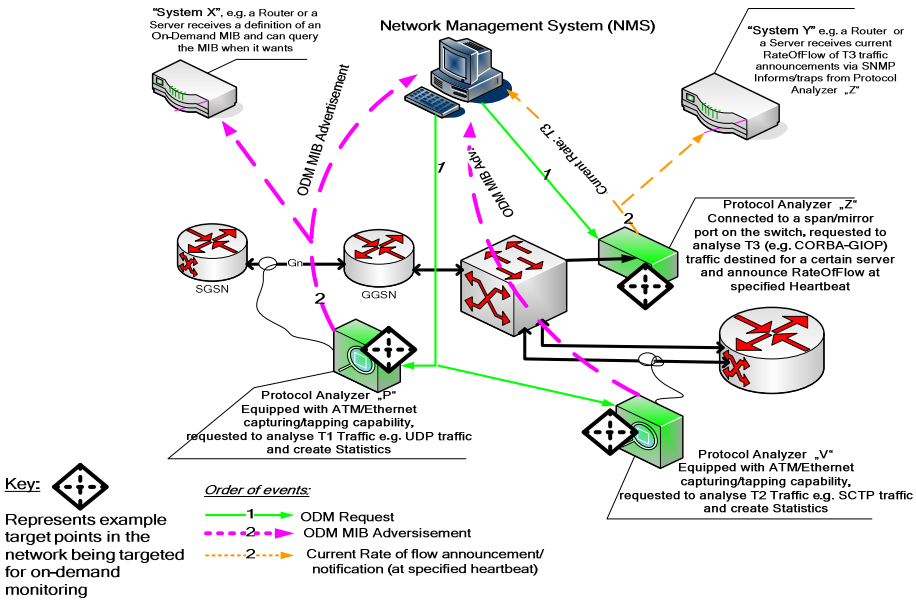


Figure 17: Example scenario of On-Demand MIB application to traffic engineering (with requirement for advertisement of the On-Demand MIB)

Figure 18 illustrates this scenario using a sequence diagram, for the case of requirement for advertisement of ODM-MIB definition (in contrast to the case that follows below).

Figure 19 illustrates that the NMS, acting as an ODM-session-creator can trigger monitoring of different flows at multiple vantage points and demand that the created monitoring-sessions send announcements of the computed current rate of flow for the traffic in question, at a heartbeat specified by the NMS. A corresponding sequence diagram is given later in **Figure 20**. The announcements i.e. notifications can be issued by the embedded ODM-Capable components via SNMP-Trap and SNMP-INFORM types of messages. The ODM-Probe we present in **Chapter 7** supports the use of such types of messages in sending out notifications.

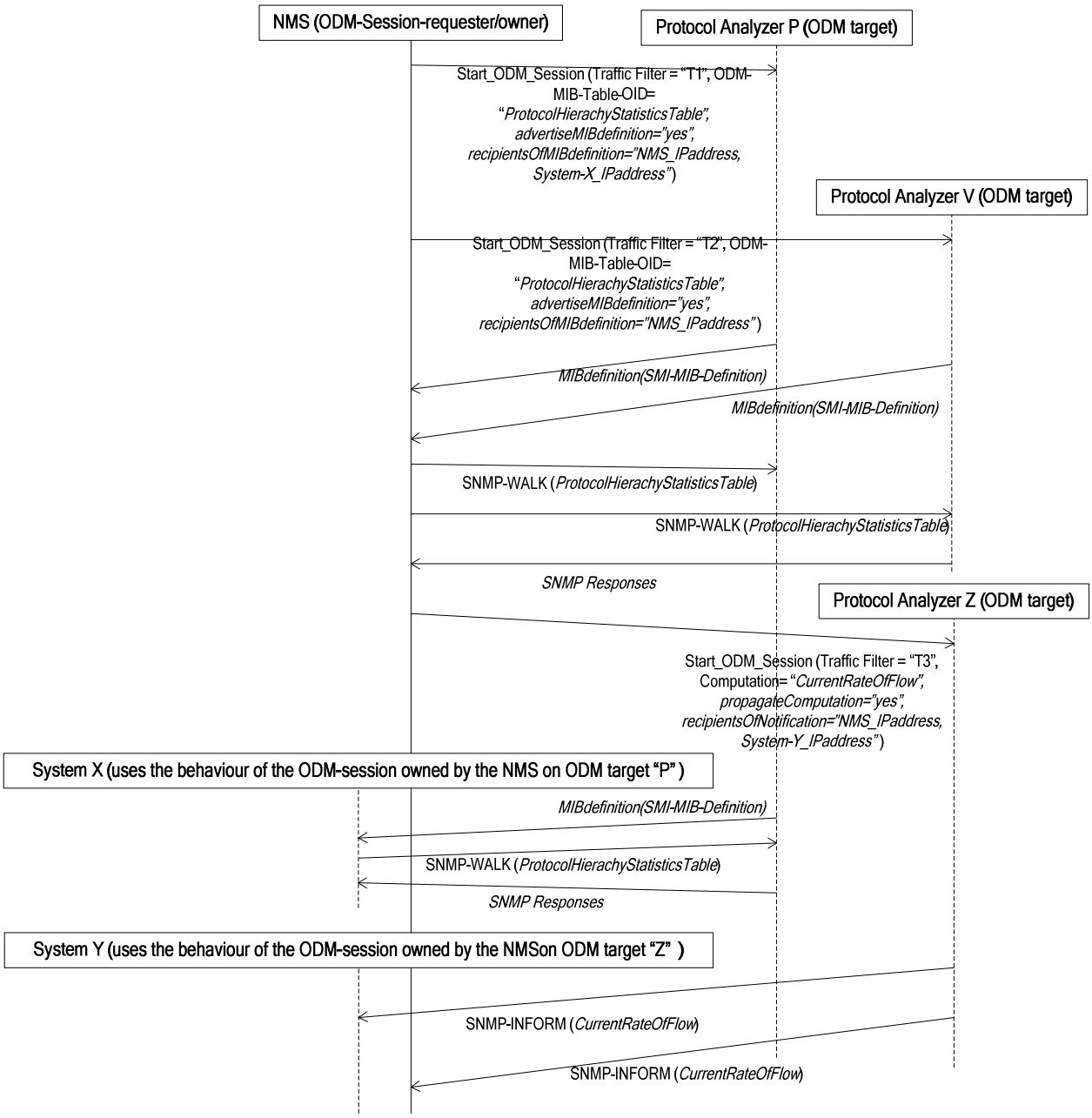


Figure 18: Sequence Diagram for the example scenario of On-Demand MIB application to traffic engineering (with requirement for advertisement of the On-Demand MIB)

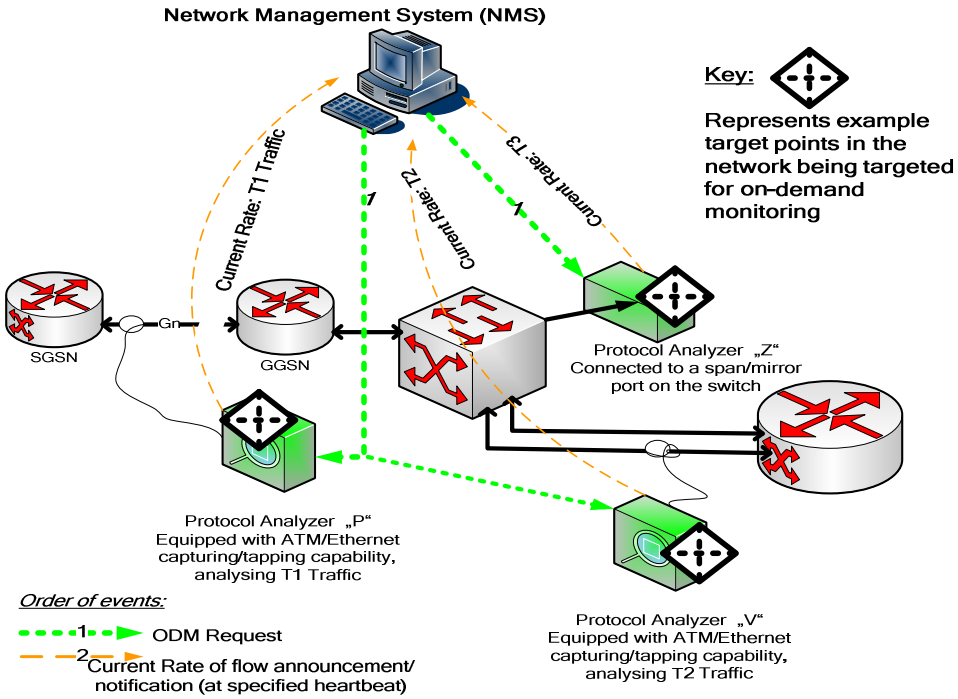


Figure 19: Example scenario of On-Demand MIB application to traffic engineering (with no requirement for advertisement of the On-Demand MIB)

Figure 20 illustrates this scenario using a sequence diagram, for the case of no requirement for advertisement of ODM-MIB definition (in contrast to the case described above).

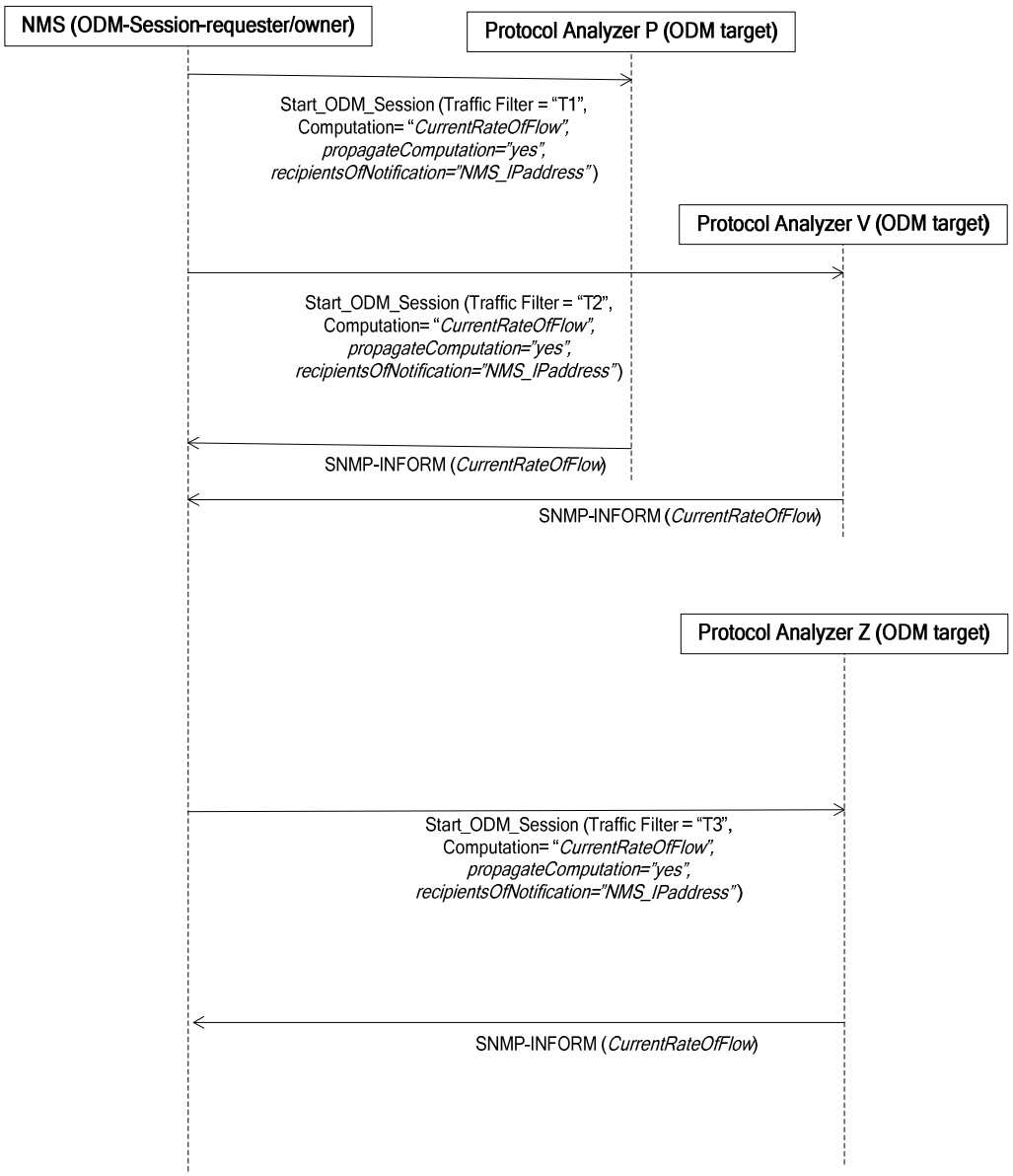


Figure 20: Sequence Diagram for the example scenario of On-Demand MIB application to traffic engineering (with no requirement for advertisement of the On-Demand MIB)

6.3.2 On-demand monitoring in dynamic automated Network Configuration Management and Troubleshooting tasks

Figure 21 shows six examples of arbitrary potential target points in the network where on-demand monitoring can be applied by an automated NMS application or a troubleshooting application. Such target points can be: *a host*, an *applications server*, a *sniffer host*, a *dedicated traffic capturing machine running a universal protocol analyzer*, a *host running a mobile IP foreign agent*, a *dedicated traffic generator and receiver or sinker host*, and an *IP←→SS7 signalling gateway*. On any of such target monitoring points, an ODM-Capable component such as the ODM-Probe described in **Chapter 7** can be instrumented into the *network element*$_{re\text{-}defined}$. On the diagram, the thick dotted bi-directional lines reflect on on-demand monitoring-requests towards the potential targets and, responses and event notifications received from the target points. The potential case for locally triggered ODM is also indicated on the diagram.

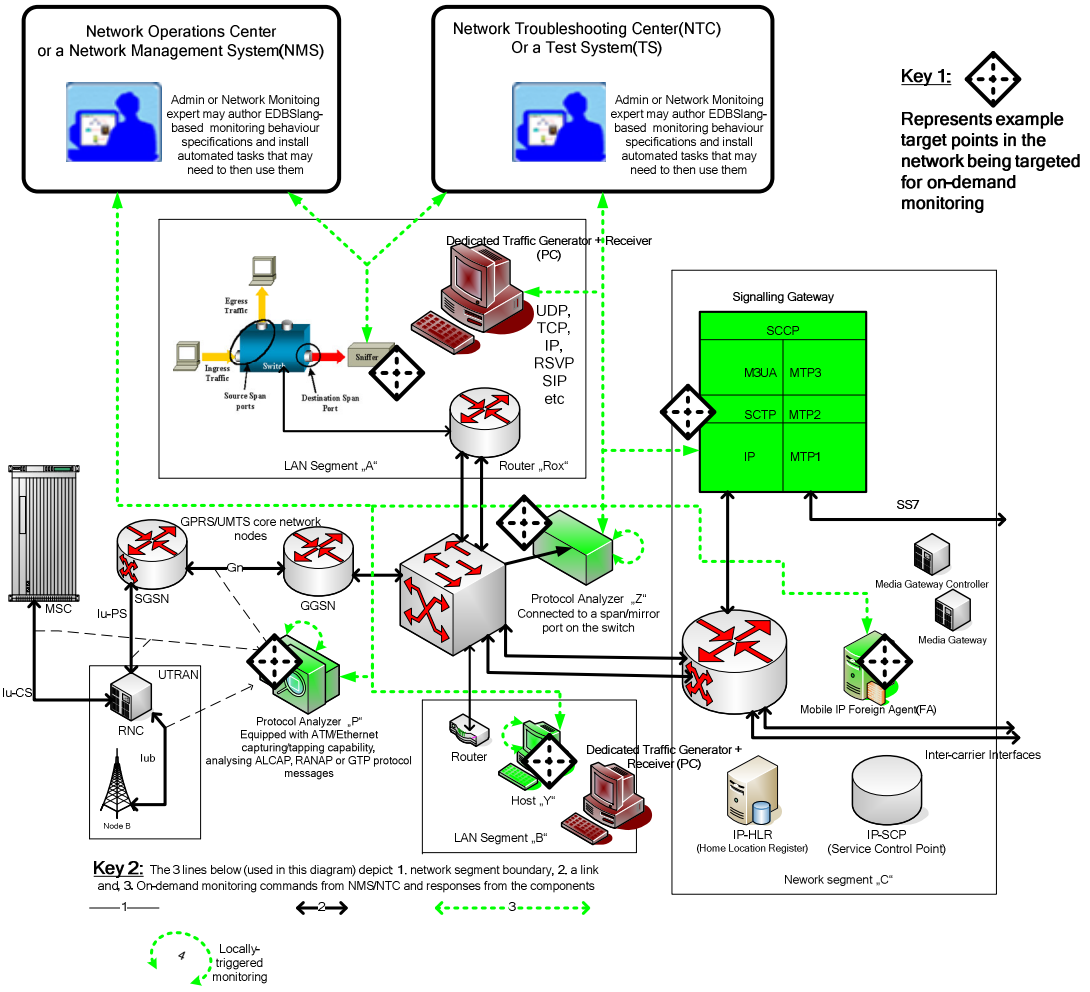


Figure 21: Examples of arbitrary target points for on-demand monitoring (locally triggered and remotely triggered) in a Multi-Service Self-Managing Network

6.3.3 An example Scenario on Network Configuration Management

Referring to **Figure 21**, we imagine the following scenario: The NMS triggers on-demand monitoring on `protocol analyzer "Z"` (connected to a mirror port) to monitor all CORBA-GIOP related messages destined for the IP address of the applications server in LAN segment "A". The NMS demands a notification from the protocol analyzer if the number of clients connecting from the GPRS/UMTS network is greater than 300. Upon the reception of the notification, the NMS intends to increase the bandwidth used by the traffic class "*PremiumServiceMobile*" configured on the DiffServ router "*Rox*" in LAN segment "A" to 15% of the total egress link capacity of both, the link towards the server (up-link) and the link

towards the GPRS/UMTS network (down-link). The NMS achieves all this by creating monitoring-sessions on the selected monitoring components and particular points based on the capabilities of the monitoring components, installing EDBSLang-based monitoring-behaviour-specifications dynamically, and terminating (or pausing) the monitoring-sessions when the monitoring is no longer required.

6.3.4 Automated Troubleshooting Scenario

Referring to **Figure 21**, in the Network Troubleshooting Centre (NTC), we assume that there is a troubleshooting repository containing troubleshooting applications that implement some service specific automated troubleshooting logic and can control and coordinate troubleshooting tasks. Router "*Rox*" in LAN segment "*A*" is supposed to behave in the following manner:

- Convert RSVP based QoS signalling messages originating from LAN segment "A" to out-bound MPLS specific (i.e. RSVP-TE) QoS signalling, in the case that the out-bound traffic will be routed via an MPLS backbone towards a certain MPLS-based carrier (peer carrier domain) and,
- Translate RSVP signalling to ATM related QoS signalling into the ATM backbone network, in the case that the out-bound traffic will be routed via an ATM-based carrier.

Now, users connected to LAN segment "*A"* are complaining about QoS related problems when accessing local and remote servers. In the troubleshooting repository in the NTC, there is a *QoS troubleshooting application* the administrator can run, which triggers the generation of RSVP signalling test packets on Host *"X"* in segment "A" as a sender and some host located in segment "*C"* as the receiver. The *troubleshooting application* also triggers on-demand monitoring of all incoming and outgoing RSVP messages on both hosts and demands that each host sends a notification if no RSVP packets are received or sent within a specified time frame from the ODM-session start time. Meanwhile, on the `protocol analyzer "Z"` capturing traffic from router "*Rox*" via mirror or span ports on the multi-service switch, the application triggers on-demand monitoring of out-bound MPLS related RSVP-TE messages towards any network, demanding a notification if at all there is such traffic and that the notification should indicate the interface on which the detected traffic is outgoing. By finding out which traffic is present or absent at which points (or interfaces), the troubleshooting application can discover the malfunctions of router "*Rox*", which may be due to a wrong configuration. **Note:** There may be mechanisms to detect QoS degradation in the network that can be used to trigger the automated troubleshooting application, without human intervention, in order to implement self-manageability of the network. The automated troubleshooting application would achieve all this by creating monitoring-sessions on the targeted monitoring components at particular points, installing EDBSLang-based monitoring-behaviour-specifications dynamically, and terminating (or pausing) the monitoring-sessions when the monitoring is no longer required.

7 Specification and Implementation of an ODM-Capable Prototypical System: The ODM-Probe

7.1 Overview

In this chapter, we describe an ODM-Capable prototypical system called the ODM-Probe, including its *Functional Specification, Architecture, and Implementation* we developed as proof of concept for the ODM-Paradigm. The ODM-Probe is a traffic monitoring probe derived from the conceptual architecture of an ODM-Capable component described in **chapter 5—section 5.8**. In [Chaparadza06a] [Chaparadza05b] [Chaparadza05d], we presented our early ideas on the development of the ODM-Probe architecture, which has since evolved significantly along with principles and concepts of the ODM-Paradigm. In this dissertation, the ODM-Probe's functional specification is presented with the inclusion of the *Finite-State-Machine (FSMs)* of the main components of the ODM-Probe. The rest of this chapter is organized as follows: **Section 7.2** describes the ODM-Probe, its functional specification, architecture, and implementation. **Section 7.3** presents the Finite-State-Machine (FSMs) of the main components of the ODM-Probe, in form of pseudo SDL models [ITU-T Z.100] to aid in understating the functional specification of the ODM-Probe, in complement to the textual description provided in the next sections. All the design and operational principles (**Principle-P1** to **Principle-7**) of the ODM-Paradigm defined in **chapters 4 and 5** are incorporated into the functional specification of the ODM-Probe. Throughout the functional specification of specific components of the ODM-Probe, the "design and operational principles" being applied to the design of the ODM-Probe component are pointed out.

As was discussed earlier, the GANA Reference Model [Chaparadza08b] [Chaparadza09] [Chaparadza10a] defines a "Function-Level" Decision Element (DE) that must be embedded into a *network element*$_{re\text{-}defined}$ architecture called the Monitoring_Decision Element (*Mon_DE*) [Liakopoulos10] [Zafeiropoulos09] [Liakopoulos08] , which must implement a control-loop over the "management interfaces" of monitoring protocols and mechanisms as its associated Managed Entities (MEs). The DE must apply configuration profiles (which include policies) on this type of MEs, and then react to incidents, state changes and context changes by communicating with other DEs to enforce changes on the behaviour of various types of MEs of the network elements (see definition of *network element*$_{re\text{-}defined}$) to ensure optimal conditions of network operation. The MEs are orchestrated and parameters of the MEs are dynamically adjusted by the

Monitoring-DE(s). A "Network-Level" Monitoring Decision Element may also be designed to work with "Function Level" Monitoring DEs in network elements (see definition of *network element*$_{re\text{-}defined}$). From the perspective of traffic monitoring, the ODM-Principles and the ODM-Probe can be incorporated into the design of Monitoring-DEs.

7.2 The ODM-Probe: Specification, Architecture, and Implementation

In this section we present a functional specification and implementation of an ODM-capable prototypical system that serves as proof of concept. The ODM-Probe serves as an example of systems called ODM-Capable systems described in **chapter 5—section 5.8**. And so, the ODM-capable system presented here is a probe simply called the **ODM-Probe**. Therefore, following the design and operational principles for support of the ODM-Paradigm in a monitoring component(s), we developed the ODM-Probe presented below. The ODM-Probe is a probe that is meant to evolve through contributions from the likes of the open source community and other communities. Therefore, we first describe the intended functionality and then indicate those functional features that are subject for further implementation or extensions as the probe evolves. This is because the implementation of the probe has been done through finding suitable open source software tools and modules and innovatively modifying and integrating them in order to demonstrate the key principles of the ODM-Paradigm. This implies that implementing all the intended functionality is made difficult by the fact that some of the software tools and components used are not easy to modify due to lack of proper documentation and the fact that the selected software tools and components were designed for some other specific purposes. As such, the philosophy i.e. approach behind the design of the ODM-Probe and its evolvable architecture was one based on finding and integrating some software tools and components that exhibit desirable or exploitable strengths of which when combined with the other software tools and components, the result is an integrated probe that supports both design and operational principles of the ODM-Paradigm. **Figure 22** shows the design of the ODM-Probe. The architecture of the ODM-Probe is derived from the conceptual architecture of an ODM-capable system described in **chapter 5—section 5.8**. The probe consists of a number of inter-working components integrating the power of SNMP and Protocol analyzers. Here, SNMP is exploited innovatively for the purpose of providing a framework upon which we can build the concept of dynamic data models i.e. dynamic On-Demand MIBs, as well as providing the ability for remote entities to then query the data models. Here, a multi-protocol analyzer is exploited innovatively for the fact that our focus is traffic monitoring and hence the need take advantage of the fact that filtering and traffic decoding has been designed and integrated into protocol analyzers, though we would need to bring in the notion of on-demand filtering and or decoding. The SNMP components and functions are based on the Net-SNMP open source tools [Net-SNMP] and the multi-protocol analyzer functions are based on a modified Ethereal [Ethereal]. The ODM-Probe consists of the following key components each one of which is described in the subsequent sub-sections of this section, as well as in section 7.3:

- The **ODM Request Handler & Admission Control Component (odmRHACC)**,
- A **Repository for storing monitoring-behaviour-specifications**,
- A **Repository for storing run-time state information of monitoring functions**,
- A set of *trigger-able* macro monitoring functions namely; the **Multi-Protocol Analyzer Engine (MPAE)**, the **MIB Manager** and the **Event Detection & Computation Engine (EDCE)**. The trigger-able monitoring functions: MIB Manager, MPAE, and EDCE are dynamic components created on-demand, minimally for the needs of a single ODM-session (bound to single session-owner i.e. automated-task) or the collective needs of multiple sessions, depending on the needs of the requested monitoring-session-behaviour(s) and availability of resources. For example, an ODM-session might require only the capturing of traffic into a *packet trace file*, without requiring the decoding of captured the packets, so that the session-owner can retrieve the trace file and decode the packets offline. MIB Manager, MPAE, and EDCE also implement some internal, mostly code-level monitoring functions that are trigger-able within the control of the enclosing macro monitoring function. Such internal, mostly code-level trigger-able monitoring functions are functions such as: *create_trace*, *decode_packet*, etc. The triggering of the functions and the components or functions that do the triggering, are described in subsequent sections, along with the functional description of the probe's components and functions given in this section.
- The **SNMP Master Agent** -: This component is described in detail later in this chapter. In brief, it is the component that handles all SNMP communications between ODM-session-owners or users and the corresponding run-time components managing session specific On-Demand MIBs.
- The **IETF-SCRIPT-MIB Environment –:** In brief, this is an environment for storing, executing and managing the execution of scripts. The full description is given later in this chapter.
- The **Autonomic Node Manager (ANM)** is an autonomous entity of the node (i.e. a Decision-Making-Element (Node-DME) of the node GANA[Chaparadza08b]) responsible for invoking the complex node management functions as well as the complex network management functions requiring collaboration with other Autonomic Managers of other nodes, as well as overseeing the overall health and performance of the node. More information about the concept of an ANM can be found in [Chaparadza08a].

The components: **ODM Request Handler & Admission Control Component (odmRHACC); Repository for storing monitoring-behaviour-specifications; Repository for storing run-time state information of monitoring functions; SNMP Master Agent;** and **IETF-SCRIPT-MIB Environment** are shared by all ODM-sessions i.e. MPAE, MIB-Manager, EDCE instances running on the probe. The multiplicity of the components i.e. instances of those components is explained in the functional specifications of the individual components given later in this section. Following the GANA Model [Chaparadza08b], the components such as the odmRHACC, MPAE, EDCE, MIB-Manager, SNMP-Master Agent are considered as Managed Entities (MEs)

of the Monitoring-Decision-Element (MON_DE) [Liakopoulos10] [Zafeiropoulos09] [Liakopoulos08] .

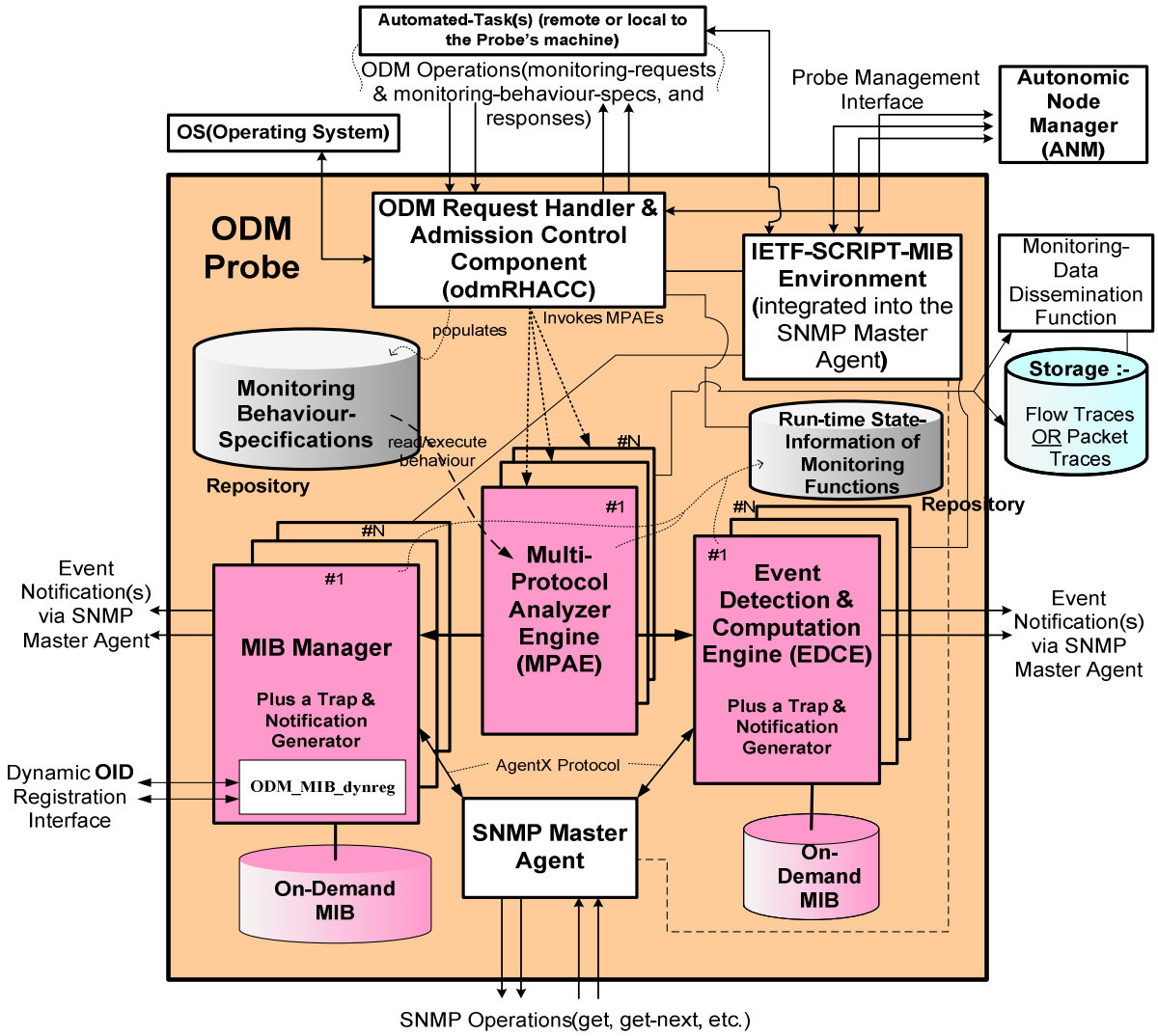


Figure 22: The ODM-Probe

7.2.1 The ODM Request Handler & Admission Control Component

This component, refered to as **odmRHACC** (see **Figure 22**) is as server component that performs a number of functions. A pseudo SDL (Specification and Description Language [ITU-T Z.100]) Model of the Finite State Machine (FSM) of the behaviour and functionality of the odmRHACC is presented in **Figure 29** and **Figure 30** in section 7.3**.** The Model augments the textual

description of this component. The purpose of the Model is to help the implementer further develop and optimize the models during code-implementation, or contrast the approach taken in the design of the probe to other approaches that may be considered to be more optimized. In the functional specification of the **odmRHACC** and the other components of the ODM-Probe, we describe how the underlying principles of the ODM-Paradigm (**Principle-P1 to Principle-P7**) defined in **chapters 4 and 5**, are addressed and incorporated into the design of the components. The functions as well as the interfaces of the component are described below:

- The **odmRHACC**, as required by **Principle-P1** and **Principle-P4**, processes ODM-requests (on-demand monitoring-requests) conveyed via *ODM-primitives* defined by **Principle-P4**, and for each intended managed monitoring-session it communicates a unique private *session-key* to the requester (ODM-session-requester/owner), useful for subsequent session-management interactions with the probe. A managed monitoring-session is a session created by a so-called ODM-session-creator/owner i.e. an automated task running in the network. The corresponding external arrows on the top (see **Figure 22**), indicate ODM operations or requests from diverse tentative session creators/owners, and the responses from the probe such as the conveyance of assigned session-keys, indications of whether requested monitoring-session-behaviour is admissible, etc. The interface for accepting monitoring requests from automated tasks is divided into two sub-interfaces: one for accepting and processing monitoring requests from local automated tasks and the other for accepting monitoring requests from remote automated tasks for which security guarantees are required so that the requested monitoring-behaviour can not be tempered with and that monitoring intentions are not intercepted. For security reasons, all remote interactions between the odmRHACC and remote clients (remote automated tasks) can use SSL (Secure Sockets Layer).
- The odmRHACC, in support of **Principle-P6**, performs *admission control* on every monitoring request as follows: Upon the reception of a monitoring-request and its corresponding monitoring-behaviour-specification, specified using the EDBSLang composition language [Chaparadza07a], to be executed by the tentative session, the ODM Request Handler & Admission Control Component (odmRHACC) runs an algorithm that determines whether there are enough resources to satisfy the needs of the requested behaviour by consulting the OS for the availability of resources. If resources are available to admit the behaviour, appropriate monitoring functions are triggered, otherwise the tentative session-creator (session-requester) receives an exception from the odmRHACC, indicating lack of resources. In **Chapter 9**, which discusses scalability issues, we address the question of whether all kinds of **Automated Tasks** in the network should be allowed to locate a monitoring component of some given capabilities and create a **Monitoring-Session** (i.e. an ODM-Session) on the component. The support for processing monitoring-behaviour-specifications (based on the EDBSLang composition language) is according to **Principle-P2**. The odmRHACC also performs admission control (**Principle-P6**) during an attempt by a session-owner to modify its session-behaviour as this may mean that additional monitoring effort is

required. Admission control policies may be based on security, other than resource availability. Priorities mean that locally running tasks can cause sessions belonging to remote session-owners to pause in the event of resource starvation or when the hosting system is self-degrading i.e. instantaneously reducing its available functionality, due security to violations or suspicion of a potential deadly attack. When sessions are forcefully paused, a notification is issued to affected session-owners. Priorities among local automated tasks are set through the Probe management interface by the Autonomic Node Manager (ANM) of the node on which the ODM-Probe is running.

- For a requested monitoring-session-behaviour that is not admissible or permissible either due to lack of resources or policies not allowing the installation of the behaviour, the odmRHACC sends a response, back to the associated tentative session-creator via the use of marked EDBSLang tags, whereby the monitoring-behaviour-specification of the intended monitoring-session is returned to the tentative session-creator but having the non-admissible EDBSlang tags marked with the <NotAdmissible> tag.
- The odmRHACC takes the monitoring-behaviour-specification uploaded i.e. transferred along with a monitoring request and pushes it into the **Repository for storing monitoring-behaviour-specifications.** In this repository, the monitoring-behaviour is stored into the `tree of monitoring-behaviour-specifications` and `behaviour-identifiers` are assigned to particular instances of cached monitoring-behaviour-specifications and to particular branches within an instance of a monitoring-behaviour-specification. The tree of monitoring-behaviour-specifications built and kept by the ODM-Probe must be included in the global capability model of the probe. Therefore, because the ODM-Probe, according to **Principle-P7**, must support the creation and export of its `capability model` into the `Capability Models Database of the network` (described in **Chapter 6**), the identifiers can be seen by tentative session creators/owners that read capability models of monitoring components from the database in order to select a monitoring component and install a monitoring-behaviour on the target. The tentative session creators/owners can use the `behaviour-identifiers` to request for the installation of the monitoring-behaviour corresponding to the identifier, instead of using a file transfer for sending a monitoring-behaviour-specification which would consume some bandwidth, though not significantly.
- The odmRHACC maintains a data structure: Global SessionsTable that stores information about monitoring-sessions running on the probe such as the session-keys and associated parameters conveyed by the monitoring-request and its corresponding monitoring-behaviour-specification, priority levels of sessions, etc.
- The odmRHACC also stores in the GlobalSessionsTable, the identifiers of MPAEs and the corresponding monitoring-behaviour identifiers of the monitoring-behaviours assigned by the odmRHACC to specific MPAEs. The array of session-keys of sessions that share a particular MPAE is also stored along with the associated MPAE. An MPAE, described later in this section, is a run-time component, invoked by the odmRHACC, which does the capturing, filtering and decoding (if required) of the traffic flow specified

by a monitoring-request's traffic filter. Multiple MPAE instances are invoked by the odmRHACC on the basis of diverse traffic filters specified in arriving monitoring requests, as described later in this sub-section. In order to limit the *MPAE<->Traffic-Capture-Filter* mappings and hence limit the number of MPAEs invoked, the odmRHACC should support *filter aggregation* (refer to [Wang07] and to the subject of composability of filters in chapter 8 of [Varghese05]) whereby some *hierarchical filtering* can be employed such that an MPAE is created to serve monitoring requests whose traffic capture filters belong to a filter hierarchy defined to capture some traffic of some characteristics defined by the filter hierarchy. In such cases an instance of an MPAE and associated child components would be expected to do the further splitting of the traffic filter hierarchy assigned and perform the operations requested by the individual filters which belong to the assigned filter hierarchy and form the traffic filters specified in monitoring requests received by the odmRHACC.

- The odmRHACC stores in some data structure, namely the PerMPAE_SessionsTable, the session-keys for sessions that share an MPAE. The odmRHACC also assigns priority levels to individual monitoring-sessions and stores such information in the PerMPAE_SessionsTable structure, which is specific to an MPAE instance. However, each session has an associated SessionInfoStructure that stores information about the specific session. Pointers to SessionInfoStructure(s) are also stored in the PerMPAE_SessionsTable. The information about running MPAEs is stored in the GlobalSessionsTable, which also contains pointers to particular PerMPAE_SessionsTable. The odmRHACC allows individual MPAEs to access their corresponding PerMPAE_SessionsTable via shared memory.

- The odmRHACC should implement an algorithm that determines the amount of resources required to execute a requested monitoring-behaviour. In order to achieve this, the odmRHACC should maintain a table containing benchmarks of resource requirements for diverse monitoring-behaviours ranging from less demanding to highly demanding behaviours. The benchmarks should be carried out at the point of deployment of the ODM-Probe. Once values for resource demands are obtained based on determining the minimal resources required to run a monitoring-behaviour with satisfactory results, the values can then be provided as input to the odmRHACC to help the algorithm estimate the amount of resources required to run a requested behaviour. The odmRHACC, knowing the overall amount of resources e.g. memory and CPU share allocated for monitoring tasks by the Operating System, as well as the resources required by the requested monitoring-behaviour, can then decide whether to admit the requested monitoring-behaviour for execution.

- The odmRHACC does the assignment of an admitted monitoring-behaviour on the bases of the traffic capture specified in the behaviour spec, by assigning the behaviour to an MPAE whose traffic filter matches the one specified by the monitoring-behaviour, after first checking that new monitoring-behaviour does not add unacceptable overhead on the

MIB-Manager and EDCE instances associated with the MPAE instance to handle the behaviour.

- The odmRHACC manages the On-Demand MIB region sub tree hook assignments (sub tree hooks to the global SNMP MIB tree) for ODM-sessions requiring an On-Demand MIB creation (see **Figure 11**). Support for On-Demand Data Models such as On-Demand SNMP MIBs defined by **Principle-P3**. **Section 7.2.4** and **section 7.2.5** cover the realization of **Principle-P3** in more detail. When a monitoring request that requires an On-Demand MIB creation arrives, the odmRHACC generates and assigns a unique MIB region root identifier (sub tree hook) to be dynamically registered by components inter-working with the SNMP Master Agent, in particular by the MIB Manager instances and the EDCE instances. These two types of run-time components are associated with an MPAE instance that manages the two components. An On-Demand MIB sub tree hook is the value "**X**" as in the branch "**1.3.6.1.3.X**", where **X** is a root sub-tree hook, which becomes free when all the ODM-sessions bound to the branch have died (cease to exist) i.e. have been terminated. In our current implementation of the ODM-Probe, the `session-key` is used as the value of "**X**" (sub tree hook) as it is unique across monitoring-sessions and the current ODM-Probe implementation supports a single ODM-session, not multiple sessions on a single set of MPAE, EDCE, and MIB-Manager instances. This current implementation approach allowed us to first test parallel sub tree MIBs created by multiple ODM-sessions, before our next step of moving on to the final goal of using specially generated sub-tree hooks instead of session-keys, as well as allowing multiple ODM-sessions to share a set of MPAE, EDCE, and MIB-Manager instances provided that the multiple ODM-sessions share the same traffic capture filter.
- The odmRHACC checks for the authorization of a particular ODM-session-owner to modify any monitoring-session on the basis of the provided `session-key` argument supplied by the session modifier.
- The odmRHACC starts a Multi-Protocol Analyzer Engine (MPAE), passing to it the assigned MIB region's root ID (ODM MIB sub tree hook) as well as other data such as arguments conveyed by a monitoring request primitive issued by a tentative session-creator, via a SessionInfoStructure, which is made accessible to the MPAE using shared-memory. It also maintains a list of started multi-protocol analyzer engines (MPAEs) and their associated traffic capture filters.
- The odmRHACC provides an interface to the Autonomic Node Manager (ANM) to allow the ANM to configure and monitor the behaviour of the odmRHACC, as well as getting information about threads and processes started by the odmRHACC. This would enable the ANM to kill all processes and threads associated with the odmRHACC and restart the odmRHACC in case of severe problems that require the odmRHACC to be re-started.
- The odmRHACC, in support of **Principle-P4**, processes the following ODM-primitives (requests) from tentative session-creators/owners. The ODM-primitives defined by **Principle-P4** are defined in more detail here**:**

- **1.** *Start-Session* ("input-parameters"), which takes as parameters: the ODM-Traffic *Filter* to be used e.g. "ip proto 17" or "*ip proto UDP*" used in the *pcap* [libpcap] based traffic filter language, and other parameters such as the default session TTL (Time-To-Live), the name of the associated monitoring-behaviour-specification and Packet Discard Rate (PDR), that is, the negotiated rate at which captured packets are deleted from memory on the target after being used in some computations. This case applies to cases where a certain amount of certain types of packets can be stored, such as diagnostic packets, under the condition that the monitoring system has some resources to store some packets in memory. An ODM-Traffic Filter is a generic filter syntax conforming to the filter language understood by the traffic monitoring component i.e. probe. When a tentative session-creator/owner i.e. an automated task has received a response from the odmRHACC containing the uniquely assigned session-key, the session-creator uploads the monitoring-behaviour-specification to be executed by the desired session. Monitoring-behaviour-specifications should be created using the EDBSLang [Chaparadza07a] composition language. All remote interactions with the odmRHACC can use the SSL (Secure Sockets Layer) protocol.
- **2.** Pause-session(input-param: *session-key)*. This primitive causes the odmRHACC to send a pause-signal to the MPAE instance running the assigned monitoring-behaviour-specification bound to this particular session-owner. The serving MPAE instance, in turn, relays the signal to its child threads and all threads suspend execution, temporarily freeing resources. The primitive is ignored if the shared resources (MPAE instance and associated threads) serving the session are still being used by other sessions. What happens with respect to admission control when a new request arrives during the pause-state of some session?. One approach to this issue is to say that a long pausing session, as defined by some time duration, looses to a newly arrived request. The other approach is to determine resources required by the newly requested behaviour taking into account the overall resources required by the pausing session as if they are still needed. If resources are available to meet the needs of the newly received behaviour request, then the behaviour is admitted.
- **3.** Resume-session(input-param: *session-key)*, This primitive causes the odmRHACC to send a resume-signal to the MPAE instance running the assigned monitoring-behaviour-specification, which in turn, relays the signal to its child threads and all threads resume execution. The primitive is ignored if the shared resources (threads) serving the session are still being used by other sessions.
- **4.** Set-Filter(input-params: *session-key*, ODM-Traffic-*Filter*). This primitive causes the odmRHACC to send a modify-signal to the MPAE instance running the assigned monitoring-behaviour-specification,

which in turn modifies the traffic capture filter in use. The odmRHACC respects this primitive only if the MPAE in question is serving one and only one session-owner.

- **5.** `Refresh-session(input-params: session-key, new TTL).` This primitive causes the odmRHACC to refresh the TTL associated with a monitoring-behaviour of a particular session-owner, otherwise when the default TTL specified by the session-owner at creation time has expired and the associated MPAE instance is currently serving only this particular session-owner, the odmRHACC terminates all the threads running the requested behaviour and resources are freed. A session-owner must keep track of its session-behaviour lifetime and refresh it accordingly.
- **6.** `Terminate(Stop)-session(input-param: session-key),` This primitive causes the odmRHACC to send a terminate-signal to the MPAE instance running the assigned monitoring-behaviour-specification, which in turn, relays the signal to its child threads and all threads terminate execution, freeing resources completely. The primitive is ignored if the shared resources (threads) serving the session are still being used by other sessions.

- **7.** `Modify-session(input-params: session-key, new monitoring-behaviour-specification).` This primitive causes the odmRHACC to determine if the traffic filter has been changed, and to reassign the behaviour to a different MPAE if the traffic filter has been changed while the previously assigned MPAE is still serving other session-owners. If the traffic filter has been changed and the MPAE is not serving other session-owners other than the modifying session-owner, the serving MPAE remains the same. If the traffic filter has not been challenged in the modified monitoring-behaviour-specification, the serving MPAE remains the same. There are three ways a session-owner can modify a session-behaviour: **1.** by issuing a set-filter primitive without changing the overall monitoring-behaviour-specification; **2.** by issuing a combination of *terminate_session(..)* followed by a completely new *start_session(..)* primitive with the modified monitoring-behaviour-specification; **3.** by sending the same monitoring-behaviour-specification as before but with some of the tags marked with <absolete> flag in undesired behaviour tags of the EDBSLang based monitoring-behaviour-specification. The question of how monitoring-behaviour modification by an automated task plays with traffic speeds and link speeds is an issue that needs to be considered when designing the automated task in question. As such, some automated tasks may require modification of a monitoring-behaviour installed at a given monitoring point, regardless of how the modification plays with the traffic speed or link speed. For example, some automated troubleshooting tasks, automated network-debugging tasks or automated Fault-diagnosing tasks, may require injecting some test or diagnostic packets into the network, and then modify the monitoring-behaviour according to

the events and monitoring data communicated by the monitoring components requested to first observe the test or diagnostic traffic and detect traffic presence or some other derived events.

- **8.** `Monitoring-Query (input-params:` "*`Query formulation`*"). The odmRHACC also does the handling of what we call *Query based sessions*, which are un-managed by automated tasks of the network but rather are managed by the probe itself, and having no notion of session-owner in the sense of automated tasks but are considered as owned by the probe itself. This is to allow automated tasks to issue queries such as: "*what is the rate of UDP traffic flow at time_x, from the current time*". In order to support such queries, the odmRHACC should maintain a query table from which it determines when to trigger monitoring functions that provide answers to queries at the appropriate time.

- The odmRHACC component also has an interface with the IETF-SCRIPT MIB environment (described later) for the following purposes: **1)** for configuring the SCRIPT MIB with parameters required by the SCRIPT-MIB for the management of a particular invoked script, and **2)** for allowing scripts to convey error information to the odmRHACC, especially when the script can not communicate this information to the ODM-session-owner i.e. the automated task that owns or influences the execution of the script. Scripts are downloaded into the ODM-Probe by a session-owner(s) as "Action-Scripts" to be invoked when some specified events have been detected by the session, such as events computed or deduced from the captured traffic. The scripts may also be assumed by a session-owner (automated task), to be already existing on the ODM-Probe.
- The interface between the Operating System (OS) and the ODM Request Handler & Admission Control Component (odmRHACC) serves two purposes: **1)** the odmRHACC consults the OS for availability of resources during the time it performs admission control on a monitoring request. **2)** The OS may also send interrupts to the ODM Request Handler & Admission Control Component (odmRHACC), which in turn sends signals to running sessions to terminate or to react to events reported by the OS.
- Using the CMDL language (in support of **Principle-P7**), the odmRHACC creates the ODM Capability Model describing the monitoring capabilities supported by the ODM-Probe, disseminates to and updates the network when the capability model has changed, as described in **chapter 5—section 5.6**.
- The odmRHACC aggregates monitoring requests and merges the associated tentative ODM-sessions that must share the same MPAE and associated components. This means the odmRHACC maintains knowledge about monitoring requests and their associated monitoring-behaviour-specifications being served by an instance of MPAE and its associated MIB Manager and EDCE instances;
- The odmRHACC can be configured to fork for the need for parallel processing, subject to availability of resources on the system.

7.2.2 The Repository for storing Monitoring-Behaviour-Specifications

This repository (see **Figure 22**) is used for storing EDBSLang based monitoring-behaviour-specifications uploaded by automated tasks or by humans. The repository implements an algorithm for assigning a `behaviour-identifier` (*Behaviour-Id*) to every unique monitoring-behaviour-specification, and maintains a mapping table of mappings between Behaviour-Ids and filenames of the behaviour specs. When a monitoring-behaviour-specification has been uploaded into the repository, the repository parses the monitoring-behaviour-specification to check if the behaviour is unique i.e. the order of appearance of tags, the tags and their values in the specification are unique (no such behaviour has been previously uploaded). If the behaviour spec is unique, it assigns a behaviour-identifier and updates the appropriate mapping table. Identifiers can also be assigned to individual EDBSLang tags of a monitoring behaviour specification, which is then considered as a tree of nodes, whereby successive nodes are EDBSLang tags (according to their ordered appearance in the specification file) and a branch in the tree is a possible path from the root (the top most) tag down to a leaf tag, with all tags on the same level (within the same enclosing tag) being considered as sibling nodes. A leaf tag contains no sub-tags. The identifier of the top most (root) tag is preceded (pre-pended) by the unique behaviour-identifier assigned to the containing monitoring-behaviour-specification, to guarantee uniqueness.

As mentioned earlier, the `behaviour-identifiers` of monitoring-behaviour-specifications (considered as instances) known by the repository, along with identifiers of tags belonging to particular behaviour spec instances are included in the Capability model of the ODM-Probe that is pushed into the `Capability Models Database` of the network (described in **Chapter 6**), which is kept updated by capability model exporters.

Automated tasks intending to upload a monitoring-behaviour-specification for execution by the probe can use `behaviour-identifiers` to indicate the intended monitoring-behaviour by running an algorithm to match their locally kept monitoring-behaviour-specification against the Capability Model of a targeted probe to find if a `behaviour-identifier` has been assigned for the intended behaviour. Using identifiers to indicate the intended behaviour as opposed to uploading (streaming) a specification would save network bandwidth. Automated tasks can also identify a combination of identifiers assigned to tags of a monitoring-behaviour-specification from the capability model and concatenate the tag identifiers into branch-identifiers, and use the resulting branch identifiers to indicate the behaviour of interest for execution by the probe.

All uploaded monitoring-behaviour-specifications are received by the odmRHACC server, which then populates the repository. The MPAE uses the filename or identifier passed to it by the

odmRHACC to read directly or request the repository to give it the file containing the monitoring-behaviour-specification assigned to it by the odmRHACC for execution.

7.2.3 The Multi-Protocol Analyzer Engine

This component, refered to as MPAE, is the component (see **Figure 22**) that does the capturing, filtering and decoding (if required) of the traffic flow specified by a monitoring-request's traffic filter and can serve multiple monitoring requests that share the same traffic filter. It is a multi-protocol analyzer in the sense that the component can be requested to change the traffic filter at runtime. A "*modify-session"* primitive or a *Set-filter* primitive issued by a session-owner causes the MPAE bound to the session to change the traffic filter, for example, from capturing say MPLS traffic to UDP traffic, or so. The odmRHACC, which serves all primitives before forwarding them to targeted MPAEs, respects the modification of traffic filter if and only if the MPAE serving the requesting session-owner is currently serving only this requesting monitoring-session-owner, otherwise the primitives attempting to modify the traffic filter will cause the odmRHACC to re-assign the requesting session to an MPAE that is running the intended traffic filter, provided the change is permissible by policies. Coupled with the functionality of the odmRHACC, the MPAE's core functionality is designed to support the principles **Principle-P1**, **Principle-P4**, and **Principle-P5**.

The MPAE component is implemented as a modified `Ethereal` [Ethereal], in particular `Tethereal`[Ethereal], the console version of Ethereal. Internally the code of the MPAE, is decomposed into specific code-level monitoring functions in addition to the thread-level monitoring functions. An instance of an MPAE is started and controlled by the ODM Request Handler & Admission Control Component. If the monitoring needs of a subsequent admissible monitoring request can be satisfied by an already running MPAE the monitoring-request is assigned to an appropriate running instance of an MPAE. The ODM Request Handler & Admission Control Component (odmRHACC) starts a new instance of an MPAE to serve a newly arrived admitted monitoring request if there are enough resources to start multiple MPAEs serving multiple monitoring requests. **Note:** Wherever we say "the MPAE" in the text, we are referring to a particular instance of the MPAE, bearing in mind that multiple instances can be created on the probe by the odmRHACC, with a single MPAE instance serving a single traffic filter or an aggregate traffic filter in circumstances where hierarchical filtering can be supported. A pseudo SDL (Specification and Description Language [ITU-T Z.100]) Model of the Finite State Machine (FSM) of the behaviour and functionality of an MPAE instance is presented in **Figure 31** in section 7.3. The Model augments the textual description of this component. The purpose of the Model is to help the implementer further develop and optimize the models during code-implementation, or contrast the approach taken in the design of the probe to other approaches that may be considered to be more optimized. The functions as well as the interfaces of the MPAE are described below:

- The MPAE reads the monitoring-behaviour-specification indicated by the ODM Request Handler & Admission Control Component from the **Repository for storing monitoring-behaviour-specifications**, parses the monitoring-behaviour-specification and triggers(invokes) the necessary monitoring functions. In order to continue to serve newly assigned monitoring-behaviour-specifications from the odmRHACC, the MPAE creates a mirror thread: **MPAE′** (MPAE-prim) that serves the purpose of listening for signals from the odmRHACC for newly assigned monitoring-behaviours specifications that should then be parsed, resulting in the triggering if necessary, of necessary functions required to serve the monitoring needs expressed in those newly assigned monitoring-behaviour-specifications, provided the required functions have not already been triggered by previously assigned monitoring-behaviour-specifications considered to be still running on the MPAE. If all the functions that are required by the newly assigned monitoring-behaviour-specs are already triggered and running, then the tentative session-owners whose monitoring-behaviour specs are the newly assigned specifications, are then bound to the already running monitoring functions so that they receive either event notifications or can access gathered monitoring data stored in the dynamically created On-Demand MIBs (see definition in **chapter 5—section 5.3.1** or [Chaparadza06a]). The *MPAE′*also updates state information regarding the sessions currently being served by the MPAE. A pseudo SDL (Specification and Description Language [ITU-T Z.100]) Model of the Finite State Machine (FSM) of the behaviour and functionality of an *MPAE′* instance is presented in **Figure 32** in section 7.3. The Model augments the textual description of this component. The purpose of the Model is to help the implementer further develop and optimize the models during code-implementation, or contrast the approach taken in the design of the probe to other approaches that may be considered to be more optimized. Meanwhile, the MPAE (not *MPAE′*) is the one that actually does the traffic capturing and the triggering of monitoring functions (e.g. threads) such as the MIB-Manager instance and an EDCE instance the first time the functions are required and the first time the MPAE instance has been started by the odmRHACC. Otherwise, the MPAE′ is the one that does the invocation of the additional required thread components if the MPAE is already capturing traffic. The MPAE and the MPAE′ run in parallel to avoid possible packet losses during the traffic capturing, caused by interrupts to serve additionally assigned monitoring requests if the MPAE instance doing the capturing were to serve interrupts.
- The MPAE creates the following two threads on-demand: the **MIB Manager**, the **Event Detection and Computation Engine (EDCE)**, depending on the monitoring needs expressed in the monitoring-behaviour-specification requested to be executed. If an On-Demand MIB is required or event notifications are required a MIB Manager is created. If some computations, event detections and notifications to designated recipients are demanded in the monitoring-behaviour-specification, an EDCE thread is created. In our current implementation of the ODM-Probe, only the MIB Manager is the one used to send out notifications to the recipients designated in the monitoring-behaviour-specification. This is because of problems with running two threads (not processes) as

AgentX [RFC 2741] sub-agents on the same node, a limitation of the current implementation of the open source Net-SNMP libraries [Net-SNMP], which rather supports multiple subagents to run as processes (not threads) on the same node.

- The MPAE also listens and reacts to ODM monitoring-session modifications issued by a binding session-owner if the MPAE is serving one session or the modifications do not involve a change of the traffic filter being used by the MPAE while serving multiple monitoring-sessions. When an MPAE is already serving multiple monitoring-sessions at the point of an attempt to modify one of the session-behaviours, the primitives issued by any of the binding monitoring-session-owners are not allowed by the odmRHACC to influence the behaviour of the targeted MPAE if the modification intends to change the traffic filter. Instead, in the later case, the odmRHACC re-assigns the modified monitoring-behaviour-specification to an MPAE whose traffic capturing filter is the same as the one requested by the modified session-behaviour or starts a new MPAE to serve the modified monitoring-behaviour-specification if no appropriate running MPAE has been found.
- The MPAE stores the decoded protocol tree of a captured packet for a small number of the last captured packets if any of the monitoring-behaviour-specifications has indicated via the EDBSLang tag **<packet_summary_table>** in the monitoring-behaviour-specification (refer to [Chaparadza07a]).
- The interface to the **Repository for storing run-time state information of monitoring functions** is used for exporting run-time state information of all running components triggered by the odmRHACC or by MPAE instances. The MPAE and all child threads, including the *MPAE´* push state information into the repository. Run-time state information of an MPAE instance includes the current state of execution e.g. "*capturing*" or "*error-state*", "*paused*", "*terminated*", and other type of state-related information such as the previously handled *signal(s)* from the odmRHACC. Run-time state information is required by entities such as the odmRHACC and the Autonomic Node Manager (ANM) of the node in determining problems to do with monitoring functions and take actions accordingly, such as re-starting affected components.
- The MPAE provides input (packet traces or flow traces) to an external *monitoring-data dissemination function* that disseminates traces, which should be implemented outside the ODM-Probe as is the case with the Monitoring DataBase(DB) as depicted in both the conceptual architecture (see **Figure 15**) and the ODM-Probe (see **Figure 22**). The external monitoring data dissemination function and the Monitoring Database should be implemented outside the ODM-Probe since their implementation and the mechanisms they employ have nothing to do with the core operational principles of the ODM-probe.

7.2.4 The MIB-Manager

The MIB-Manager (see **Figure 22**) is a component that implements a number of SNMP related functions. The MIB-Manager is a thread created by an MPAE instance only if an On-Demand

MIB is required or event notifications are required by the monitoring-behaviour-specification(s) to be executed by the MIB-Manager together with its associated MPAE and EDCE instances. The MIB-Manager, together with the associated MPAE and EDCE instances, support **Principle-P2**, which covers programmable monitoring. As such, a MIB-Manager instance is managed by the associated MPAE, which influences its pausing, resumption or termination of execution depending on the *ODM-primitives* received by the associated MPAE instance. **Note:** Wherever we say "the MIB-Manager" in the text, we are referring to a particular instance of the MIB-Manager, bearing in mind that multiple instances can be created on the probe by their associated individual MPAE instances. A pseudo SDL (Specification and Description Language [ITU-T Z.100]) Model of the Finite-State-Machine (FSM) of the behaviour and functionality of an MIB-Manager instance is presented in **Figure 33** in section 7.3. The Model augments the textual description of this component. The purpose of the Model is to help the implementer further develop and optimize the models during code-implementation, or contrast the approach taken in the design of the probe to other approaches that may be considered to be more optimized. The functions as well as the interfaces of the MIB Manager are described in the bullet points and sub-sections below:

- The MIB Manager creates the On-Demand MIB required by the session-owner (automated task) or by all sessions sharing the services of the instance of the MIB Manager in question. Support for On-Demand Data Models such as On-Demand SNMP MIBs by **Principle-P3**. New OIDs are created on demand during the lifetime of the MIB Manager, as demanded by newly arriving monitoring requests assigned to this MIB Manager instance. The On-Demand MIB created by an instance of the MIB Manager as well as its dynamics supports the creation and destruction of the three types of OIDS described below, namely **I.** EDBSLang-based (i.e. those specifiable by session-creators/owners) OIDs that are specified (demanded) in the monitoring-behaviour-specification; **II.** OIDs automatically generated for storing session (s) related information; **III.** Dynamically registered OIDs that are created and registered on the fly at run-time by entities such an *Action-Script(s)*. An *Action-Script* is a script specified in a *monitoring-behaviour-specification* as a script to be executed by the associated monitoring-session upon the detection of *specified-event*. The MIB Manager takes the odmRHACC assigned ODM MIB sub tree hook, i.e. the value of "**X**" in the branch "`1.3.6.1.3.`**X″**, and for each required OID, generates a node hook (an integer value) for the OID and registers the OID under the assigned sub-tree (`1.3.6.1.3.`**X**) . In our current implementation of the ODM-Probe, node hooks for OIDs are statically assigned (fixed), except for the dynamically registered OIDs (the third type of OIDs). The values of the statically assigned (fixed) node hooks are shown in the Tables 1, 2 and 3 below, i.e. the values in dot-notation that follow after the root ID ("**X**") in "`1.3.6.1.3.`**X″ .**

In the following paragraphs we describe the three types of Object Identifiers (OIDs) that pertain to On-Demand SNMP MIBs that were implemented and evaluated in the implementation of the

ODM-Probe. The types of OIDs currently supported by the current implementation of the ODM-Probe are described in the subsequent sub-sections below.

I. EDBSLang-specifiable OIDs i.e. those that can be specified by session-creators/owners

These types of OIDs are listed in Table 8 (defined in depth in **Table 5**) are the type that can be specified in the monitoring-behaviour-specification using special EDBSLang tags as described in **chapter 5—section 5.2.1** as well as in [Chaparadza07a]. The kind of OIDs presented here are simply example OIDs we have considered only for prototyping purposes, meaning that other types of OIDs can be defined in the further development of the concept of On-Demand SNMP MIBs as well as the ODM-Probe. The example below (see **Figure 23**) shows a fragment of an example EDBSLang-based monitoring-behaviour-specification that specifies that: *UDP* traffic be monitored; if some *UDP* packet/traffic has been seen after *30 seconds* then send notifications to two designated recipients indicating the action they need to execute; a *protocol hierarchy statistics MIB table* as well as the other indicated OIDs, be created for later reading by the session-owner; some computations on the *rate of flow* be performed every *600 seconds* and the value be propagated to the session-owner, computed values must also be stored for later reading, and if the rate of flow is greater than *1Mbit/s* threshold then an event notification must be issued and the specified action must be taken by the installed monitoring-session-behaviour i.e. an *Action-Script* execution.

Table 8

Node(Hook) of attachment to the On-Demand MIB tree	Name of the OID
1.3.6.1.3.**X.2**	*ProtocolHierachyStatisticsTable*
1.3.6.1.3.**X.3**	*CurrentRateOfFlow*
1.3.6.1.3.**X.4**	*GenericEventDescription*
1.3.6.1.3.**X.5**	*GenericPacketHexdump*
1.3.6.1.3.**X.6**	*PacketSummaryTable*
1.3.6.1.3.**X.12**	*PacketCounter*
1.3.6.1.3.**X.13**	*PacketArrivalRate*
1.3.6.1.3.**X.1**	*DetectedEventsTable*

```
<ODM_Traffic_Filter>“ip proto udp” </ODM_Traffic_Filter>
<packet_captured reference=”from_capture_start_time”  seconds=”30s” event_notification=“yes”
        notification_dest =”ipaddr1, ipaddr2” event_description=”30 seconds elapsed from start
                capture and a packet has been captured” notification_sink_action=”ScriptY.pl”>
</packet_captured>
 < ODM_MIB>
        <table_OID> Protocol_Hierachy_Statistics traffic_capture_window_size=”900s”
        </table_OID >
        < table_OID > EventsDetected_Table
        </ table_OID >
        <scalar_OID> Current_RateOfFlow        </scalar_OID>
        <scalar_OID> PacketCounter </scalar_OID>
        <scalar_OID>Generic_Event_Description
        </scalar_OID>
        <scalar_OID>Generic_Packet_HexDump</scalar_  OID>
< /ODM_MIB>
<ComputationAndChecks computation=”Rate” computationStep=”600s”  storeComputations=”Yes”
        propagateComputedValue=”yes”>
        <ComputedEventCheck computation=”Rate” threshold=”1Mbit/s”  booleanCheck=”threshold<
        Rate”  event_nofitification=”yes” notification_dest=”ipaddr1, ipaddr5” event_description=”Rate
        exceeds threshold” own_action=”ScriptG.pl”>
        </ComputedEventCheck>
</ComputationAndChecks>
```

Figure 23: Fragment of an example of an EDBSLang-based monitoring-behaviour-specification (with specified OIDs to be instantiated)

II. Automatically created OIDs—created by the ODM-Probe, mainly for storing ODM-Session related Information

Except for the "*Time*" OID, these OIDs—listed in **Table 9** (defined in depth in **Table 6**), are created for each monitoring-session that is bound to the MIB Manager instance.

Table 9

Node(Hook) of attachment to the On-Demand MIB tree	Name of the OID
1.3.6.1.3.**X.7**	*Time*
1.3.6.1.3.**X.8**	*ErrorsTable*
1.3.6.1.3.**X.9**	*FailuresTable*
1.3.6.1.3.**X.10**	*SessionDescriptionTable*
1.3.6.1.3.**X.11**	*SessionStatus*

III.

Dynamically registered OIDs (see Table 10)

The MIB Manager provides a special API(Application Programming Interface) that allows entities running on the box, in particular Action-Scripts invoked by monitoring-session-behaviours assigned to the MIB Manager instance in question, to register new Object Identifiers (OIDs) into the MIB tree managed by the MIB Manager instance. The newly created OIDs and their values are disseminated to session-owner associated with the executing *Action-Script* upon the completion of the registration process, as described later in the description of the ODM_MIB_dynreg component, which is considered to be part of the MIB Manager. These types of OIDs are listed in Table 10 (defined in depth in Table 7).

Table 10

Node(Hook) of attachment to the On-Demand MIB tree	Name of the OID
1.3.6.1.3.**X.20**	*DynamicOIDregistrations*
1.3.6.1.3.**X.20.1**	*RegistrationSessionNode*
1.3.6.1.3.**X .20.1.1**	*Generic"Scalar1" or Generic"Table1"*
1.3.6.1.3.**X.20.1.2**	*Generic"Table2" or Generic"Scalar2"*
1.3.6.1.3.**X.20.1.N**	*Generic"TableN" or Generic"ScalarN"*

Figure 24 shows the structure of an On-Demand MIB tree created and managed by an instance of a MIB Manager i.e. branch ($\mathbf{X_n}$). The figure shows how On-Demand MIB branches are treated as branches registered under a node that is bound to the SNMP Master Agent. The figure also shows how nodes are allocated, as well as reserved nodes. A number of branches under the responsibility of multiple subagents, running as threads and bound to a single MPAE can only be supported for AgentX environments in which multiple threads can be allowed to run on one machine as AgentX subagents. In that case, the subagents would either register branches with the SNMP Master Agent, whose root nodes have to be generated uniquely by the associated MPAE under "`1.3.6.1.3.X`", where "**X**" is the hook-ID assigned and passed to the MPAE by the odmRHACC **or** the odmRHACC would have to assign and pass multiple unique "**X**"s to an MPAE, to be used by such subagents when registering with the SNMP Master Agent. In our implementation, which uses NET-SNMP AgentX libraries, the sub-branches $\boldsymbol{X_1 \ldots X_n}$ can only mean that the branches are assigned to different MIB-Manager instances which always run as threads associated with different MPAE instances running as processes. This means that the SNMP-Master agent and associated Net-SNMP AgentX libraries view the multiple MIB-Manager instances running on the ODM-Probe as separate processes, not threads, since they are encapsulated as threads owned by the separate MPAE processes.

- An On-Demand MIB created by a MIB Manager instance can be used together with the IETF Event MIB [RFC 2981] and the IETF Expression MIB [RFC 2982] running on the box(running within the SNMP Master Agent) in order to generate notifications based on the events and expressions specified in the two MIBs respectively. An ODM-session-owner (i.e. an automated task) can also create events in the Event MIB and or create expressions in the Expression MIB. The events created in the Event MIB or expressions in the Expression MIB can be related to OIDs created in the ODM-MIB tree or can also be related to the "traditional" statically instantiated MIBs on the system.

- The MIB Manager must also implement the algorithm for the SMI definition of the On-Demand MIB and its advertisement to interested parties specified by a particular binding ODM-session-owner or session-owners. As already established in SNMP, an SMI [RFC 2578] definition is a definition of a MIB data model that allows systems to use SNMP to query for values of variables (OIDs) stored in the MIB. As already known in the SNMP world, any entity interested in reading the MIB can access the MIB at any time depending on MIB access rights such as the VACM [RFC 2275] and USM [RFC 2574] based rights. The SMI MIB data model definition and advertisement is needed in cases whereby the OIDs created by a MIB Manager instance are hooked to MIB hooks i.e. "`1.3.6.1.3.X`" and sub-branches that are assigned dynamically, as opposed to the case of having fixed hooks assigned *a priori* by default to specific OIDs. An ODM-session-owner may require that the MIB created by the associated monitoring-behaviour be accessible to other entities. This is the case of say a Network Management System (NMS), playing the role of ODM-session-owner, having the control of the network, and installing (triggering) monitoring-behaviours in a selected monitoring point(s), whereby the installed monitoring-behaviour(s) is intended to create an On-Demand MIB that can

be accessed by distributed automated tasks that either poll the MIB or receive events from the monitoring-behaviour in order to use the gathered information in triggering self-adaptive functions (refer to [Chaparadza06a] for such cases). The session-owner indicates i.e. specifies the recipients of the SMI MIB definition via the use of the <dest> EDBSLang tag (for destination) within an enclosing <ODM_MIB> EDBSLang tag. The MIB definition and advertisement algorithm reads the monitoring-behaviour-specifications assigned to the MIB Manager instance in order to determine the OIDs that should be advertised to specific parties specified by a particular session-owner, since the different session-owners bound to the MIB Manager instance may have different parties specified as interested targets for the SMI MIB model definition.

Event notifications issued by the MIB-Manager are based on `SNMP-Inform` and `SNMP-Trap` messages. The ODM-Probe employs SNMP-Traps for *announcements* such as:

1. "*The current rate of flow or arrival-rate or traffic shape description of the ODM specified traffic",*

2. "*Traffic X currently consumes the greatest bandwidth",*

3. "*System Y is currently the top most "top-talker"".*

The ODM-Probe uses SNMP INFORM messages (Acknowledged) *for Event Notifications* such as:

1. *"X seconds elapsed from traffic capture start-time, No Z packet/traffic captured",*

2. *"Captured Z packet/traffic after Y seconds from capture start-time(....)",*

3. *"Detected a period of silence for R traffic, of more than P seconds",*

4. *"Detected Problem M, all notification sinks should execute Action Q",*

5. *"Detected Event E, all notification sinks should execute Action N".*

As can be noted, an event notification can include the actions to be taken by notification receivers (sinks) specified in a monitoring-behaviour-specification by an ODM-session-owner.

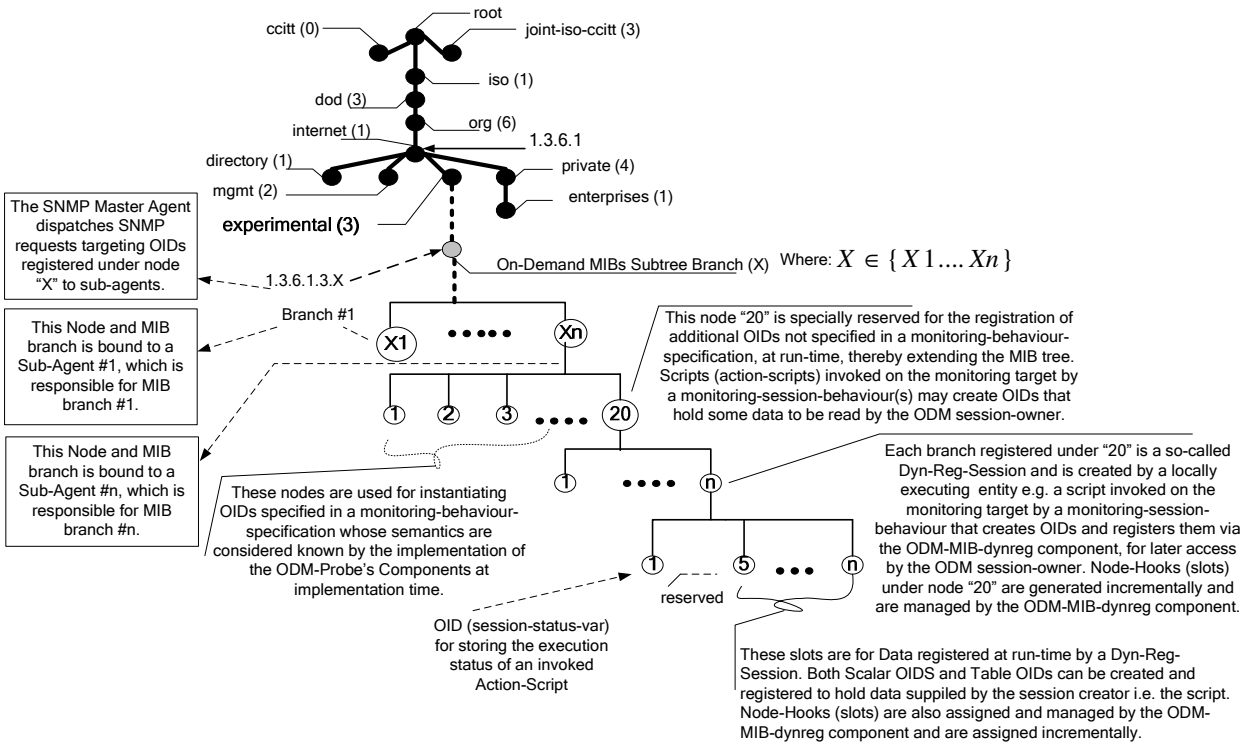


Figure 24: The ODM-MIB Tree

7.2.5 The ODM_MIB_dynreg Component (part of the MIB Manager)

The ODM_MIB_dynreg component (see **Figure 22**) is part of MIB-Manager and serves the purposes of run-time registration of OIDs into the ODM MIB tree. This component provides support for On-Demand Data Models such as On-Demand SNMP MIBs defined by **Principle-P3**. Therefore, the ODM_MIB_dynreg is a server daemon that serves client requests for registering Scalars and Tables of various types. The following is a description of the communication protocol syntax used in the interaction between the server and the entities(clients) running locally on the monitoring target such as Action-Scripts invoked by a monitoring-session-behaviour, which intend to register data into the ODM MIB tree. The registered data i.e. the newly created OIDs can be read by the session-owner after the indication of successful registration by the registering entity i.e. an *Action-Script*. The functional aspects presented below are reflective of the current implementation of the ODM_MIB_dynreg component part of the ODM-Probe.

For registering a *SCALAR*, a client request has the following ASCII-based *COMMAND* form consisting of commands and some corresponding arguments terminated by a new-line ("\n") symbol as indicated in **Figure 25**:

```
REGS <STR> <STR>\n
NAME <STR>\n
SCAL <STR> TYPE <ASN>\n
END\n
```

Figure 25: The COMMAND used for the registration of a Scalar OID

Figure 26 gives the ASCII syntactical form for registering a *TABLE*:

```
REGT <STR> <STR>\n
NAME <STR>\n
INDX|COLM <STR> TYPE <ASN>\n ...
DATA <STR> ...\n
END\n
```

Figure 26: The COMMAND used for the registration of a Table OID

Where: the *COMMAND* can be any of the following:

- **REGS** – for requesting for the registration of a *Scalar*.
- **REGT** – for requesting for the registration of a *Table.*
- **TRAP** (**NOTIFY**) **<OID_List>** – used by an executing Action-Script for requesting for the generation of an SNMP Trap or INFORM type of notification message that must be sent to the ODM-Session-owner whose monitoring-session-behaviour invoked the Action-Script. Such a notification message may be issued by an entity e.g. Action-Script that has completed the task of creating and registering OIDs and wishes to inform the ODM-Session-owner about the values of computations (the dynamically created OIDs indicated by **<OID_List>**), the values of which are conveyed within the notification message (a Trap or an Inform message), and/or to inform the ODM-Session-owner that registration has been done and the dynamically created OIDs are now available for reading should the session-owner want to read them. As such Variable bindings (VarBinds) are conveyed by the notification message, thereby causing the session-owner to learn about the dynamically created OIDs, to enable the session-owner or any other entities to read the OIDs in a future time, should a need arise.
- **ERROR <STR>** – used by an invoked Action-Script for overwriting the value stored in a special *Scalar OID:* "*Session-Status-Var*" variable that stores the execution status of the invoked Action-Script, with the value: "ERROR-STATE", to overwrite the default "SUCCESSFULLY-EXECUTING", in order to indicate to the ODM-Session-owner that

the "*action*" did not execute successfully. The session-owner can learn about the execution status and the name of the currently executing Action-Script by receiving a notification automatically generated by the associated MIB-Manager instance.

The *COMMAND* meant for registering a Scalar or Table i.e. the REGS or REGT respectively, is always followed by a number of string arguments, which are described below:

* The first string ("**STR**") identifies the Event Description as specified by the corresponding "*event_name*" *Tag* in the *ODM_Behaviour_Specification*, which is specified using the *EDBSLang* language and is the monitoring-behaviour-specification that is read from the **Repository for storing monitoring-behaviour-specifications** for execution by the associated MPAE.

* The second string ("**STR**") identifies the Action Name as specified by the corresponding "*own_action*" *Tag* in the *ODM_Behaviour_Specification*, which is specified using the *EDBSLang* language and is the monitoring-behaviour-specification that is read from the **Repository for storing monitoring-behaviour-specifications** for execution by the associated MPAE.

In addition the REGS or REGT command consists of *sub-commands* or *tokens* that are used for indicating additional information as described below:

* **NAME** is used for indicating the name of the MIB-object (OID) by the string that follows the token (sub-command).

*** END** is used for indicating the end of the request for OID registration.

*** SCAL** is used for indicating the scalar value by the string that follows the token (sub-command).

* **TYPE** is used for indicating the ASN.1 type by the string that follows the token (sub-command).

*** INDX** is used for indicating the "_index_" of the table by the string that follows the token (sub-command).

* **COLM** is used for indicating a successive column of the table and its value by the string that follows the token (sub-command).

* **DATA** is used for indicating that the strings that follow represent the values to be inserted into the Table columns, and are to be put into the registered table one after the other.

A **STR** is any string value of up to 26 bytes of characters, and should not contain any white spaces.

A **NUM** is any numeric value up to 26 digits.

As for ASN.1 types that must be bound to the registered OIDs and are specified using the sub-command: "**TYPE <ASN>** \n", the following types are supported (valid):

- ASN_BOOLEAN
- ASN_INTEGER
- ASN_BIT_STR

- ASN_OCTET_STR
- ASN_NULL
- ASN_OBJECT_ID
- ASN_SEQUENCE
- ASN_SET
- ASN_UNIVERSAL
- ASN_APPLICATION
- ASN_CONTEXT
- ASN_PRIVATE
- ASN_PRIMITIVE
- ASN_CONSTRUCTOR
- ASN_LONG_LEN
- ASN_EXTENSION_ID
- ASN_BIT8
- ASN_APP_OPAQUE
- ASN_APP_COUNTER64
- ASN_APP_FLOAT
- ASN_APP_DOUBLE
- ASN_APP_I64
- ASN_APP_U64
- ASN_APP_UNION
- ASN_PRIV_INCL_RANGE
- ASN_PRIV_EXCL_RANGE
- ASN_PRIV_DELEGATED
- ASN_PRIV_IMPLIED_OCTET_STR
- ASN_PRIV_IMPLIED_OBJECT_ID
- ASN_PRIV_RETRY

Examples of dynamic OID registrations at run-time:

========

Figure 27 gives an example on how an entity i.e. an Action-Script can register a **Scalar OID** and bind data to it. The *Action-Script* (invoked by an EDCE instance) performing the dynamic OID registration must also provide the name of the event (**event_name1**) whose detection caused the action-script to be executed by the associated session, the action name (**action_name1),** and the session-key (**session-key1**) that identifies the associated session, to distinguish the session among the multiple sessions being served by the associated MIB-Manager instance. If the action-script was found specified in multiple session-behaviour-specifications for the same event name, then the associated multiple **session-keys** are supplied by the executing action-script. Our current implementation of the ODM-Probe supports only one ODM-session per "MPAE - MIB

Manager" association and so the need for the third argument: **session-key**, is not yet implemented. This is why it is not included in the examples below.

```
REGS "event_name1"  "action_name1"\n
NAME MyScalar\n
SCAL str_str_str TYPE ASN_OCTET_STR\n
END\n
```

Figure 27: Example on how an entity i.e. an Action-Script can register a Scalar OID **and bind data to it**

Figure 28 gives an example on how to register a **Table OID** and fill it with data. The need for arguments: event name, action name and session-key has already been explained above.

```
REGT "event_name2"  "action_name2"\n
NAME "My-TCP-TrafficStatistics-ComputationTable"\n
INDX Column1 TYPE ASN_OCTET_STR\n
COLM Column2 TYPE ASN_OCTET_STR\n
COLM Column3 TYPE ASN_INTEGER\n
END\n
```

Figure 28: Example on how to register a Table OID **and fill it with data**

The **ODM_MIB_dynreg** server component returns "*return codes*" during the interaction with an entity registering an OID, which are described below.

Success codes:

200 OK

201 Scalar successful registered

202 Table successful registered

Error codes:

400 Registering failed

401 Registering duplicated Name

Parsing error codes:

500 Parsing error

501 Wrong ASN.1 Type

The **ODM_MIB_dynreg** server responds with message: "200 OK" after receiving every subsequent request terminated by the new-line symbol "\n".

7.2.6 Event Detection & Computation Engine

This component is refered to as EDCE (Event Detection and Computation Engine). An instance of an EDCE thread (see **Figure 22**) is created by an MPAE if some computations and or event detections are demanded in the monitoring-behaviour-specification to be executed. An MPAE instance creates an associated EDCE instance when any of the monitoring-behaviour-specifications (belonging to different tentative session-owners) specifies some computations to be performed and the need for event notifications based on the result of a computation(s) as event-notification triggers. The key ODM-principle behind the design of the EDCE component is **Principle-P2**, which covers programmable monitoring. As such, an EDCE instance is managed by an MPAE, which influences its pausing, resumption or termination of execution, according to the session management primitives received by the associated MPAE instance. **Note:** Wherever we say "the EDCE" in the text, we are referring to a particular instance of the EDCE, bearing in mind that multiple instances can be created on the probe by their associated individual MPAE instances. A pseudo SDL (Specification and Description Language [ITU-T Z.100]) Model of the Finite State Machine (FSM) of the behaviour and functionality of an EDCE instance is presented in **Figure 34** in section 7.3. The Model augments the textual description of this component. The purpose of the Model is to help the implementer further develop and optimize the models during code-implementation, or contrast the approach taken in the design of the probe to other approaches that may be considered to be more optimized. The functions as well as the interfaces of the EDCE are described below:

- The EDCE component accesses the packets captured by the associated MPAE instance, examines the packets, performs computations, detects specified events, which could be based on packet details i.e. protocol fields or could be based on some computations, and sends notifications to the designated recipients or triggers the execution of some special task programs called "*Action-Scripts*". There are two types of event triggers whose Boolean values are derived from captured traffic: the first event trigger is based on the detection of the traffic specified by a traffic filter resulting in logic values that express the "*presence*" or "*absence*" of traffic within a specified *traffic-detection-attempt window*. The EDBSLang includes the following tags for use in monitoring-behaviour-specifications: <*packet_captured*> and <*no_packet_captured*> to request for an action if the traffic under detection has been found to be present or absent, respectively. For more information on the semantics of these tags we refer to [Chaparadza07a]. An event trigger(s) can be based on analyzing captured traffic and performing some computation(s) in order to determine that event(s) specified in the monitoring-behaviour-specification(s) and inferred from the computations have occurred. When an event trigger has fired, either an event notification is sent to designated recipients specified in the monitoring-behaviour(s) under execution or an "*Action-Script*" is executed. The EDCE is meant to perform some computations such as some statistics on the captured traffic, comparing against thresholds specified in the monitoring-behaviour-specification(s), checking against events specified as events to be watched for and issuing event notifications to notification recipients designated in the monitoring-behaviour-specification(s).

- The EDCE may also need to create a `packet discarder` thread that takes care of discarding the packets that have already been used in computations and are no longer required. This is done in order to free memory by discarding redundant packets from the packet buffer of captured packets at regular intervals as specified by the ODM-session-owner by indicating the PacketDiscard Rate as a parameter in the *start-session* primitive (see earlier descriptions of ODM-primitives). The PacketDiscard Rate specified by a session-owner is respected only when there are enough resources to store a number of packets while the EDCE is serving only one session that specified a PacketDiscard rate or that among the sessions being served by the EDCE instance, only one has specified a PacketDiscard rate. The EDCE also employs aggregation of the PacketDiscard rate to keep respect of the largest value, all subject to availability of resources, otherwise the EDCE uses its own discard rate to adapt to the rate at which captured packets are filling the buffer used to temporarily store captured packets.
- To enhance performance for SNMP operations on OIDs when the ODM-Probe is implemented on multi-processor architectures that support threading, the EDCE can be made to implement some of the OIDs related to computations such as traffic statistics, computed rate of flow in order to share the burden with the associated MIB Manager instance. Therefore, the EDCE can be implemented as an SNMP subagent [RFC 2741]. In our current implementation, only the MIB Manager code is implemented as a subagent, due to the limitation of NET-SNMP libraries in supporting multiple subagents to run as threads (not processes) on a network node, as explained earlier in the functional description of the MIB-Manager. Event notifications are based on `SNMP-Inform` and `Trap` types of messages
- Apart from being influenced by the associated MPAE in signals to pause or resume execution, the behaviour of an EDCE instance depends on the EDBSLang-based monitoring-behaviour-specification parsed by the associated MPAE instance.
- The EDCE invokes *Action-Scripts* for each specific event it has detected. The EDCE supplies the following arguments to each *action-script* it invokes: the **event_name**, the **action_name** (which must be the same as the filename of the action-script). If the action-script was found specified in multiple session-behaviour-specifications for the same event name, then the associated multiple **session-keys** are supplied as arguments to the invoked *action-script*. The invoked action uses the arguments supplied by the EDCE at its invocation time, during the dynamic, on-the-fly registration of OIDs to the associated MIB Manager instance (refer to the examples of dynamic MIB registration described earlier). Therefore the EDCE interfaces with the IETF-SCRIPT-MIB in order to invoke action scripts. The EDCE can also be made to access a scripts-directory to invoke a script (s) directly instead of using the IETF-SCRIPT-MIB to invoke scripts stored in the SCRIPT-MIB. Invoking scripts such as *Perl scripts* via accessing a scripts-directory would rely on the fact that the execution engine used to execute the scripts is installed into the operating system, and not necessarily being part of the SCRIPT-MIB.
- In its overall behaviour, the EDCE also performs (re)-scheduling and (re)-ordering of all "*Computations to be performed*" specified in the assigned monitoring-behaviour-

specifications. This results in the "*Computations to be performed*" getting (re)-ordered as follows:

{**Computation X_1** to be computed at **Time T_1**; **Computation X_2** to be computed at **Time T_2**; …… **Computation X_N** to be computed at **Time T_N**}

Whereby the (re)-ordering is done according to the **Time** at which the **Computation** must be performed, such that $T_1 < T_2 < T_3$ …… T_N.

- In its overall behaviour, the EDCE also performs (re)-scheduling and (re)-ordering of all "*Events to be detected*" specified in the assigned monitoring-behaviour-specifications. This results in the "*Events to be detected*" getting (re)-ordered as follows:

  {**Event X_1** to be detected at **Time T_1**; **Event X_2** to be detected at **Time T_2**; …… **Event X_N** to be detected at **Time T_N**}

  Whereby the (re)-ordering is done according to the **Time** at which the **Event** must be detected, such that $T_1 < T_2 < T_3$ …… T_N.

- In its overall behaviour, the EDCE also performs (re)-scheduling and (re)-ordering of all "*Event associated Actions*" specified in the assigned monitoring-behaviour-specifications. This results in the "*Event associated Actions*" getting (re)-ordered as follows:

  {**Action X_1** to be performed i.e. executed at **Time T_1**; **Action X_2** to be perfomed i.e. executed at **Time T_2**; …… **Action X_N** to be computed at **Time T_N**}

  Whereby the (re)-ordering is done according to the **Time** at which the **Action** must be performed, such that $T_1 < T_2 < T_3$ …… T_N.

7.2.7 Repository for storing run-time state information of monitoring functions

This repository (see **Figure 22**) is used for storing run-time state information of monitoring functions i.e. processes and threads such as the MPAE, MIB-Manager and ECE instances. The processes and threads export run-time state information into the repository. For example, the MPAE and all child threads, including the *MPAE´* push state information into the repository. Run-time state information of an MPAE instance includes the current state of execution e.g. "*capturing*" or "*error-state*", "*paused*", "*terminated*", and other type of state related information such as the previously handled *signal(s)* from the odmRHACC. Run-time state information is required by entities such as the odmRHACC and the Autonomic Node Manager (ANM) of the node in determining problems to do with monitoring functions and take actions accordingly, such as re-starting affected components.

7.2.8 SNMP Master Agent

This component's role (see **Figure 22**) is the marshalling and dispatching of SNMP communications between automated tasks (ODM-session-owners) and the MIB Manager and EDCE instances running on the box (system or node). MIB Manager and EDCE instances that register MIB regions and communicate with the Master Agent are called sub-agents [RFC 2741]. All interactions between sub-agents and the Master use the AgentX [RFC 2741] protocol. The master agent handles all the data collection requests (SNMP messages) such as ***snmp-get***, ***snmp-getnext, snmp-walk*** by dispatching those requests to the appropriate running sub-agents i.e. the MIB Manager and EDCE instances. The dispatching is done on the basis of the MIB sub tree hook registrations for unique branches "`1.3.6.1.3.`**`X`**" where unique values "X" are assigned by the odmRHACC. The registrations of branches are done by the sub-agent instances at their instantiation time. The SNMP Master Agent also relays SNMP messages such as SNMP-Trap and SNMP-INFORM messages coming from MIB Manager and EDCE instances running on the box, to intended automated tasks.

7.2.9 IETF-SCRIPT-MIB Environment

The SCRIPT-MIB [ScriptMIB] (see **Figure 22**) is an environment for storing, executing and managing the execution of scripts. Automated tasks can upload scripts into the SCRIPT-MIB. These kinds of scripts are what we refer to as *Action-Scripts* that are executed by EDCE instances upon the detection of specified events. The Autonomic Node Manager (ANM) has an interface to the SCRIPT-MIB in order to use the SCRIPT-MIB for managing the ODM-Probe by uploading special types of scripts that configure and restrict the behaviour of the SCRIPT-MIB, to allow it to only accept script uploads and disallow their direct execution by automated tasks. The odmRHACC is the one that then configures the SCRIPT-MIB to indicate the remote systems and identifiers of session-owners i.e. automated tasks that should be allowed by the SCRIPT-MIB to upload scripts and to enable them to directly execute scripts, bypassing execution by proxy via an EDCE instance. This is one of the issues for further research.

7.3 Pseudo SDL Models and algorithms of the ODM-Probe

This section presents the pseudo SDL models that illustrate the state-transitions and behaviors of the main components of the ODM-Probe. The purpose of the models is to help the implementer further develop and optimize the models during code-implementation, or contrast the approach taken in the design of the probe to other approaches that may be considered to be more optimized. The models presented here augment the textual description of the functionality of the

components given in sections 7.2.1 - 7.2.6, namely the odmRHACC, MPAE, EDCE, and MIB-Manager.

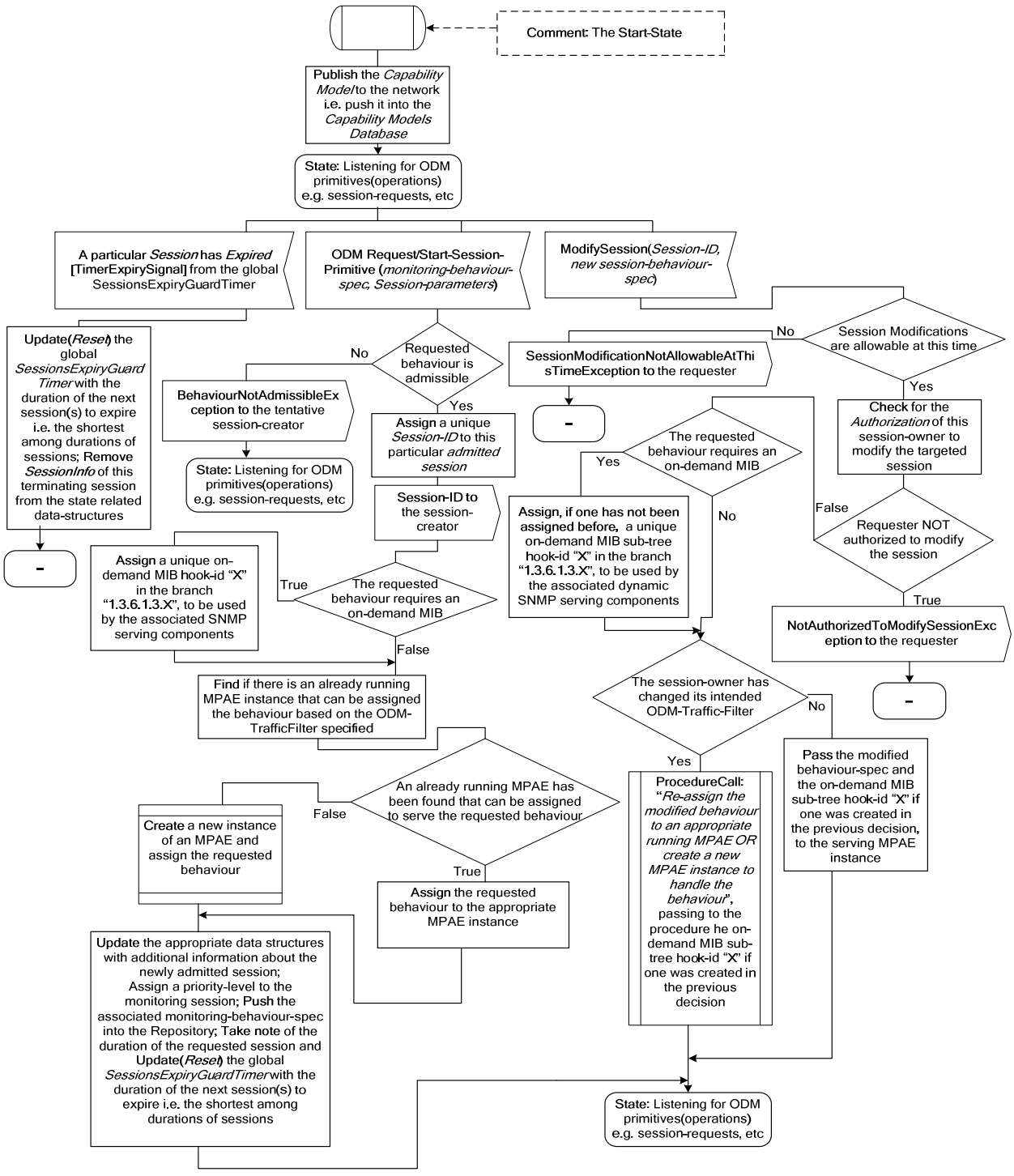


Figure 29: Pseudo SDL Model of the Finite State Machine of the odmRHACC (part 1 of 2)

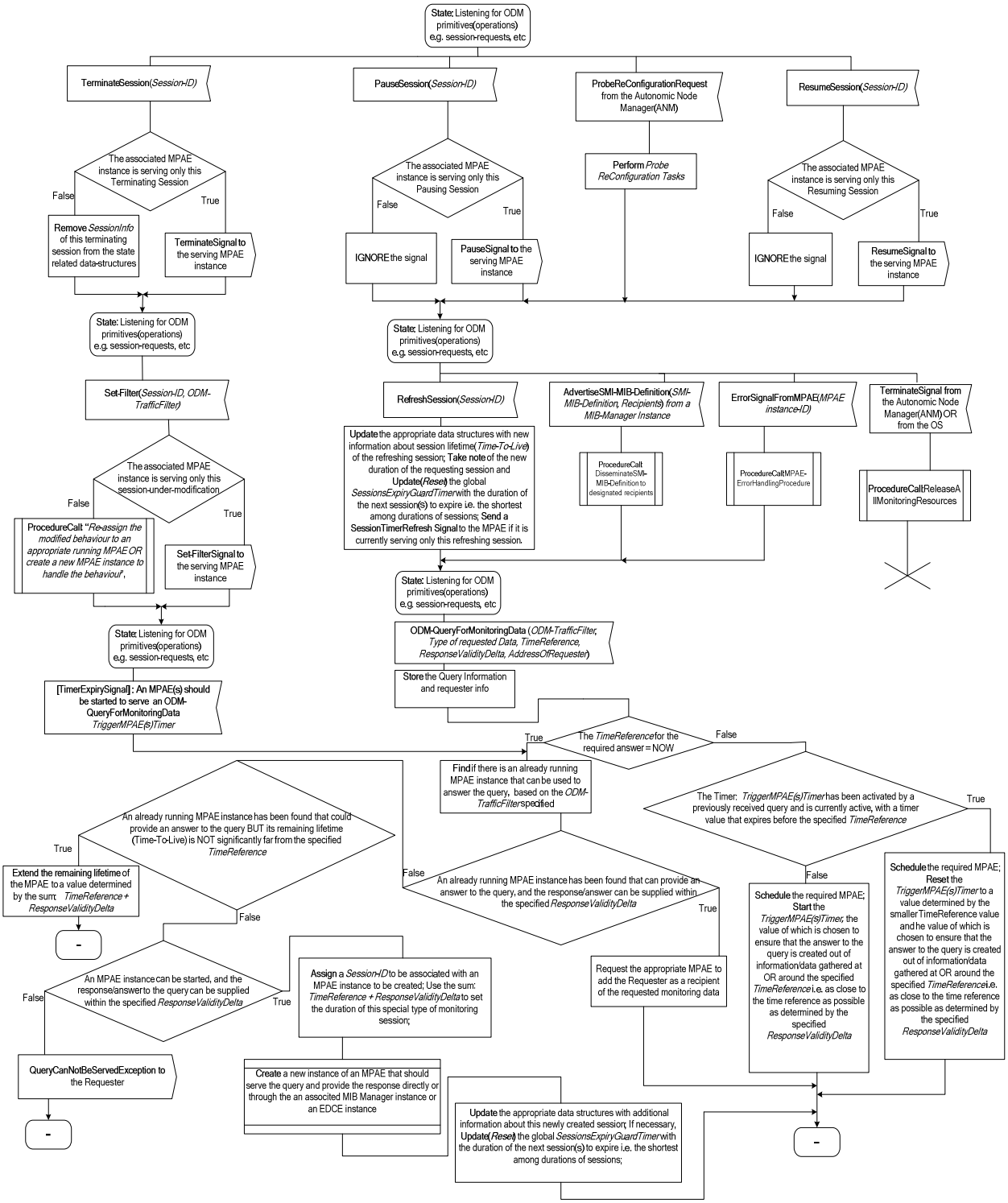


Figure 30: Pseudo SDL Model of the Finite State Machine of the odmRHACC (part 2 of 2)

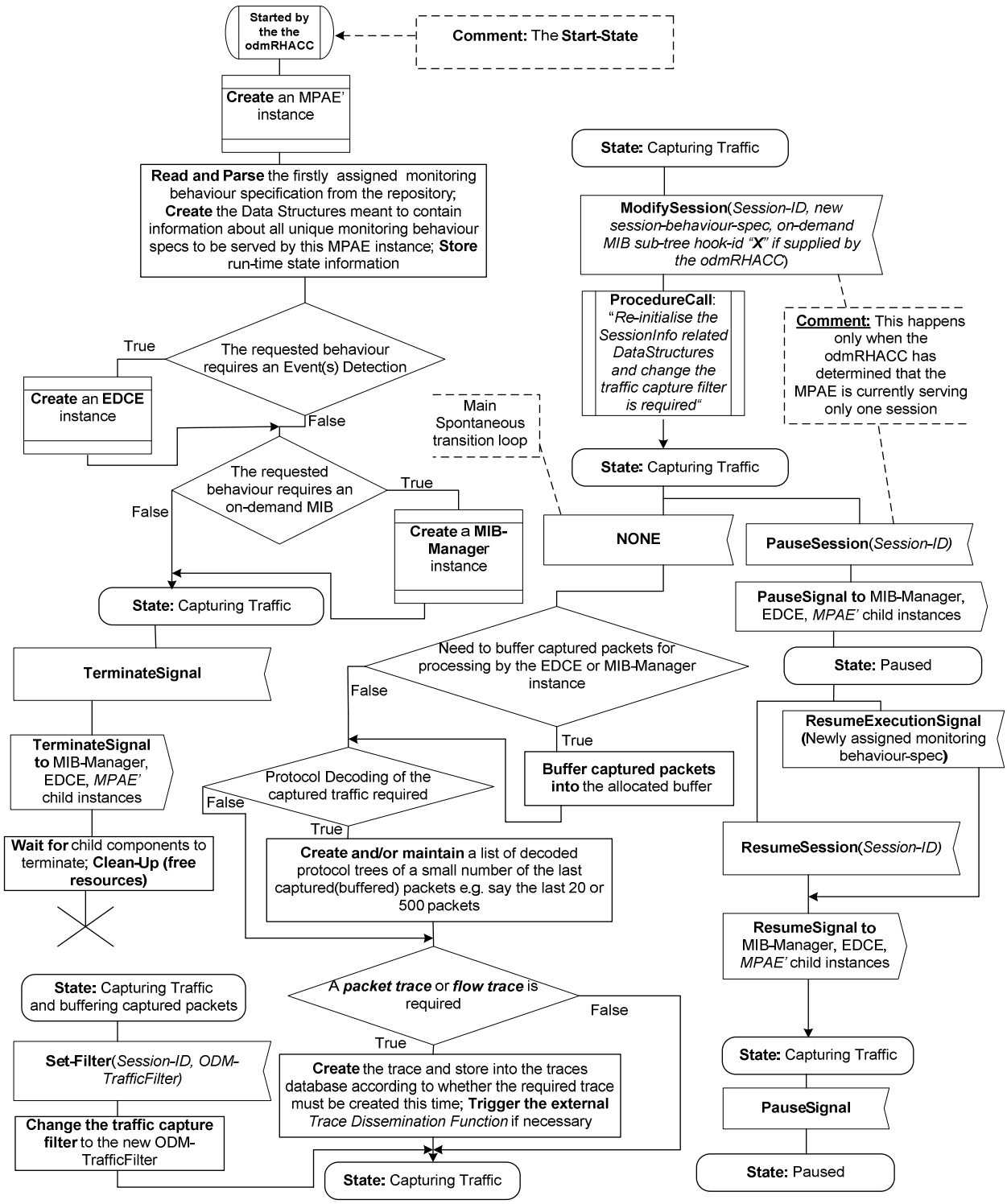


Figure 31: Pseudo SDL Model of the Finite State Machine of an MPAE instance

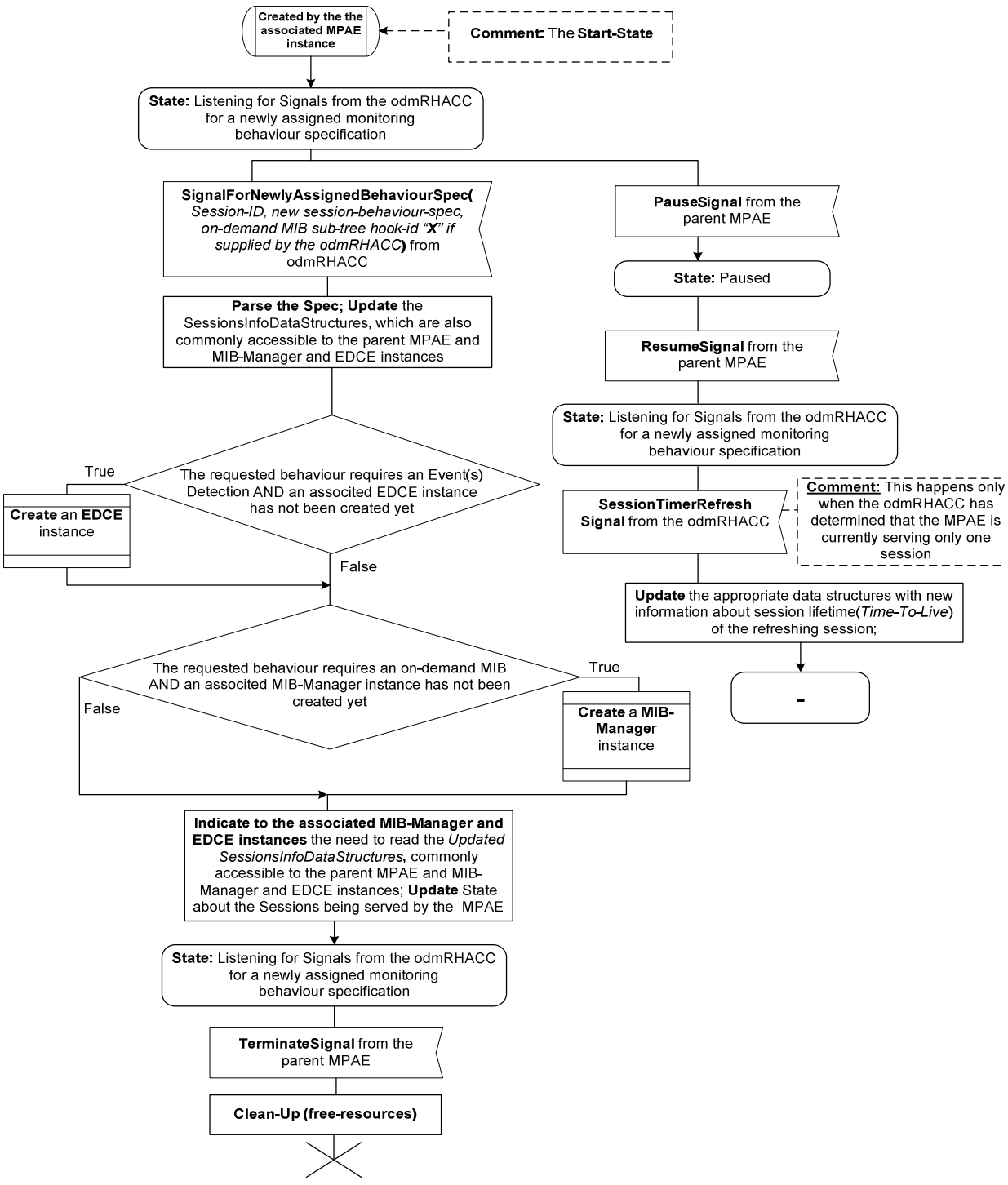


Figure 32: Pseudo SDL Model of the Finite State Machine of an MPAE' (MPAE-prim) instance

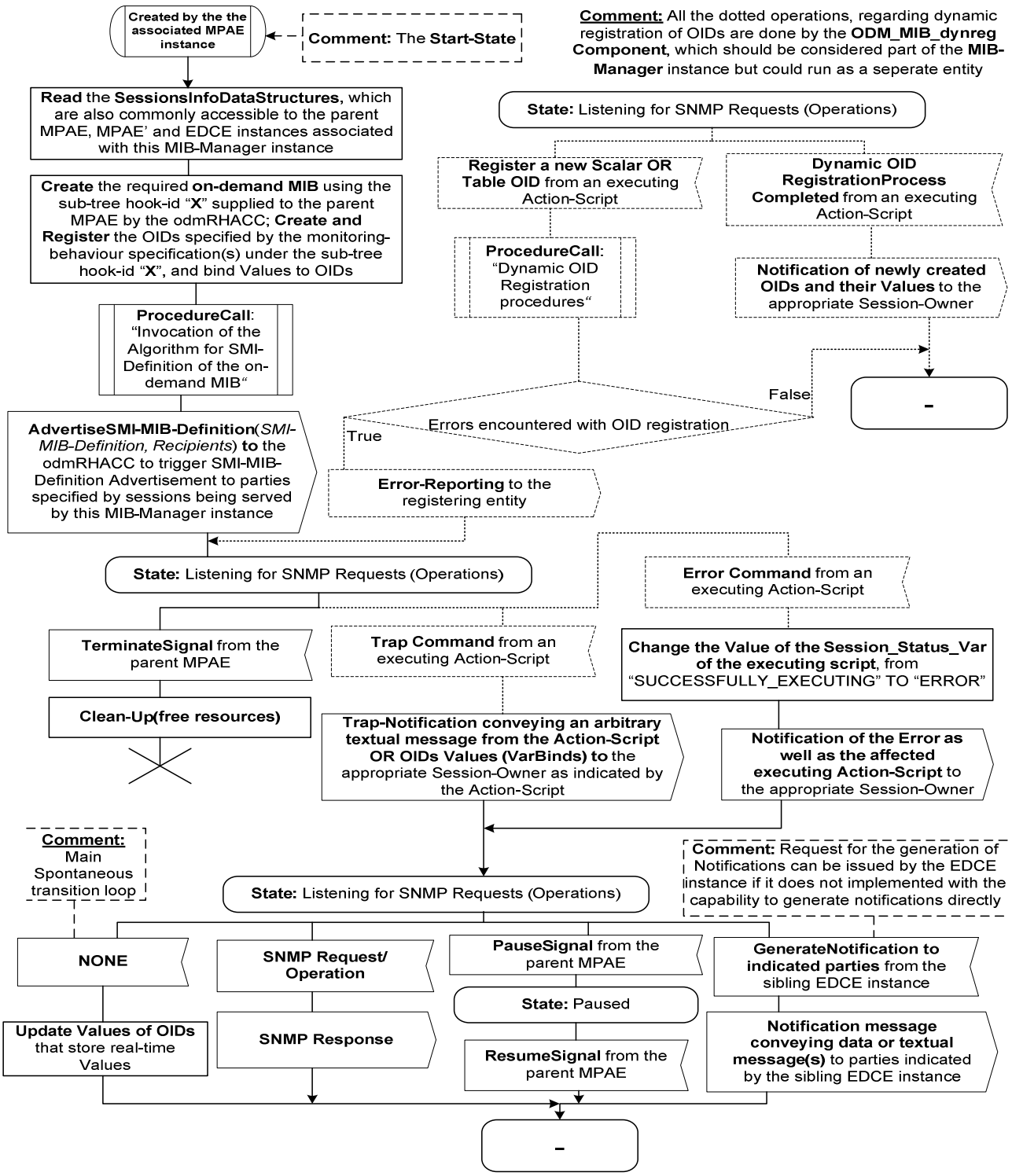


Figure 33: Pseudo SDL Model of the Finite State Machine of a MIB-Manager instance

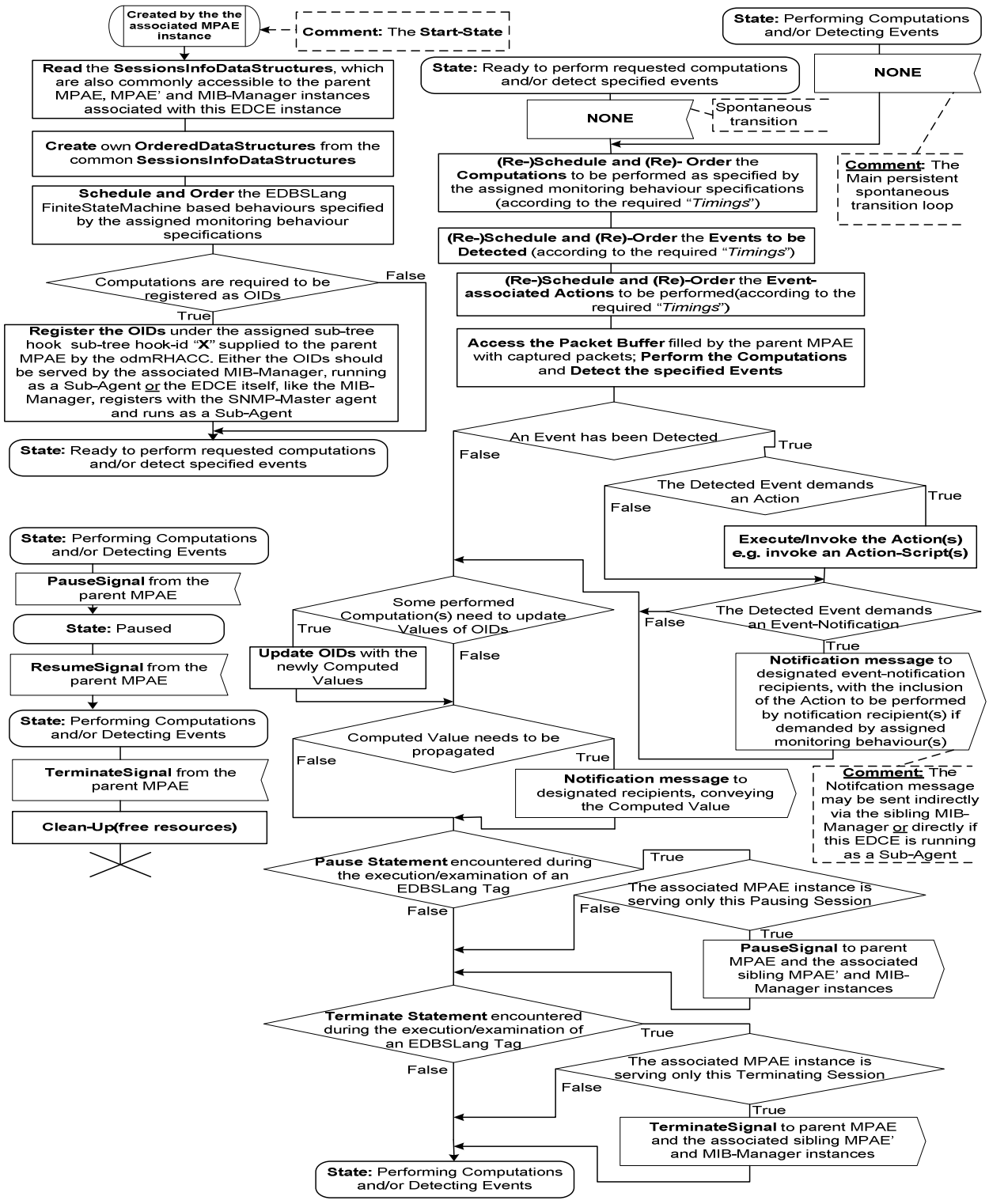


Figure 34: Pseudo SDL Model of the Finite State Machine of an EDCE instance

7.4 The Implementation Platform Used

The ODM-Probe was implemented and evaluated on a Linux platform. Most of the modules taken from the open source tools and libraries used to implement the probe can be used on other platforms mentioned in the documentation of the respective libraries. A SUSE Linux 9.3 release 2.6.11.4-20a-smp on an i686 machine platform was used. The versions of NET-SNMP (net-snmp-5.2.1) [Net-SNMP], Ethereal (ethereal-0.10.11) [Ethereal] and the other libraries are all documeneted in the prototype code of the ODM-Probe [ODM-Probe-Source-Code]. The ODM-Probe is implemented in user-space, not in the kernel, due to the nature of the open-source libraries that were modified to implement some functions of the probe. Details of the maturity level of the implementation are provided later in section 8.4.

8 Evaluation of the ODM-Probe and Case Study

In this chapter, we present an evaluation of the ODM-Probe and the key ODM-Principles that were selected for evaluation. We present the implementation platform used for the prototype implementation of the ODM-Probe, and a Case Study performed on the interaction of the Automated Tasks with an ODM-capable component in a distributed environment in which On-Demand MIBs are instantiated. Also, as part of the Case Study, we also present *Metrics* associated with the Components of the ODM-Probe; the maturity-level of the current implementation of the ODM-Probe; Contrasting the ODM-Probe to today's well-known monitoring approaches (tools); how to instrument ODM-Probes into network elements (see definition of *network element*$_{re\text{-}defined}$) and migrating current monitoring frameworks to supporting the ODM-Paradigm; and some concluding remarks and insight into the required further work.

8.1 Interaction of the components of On-Demand Monitoring

Figure 35 shows the distributed components that play a role in On-Demand Monitoring in a distributed environment. The diagram is based on the current implementation of the ODM-Probe, which currently does not support multiple sessions per single set of MPAE, MIB-Manager and EDCE instances.

On the upper part of the figure, a network node "X" is shown where an ODM-session creator (owner) i.e. an automated task is running. The session-creator (owner) uses an SSL based client to interact with the odmRHACC component of the ODM-Probe running on node "Y", assuming that it has targeted this component for triggering a monitoring-behaviour. The GUI (Gprobe) can be used by a human to emulate a session-creator. The SNMP components that interact with each other and with the session-owner during event notifications and the reading of monitoring data from an On-Demand MIB created by the ODM-Probe are also shown, namely:

- the SNMP Trap/Inform daemon that receives notifications on behalf of the session-owner and informs the session-owner;
- the SNMP applications used by the session-owner for reading data from MIBs(including On-Demand MIBs created for the session-owner on the ODM-Probe);

- An On-Demand MIB File Receiver daemon that receives the definitions in SMI [RFC 2578] of On-Demand MIBs generated and advertised by the ODM-Probe.

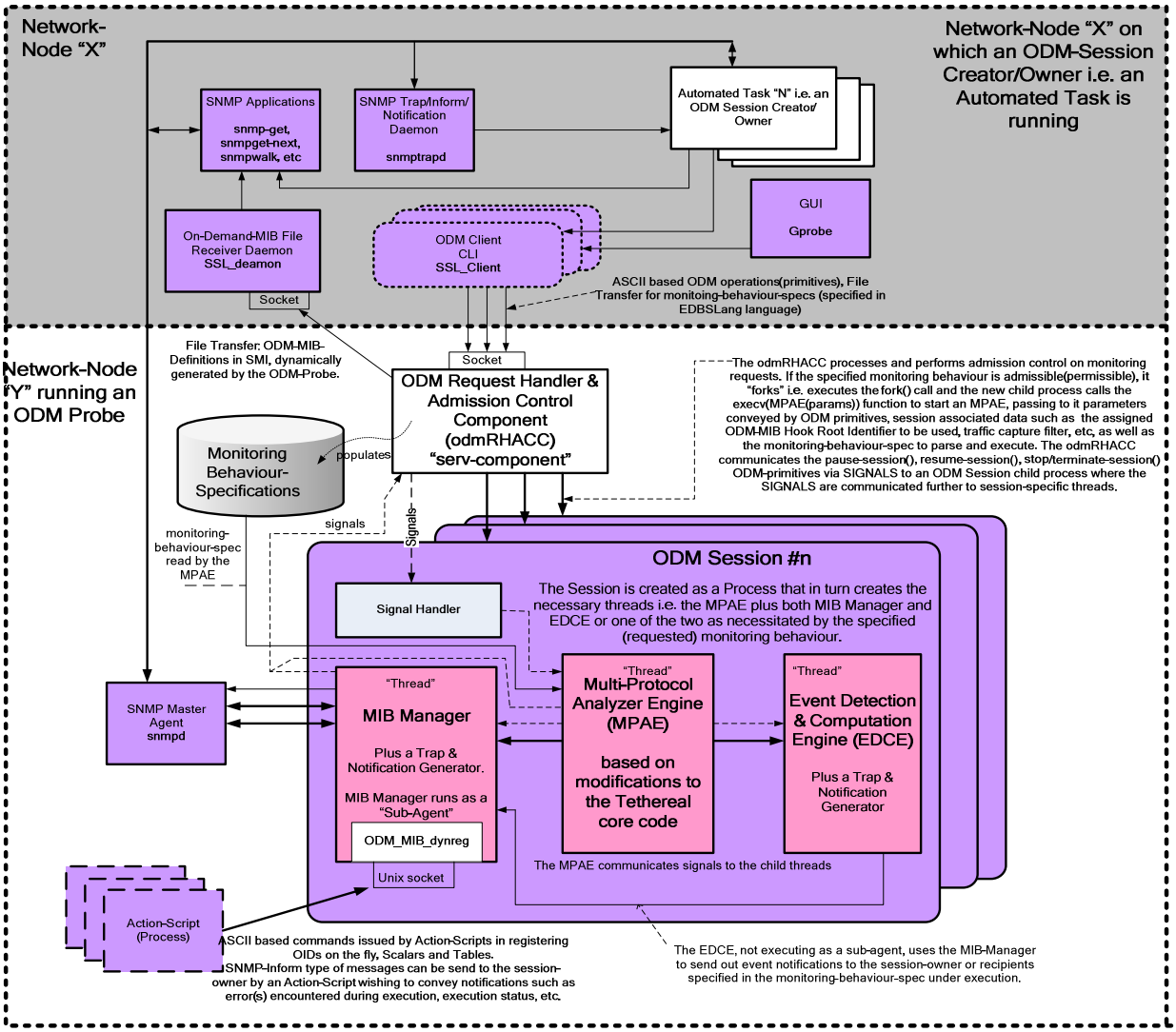


Figure 35: Components for On-Demand Monitoring interacting with an Automated Task in a distributed environment

On the lower part of the figure, the odmRHACC component is shown and the components it interacts with, namely:

- The Repository for storing monitoring-behaviour-specifications; the ODM-sessions started as processes which in turn create the necessary threads: the MPAE, MIB-Manager and EDCE instances. The current implementation does a one-to-one mapping between an ODM-session and an MPAE and its child threads: MIB-Manager and EDCE instances;
- a signal handler for relaying signals pertaining to primitives issued by the session-owner;

- The SNMP Master Agent;
- The interaction between *Action-Scripts* and the associated MIB-Manager instance for dynamic OID registrations.
- Signals received from the odmRHACC are relayed by the MPAE instance to its child threads as described earlier along the functional description of MPAE run-time component.
- The MIB Manager's algorithm for SMI [RFC 2578] definition and advertisement of the created On-Demand MIB, issues signals to the odmRHACC, which then sends (advertises) the ODM-MIB-Definitions to the session-owner via the On-Demand MIB File Receiver daemon on node "X".
- In the current implementation, an EDCE can not run as an AgentX subagent as explained earlier, and so for event notifications, it uses the associated MIB-Manager instance to send out event notifications to designated recipients.
- An MPAE instance can also send signals to the odmRHACC when it experiences critical problems such as errors during its execution lifetime.

8.2 Metrics associated with the Components of the ODM-Probe

8.2.1 Processing Times

The following are the key processing times associated with the components. Clearly, a number of the processing times associated with the presented ODM-Probe architecture and its components can be identified. Here, we give only a handful of some of the processing times already identified. We distinguish between *composite processing times* on the *monitoring-service* level and the *internal processing times associated with transitions* within the individual Extended Finite State Machines (EFSMs) of the individual components of the ODM-Probe. The processing times can be measured on any platform on which the ODM-Probe is implemented, by "timing within the code", the relevant operations i.e. measuring the time taken to execute the operations. In this section, we also discuss the determining factors that influence the processing times.

(A) Composite Processing Times on the *monitoring-service* level

Monitoring_BehaviourAdmission_Time (MBAdmTime) -: is the overall time it takes for a monitoring request to be examined by the odmRHACC and for the associated monitoring-behaviour-specification to be parsed by the MPAE such that all the monitoring functions are ready to execute. The determining factors that influence this processing time are discussed below—in part (B) of this sub-section.

Monitoring_Behaviour_Spec_Parsing_Time (MBSParse_Time) -: is the time it takes an MPAE to parse a monitoring-behaviour-specification. The determining factors that influence this processing time are discussed below—in part (B) of this sub-section.

Monitoring_Behaviour_Modification_Time (MBMod_Time) -: is the time it takes the odmRHACC to process a modify-session request from a session-owner that does not result in reassigning the new behaviour to a different MPAE instance, such that the new behaviour is ready to commence execution. The determining factors that influence this processing time are discussed below—in part (B) of this sub-section.

Monitoring_Behaviour_Reassignement_Time (MBReass_Time) -: is the time it takes the odmRHACC to reassign a behaviour requested by a modify-session primitive that results in the modified behaviour being reassigned to a different MPAE than in the case of previous behaviour. The determining factors that influence this processing time are discussed below—in part (B) of this sub-section.

Monitoring_Behaviour_Pausing_Time (MBPause_Time) -: is the time it takes all the components (MPAE, MIB-Manager, EDCE instances) associated with the one and only session to pause execution, after receiving a pause signal. The determining factors that influence this processing time are discussed below—in part (B) of this sub-section.

Monitoring_Behaviour_Resumption_Time (MBResume_Time) -: is the time it takes all the components (MPAE, MIB-Manager, EDCE instances) associated with the one and only session to resume execution, after receiving a resume signal. The determining factors that influence this processing time are discussed below—in part (B) of this sub-section.

ODM-MIB_Creation Time -: is the time it takes for a MIB-Manager instance to create an On-Demand MIB—whose OIDs were specified in a monitoring-behaviour specification, and advertise the SMI MIB definition to the session-owner. The determining factors that influence this processing time are discussed below—in part (B) of this sub-section.

OID-Registration Time-: is the time it takes for a MIB-Manager instance to register a new OID into the MIB tree. The determining factors that influence this processing time are discussed below—in part (B) of this sub-section.

(B) Internal Processing Times associated with FSM transitions, and the determing factors

These types of processing times relate to the time it takes for any of the components of the ODM-Probe to perform any of the *transitions* described by the FSM Models presented in chapter 7—section 7.3. The following factors determine the *processing times associated with FSM transitions* which in turn have impact on the *composite processing times* on the *monitoring-service* level described above—see part (A).

- Platform related factors such as memory access and speed, processor power, etc.
- Packet filtering and the hierachical filtering employed by the ODM-Probe, demultiplexing of traffic flows and data coyping within and among the components. These factors have their own performance related issues described in detail in [Varghese05].
- Code optimization techniques employed in the threads, processes, timers and context switching. These factors have their own performance related issues described in detail in [Varghese05].
- The number of monitoring-sessions running on the system has impact on the processing times that result in lookups being performed in using monitoring-session-identifiers to locate and modify session-specific data, or in the cases where global variables need to be adjusted.

8.2.2 Evaluating the *Metrics* related to *Resource-Consumption* dynamics

8.2.2.1 Overview

The methodology for determining the dynamics of the **Monitoring Effort** defined in chapter 4—section 4.1.1 is applicable to monitoring probes such as the ODM-Probe. Here, in this section we present and illustrate the methodology as applied to ODM-Probe. It is worthy pointing out that an exhaustive evaluation requires a full implementation of the functional specification given in the previous chapter. Hence, due to the fact that the ODM-Probe implementation is not a full implementation, we focus on how to evaluate the dynamics of the function below (defined in chapter 4–section 4.1.1, based on the selected key composable monitoring-services and associated monitoring-functions implemented and evaluated.

> $\boldsymbol{M_{effC}}$ = ***MonitoringEffort*** *on a Monitoring Component* that processes *monitoring-requests* and executes the required *monitoring-functions, is expressed as total amount of resources consumed by the monitoring-functions at a given time:*
>
> $$"Total\ \_\ R\ \_\ in\ \_\ use_t" = \quad f(t)$$

As discussed earlier, in chapter 4–section 4.1.1, determining the characteristic of the above function helps in two things:

(1): *Determining the way an "actual deployment" of the ODM-Probe should queue and process monitoring-requests*;

(2) *Using the following measurements of resource-demand variables or estimations thereof in the design of the admission control algorithm, as well as tuning the admission control policies employed by the Monitoring Component on monitoring-requests*: (a) The pre-determined measurements of resources required by each individual monitoring-functions under varying load on the system hosting the ODM-Probe; (b) The estimations of the following resource measurement variables (defined in chapter 4–section 4.1.1) at a particular time "t" of processing monitoring-requests:

$$\{ \; Total\_R\_already\_seized_t, \; Total\_R\_shared\_Funcs\_Not\_yet\_executing_t, \; Total\_R\_for\_Funcs\_unique\_to\_each\_monitoring\text{-}request_t \; \}$$

It is worthy pointing out that approaches applied to systems supporting job assigments and scheduling such as Grids, like those presented in [Chtepen09] [Chtepen08] may also be applied to on-demand monitoring systems like the ODM-Probe.

To illustrate how to determine the dynamics of the $\boldsymbol{M}_{\mathbf{effC}}$ function and the associated resource consumption dynamics defined in chapter 4–section 4.1.1, we consider a number of "*cases"* for which monitoring-requests are issued with varying monitoring-functions in the different cases being requested for execution by the ODM-Probe.

Assumption: Assuming that an ODM-Probe is ready for serving *monitoring-requests*.

Let's consider a *monitoring-behaviour-specification* such as the one in Figure 14. When such a monitoring-behaviour-specification is conveyed by a *monitoring-request*, a *monitoring-session* should be created by the ODM-Probe if the request is admitted. From the example *monitoring-behaviour-specification* we see that the following set of *monitoring-functions* that compose a monitoring-service when executed by the ODM-Probe:

{F_1 = *packet-capturing-function* that captures the specified traffic; F_2 = *event-detection-and-notification-function*; F_3 = *on-demand-MIB-creation-function* for some specified SNMP Object-Identifier (OID)}

The requested monitoring-functions, considered as micro-functions can be maped to macro-functions that are instantiated by the ODM-Probe to satisfy the needs of a monitoring-session, and are created to run as "*threads*". These macro-functions are an *MPAE instance*, an *EDCE instance* and a *MIB-Manager instance*. Different types of monitoring-behavior-specifications can be conveyed by different types of monitoring-requests issued to the ODM-Probe. For example, a monitoring-request may request an *on-demand-MIB-creation-function* to create more SNMP Object-Identifiers (OIDs) such as those specified in Figure 12.

Since the macro-functions such as F1, F2, F3 would execute as threads on the ODM-Probe, it means methods and tools for measuring resource demands and consumption such as memory by processes and threads on platforms such as Linux [Linux_runtime_resource_measurement] [Linux_runtime_resource_measurement2], which include instrumenting code within the source code, for measuring resource-consumption by threads and processes, can be applied for measuring the resource demands for the macro-functions. The measurements help in estimating at any given time "*t*" of processing monitoring-requests, the following resource requirements estimation variables: $Total\_R\_already\_seized_t$, $Total\_R\_shared\_Funcs\_Not\_yet\_executing_t$, $Total\_R\_for\_Funcs\_unique\_to\_each\_monitoring\text{-}request_t$.

Definition: "**FunctionUniqueness"**: In order to determine whether a function is a "shared" or "unique" monitoring-function, we say that we have a case of a "shared Function" for at least 2 monitoring-requests if:

> **(**The *Traffic Filter* for a specific monitoring-request under examination matches that of another monitoring-request being processed at time "t") ***AND***
>
> **((**The *Requirements for Events-Detection* are the same for the monitoring-requests being examined) OR (The *Requirements for on-demand monitoring data generation or access* are the same for the monitoring-requests**))**

In order to make the measurements and estimations of the dyanmics of the $\boldsymbol{M_{effC}}$ function, we propose **<u>two steps</u>**:

Step-1:

Measuring the resources (memory, CPU and bandwidth in data transfer) required by each individual monitoring-functions under varying load on the system hosting the ODM-Probe. This means measuring and benchmarking each monitoring-functions supported by the probe, under varying load conditions. For the macro-functions supported by the ODM-Probe, i.e. an *MPAE instance*, an *EDCE instance* and a *MIB-Manager instance*, which run as threads, the measurements concern the resources demanded by each instantiated thread under varying load conditions.

In order to measure memory consumption for the individual macro functions that run as threads of a particular monitoring-session process created by the ODM-probe, a number of tools such as those proposed in [Linux_runtime_resource_measurement] can be used. For example Linux commands "*cat /proc/<pid>/statm*" and "*cat /proc/<pid>/status*" reveal resources consumed by a particular of process-id "*pid*". The commad "*cat /proc/<pid>/maps*" shows the actual memory

areas as they have been mapped into the address space of a particular process, as well as their associated permissions (refer to [Linux_runtime_resource_measurement]).

The trace below is an illustration of the result of executing the "*cat /proc/<pid>/maps*" for a single monitoring-session represented by a single process (an MPAE instance shown at run-time as a modified "*lt-tethereal"*) with its threads for representing an EDCE instance and SNMP related MIB-Manager instance running an AgentX sub-agent, started on the ODM-Probe. The output of this command can be used by the tool provided in [Linux_runtime_resource_measurement2] for determing the memory consumed by the monitoring-session process and its threads. Since there are "shared libraries" as can be seen on the trace below, it means that successive instantiation of MPAEs for newly created monitoring-sessions will not "share" resources (shared-libraries and corresponding memory) since linux loads a shared-library only once. That means in the calculation of memory consumed by individual MPAE processes, shared memory consumed by their shared libraries mut be taken into consideration.

```
 08048000-08092000 r-xp 00000000 08:07 367779    /root/ODM/ethereal-0.10.11/.libs/  lt-tethereal
08092000-08093000 rwxp 0004a000 08:07 367779    /root/ODM/ethereal-0.10.11/.libs/  lt-tethereal
08093000-082de000 rwxp 08093000 00:00 0
40000000-40016000 r-xp 00000000 08:07 12563     /lib/ld-2.3.4.so
40016000-40018000 rwxp 00015000 08:07 12563     /lib/ld-2.3.4.so
40018000-40019000 rwxp 40018000 00:00 0
40019000-4003a000 r-xp 00000000 08:07 356968    /root/ODM/ethereal-0.10.11/wiretap/.libs/libwiretap.so.0.0.1
4003a000-4003b000 rwxp 00021000 08:07 356968    /root/ODM/ethereal-0.10.11/wiretap/.libs/libwiretap.so.0.0.1
4003b000-40831000 r-xp 00000000 08:07 358274    /root/ODM/ethereal-0.10.11/epan/.libs/libethereal.so.0.0.1
40831000-40a5f000 rwxp 007f6000 08:07 358274    /root/ODM/ethereal-0.10.11/epan/.libs/libethereal.so.0.0.1
40a5f000-40a7b000 rwxp 40a5f000 00:00 0
40a7b000-40a83000 r-xp 00000000 08:07 358890    /usr/local/lib/ethereal/plugins/0.10.11/v5ua.so
40a83000-40a86000 rwxp 00007000 08:07 358890    /usr/local/lib/ethereal/plugins/0.10.11/v5ua.so
40a86000-40a8a000 r-xp 00000000 08:07 358854    /usr/local/lib/ethereal/plugins/0.10.11/lwres.so
40a8a000-40a8b000 rwxp 00004000 08:07 358854    /usr/local/lib/ethereal/plugins/0.10.11/lwres.so
40a8b000-40a8d000 r-xp 00000000 08:07 358887    /usr/local/lib/ethereal/plugins/0.10.11/stats_tree.so
40a8d000-40a8e000 rwxp 00001000 08:07 358887    /usr/local/lib/ethereal/plugins/0.10.11/stats_tree.so
40a8e000-40a90000 r-xp 00000000 08:07 358842    /usr/local/lib/ethereal/plugins/0.10.11/cosnaming.so
40a90000-40a91000 rwxp 00001000 08:07 358842    /usr/local/lib/ethereal/plugins/0.10.11/cosnaming.so
40a91000-40a95000 r-xp 00000000 08:07 358824    /usr/local/lib/ethereal/plugins/0.10.11/agentx.so
40a95000-40a96000 rwxp 00004000 08:07 358824    /usr/local/lib/ethereal/plugins/0.10.11/agentx.so
40a96000-40a97000 r-xp 00000000 08:07 358845    /usr/local/lib/ethereal/plugins/0.10.11/coseventcomm.so
40a97000-40a98000 rwxp 00001000 08:07 358845    /usr/local/lib/ethereal/plugins/0.10.11/coseventcomm.so
40a9a000-40aa6000 r-xp 00000000 08:07 15801     /usr/lib/libpcre.so.0.0.1
40aa6000-40aa7000 rwxp 0000b000 08:07 15801     /usr/lib/libpcre.so.0.0.1
40aa7000-40aa9000 r-xp 00000000 08:07 354240    /usr/local/lib/libgmodule-2.0.so.0.600.4
40aa9000-40aaa000 rwxp 00002000 08:07 354240    /usr/local/lib/libgmodule-2.0.so.0.600.4
40aaa000-40b25000 r-xp 00000000 08:07 354157    /usr/local/lib/libglib-2.0.so.0.600.4
40b25000-40b26000 rwxp 0007a000 08:07 354157    /usr/local/lib/libglib-2.0.so.0.600.4
40b26000-40b36000 r-xp 00000000 08:07 16542     /lib/libz.so.1.2.2
40b36000-40b37000 rwxp 0000f000 08:07 16542     /lib/libz.so.1.2.2
40b37000-40b38000 rwxp 40b37000 00:00 0
```

```
40b38000-40b46000 r-xp 00000000 08:07 115640    /lib/tls/libpthread.so.0
40b46000-40b48000 rwxp 0000d000 08:07 115640    /lib/tls/libpthread.so.0
40b48000-40b4a000 rwxp 40b48000 00:00 0
40b4a000-40b7a000 r-xp 00000000 08:07 365963    /usr/local/lib/libnetsnmpagent.so.5.2.1
40b7a000-40b7c000 rwxp 0002f000 08:07 365963    /usr/local/lib/libnetsnmpagent.so.5.2.1
40b7c000-40bc5000 r-xp 00000000 08:07 365958    /usr/local/lib/libnetsnmpmibs.so.5.2.1
40bc5000-40bc9000 rwxp 00049000 08:07 365958    /usr/local/lib/libnetsnmpmibs.so.5.2.1
40bc9000-40c0a000 rwxp 40bc9000 00:00 0
40c0a000-40c21000 r-xp 00000000 08:07 365967    /usr/local/lib/libnetsnmphelpers.so.5.2.1
40c21000-40c22000 rwxp 00016000 08:07 365967    /usr/local/lib/libnetsnmphelpers.so.5.2.1
40c22000-40c96000 r-xp 00000000 08:07 365954    /usr/local/lib/libnetsnmp.so.5.2.1
40c96000-40c98000 rwxp 00074000 08:07 365954    /usr/local/lib/libnetsnmp.so.5.2.1
40c98000-40cb6000 rwxp 40c98000 00:00 0
40cb6000-40cb8000 r-xp 00000000 08:07 115630    /lib/libdl.so.2
40cb8000-40cba000 rwxp 00001000 08:07 115630    /lib/libdl.so.2
40cba000-40cbb000 rwxp 40cba000 00:00 0
40cbb000-40d98000 r-xp 00000000 08:07 50453     /usr/lib/libcrypto.so.0.9.7
40d98000-40dab000 rwxp 000dc000 08:07 50453     /usr/lib/libcrypto.so.0.9.7
40dab000-40dae000 rwxp 40dab000 00:00 0
40dae000-40dcf000 r-xp 00000000 08:07 115639    /lib/tls/libm.so.6
40dcf000-40dd1000 rwxp 00020000 08:07 115639    /lib/tls/libm.so.6
40dd1000-40ee4000 r-xp 00000000 08:07 115638    /lib/tls/libc.so.6
40ee4000-40ee5000 r-xp 00113000 08:07 115638    /lib/tls/libc.so.6
40ee5000-40ee8000 rwxp 00114000 08:07 115638    /lib/tls/libc.so.6
40ee8000-40eec000 rwxp 40ee8000 00:00 0
40eec000-40ef0000 r-xp 00000000 08:07 358881    /usr/local/lib/ethereal/plugins/0.10.11/rtnet.so
40ef0000-40ef2000 rwxp 00003000 08:07 358881    /usr/local/lib/ethereal/plugins/0.10.11/rtnet.so
40ef2000-40ef4000 r-xp 00000000 08:07 358884    /usr/local/lib/ethereal/plugins/0.10.11/rudp.so
40ef4000-40ef5000 rwxp 00001000 08:07 358884    /usr/local/lib/ethereal/plugins/0.10.11/rudp.so
40ef5000-40f03000 r-xp 00000000 08:07 358857    /usr/local/lib/ethereal/plugins/0.10.11/mate.so
40f03000-40f04000 rwxp 0000d000 08:07 358857    /usr/local/lib/ethereal/plugins/0.10.11/mate.so
40f04000-40f16000 rwxp 40f04000 00:00 0
40f16000-40f20000 r-xp 00000000 08:07 358848    /usr/local/lib/ethereal/plugins/0.10.11/gryphon.so
40f20000-40f22000 rwxp 0000a000 08:07 358848    /usr/local/lib/ethereal/plugins/0.10.11/gryphon.so
40f22000-40f36000 r-xp 00000000 08:07 358836    /usr/local/lib/ethereal/plugins/0.10.11/docsis.so
40f36000-40f3d000 rwxp 00014000 08:07 358836    /usr/local/lib/ethereal/plugins/0.10.11/docsis.so
40f3d000-40f45000 r-xp 00000000 08:07 358863    /usr/local/lib/ethereal/plugins/0.10.11/mgcp.so
40f45000-40f47000 rwxp 00008000 08:07 358863    /usr/local/lib/ethereal/plugins/0.10.11/mgcp.so
40f47000-40f4b000 r-xp 00000000 08:07 358822    /usr/local/lib/ethereal/plugins/0.10.11/acn.so
40f4b000-40f4c000 rwxp 00004000 08:07 358822    /usr/local/lib/ethereal/plugins/0.10.11/acn.so
40f4c000-40f54000 r-xp 00000000 08:07 358872    /usr/local/lib/ethereal/plugins/0.10.11/profinet.so
40f54000-40f57000 rwxp 00007000 08:07 358872    /usr/local/lib/ethereal/plugins/0.10.11/profinet.so
40f57000-40f59000 r-xp 00000000 08:07 358875    /usr/local/lib/ethereal/plugins/0.10.11/rdm.so
40f59000-40f5a000 rwxp 00001000 08:07 358875    /usr/local/lib/ethereal/plugins/0.10.11/rdm.so
40f5a000-40f5b000 r-xp 00000000 08:07 358878    /usr/local/lib/ethereal/plugins/0.10.11/rlm.so
40f5b000-40f5c000 rwxp 00001000 08:07 358878    /usr/local/lib/ethereal/plugins/0.10.11/rlm.so
40f5c000-40f60000 r-xp 00000000 08:07 358893    /usr/local/lib/ethereal/plugins/0.10.11/xml.so
40f60000-40f61000 rwxp 00003000 08:07 358893    /usr/local/lib/ethereal/plugins/0.10.11/xml.so
40f61000-40f63000 r-xp 00000000 08:07 358833    /usr/local/lib/ethereal/plugins/0.10.11/ciscosm.so
40f63000-40f64000 rwxp 00001000 08:07 358833    /usr/local/lib/ethereal/plugins/0.10.11/ciscosm.so
40f64000-40f65000 r-xp 00000000 08:07 358869    /usr/local/lib/ethereal/plugins/0.10.11/pcli.so
40f65000-40f66000 rwxp 00001000 08:07 358869    /usr/local/lib/ethereal/plugins/0.10.11/pcli.so
```

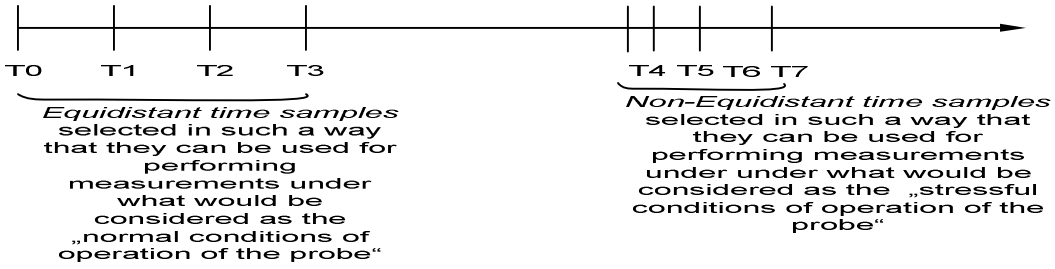
```
40f66000-40f74000 r-xp 00000000 08:07 358830     /usr/local/lib/ethereal/plugins/0.10.11/asn1.so
40f74000-40f75000 rwxp 0000e000 08:07 358830     /usr/local/lib/ethereal/plugins/0.10.11/asn1.so
40f75000-40f79000 rwxp 40f75000 00:00 0
40f79000-40fae000 r-xs 00000000 08:07 354643     /var/run/nscd/passwd
40fae000-40fb2000 r-xp 00000000 08:07 358866     /usr/local/lib/ethereal/plugins/0.10.11/opsi.so
40fb2000-40fb4000 rwxp 00003000 08:07 358866     /usr/local/lib/ethereal/plugins/0.10.11/opsi.so
40fb4000-40fb6000 r-xp 00000000 08:07 358839     /usr/local/lib/ethereal/plugins/0.10.11/enttec.so
40fb6000-40fb7000 rwxp 00002000 08:07 358839     /usr/local/lib/ethereal/plugins/0.10.11/enttec.so
40fb7000-40fbe000 r-xp 00000000 08:07 358860     /usr/local/lib/ethereal/plugins/0.10.11/megaco.so
40fbe000-40fbf000 rwxp 00007000 08:07 358860     /usr/local/lib/ethereal/plugins/0.10.11/megaco.so
40fbf000-40fc7000 r-xp 00000000 08:07 358851    /usr/local/lib/ethereal/plugins/0.10.11/irda.so
40fc7000-40fc9000 rwxp 00008000 08:07 358851     /usr/local/lib/ethereal/plugins/0.10.11/irda.so
40fc9000-40fcb000 rwxp 40fc9000 00:00 0
40fcb000-40fd2000 r-xp 00000000 08:07 358827     /usr/local/lib/ethereal/plugins/0.10.11/artnet.so
40fd2000-40fd5000 rwxp 00006000 08:07 358827     /usr/local/lib/ethereal/plugins/0.10.11/artnet.so
40fd5000-40fdb000 r-xs 00000000 08:07 115619     /usr/lib/gconv/gconv-modules.cache
40ffb000-41003000 r-xp 00000000 08:07 115634     /lib/libnss_files.so.2
41003000-41005000 rwxp 00007000 08:07 115634     /lib/libnss_files.so.2
41005000-41038000 r-xp 00000000 08:07 15792      /usr/lib/locale/en_GB.utf8/LC_CTYPE
41038000-41039000 ---p 41038000 00:00 0
41039000-41239000 rwxp 41039000 00:00 0
41239000-4126e000 r-xs 00000000 08:07 378106     /var/run/nscd/dba9tRJj (deleted)
4126e000-4126f000 ---p 4126e000 00:00 0
4126f000-4146f000 rwxp 4126f000 00:00 0
bffea000-bfffe000 rwxp bffea000 00:00 0
bfffe000-c0000000 rw-p bfffe000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0
```

Step-2:

In this step, we consider the following *Cases* for which a set of monitoring-requests are issued to the ODM-Probe during a window of experiment *[T_0 –T_n]* described in chapter 4–section 4.1.1. Today's well established methods for benchmarking, determining performance and scalability of software systems, such as given in [Weyuker00] can be applied for the ODM-Probe. We devide the experiment window into some time samples, say in multiples of some fixed number of time units (say, nanoseconds, microseconds, milliseconds or seconds), in such a way that when we execute each of the following cases, we are able to measure the changes in the resources seized (consumed). Here, we consider arrival-rate for a volume of monitoring-requests at the specific time samples. We consider two approaches to running such experiments: (a) a *non-stressing normal condition of operation of the ODM-Probe*; and (b) a *stressing condition of operation of the ODM-Probe*.

In the non-stressing condition of operation: We consider that at first, the experiment is executed with equidistant time samples that could reflect real-usage requirements of monitoring-task invocations on the ODM-Probe. For example, we can consider the magnitude of multiples of *5 seconds [0s, 10s, 20s, 30s]* and then the ODM-Probe is re-initiazed and the experiment is repeated after some time, but this time with non-equidistant time samples emulating arrival-rate for such kind of volume requests, sufficient enough to stress the probe, from *T4-T7* such that *T5-T4=0.3s; T6-T5=0.6s; T7-T6=1s.*

In the stressing-condition of operation, same experiments should be repeated with selected time-sampling that ensures that a volume of monitoring-requests arrive while the probe is still invoking functions pertaining to previously received requests. The order of executing the "cases" described below need to be swapped when conducting more experiments. Experiments that consider "single requests" arriving on the probe at the time smaples, and not "volume requests", should also be conducted.



To derive a set of monitoring-functions that can be associated with different monitoring-requests we consider different parameterizations of the following macro-functions in order to have variations in the monitoring needs of the monitoring-requests. The spectrum of the parameterizations possible is defined by the ODM-primitives defined in the ODM-Probe specification chapter 7, as well as by the EDBSLang presented in section 5.2.1.

{$\mathbf{F_1}$ = *packet-capturing-function ("TrafficFilter", "with or without TrafficTraceCreation")* that captures the specified traffic and with OR without creation of TrafficTraces; $\mathbf{F_2}$ = *event-detection-and-notification-function ("Types of Events")*; $\mathbf{F_3}$ = *on-demand-MIB-creation-function ("Types of SNMP Object-Identifiers")* for some specified Object-Identifiers (OIDs)}.

8.2.2.2 The Experiment Cases

As discussed above, we propose to execute the following cases, while ensuring that the monitoring-sessions created for each monitoring request runs till the very end of the whole experiment when all cases have been executed.

Case-1: A number of *monitoring-requests N* whereby *some of the requested monitoring-functions are not the same (across the requests).* We choose a number of N requests that stresses the system's and probe's resources suffient enough to observe and measure the amount of resources (memory usage, CPU, bandwidth in data transfer) consumed by the monitoring-functions once they are executing. For illustration purposes, we focus on memory consumption and not on the other types of resources that also need to be considered.

$MonReq_1$ = {**F_1** *(filter="ip proto udp", "no trace-generation")*, **F_2** *($Event_1$="traffic detected", $Event_2$="traffic exceeds threshold of 10Mbit/s")*, **F_3** *(OID_1="Protocol-Hierarchy-StatisticsTable", OID_2="EventsDetectedTable")*}

$MonReq_2$ = {**F_1** *(filter="ip proto tcp", "with trace-generation")*, **F_2** *($Event_1$="traffic detected", $Event_2$="traffic exceeds threshold of 10Mbit/s")*, **F_3** *(OID_1="Protocol-Hierarchy-StatisticsTable", OID_2="EventsDetectedTable")*}

$MonReq_3$ = {**F_1** *(filter="ip proto udp", "no trace-generation")*, **F_3** *(OID_1="Protocol-Hierarchy-StatisticsTable", OID_2="rateOfFlow")*}

$MonReq_4$ = {**F_1** *(filter="ip proto rsvp", "with trace-generation")*, **F_2** *($Event_1$="traffic detected", $Event_2$="traffic exceeds threshold of 10Mbit/s")*, **F_3** *(OID_1="Protocol-Hierarchy-StatisticsTable", OID_2="EventsDetectedTable")*}

$MonReq_5$ = {**F_1** *(filter="ip proto rsvp", "with trace-generation")*, **F_2** *($Event_1$="traffic detected", $Event_2$="traffic exceeds threshold of 10Mbit/s")*, **F_3** *(OID_1="Protocol-Hierarchy-StatisticsTable", OID_2="EventsDetectedTable")*}

$MonReq_n$ = {**F_1** *(filter="ip proto sctp", "with trace-generation")*, **F_3** *(OID_1="EventsDetectedTable")*}

The requested monitoring-function **F_1** in both **$MonReq_1$** and **$MonReq_3$** is the same. **$MonReq_4$** and **$MonReq_5$** are identical. When it comes to the handling of a monitoring-request by the current implementation of the ODM-Probe i.e. the creation of monitoring-session, a request results in the creation of either of the following run-time cases:

A *linux process (MPAE)* with *2 threads (EDCE, MIB-Manager)*; ……(*Request-Type-1*)

A *linux process (MPAE)* with *1 threads (EDCE)*;……..…(*Request-Type-2*)

A *linux process (MPAE)* with *1 threads (MIB-Manager)*; ……(*Request-Type-3*)

A *linux process (MPAE)* only i.e. with *no child threads*; ……..(*Request-Type-4*)

The **linux platform** on which the ODM-Probe was prototyped assigns by default approx *2 MB* for a thread upon its creation [Linux_runtime_resource_measurement2]. For simplified illustration, let's ignore the fact that in the calculation of memory consumed by individual MPAE processes, shared memory consumed by their shared libraries must be taken into consideration. Let's consider only the resident memory obtained by executing the linux command "*cat /proc/<pid>/status": VmRSS: 13128 kB*.

This means for *Request-Type-1* memory required is approximately:

2 x *thread-memory*+1 x *process-memory ~= 13 MB*

For *Request-Type-2* memory required is approximately:

1 x *thread-memory*+1 x *process-memory ~= 13-2 MB ~=11MB*

For *Request-Type-3* memory required is approximately:

1 x *thread-memory*+1 x *process-memory ~= 13-2 MB ~=11MB*

For *Request-Type-4* memory required is approximately:

1 x *process-memory ~= 13-4 MB ~=9MB*

In the experiment of issuing a number of such monitoring-requests N, the following resource consumption measurement metrics must be determined. For the ODM-Probe, this is done by measuring the resources consumed by each process and its threads started by each individual monitoring-request.

Total_R_already_seized$_t$ = 0 at time samples {T0} and {T4}

Total_R_shared_Funcs_Not_yet_executing$_t$, *at time samples {T0} and then {T4}*

Total_R_for_Funcs_unique_to_each_monitoring-request$_t$, *at time samples {T0} and then {T4}*

For illustration purposes, let's consider that *N=100 monitoring-requests* are being processed after they have been queued, and that *25 monitoring requests* are considered for each of the "*Request-Types*" described above, and that out of the *25 requests* for each of "*Request-Type*" 10 requests require *10 "shared monitoring-functions"* (see definition of *FunctionUniqueness* given earlier) but are not shared with requests of the other three "*Request-Types*" and have "0" (zero) "unique monitoring-functions". That means we have 4 x 10 "shared monitoring-functions". Using the memory measurement tools available on linux, we determine resources required by the different types of requests.

Request-Type	*Number of Monitoring-Requests having **only** "shared monitoring-functions" and the required memory resources*	*Number of Monitoring-Request having **only** "unique monitoring functions" and the required memory resources*
Request-Type-1	10 Requests:→ 10x13MB	15 Requests:→ ~15x13MB
Request-Type-2	10 Requests:→ 10x11MB	15 Requests:→ ~15x11MB
Request-Type-3	10 Requests:→ 10x11MB	15 Requests:→ ~15x11MB
Request-Type-4	10 Requests:→ 10x9MB	15 Requests:→ ~15x9MB

Total_R_shared_Funcs_Not_yet_executing$_t$, *at time samples {T0} and then {T4} = 440 MB*

Total_R_for_Funcs_unique_to_each_monitoring-request$_t$ *at time samples {T0} and then {T4} =495 MB.*

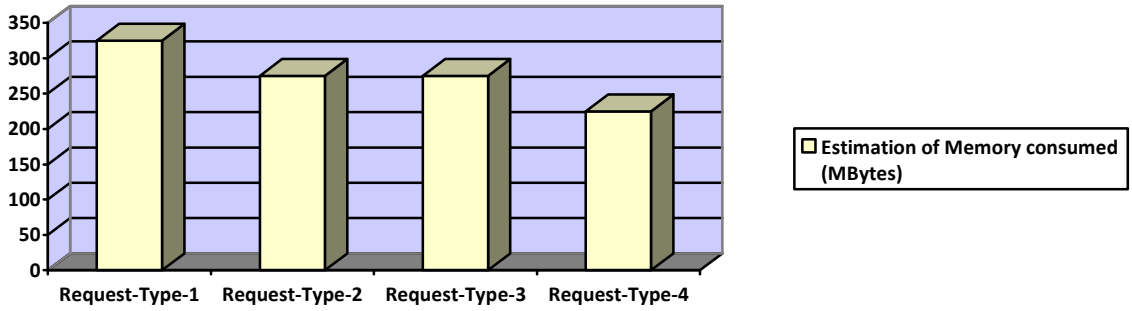
In addition to measuring these resource estimation variables, the relevant "processing times" defined in section 8.2.1 need to be measured through instrumented measurement code (i.e. via the use of appropriate calls to relevant system functions). The relevant processings times include:

Monitoring_BehaviourAdmission_Time (MBAdmTime), and

Monitoring_Behaviour_Spec_Parsing_Time (MBSParse_Time).

The measurements and characterization of such "processing times" give a picture on how the ODM-Probe performs under normal conditions of operation and state of the probe during the execution of the particular experiment case.

The bar chat below illustrates how to estimate the memory requirements for the different types of "Request-Types" at a particular time of issuing a volume of the different types of monitoring-requests and performing some measurements.



Case-2: A number of *monitoring-requests N* whereby *the requested monitoring-functions are all the same*. We choose a number of N requests that stresses the system's and probe's resources suffient enough to observe and measure the amount of resources (memory usage, CPU, bandwidth in data transfer) consumed by the monitoring-functions once they are executing. For illustration purposes, we focus on memory consumption and not on the other types of resources that also need to be considered.

MonReq$_1$ = {**F$_1$** *(filter="ip proto udp", "with trace-generation")*, **F$_2$** *(Event$_1$="traffic detected", Event$_2$="traffic exceeds threshold of 10Mbit/s")*, **F$_3$** *(OID$_1$="Protocol-Hierarchy-StatisticsTable", OID$_2$="EventsDetectedTable")*}

MonReq$_2$ = {**F$_1$** *(filter="ip proto udp", "with trace-generation")*, **F$_2$** *(Event$_1$="traffic detected", Event$_2$="traffic exceeds threshold of 10Mbit/s")*, **F$_3$** *(OID$_1$="Protocol-Hierarchy-StatisticsTable", OID$_2$="EventsDetectedTable")*}

MonReq$_n$ = {**F$_1$** *(filter="ip proto udp", "with trace-generation")*, **F$_2$** *(Event$_1$="traffic detected", Event$_2$="traffic exceeds threshold of 10Mbit/s")*, **F$_3$** *(OID$_1$="Protocol-Hierarchy-StatisticsTable", OID$_2$="EventsDetectedTable")*}

The requested monitoring-function **F$_1$**, **F$_2$**, **F$_3$**, in all the requests are the same. ***Case-2*** may be executed four times, each time with with say N=100 monitoring-requests of a particular "Request-Type" as shown below.

Request-Type	*Number of Monitoring-*

	*Requests having **only** "shared monitoring-functions" and the required memory resources*
Request-Type-1	100 Requests:→ 100x13MB
Request-Type-2	100 Requests:→ 100x11MB
Request-Type-3	100 Requests:→ 100x11MB
Request-Type-4	100 Requests:→ 100x9MB

In the experiment of issuing a number of such monitoring-requests, the following resource consumption measurement metrics must be determined. For the ODM-Probe, this is done by measuring the resources consumed by each process and its threads started by each individual monitoring-request. The same approach of determining memory consumption as taken in the experiment with Case-1 should be taken for this Case-2.

***Total_R_already_seized**$_t$*, at time sample {T1} and then {T5}
***Total_R_shared_Funcs_Not_yet_executing**$_t$, at time sample {T1} and then {T5}*
***Total_R_for_Funcs_unique_to_each_monitoring-request**$_t$, at time samples {T1} and then {T5}*

Total_R_already_seized$_t$, at time sample {T1} and then {T5} = *Total memory consumed by the N monitoring-sessions created in Case-1 (assuming that Case-2 is being executed only once).*

Total_R_shared_Funcs_Not_yet_executing$_t$*, at time sample {T1} and then {T5} = N* x *memory consumed by a single monitoring-session created by a monitoring-request of a particular "Request-Type" selected for executing Case-2.*

Total_R_for_Funcs_unique_to_each_monitoring-request$_t$*, at time samples {T1} and then {T5} =0;*

In addition to measuring these resource estimation variables, the relevant "processing-times" defined in section 8.2.1 need to be measured through instrumented measurement code (i.e. via the use of appropriate calls to relevant system functions). The relevant processings times include:

Monitoring_BehaviourAdmission_Time (MBAdmTime), and

Monitoring_Behaviour_Spec_Parsing_Time (MBSParse_Time).

The measurements and characterization of such "processing times" give a picture on how the ODM-Probe performs under normal conditions of operation and state of the probe during the execution of the particular experiment case.

Case-3: A number of *monitoring-requests N* whereby *the requested monitoring-functions are all already executing (i.e. they were invoked by previously admitted monitoring-requests).* We choose a number of N requests that stresses the system's and probe's resources suffient enough

to observe and measure the amount of resources (memory usage, CPU, bandwidth in data transfer) consumed by the monitoring-functions once they are executing. For illustration purposes, we focus on memory consumption and not on the other types of resources that also need to be considered.

Since the monitoring-sessions created by each monitoring-request in Case-1 and Case-2, are supposed to be still running at the point of executing Case-3, it means in Case-3 we can simply re-execute Case-1 with the same monitoring-requests.

In the experiment of issuing a number of such monitoring-requests, the following resource consumption measurement metrics must be determined. For the ODM-Probe, this is done by measuring the resources consumed by each process and its threads started by each individual monitoring-request.

Total_R_already_seized$_t$, at time sample {T2} and then {T6}
Total_R_shared_Funcs_Not_yet_executing$_t$, *at time sample {T2} and then {T6}*
Total_R_for_Funcs_unique_to_each_monitoring-request$_t$, *at time samples {T2} and then {T6}*

Since Case-3 can be executed and analysed by executing Case-1 again but with some "additional requeremnts to the event detection and notification functions and generated monitoring-data", the memory consumption can be measured in the same way as was done in Case-1 but with need to take into consideration the state introduced by the execution of Case-2 (i.e. memory consumed by monitoring-sessions created by monitoring-requests used in Case-2). What is the impact of executing monitoring-requests requiring the same monitoring-functions as previously executed, with same traffic-filters but with "additional requirements" to the event detection and notification functions and generated monitoring-data? The expected observation of the impact is that since no new processes and threads are created, as the required ones already exist, the only increase in memory consumption comes mainly from the need to create and store and track state information about the additional monitoring-sessions created by these types of new monitoring-requests.

In addition to measuring these resource estimation variables, the relevant "processing times" defined in section 8.2.1 need to be measured through instrumented measurement code (i.e. via the use of appropriate calls to relevant system functions). The relevant processing-times include:

Monitoring_BehaviourAdmission_Time (MBAdmTime), and

Monitoring_Behaviour_Spec_Parsing_Time (MBSParse_Time).

The measurements and characterization of such "processing times" give a picture on how the ODM-Probe performs under normal conditions of operation and state of the probe during the execution of the particular experiment case.

Case-4 (worst-case scenario): A number of *monitoring-requests N* whereby the *requested monitoring-functions are all different and are selected in such a way that their parameters vary enough to stress the ODM-Probe*. We chose a number of *N* requests that stresses the system's and probe's resources suffient enough to observe and measure the amount of resources (memory usage, CPU, bandwidth in data transfer) consumed by the monitoring-functions once they are executing. For illustration purposes, we focus on memory consumption and not on the other types of resources that also need to be considered.

The worst-case scenario is realized by executing a case with differing monitoring-functions being requested and with large monitoring-behaviour-specifications being conveyed by individual requests.

In the experiment of issuing a number of such monitoring-requests, the following resource consumption measurement metrics must be determined. For the ODM-Probe, this is done by measuring the resources consumed by each process and its threads started by each individual monitoring-request.

Total_R_already_seized$_t$, at time sample {T3} and then {T7}
Total_R_shared_Funcs_Not_yet_executing$_t$, *at time sample {T3} and then {T7}*
Total_R_for_Funcs_unique_to_each_monitoring-request$_t$, *at time samples {T3} and then {T7}*

The memory consumption can be measured in somewhat similar way as was done in Case-1 but with need to take into consideration the state introduced by the execution of Case-2 and Case-3 (i.e. memory consumed by monitoring-sessions created by monitoring-requests used in Case-2 and Case-3).

In addition to measuring these resource estimation variables, the relevant "processing times" defined in section 8.2.1 need to be measured through instrumented measurement code (i.e. via the use of appropriate calls to relevant system functions). The relevant processings-times include:

Monitoring_BehaviourAdmission_Time (MBAdmTime), and
Monitoring_Behaviour_Spec_Parsing_Time (MBSParse_Time).

The measurements and characterization of such "processing times" give a picture on how the ODM-Probe performs under normal conditions of operation and state of the probe during the execution of the particular experiment case.

A number of "*other experiment cases*" can be derived and experimented with, in order to deduce the dynamics of the **Monitoring Effort** function. For example, the *number of monitoring-requests N* processed at the time samples can be varied, and also some *monitoring-sessions can*

be made to pause or terminate at the time when new requests are being admitted—thereby freeing resources. Such cases should also be experimented with in order to determine the dynamics of **Monitoring Effort** function**.** In addition to the different additional cases that can be experimented with, querying or retrieval of monitoring data by ODM-Session-Owners has not been considered in the cases above, which means, experiments that repeat the cases but with inclusion of querying for monitoring data also need to performed in order to determine the dynamics of the **Monitoring Effort** function.

Analyzing experiment cases performed for the "*stressing condition of operation of the ODM-Probe":*

Tests perfomed within the interval *{T4……T7}* are meant to stress the probe so as to determine the limits of it processing a large volume of monitoring-requests of varying monitoring needs (i.e. monitoring-functions) within a short time. The metrics that are critical to be measured and analyzed during such experiments are mainly the "processing times" defined in section 8.2.1 that need to be measured through instrumented measurement code (i.e. via the use of appropriate calls to relevant system functions). The relevant processing-times include:

Monitoring_BehaviourAdmission_Time (MBAdmTime), and
Monitoring_Behaviour_Spec_Parsing_Time (MBSParse_Time).

8.2.2.3 Using the results obtained in the experiments to tune admission control behaviour of the ODM-Probe on monitoring-requests

In order to use the dynamics of the **Monitorig Effort** function to tune the behaviour of the admission control algorithm and policies employed by the **odmRHACC component of the ODM-Probe**, the odmRHACC component should do the following:

At at the point of processing monitoring-requests accumulated over a short time interval of caching requests, the admission control can use constraints such as (*1) Maximum number of threads that the ODM-Probe is permitted to create on the system*, and (*2) Total amount memory reserved for traffic monitoring on the system*, to decide on whether to admit a monitoring-request i.e. execute its required monitoring-functions (assuming that the required functions are not already executing due to a previously admitted request with similar needs).

if ("Maximum-number-of-threads-permitted has not been reached"){
if ***SUM(****Total_R_already_seized*$_t$*,*
Total_R_shared_Funcs_Not_yet_executing$_t$*,*
Total_R_for_Funcs_unique_to_each_monitoring-request$_t$***)***
"is less than" *Total_Amount_of_Memory_assigned_for_Monitoring*
Then admit *the monitoring-request;*

}

It is worthy pointing out that other approaches applied to systems supporting job assigments and scheduling such as Grids, like those presented in [Chtepen09] [Chtepen08] may also be applied to on-demand monitoring systems like the ODM-Probe.

8.3 Limitations of On-Demand MIBs

On-Demand MIBs have their own associated limitations as described in chapter 5—section 5.3.1.2. The size of the On-Demand MIB tree is determined by number of OIDs dynamically registered into the MIB. The dynamics (growth and shrinking) of the On-Demand MIB tree are determined by the OIDs registered by some monitoring-sessions started at different times as well as by the destruction of some OIDs by some terminating monitoring-sessions at different times during the operation of the ODM-Probe. The performance of On-Demand MIBs does not differ from that of the statically instantiated MIBs of the system.

8.4 The maturity-level of the current implementation of the ODM-Probe

In this section, we give details on the maturity of the implementation of the ODM-Probe and a contrast of the current implementation to functional specification presented Chapter 7, in terms of what is still to be implemented and the perspectives related to conformance to the functional specification. The functional specification of the ODM-Probe presented in Chapter 7, is better suited for an implementation approach that seeks to implement the components from scratch, than an approach that is based on modifying some selected existing software tools and libraries. This is because of the following reasons: **(1)** The problem of the fact that today's monitoring tools, libraries and architectures may not be flexible enough to allow for modifications; **(2)** an approach that is based on modifying some existing selected software tools and libraries, implies that some of the limitations of those tools and libraries may become inherent in the architecture that then emerges. The implementation approach we took for prototyping purposes is one based on modifying some existing selected software tools and libraries. Therefore, the implementation approach we took suffers the problems of inflexibilities and limitations inherent in the selected software tools and libraries. In the following paragraphs, we provide a summary of the key functional features implemented in our current implementation of the ODM-Probe.

Our current implementation of the ODM-Probe has the concept of On-Demand SNMP MIBs implemented, including all the concepts behind dynamic OID registrations. The next thing required will be to consider extending the NET-SNMP library to allow AgentX subagents to run as threads on the same machine, served by a single SNMP Master Agent. The reason why this is

needed is because, the MPAE, MIB-Manager and EDCE components are meant to operate on shared (common) variables containing data that is created and updated in real time by the packet capturing and decoding component—the MPAE, a task which is easily achieved with threads than processes. Currently, the implementation supports only one ODM-session per single set of MPAE, MIB-Manager and EDCE instances, but with support for all the session-management primitives described earlier. The next thing required is to support multiple sessions with varying monitoring-behaviour-specifications but sharing the same traffic capture filter (i.e. the *ODM-Traffic-Filter*), per single set of MPAE, MIB-Manager and EDCE instances. The implementation of the advanced admission control mechanisms and policies to be employed by the odmRHACC server has not been done and so, it still requires development. The creation of a Capability Model by the odmRHACC, the exporting and updating of the Capability Model into a Capability Models Database described previously in chapter 6**,** which presents the full perspective of the ODM-Paradigm, is also work that still requires development.

The interface(s) that allows automated tasks that are local to the ODM-Probe to interact with it directly and bypass SSL-enabled *ODM-primitives* and SNMP based interactions still needs to be developed. Local automated tasks can still use the same interfaces as remote automated tasks, but for performance optimizations of the whole ODM-Probe architecture, it is desirable that local automated tasks use a different type of interface(s) that bypasses SNMP encodings in data reading and SSL enabled interactions.

For achieving high performance and taking advantage of functions available in advanced hardware, the ODM-Probe components such as the odmRHACC, MPAE, EDCE and MIB-Manager should be implemented in kernel, and components such as the MPAE must be implemented with support of functions in Network Interface Cards (NICs). The current implementation is not in the kernel due to the approach we took, of modifying and integrating some existing Open-Source Tool-Kits such as NET-SNMP [Net-SNMP] and Ethereal [Ethereal], which are implemented for use in user-space, only to suit our prototyping purposes.

The hope is that, as the ODM-Probe is contributed to the Open-Source community [ODM-Probe-Source-Code], the functional specification of the ODM-Probe provided in this dissertation, as a blueprint, will continue to help in the further research and development of the ODM-Probe. Because the current implementation of the ODM-Probe was driven by the need to prototype individual concepts such as On-Demand MIBs, separately, it means some work is till required to integrate all the pieces together and further develop the ODM-Probe. For more information on the required integration activities, we refer to the source code documentations of the ODM-Probe, as well as the README files.

In conclusion, the maturity of the implementation of the ODM-Probe can not be really called a mature implementation since some further work is still required to be carried out as pointed out in this section.

8.5 Contrasting the ODM-Probe to today's well-known monitoring approaches

Here we provide a brief discussion on the design principles employed by some of the well-known monitoring systems or tools that we consider relevant to the further development of the ODM-Probe. We provide a comparison of the ODM-Probe to other types of well-known monitoring components whose design principles can contribute to further development of the ODM-Probe.

It is clear that the functional specification of the ODM-Probe and implementation still needs to undergo rigorous revisions and evolutions by considering a number of vital design principles and experiences gained in the implementation of monitoring functions and systems such as the ones examined in **chapter 5—section 5.9**. We note the following well-known monitoring systems and approaches, as having characteristics, features and design principles that can be contributed (partially or fully) to the further development of the functional specification and implementation of the ODM-Probe: DiMAPI [Trimintzios06], MAPI [Polychronakis04], FFPF [Bos04] [Hruby05], CoralReef [CoralReef] [Keys01], NetFlow [NetFlow], IPFIX [IPFIX], SNMP based architectures [SNMP]. These systems and approaches have already been examined in **chapter 5—section 5.9**, with respect to their potential contribution to the further development of ODM-Principles (**Principle-1** to **Principle-7**) and/or their potential evolution towards supporting ODM-Principles (**Principle-1** to **Principle-7**). In the design of the ODM-Probe, we have not covered any need for communication between two or more monitoring components (e.g. two or more ODM-Probes). DiMAPI [Trimintzios06] deals with inter communication between two or more monitoring components, an issue that needs further study for the evolution of the ODM-Probe, to enable inter communication between two probes. How this would interplay with the ODM-Principles, would need some investigation. MAPI [Polychronakis04], CoralReef [CoralReef] [Keys01] and FFPF [Bos04] provide a number of insights on how to address the monitoring needs of applications that need to run on the same box as the monitoring components or modules, including design principles for the monitoring components or modules such as support for heterogeneous monitoring drivers and hardware; traffic filtering and classification at high speeds; traffic/packet processing in the user-space, kernel space and/or in network processors; and minimising context-switching and packet copying for multiple monitoring applications, etc. In the functional specification of the ODM-Probe, the interface between *Automated Tasks* and an ODM-Probe running on the same box, which is meant to specifically address the need of automated tasks that need to directly receive packets of a "traffic flow" has not been addressed. Therefore, we consider that for the specification of this specialized interface, the approaches taken in MAPI [Polychronakis04], FFPF [Bos04] and CoralReef [CoralReef] [Keys01] can be adopted for the specification and the design of the this interface of the ODM-Probe. Netflow [NetFlow], IPFIX [IPFIX] and SNMP MIB data models are relevant for the support of on-demand creation of On-Demand Data Models (a concept defined by **Principle-P3**) by the ODM-Probe.

Concerning the design of some of the core components of the ODM-Probe, such as the MPAE and EDCE components, the approaches and design principles adopted in the design of FFPF: Fairly Fast Packet Filters [Bos04], can be adopted in the further development and evolution of the MPAE and EDCE components. This is because the FFPF framework addresses a number of vital design principles, such as traffic filtering and classification at high speeds; traffic/packet processing in the user-space, kernel space and/or in network processors; and minimising context-switching and packet copying for multiple monitoring applications i.e. tasks, scalability (for the support of multiple monitoring applications i.e. tasks), flexibility, etc. The ODM-Probe is meant to support multiple monitoring tasks via the notion of monitoring-sessions that are handled by multiple MPAE instances running on the probe. Therefore, the lessons learnt from the design for scalability principle employed by the FFPF, can be used in the refinement of the ODM-Probe architecture. In FFPF, the same generic concept of a "traffic flow" as adopted in the ODM-Paradigm, is used. The concept of "flow-specific state" such as "counters" can contribute to further development of the EDBSLang composition language we presented in **Chapter 5—section 5.2.1**, in order to enable automated tasks to specify in a monitoring-behaviour-specification the need for the creation of such flow-state data such as "counters". The FFPF framework is language neutral when it comes to traffic filter languages, and supports filter languages such as the well-known BPF [BPF] based filters, as well its own native filter languages. The issue of support for multiple filtering languages needs to be a feature of the MPAE component of the ODM-Probe. This means the packet filtering and classification modules of the FFPF framework could be made to become part of the core functions of the MPAE. Another concept introduced in the FFPF framework worthy to consider for adoption in the development of the MPAE component is what is called a "flow_grabber", which works like a packet filtering component. The MPAE component could be viewed as having some of the functions provided by a "flow_grabber", apart from its core functions defined in this functional specification. Also important for consideration from the FFPF framework, are some concepts such as "flow_group", which is group of applications (i.e. flow receivers) that share a common packet buffer. This concept, as well as the buffer management schemes employed by the FFPF framework could offer some insights into the evolution of the design of the MPAE. The subject of adopting some of the design principles from frameworks like FFPF framework, DiMAPI, MAPI, etc, in the further design of some functions of components of the ODM-Probe, such as the odmRHACC, the MPAE and EDCE, would amount to a subject for further research in the evolution of the ODM-Probe.

8.6 Instrumenting ODM-Probes into network elements

In this section, we present an approach on how to instrument ODM-Probes into network elements (see definition of *network element*$_{re\text{-}defined}$) and migrating current monitoring frameworks to supporting the ODM-Paradigm. The ODM-Probe can be instrumented into different types of network nodes (systems): *routers, switches, hosts, signalling gateways,*

specially-made dedicated traffic monitoring probes etc, making them what we would call **ODM-Capable Systems**. In general, the envisioned ODM-Capable Systems can be developed according to the following two approaches:

a) By taking the ODM-Probe architecture and its functional specification presented in this chapter as the starting point, and then evolving the ODM-Probe according to the aspects and issues discussed in **chapter 5-section 5.9 and section 8.5**.
b) By producing a different functional specification and architecture of an ODM-Capable Probe that follows the fundamental theory of the ODM-Paradigm defined in **Chapter 4** as well as the aspects and issues discussed in **chapter 5-section 5.9 and section 8.5**.

8.7 Some remarks

Chapters 4 and 5 presented the ODM-Principles i.e. the design and operational principles of monitoring components that are required in order to address problems of traffic monitoring in self-managing networks, such as the need to take into account the dynamics of automated tasks. An ODM-Capable monitoring component needs to support the seven principles (**Principle-P1 to Principle-P7** defined in **Chapter 4**) such as support for intelligent and opportunistic use and allocation of resources; the need to introduce "admission-control" along with the concepts of a "monitoring-request" and a "monitoring-behaviour-specification" for monitoring; the need for customizable, flexibly configurable and re-configurable monitoring, programmable and adaptive monitoring; and the need to support on-demand in-memory dynamic models of monitoring data on monitoring components. The ODM-Paradigm enables individual dynamic automated tasks to locate monitoring components of interest, given their point(s) of attachment to the network topology and their capabilities, specify their monitoring needs via monitoring-behaviour-specifications and upload the behaviour into the monitoring component(s) for execution. In the previous chapter we provided the functional specification of a prototype ODM-supporting probe i.e. an ODM-Capable prototypical probe as well as its current implementation, which is a continuing effort because, as pointed out in **section 8.4**, some of the concepts described by the functional specification have not yet been implemented into the current implementation of the ODM-Probe. Therefore, the functional specification serves as a blueprint for further development of the probe under open source efforts as the **ODM-Probe** is released into the open source community [ODM-Probe-Source-Code].

However, it is important to note that, by following the ODM-Paradigm as presented in **chapters 4 and 5**, *diverse architectures for monitoring components for multi-service self-managing networks,* different from the architecture of the ODM-Capable probe described in Chapter 5, can be developed, evaluated and contrasted to our ODM-Probe. Future work on the ODM-Probe will need to involve the remaining implementation aspects and evaluating them. Therefore, scalability issues in a larger holistic context of the ODM-Probe, as well as scalability tuning, is a

subject for further research. However, in **Chapter 9,** we provide a discussion on scalability issues related to the ODM-Paradigm in general.

9 Discussion on Scalability, Resource Allocation and Utilization in using ODM-Principles

In this chapter we address the following question: `Does it mean that any kind of` **`Automated Task`** `in the network should be allowed to locate a monitoring component of some given capabilities and create a` **`Monitoring-Session`** `(i.e. an ODM-Session) on the component?` This question is very important in the sense that, as expected in practice, it is not possible to allow any Automated Task in the network to locate and create a monitoring-session on monitoring component for the following reasons:

- ***Scalability*** – The resources i.e. computation power (CPU cycles), memory and bandwidth dedicated to monitoring functions i.e. tasks on a single monitoring *network element*$_{re\text{-}defined}$ and in the network as whole, are obviously limited. As it is known in today's networking devices such as routers and switches, some share of the overall resources on the box can be allocated to monitoring tasks such as SNMP or NetFlow data gathering and dissemination operations, while the other share of resources is dedicated to the key functions of the device such as routing, forwarding, switching, etc. With respect to scalability in the context of the ODM-Paradigm, we define the *scalability bound* as: *The degree to which the monitoring needs (expressible through monitoring-behaviour-specifications or monitoring queries) for the diverse automated tasks of a network can be covered by the amount of the aggregate resources that are dedicated or allocated to monitoring tasks and related operations on individual network elements (see definition of network element*$_{re\text{-}defined}$*) and within the network as a whole*. Clearly, at any given time during the operation of a self-managing network, achieving 100% coverage for monitoring needs can not be achieved in reality due to resource and capability limitation of devices. As defined by the principles of the ODM-Paradigm, monitoring functions should be invoked on-demand and resources freed whenever monitoring is temporally or no longer required by an **Automated Task** that triggered the monitoring functions. This means *the aggregate resources that are dedicated or allocated to monitoring tasks and related operations on individual network elements (see definition of network element*$_{re\text{-}defined}$*) and within the network as a whole* get consumed and freed dynamically by the **Automated Tasks** of the network.

- **The need for intelligent use of the limited** *aggregate resources that are dedicated or allocated to monitoring tasks and related operations on individual network elements (see definition of network element$_{re\text{-}defined}$) and within the network as a whole*. Allowing (without restrictions) any **Automated Task** in the network to locate a monitoring component of some given capabilities and create a **Monitoring-Session** (i.e. an ODM-Session) on the component need to be avoided because this would unnecessarily consume resources associated with maintaining state information about monitoring-sessions and session-owners maintained by monitoring components.

In this chapter, we propose some solutions to the optimal setups that consist of example configurations which can be considered for allowing some Automated Tasks in the network, acting as ODM-session-owners, to configure and trigger monitoring tasks on behalf of some Automated Tasks which then use the services offered by the monitoring tasks (i.e. ODM-Sessions) and need not manage the monitoring tasks (i.e. ODM-Sessions) by themselves.

The first setup that should be considered is depicted on **Figure 36**. Similar setups or derivatives of this setup may be worked out. Here, an **Automated Task "0"** running on ***network element$_{re\text{-}defined}$*** **"0"**, plays the role of an ODM-Session requester/owner on the local ODM-Capable Component (or sub-system) by triggering and managing the execution of a monitoring-behaviour-specification that addresses the monitoring needs of not only this Automated Task "0" but other Automated Tasks in the network. For example, this Automated Task "0" could be an instance of routing protocol such as OSPF, running on the *network element$_{re\text{-}defined}$* and triggering a local monitoring-behaviour that creates monitoring data or issues event-notifications that are required to drive the behaviour of the local OSPF instance as well as similar instances of OSPF running in the network. The unbroken bi-directional arrow depicts the communications related to ODM-Session Management (establishment, pausing, modifications, termination) while the dotted bi-directional arrows depict communications related to event-notifications and data retrieval from the executing ODM-Session. **Automated Tasks** "**1**", "**2**" and "**3**" on network elements (see definition of *network element$_{re\text{-}defined}$*) "1", "2" and "3" are shown playing the role of "User" of an ODM-Session created by Automated Task "0" on the ODM-Capable component on *network element$_{re\text{-}defined}$* "0".

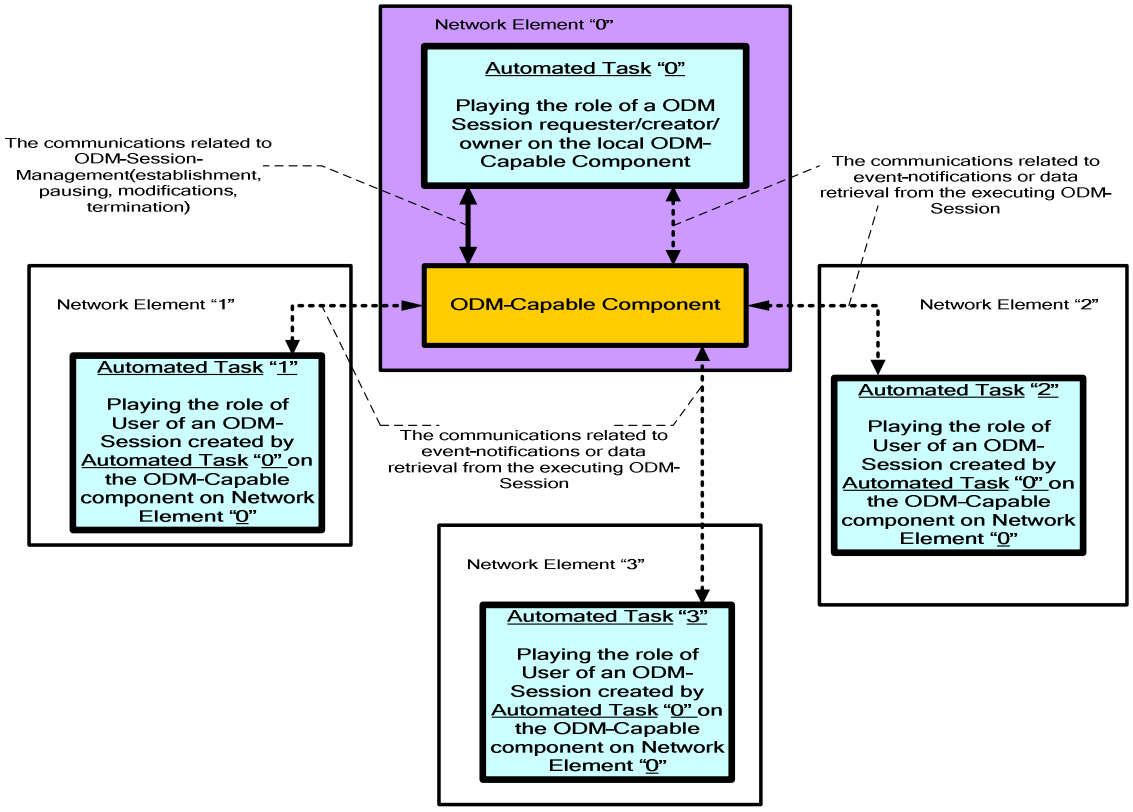


Figure 36: A single Automated Task playing the role of an ODM-Session requester and owner on a local ODM-capable component, on behalf of other Automated Tasks

The second setup that should be considered is depicted on **Figure 37**. Similar setups or derivatives of this setup may be worked out. Here, an **Automated Task "0"** running on ***network element*$_{re\text{-}defined}$ "0"**, plays the role of an ODM-Session requester/owner on ODM-Capable Component (or sub-system) running on network elements (see definition of *network element*$_{re\text{-}defined}$) "X" and "Y" by triggering and managing the execution of monitoring-behaviour-specifications that address the monitoring needs of other Automated Tasks in the network. For example, this Automated Task "0" could be a centralized network management application of a Network Management System (NMS). The monitoring-behaviours whose monitoring-sessions are under the management of Automated Task "0" create monitoring data or issue event-notifications that are required to drive (respectively) the behaviours of Automated Tasks "1", "2", "3" and "4" on network elements (see definition of *network element*$_{re\text{-}defined}$) "1", "2", "3" and "4". The unbroken bi-directional arrow depicts the communications related to ODM-Session Management (establishment, pausing, modifications, termination) while the dotted bi-directional arrows depict communications related to event-notifications and data retrieval from the executing ODM-Session. Automated Tasks "1", "2", "3" and "4" on network elements (see definition of *network element*$_{re\text{-}defined}$) "1", "2", "3" and "4" are shown playing the role of "User" of an ODM-Session created by Automated Task "0" on the ODM-Capable component on

network element$_{re\text{-}defined}$ "X" or "Y" respectively. In comparison to the other three Automated Tasks, Automated Task "2" is shown as playing the role of User of two ODM-Sessions created by Automated Task "0" on the ODM-Capable components on *network element*$_{re\text{-}defined}$ "X" and *network element*$_{re\text{-}defined}$ "Y". In **Chapter 6** (**the full perspective of the ODM-Paradigm**), two scenarios are presented, which can serve as example scenarios that can be realized on the basis of the setup presented on **Figure 37**. The first one is an example scenario of On-Demand MIB application to traffic engineering (with requirement for advertisement of the On-Demand MIB). The second scenario is an example scenario of On-Demand MIB application to traffic engineering (with no requirement for advertisement of the On-Demand MIB).

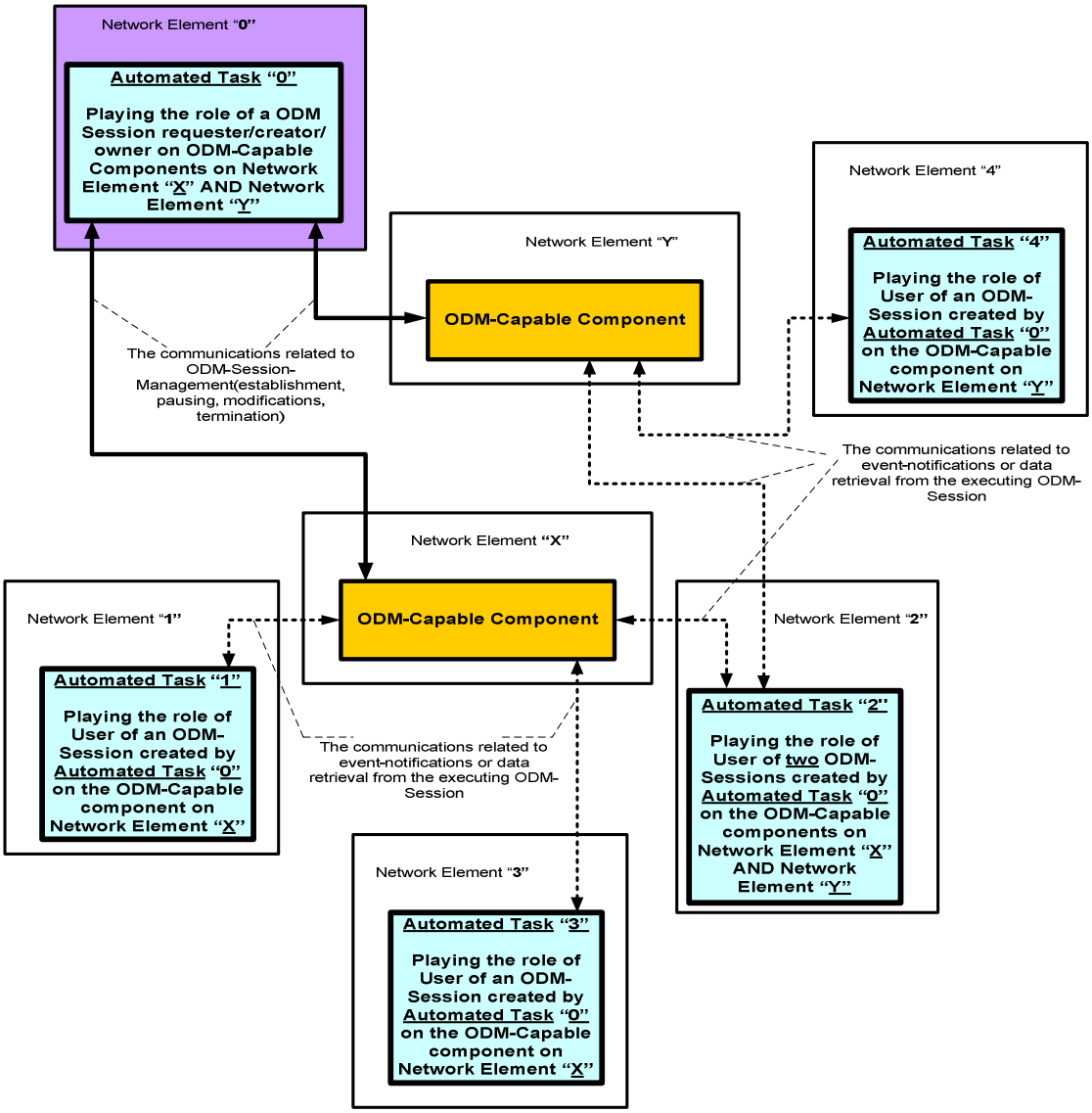


Figure 37: A single Automated Task playing the role of an ODM-Session requester/owner on multiple distributed ODM-capable components, on behalf of other Automated Tasks

A subject that remains for further research and study concerns the dynamics related to the seizure and freeing of the ***aggregate resources that are dedicated or allocated to monitoring tasks and related operations on individual network elements (see definition of network element$_{re\text{-}defined}$) and within the network as a whole***, by the Automated Tasks of some self-managing networks of some sizes i.e. scale. Such a study would give some indications and estimations on how much of some freed system and network resources i.e. a part of the aggregate resources allocated by default to monitoring tasks and operations, can be dynamically available for "opportunistic" use by other functions of the network elements (see definition of *network element$_{re\text{-}defined}$*) and the

network as a whole. The study would also indicate the rates of resource-seizure and resource-freeing, as well as the factors determining the dynamics of the *aggregate resources that are dedicated or allocated to monitoring tasks and related operations on individual network elements (see definition of network element$_{re\text{-}defined}$) and within the network as a whole***,** including the behaviours of those kinds of Automated Tasks that trigger monitoring functions and free resources in a given self-managing network of a particular size i.e. scale.

10 Conclusions and Further Work

In this dissertation, we presented a traffic monitoring paradigm we are calling the ODM-Paradigm, a monitoring paradigm we consider suitable for automated tasks that drive multi-service self-managing networks. The ODM-Paradigm addresses the problem statement given in Chapter2-section 2.3. It takes into account the requirement for context-driven (re)-configuration and self-adaptation of network tasks, dynamicity of their monitoring needs of individual network tasks, the need to use system and network resources intelligently, optimally, and opportunistically throughout the network, as well as the need for monitoring components to self-describe their monitoring capabilities to the network in order to enable automated tasks in the network to locate and select a monitoring component of desired capabilities when a need arises, trigger monitoring functions and manage the execution of those functions, and free resources on the targeted component whenever monitoring is temporarily not required or no longer required by an automated task(s). Although the main focus of this dissertation is about design and operational principles for traffic monitoring components, the research work also introduced an architectural Reference Model for Autonomic Networking and Self-Management called the GANA (Generic Autonomic Network Architecture) Reference Model, a *Hierarchical Autonomic Management and Control Architectural Framework.*

The ODM-Paradigm is based on our research on design and operational principles of traffic monitoring components, which we call ODM-Principles, and are meant to address the problems mentioned above, regarding traffic monitoring in the emerging, complex and highly dynamic self-managing networks, which are characterized by high resource demands and the requirement for context-driven (re)-configuration and self-adaptation of network tasks.

The research work presented in this dissertation was about finding and validating the key design and operational principles for traffic monitoring components and platforms that are suitable for the emerging multi-service self-managing networks—expected to be resource demanding, namely the ODM-Paradigm principles defined and described in **Chapter 4**, and whose technical solutions are presented in **Chapter 5**. The research work presented here, is not about a call to the design of new traffic monitoring functions, tailored to solving a particular traffic monitoring problem(s) and being complementary to the existing diverse traffic monitoring functions and architectures already covered by other research work, such as, say, packet capturing and processing functions on high speed links, but rather, we sought to answer the question of how the traffic monitoring functions of monitoring tools and components or devices can be made to fit into the ODM-Paradigm by making them evolve towards supporting the ODM-Principles. Answers to how ODM concepts can be integrated into existing monitoring tools and methods

have been also provided. Also presented is an approach on how to instrument ODM-Probes into network elements (see definition of *network element*$_{re\text{-}defined}$) and migrate today's well-known monitoring frameworks, to supporting the ODM-Paradigm.

As described in **Chapter 4**, intelligent and opportunistic use of resources is achievable when the monitoring functions of traffic monitoring components are designed following the ODM-Principles described in more detail in **Chapters 4 and 5**. This is because, the ODM-Principles include the freeing of resources of the system by a monitoring component whenever monitoring services are temporarily or no longer required, when indicated by automated tasks that triggered the monitoring-behaviour(s) in the first place, such that the freed resources can be dynamically used for some tasks other than monitoring tasks on the system.

The ODM-Paradigm is open to growth in the following dimensions: **(1)** The refinement of the ODM-Principles laid in **Chapter 4**. We expect that as the field of autonomics and self-managing networks continues to grow, the ODM-Principles may need to be further refined; **(2)** The second dimension may be that new principles may be captured and introduced to extend the ODM-Principles laid in **Chapter 4**; **(3)** The third dimension is that of newly discovered characteristics of automated tasks (possibly new applications and services) of self-managing networks that impose new requirements (not captured in current research) on traffic monitoring components and platforms of a self-managing network.

The design principles for traffic monitoring components presented in this dissertation that allow for on-demand triggering of monitoring functions and termination of their execution when temporarily not required or longer required, can equally be applied to the design of other types of system functions or network functions (apart from monitoring functions) for which such flexibility is a necessity.

Evaluation of the ODM-Probe prototype and the Case Study conducted with On-Demand MIBs shows that the ODM-Principles evaluated can be feasibly implemented and applied in practice.

10.1 Summary of what the key chapters presented

Chapter 2 presented the *Problem Formulation*, the limitations of today's monitoring paradigms, and why the paradigms are not suitable for multi-service self-managing networks. More limitations are revealed in Chapter 5—**section 5.9** where today's monitoring approaches and paradigms are examined against individual ODM-Principles laid in **Chapter 4**.

First, in **chapter 3**, design-principles for self-managing networks are presented, as this helps in understanding the concepts of "*task automation*", an "*automated task*", and "*autonomic feedback*

control structures" that enable the design and implementation of self-managing networks. The evolving GANA architectural Reference Model for Autonomic Networking and Self-Management has also been presented in brief, while pointing the reader to research work on GANA and its evolution, as the detailed GANA related work are beyond the scope of this document.In the same chapter, we outlined the role of monitoring in self-managing networks, and the requirement for a monitoring paradigm that is suitable for multi-service self-managing networks.

In the same chapter, we discussed the requirement for a traffic monitoring platform we consider suitable for multi-service self-managing networks, including the basic concepts and elementary building blocks of such a platform. In the same chapter, we introduced the notions and concepts such as *monitoring-request*, *monitoring-session*, *session-identifier*, *session-view, monitoring-behaviour-specification*, *session-management, monitoring-query, traffic monitoring capability models,* etc, that must be associated with the monitoring components i.e. the elementary building blocks of the required traffic monitoring platform. The ODM-Principles are briefly introduced and are later defined and described in more detail in **chapter 4**.

Chapter 4 introduced the fundamental theory, concepts and principles of the On-Demand Monitoring (ODM) paradigm. A definition of the ODM-Paradigm is provided along with the foundation upon which the definition is based. The definition of what we call ODM-Principles is provided.

Chapter 5 presented our technical solutions to the ODM-Concepts and Principles. In the same chapter, a discussion on the problems inherent in today's state-of-the-art monitoring paradigms and frameworks, with respect to supporting the ODM-concepts and what we called ODM-Principles for ODM-capable traffic monitoring components is provided. The list below is a list of the ODM-Principles (described in more detail in **Chapter 4**) that must be supported by what we call an ODM-Capable monitoring component:

- **Support for Customizable Monitoring** (referred to as **Principle-P1** in **Chapter 4**). A technical solution to achieving this principle is presented in Chapter 5. The functional specification and implementation of the ODM-Probe provided in **Chapter 7**, demonstrates how a traffic monitoring ODM-capable component can be designed to support this ODM-Principle. However, in **Chapter 7**, we point out that the implementation of the ODM-Probe is not a mature implementation and that some refinements of the functional specification as well as further implementation work on the ODM-Probe is considered as further work in this research.
- **Support for On-Demand creation and destruction of gathered Monitoring-Data and in-memory Data Models** (referred to as **Principle-P3** in **Chapter 4**). A technical solution to achieving this principle is presented in Chapter 5. Regarding this ODM-Principle, this thesis has contributed to the introduction of the concept of what we call On-Demand SNMP MIBs that is described in **chapter 5—section 5.3.1**. The concept of

On-Demand SNMP MIBs is incorporated into the functional specification and implementation of the ODM-Probe presented in **Chapter 7**. The concept of On-Demand SNMP MIBs has also been implemented and evaluated by the implementation of the ODM-Probe.

- **Support for Triggerable, Configurable and Re-Configurable Monitoring** (referred to as **Principle-P4** in **Chapter 4**). A technical solution to achieving this principle is presented in Chapter 5. The functional specification and implementation of the ODM-Probe provided in **Chapter 7**, demonstrates how a traffic monitoring ODM-capable component can be designed to support this ODM-Principle via the use of what we called "ODM-primitives" and "monitoring-queries" described in the definition of this ODM-Principle.
- **Support for Programmable Monitoring** (referred to as **Principle-P2** in **Chapter 4**). A technical solution to achieving this principle is presented in Chapter 5. Regarding this ODM-Principle, this thesis has contributed a *Composition Language for specifying monitoring-behaviours* that includes the possibility to *specify the reactions* of a monitoring-behaviour that can be *instantiated* on an ODM-capable component. The composition language, called the *Event Description and Behaviour Specification Language* (EDBSLang*)* has been described in **chapter 5—section 5.2.1**. The use of the EDBSLang language is also incorporated into the functional specification of the ODM-Probe. However, along with the description of the language, we do not claim that the language is complete, and so we have pointed out some potential aspects related to the evolution (i.e. further development) of the EDBSLang language.
- **Support for Adaptive Monitoring** (referred to as **Principle-P5** in **Chapter 4**). A technical solution to achieving this principle is presented in Chapter 5. This ODM-Principle has not been covered in the functional specification and implementation of the ODM-Probe provided in **Chapter 7.**
- **Support for Intelligent and Opportunistic use and allocation of resources** (referred to as **Principle-P6** in **Chapter 4**). A technical solution to achieving this principle is presented in Chapter 5. The use of the notion i.e. concept of: monitoring-session-management and the associated session-management primitives (ODM-primitives) ensures that network resources are freed whenever monitoring is temporarily not required or no longer required by an automated task(s) **-** thereby allowing for opportunistic use of available system and network resources. This ODM-Principle is incorporated into the functional specification and implementation of the ODM-Probe provided in **Chapter 7.**
- **Support for Self-description of traffic monitoring Capability Models, solicitation for the Capability Model of a monitoring component and self-advertisement** to interested nodes or network by the monitoring component in order to allow automated tasks to locate, select a monitoring component of desired capabilities and point of attachment to the network, trigger monitoring functions and manage the execution and termination of those functions. This principle is referred to as **Principle-P7** in **Chapter 4**. The monitoring component must update the network of any changes to its monitoring capability model. A technical solution to achieving this principle is presented in Chapter

5. Regarding this ODM-Principle, this thesis has contributed to the introduction of a *Capability Model Description Language* (CMDL) that can be used by ODM-capable monitoring components to self-describe their *Monitoring Capability Models* and self-publish them to the network. CMDL is a meta-language for describing capability models of ODM-capable components and devices. The CMDL language is described along **Principle-P7** in **chapter 5—section 5.6** and in **Appendix B** (in much more detail). The incorporation of the support for **Principle-P7** into the functional specification and implementation of the ODM-Probe provided in **Chapter 7,** has not been fully covered. In **Appendix B,** some potential aspects of the evolution (i.e. further development) of the CMDL language are pointed out.

In **Chapter 6** we gave an insight of the "Big-Picture" of ODM-Paradigm and example application areas and real-world scenarios on the the application of the ODM-Paradigm.

In **Chapter 7** we provided a functional specification and prototype implementation of an ODM-Capable Probe that serves as proof of concept for the ODM-Paradigm. Along the functional descriptions of individual components of the ODM-Probe, some explanations as to how individual ODM-Principles are incorporated into the functional specification and implementation of the ODM-Probe are given. Therefore, the functional specification serves as a blueprint for further development of the probe under open source efforts as the **ODM-Probe** is released into the open source community.

In **Chapter 8**, we presented an *Evaluation of the ODM-Probe and its ODM-Principles, and Case Study***.** An illustration of the interaction of the components of On-Demand Monitoring in a distributed environment is given and the ***processing times*** associated with the Components of the ODM-Probe are derived and discussed. The processing times determine the dynamics of resource seizure and freeing on ODM-capable traffic monitoring components and subsystems. A contrast of the ODM-Probe to other types of monitoring components whose design principles can contribute to further development of the ODM-Probe has been given in the chapter.

In **Chapter 9** we discussed the issues related to *scalability and resource allocation and utilization efficiency in the application of the ODM-Paradigm*.

10.2 Questions answered by this research

The key questions addressed by this research, as well as the solutions that address the questions are discussed in Chapter 1-section 1.3.

10.3 Summary of key contributions

As part of the "Big-Picture" i.e. the full perspective of the ODM-Paradigm, the research introduced and presented the key achievements described in chapter 1-section 1.3.

10.4 Open Issues that were not addressed in this research

To mention a few of the open issues that were considered out of scope in this research: the querying methods and mechanisms that enable automated tasks to query the Capability Models Database for monitoring components of desired capabilities; the need for defining a special type of a packet that can be used by an automated task to request for monitoring on each ODM-Capable component along a given path in the network and is intercepted by ODM-Capable components along the path; the use of traffic flow based triggers such as SUM, MINIMUM, MAXIMUM, etc, that can be specified in relation to measurable characteristics of a single traffic flow or multiple traffic flows, to form a condition that triggers monitoring on an ODM-Capable component when the trigger fires.

What has also been considered out of focus in our research on designing the ODM-Paradigm is the interaction between ODM-capable components. Rather, the main focus we presented is the relation between an ODM-capable component as a platform for traffic monitoring, and the users—namely the automated tasks of the network that issue monitoring-queries or create monitoring-sessions on the targeted components, thereby triggering monitoring functions and managing the execution and termination of the triggered monitoring functions.

10.5 Further Work on the GANA Reference Model

The GANA Model *defines the "levels of abstractions of functionality"* at which to introduce *Autonomic Elements i.e. Managers* in a *holistic way.* However, the GANA Model is evolvable to accommodate and unify concepts from today's well-known approaches to autonomic management and communication such as the IBM-MAPE Model [MAPE], 4D architecture [Greenberg05], CONMan management model [CONMan], FOCALE [Strassner06], Knowledge Plane for the Internet [Knowledge Plane], etc. This is because, each of these approaches to autonomic communication and self-management presents unique concepts that can be combined in a harmonious way with concepts from the other approaches, thereby creating a single unified model that can be applied in reasoning and designing autonomic components, their interfaces and relations. This has been one of the main goals behind the creation of the GANA Model from its very early stages when it was first introduced [Chaparadza08b]. The on-going work of the ETSI Industry Specification Group called AFI [ETSI AFI] is evolving the GANA Model along such an

inspiration. In the detailed GANA specification (we refer to the EFIPSANS deliverable D1.7 [EFIPSANS]) and AFI Work Item#2 Specification, aspects such as the ones listed below are specified and will continue to see further developments as the Model becomes widely adopted in use:

- How the framework, as a hybrid model, enables to *combine Centralized and Distributed Decision-Making* based Management and Control of Resources while pointing out the main limitations of decentralization and distributed decision-making in network management and control.
- Methods and techniques for addressing *Stability of Control-Loops in Hierarchical Autonomic Management and Control Architectural Frameworks* such as GANA: i.e. *Stability in GANA*.
- *Reference Points* necessitated by the autonomic manager components and Information (or Knowledge) Sharing components.
- How *cross-layer information* may need to flow to *all* Decision Elements (DEs) at the Function-Level (Level-2 in GANA).
- How to incorporate *Cognition in Decision Elements (DEs) and Cognition Levels in the GANA Decision Plane Hierarchy*.
- *Knowledge Plane* as part of the *GANA Decision Plane*.
- How to accommodate *Virtualization Techniques*.
- Mappings to existing management frameworks such as TMN-LLA and FCAPS framework, and how service management and network management as well as associated systems such as OSS's would need to evolve as necessitated by the GANA framework.
- *Federation* in GANA,
- *Decision Notification* in GANA for the "Human in the Loop",

11 Bibliography, Terminology and Definitions

11.1 Publications (related to this area of research) authored or co-authored by the author

[Chaparadza05a] Chaparadza R.; Glickman, Y.; Deussen, P.H., RTLG - RSVP Enabled Traffic Load Generator for Intra-Domain and Inter-Domain QoS Signalling Tests. In the proceedings of the International Workshop on Internet Performance, Simulation and Measurements (IPS-MoMe-2005), Warsaw, Poland, 14 March 2005.

[Chaparadza05b] Chaparadza R.: Improving the automation of network management and troubleshooting tasks by applying On-Demand Monitoring of protocol(s) specific traffic in multi-service networks. 5th IEEE International Workshop on IP Operations and Management (IPOM 2005), in the UPC local proceedings, Barcelona, Spain, October 2005.

[Chaparadza05c] Chaparadza R.: On designing SNMP based monitoring systems supporting ubiquitous access and real-time visualization of traffic flow in the network, using low cost tools. In the proceedings of the IEEE International Conference on Networks and IEEE Malaysia International Conference on Communications and (MICC & ICON 2005), Kuala Lumpur, Malaysia, 16-18 November.

[Chaparadza05d] Chaparadza R., Coskun H, Schieferdecker I.: Addressing some challenges in autonomic monitoring in self-managing networks. In proceedings of IEEE International Conference on Networks and IEEE Malaysia International Conference on Communications and (MICC & ICON 2005), Kuala Lumpur, Malaysia, 16-18 November.

[Chaparadza06a] Chaparadza R: Introducing On-Demand MIBs to Traffic Engineering. In the proceedings of 2006 IEEE International Conference on Networks (ICON2006), Singapore. Networking – challenges and frontiers. September 13-15 2006, Singapore.

[Chaparadza06b] Chaparadza R, Peter M, Schieferdecker I: FANSA - a Framework for Automated Network Security Auditing. In proceedings of the International Conference On Late Advances in Networks (ICLAN'2006), December 06-08, 2006, Paris, France.

[Chaparadza07a] Chaparadza R.: A Composition Language for Programmable Traffic Flow Monitoring in Multi-Service Self-Managing Networks. In proceedings of the 6th IEEE International Workshop on Design of Reliable Communication Networks (DRCN 2007), La Rochelle, France.

[May07] Martin May M., Siekkinen M., Goebel V., Plagemann T., Chaparadza R.: Monitoring as First Class Citizen in an Autonomic Network Universe. In proceedings of the Workshop on Technologies for Situated and Autonomic Communications Co-located with BIONETICS 2007, Budapest, Hungary: December 10-12, 2007.

[Chaparadza07b] Chaparadza R.: Evolution of the current IPv6 towards IPv6++ (IPv6 with Autonomic Flavours). Published by the International Engineering Consortium (IEC) in the Review of Communications, Volume 60, December 2007.

[Chaparadza08a] Chaparadza R.: UniFAFF: A Unified Framework for Implementing Autonomic Fault-Management and Failure-Detection for Self-Managing Networks –published by the International Journal of Network Management, John Wiley & Sons 2008.

[Chaparadza08b] Chaparadza R.: Requirements for a Generic Autonomic Network Architecture (GANA), suitable for Standardizable Autonomic Behaviour Specifications of Decision-Making-Elements (DMEs) for Diverse Networking Environments: To be published by the International Engineering Consortium (IEC) in the Review of Communications, Volume 61, December 2008.

[Chaparadza09] Ranganai Chaparadza, Symeon Papavassiliou, Timotheos Kastrinogiannis, Martin Vigoureux, Emmanuel Dotaro, Alan Davy, Kevin Quinn, Michał Wódczak , Andras Toth , Athanassios Liakopoulos, Mick Wilson: *Creating a viable Evolution Path towards Self-Managing Future Internet via a Standardizable Reference Model for Autonomic Network Engineering*, Future Internet Assembly (FIA) Prague 2009 Conference, Czech:, published in the Future Internet Book produced by FIA.

[Retvari09] Gabor Retvari, Felician Nemeth, Ranganai Chaparadza, Robert Szabo: OSPF for Implementing Self-adaptive Routing in Autonomic Networks: a Case Study: In proceedings of the the 4th IEEE International Workshop on Modelling Autonomic Communication Environments (MACE 2009), October 26-27 2009, Venice, Italy.

[Chaparadza10a] R. Chaparadza, S. Papavassiliou, S. Soulhi, J. Ding: The Self-Managing Future Internet powered by the current IPv6 and Extensions to IPv6 towards "IPv6++"—a viable Roadmap Scenario for the Internet Evolution Path. In proceedings of the IEEE MENS Workshop at IEEE Globecom 2010: 6-10 December 2010, Miami, Florida, USA.

[Simon10] Csaba Simon, Ranganai Chaparadza, Péter Benko, Domonkos Asztalos, Vassilios Kaldanis: Enabling autonomicity in the future networks: In proceedings of the IEEE MENS Workshop at IEEE Globecom 2010: 6-10 December 2010, Miami, Florida, USA.

[Tcholtchev10a] N. Tcholtchev and R. Chaparadza: Autonomic Fault-Management and Resilience from the Perspective of the Network Operation Personnel: In proceedings of the IEEE MENS Workshop at IEEE Globecom 2010: 6-10 December 2010, Miami, Florida, USA.

[Rebahi10] Yacine Rebahi, Nikolay Tcholtchev, Ranganai Chaparadaza, Vassilis N. Merekoulias: „Addressing Security Issues in the Autonomic Future Internet", Work in Progress paper: In proceedings of IEEE Consumer Communications and Networking Conference 2011.

[Chaparadza10b] R. Chaparadza, N. Tcholtchev, V.Kaldanis: How *Autonomic Fault-Management* can address current challenges in *Fault-Management* faced in IT and Telecommunication Networks: In: SELFMAGICNETS'10: Proc. of the International Workshop on Autonomic Networking and Self-Management in Access Networks, in conjunction with ICST ACCESSNETS Conference '10, Budapest, Hungary (Nov 2010).

[Tcholtchev10b] Nikolay Tcholtchev and Ranganai Chaparadza: "On Self-Healing based on collaborating End-Systems, Access and Core Network Components": In the proceedings of the International Workshop on Autonomic Networking and Self-Management in the Access Networks (SELFMAGICNETS 2010), in conjunction with ICST ACCESSNETS Conference '10, 3-5 November 2010, Budapest, Hungary.

[Prakash10a] Prakash, A., Starschenko, A., Chaparadza, R.: Auto-Discovery and Auto-Configuration of Routers in an Autonomic Network. In: SELFMAGICNETS'10: Proc. of the International Workshop on Autonomic Networking and Self-Management in Access Networks, in conjunction with ICST ACCESSNETS Conference '10, Budapest, Hungary (Nov 2010).

[Prakash10b] Arun Prakash, Zoltan Theisz, and Ranganai Chaparadza: Formal Methods for Modeling, Refining and Verifying Autonomic Components of Computer Networks: To appear in the Journal: Springer Transactions on Computational Sciences, 2010.

[Prakash10c] A. Prakash, R. Chaparadza, Z. Theisz, "Requirements of a Model-Driven Methodology and Tool-Chain for the Design and Verification of Hierarchical Controllers of an Autonomic Network", 1st international Conference on Models and Ontology-based Design of Protocols, Architectures and Services (MOPAS), Athens, Greece, June 2010.

[Kaldanis10] V. Kaldanis, P.Benko, D.Asztalos, C.Simon,, R.Chaparadza, G.Katsaros: Methodology Towards Integrating Scenarios and Testbeds for Demonstrating Autonomic/self-managing Networks and Behaviors required in Future Networks: In: SELFMAGICNETS'10: Proc. of the International Workshop on Autonomic Networking and Self-Management in Access Networks, in conjunction with ICST ACCESSNETS Conference, Budapest, Hungary (Nov 2010).

[Chaparadza10c] Chaparadza, R. et al.: IPv6 and Extended IPv6 (IPv6++) Features that enable Autonomic Network Setup and Operation. In: SELFMAGICNETS '10: Proceedings of the International Workshop on Autonomic Networking and Self-Management in the Access Networks, in conjunction with ICST ACCESSNETS Conference Budapest, Hungary (Nov 2010).

[Kastrinogiannis10] T. Kastrinogiannis, N. Tcholtchev, A. Prakash, R. Chaparadza, V. Kaldanis, H. Coskun, and S. Papavassiliou, “Addressing Stability in Future Autonomic Networking,” in Proc. the 2nd Int. ICST Conf. on Mobile Networks and Management (MONAMI 2010), Santander, Spain, Sept, 2010.

[Tcholtchev09] Nikolay Tcholtchev, Ranganai Chaparadza and Arun Prakash: "Addressing Stability of Control-Loops in the context of the GANA architecture: Synchronization of Actions and Policies", Intl. Workshop on Self-Organizing Systems, IWSOS 2009, Zurich, Switzerland, 2009.

[Zafeiropoulos09] Anastasios Zafeiropoulos, Athanassios Liakopoulos, Alan Davy, and Ranganai Chaparadza: Monitoring within an Autonomic Network: A GANA Based Network Monitoring Framework: In the proceedings of the 2nd Workshop on Monitoring Adaptation and Beyond MONA+ 2009: Copyright by Springer, 2009.

[Liakopoulos08] A. Liakopoulos, A.Zafeiropoulos, A.Polyrakis, M.Grammatikou, J.M.González, M.Wodczak, R.Chaparadza, "Monitoring Issues for Autonomic Networks: The EFIPSANS Vision", 1st European Workshop on Mechanisms for the Future Internet, 2008, 10-11 July 2008, Salzburg, Austria.

11.2 Other Relevant Bibliography

[Abbas03] A. Abbas: The Autonomic Computing Report – Characteristics of Self-Managing IT Systems. White paper: Grid Technology Partners, 2003.

[**Agarwal03**] D. Agarwal, J. M. González, G.Jin, B.Tierney: An Infrastructure for Passive Network Monitoring of Application Data Streams. In the Proceedings of the 2003 Passive and Active Monitoring Workshop, PAM2003, April 6-8 2003, La Jolla, California.

[**ANA**] EC-funded IST FP6 project: http://www.ana-project.org.

[**ANA_delivD3.5**] Chaparadza R. Specification of the failure-detection and fault-management part of the ANA architecture (v1), ANA Project Deliverable D.3.5v1. http://www.ana-project.org/ [17 June 2008].

[**ANS**] The Human Autonomic Nervous System:
http://users.rcn.com/jkimball.ma.ultranet/BiologyPages/P/PNS.html#autonomic

[Aristomenopoulos10] G. Aristomenopoulos et al: Autonomic Mobility and Resource Management Over an Integrated Wireless Environment-A GANA Oriented Architecture. In proceedings of the IEEE MENS Workshop at Globecom 2010: 6-10 December 2010, Miami, Florida, USA.

[Arlos05] P. Arlos, M. Fiedler, A. A. Nilsson: A Distributed Passive Measurement Infrastructure. In proceedings of the 6th International Workshop on Passive and Active Network Measurement, PAM 2005, Boston, MA, USA, March 31 - April 1, 2005.

[AT&T04] AT&T White paper- Self-Managing Networks: Building the Infrastructure for Utility Computing, released in 2004: http://www.corp.att.com/news/2004/05/27-13088

[Autonomic_Computing] http://en.wikipedia.org/wiki/Autonomic_Computing

[Autonomic_Networking] http://en.wikipedia.org/wiki/Autonomic_Networking.

[Avizienis04] Avizienis A, Laprie J-C, Randell B, Landwehr C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 2004; **1**: 11–33.

[**Bos04**] H. Bos et al: FFPF: Fairly Fast Packet Filters. In proceedings of the 6th Symposium on Operating Systems Design & Implementation (OSDI '04), December 6-8, San Francisco, CA, USA.

[**BPF**] FreeBSD Man Pages - Manual Reference Pages - BPF (Berkeley Packet Filter).

[**Brownlee97**] N. Brownlee: RFC 2123: Traffic Flow Measurement: Experiences with NeTraMet. March 1997.

[**CAIDA**] http://www.caida.org/tools/

[**Chaparadza08c**] R.Chaparadza et al: First Draft of Autonomic Behaviours Specifications (ABs) for Selected Diverse Networking Environments. INFSO-ICT-215549-EFIPSANS-FP7-IP EU funded Project: Deliverable-D1.1: issued on 30.06.2008.

[Chtepen08] Maria Chtepen, Filip H.A. Claeys, Bart Dhoedt, Filip De Turck, Peter A. Vanrolleghem, and Piet Demeester: Scheduling of dependent grid jobs in absence of exact job length information: In Proceedings of EVGM2008, the 4th IEEE/IFIP International Workshop on End-to-End Virtualization and Grid Management, in conjunction with the 5th International Workshop on Next Generation Networking Middleware (NGNM2008).

[Chtepen09] Chtepen, M., F.H.A. Claeys, B. Dhoedt, F. De Turck, P.A.Vanrolleghem, and P. Demeester: Performance evaluation and optimization of an adaptive scheduling approach for dependent grid jobs with unknown execution time: In proceedings of 18th World IMACS / MODSIM Congress, Cairns, Australia 13-17 July 2009.

[**Cisco**] Cisco Packet Telephony Center Monitoring and Troubleshooting, the challenges to Troubleshooting, an article from Cisco: http://www.cisco.com/en/US/products/sw/netmgtsw/ps4883/

[CONMan] Ballani, H., Francis, P.: CONMan: A Step Towards Network Manageability. In: SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications. pp. 205{216. ACM, New York, NY, USA (2007)}

[**CoralReef**] http://www.caida.org/tools/measurement/coralreef/

[**Cranor02**] C. Cranor et al: Gigascope: High Performance Network Monitoring with an SQL Interface: In proceedings of the ACM SIGMOD 2002, June 3-6 2002, Madison, Wisconsin, USA.

[**Danalis03**] A. Danalis, C. Dovrolis: ANEMOS: An Autonomous NEtwork MOnitoring System. In Proceedings of 4th Passive and Active Measurements Workshop: La Jolla, California, April 6-8, 2003.

[**Day08**] John Day: Partterns in Network Architecture, A Return to Fundamentals: Prentice Hall, Copyright 2008 Pearson Education, Inc.

[**EFIPSANS**] EC FP7-EFIPSANS Project: http://efipsans.org/

[**Ethereal**] Ethereal Multi-Platform Protocol Analyzer: http://www.ethereal.com/.

[ETSI AFI] ETSI Industry Specification Group: "*Autonomic network engineering for the self-managing Future Internet*"—AFI in short**:** http://portal.etsi.org/afi/

[Greenberg05] Albert Greenberg et al: A clean slate 4D approach to network control and management: In ACM SIGCOMM Computer Communication Review, 2005.

[**Hruby05**] T. Hruby et al: Lessons learned in developing a flexible packet processor for high speed links. In Technical Report IR-CS-016, Vrije Universiteit Amsterdam, 2005.

[**Huang06**] L. Huang et al: Toward sophisticated detection with distributed triggers. In proceedings of Proceedings of the 2006 SIGCOMM workshop on Mining network data.

[MAPE] IBM article: Understand the autonomic manager concept: http://www-128.ibm.com/developerworks/library/ac-amconcept/.

[**IEC**] IEC (International Engineering Consortium) Tutorial: Internet Model for Control of Converged Networks: We ProForum Tutorials, Copyright International Engineering Consortium.

[**IPFIX**] http://www.ietf.org/html.charters/ipfix-charter.html.

[Intel] Towards an Autonomic Framework: Self-Configuring Network Services and Developing Autonomic Applications: Intel Technology Journal, Volume 8, Issue 4, 2004.

[**ITU-T Z.100**] Specification and Description Language (SDL): standardized by ITU-T as standard Z.100.

[ITU-T Rec. M.3400] The FCAPS management Framework: ITU-T Recommendation M.3400: Telecommunications management network: Copyright ITU 2001.

[**Jain04**] A. Jain, J.M. Hellerstein, S. Ratnasamy: Third Workshop on Hot Topics in Networks: HotNets-III, November 15-16, 2004, San Diego, CA USA.

[**Juniper**] http://www.juniper.net/.

[**Keys01**] K. Keys, D. Moore, R. Koga, E. Lagache, M. Tesch, and K. Claffy, "The architecture of CoralReef: an Internet traffic monitoring software suite," in Proceedings of the 2nd International Passive and Active Network Measurement Workshop, Apr. 2001.

[Knowledge Plane] David D. Clark et al: A Knowledge Plane for the Internet: *In proceedings of SIGCOMM'03*, August 25–29, 2003, Karlsruhe, Germany.

[Liakopoulos10] A. Liakopoulos et al: Applying distributed monitoring techniques in autonomic networks: In proceedings of the IEEE MENS Workshop at Globecom 2010: 6-10 December 2010, Miami, Florida, USA.

[**libpcap**] http://sourceforge.net/projects/libpcap/

[Linux_runtime_resource_measurement]: Methods for measuring runtime memory consumed by processes and threads: (A) http://elinux.org/Runtime_Memory_Measurement; http://elinux.org/Accurate_Memory_Measurement ; http://www.kernel.org/doc/man-pages/online/pages/man3/pthread_create.3.html

[Linux_runtime_resource_measurement2] http://elinux.org/images/d/d3/Mem_usage

[**MirrorAndSpan**] Port Mirroring/Spanning: http://www.cisco.com/warp/public/473/41.html

[Mortier06] R. Mortier, E. Kiciman: Autonomic Network Management: Some Pragmatic Considerations: SIGCOMM'06 Workshops, September 11-15, 2006, Pisa, Italy.

[**NetFlow**] http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html http://en.wikipedia.org/wiki/Netflow

[**Net-SNMP**] Net-SNMP Tools: http://www.net-snmp.org/.

[**Netview**] IBM Netview for AIX. [Available Online]: http://www.tivoli.com/products/index/netview/

[**nmon**] http://www.ibm.com/developerworks/aix/library/au-analyze_aix/

[**ODM-Probe-Source-Code**] http://sourceforge.net/projects/odm-probe/

[**Openview**] a network management platform from Hewlett Packard. HP Openview. [Available Online]: http://www.openview.hp.com/

[**PassiveActiveMonit**] Passive and Active Monitoring Techniques: http://www.slac.stanford.edu/comp/net/wan-mon/passive-vs-active.html

[**Patarin00**] S. Patarin, M. Makpangou: Pandora: a Flexible Network Monitoring Platform. In Proceedings of the USENIX 2000 Annual Technical Conference.

[**Paxson98**] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis, "An architecture for large-scale internet measurement," IEEE Commun. Mag., vol. 36, no. 8, pp. 48–54, Aug. 1998.

[PBNM] Strassner, J., "Policy-Based Network Management", Morgan Kaufman Publishers, ISBN 1-55860-859-1, Sep 2003.

[**Polychronakis04**] M. Polychronakis, K. G. Anagnostakis, E. P. Markatos, and A. Øslebø, "Design of an Application Programming Interface for IP Network Monitoring," in Proceedings of the 9th IFIP/IEEE Network Operations and Management Symposium (NOMS'04), Apr. 2004, pp. 483–496.

[**RFC 2205**] Resource ReSerVation Protocol (RSVP).

[**RFC 2275**] RFC 2275 - View-based Access Control Model (VACM) for the Simple Network Management Protocol (SNMP).

[**RFC 2474**] Definition of the Differentiated Services Field – IETF RFC 2474.

[**RFC 2574**] RFC 2574 - User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3).

[**RFC 2578**] Structure of Management Information Version 2 (SMIv2) – IETF RFC 2578

[**RFC 2721**] N. Brownlee: RTFM: Applicability Statement.

[**RFC 2741**] IETF RFC 2741 - Agent Extensibility (AgentX) Protocol Version 1

[**RFC 2981**] Event MIB- IETF RFC 2981.

[**RFC 2982**] Distributed Management Expression MIB - IETF RFC 2982.

[**RFC 3165**] The DISMAN ScriptMIB: RFC 3165.

[**RFC 3272**] Overview and Principles of Internet Traffic Engineering (IETF RFC 3272).

[**RFC 3577**] Introduction to the Remote Monitoring (RMON) Family of MIB Modules.

[**ScriptMIB**] IETF-SCRIPT-MIB:http://net-snmp.sourceforge.net/docs/mibs/DISMAN-SCRIPT-MIB.txt.

[Smirnov03] M. Smirnov: Autonomic Communication, Research Agenda for a New Communication Paradigm. White paper, Fraunhofer FOKUS, 2003.

[**SNMP**] http://www.ietf.org/rfc/rfc2571.txt

[**Sourdis03**] I. Sourdis and D. Pnevmatikatos. Fast, Large-Scale String Match for a 10 Gbps FPGA-based Network Intrusion Detection System. In Proceedings of the 13th International Conference on Field Programmable Logic and Applications, September 2003.

[Strassner06] John. C. Strassner, N. Agoulmine, E.Lehtihet: FOCALE – A Novel Autonomic Networking Architecture: In proceedings of the Latin American Autonomic Computing Symposium (LAACS), Campo Grande, MS; Brazil 2006.

[Steinder04] Steinder M, Sethi AS. A survey of fault localization techniques in computer networks. In *Science of Computer Programming* 2004; **53**: 165-194.

[**SUN**] SUN solstice. [Available Online]: http://www.sun.com/productsn-solutions/sw/solstice/

[**tcpdump**] http://www.tcpdump.org/

[**Tekelec**] SS7 management and Troubleshooting challenges, articles from Tekelec: http://www.tekelec.com/

[ThreadPriorityAndScheduling]: Pthread priority and scheduling:
http://cpp-applied.info/IBM.Redbooks-Developing.and.Porting.C.and.CPP.Applications.on.AIX/7338final/LiB0063.html

[**TMN**] ITU-T Recommendation M.3010: Principles for a telecommunications management network: Copyright ITU 2000.

[**Trimintzios06**] P. Trimintzios et al: DiMAPI: An Application Programming Interface for Distributed Network Monitoring: In proceedings of the Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP. Digital Object Identifier: 10.1109/NOMS.2006.1687568.

[**Varghese05**] George Varghese: Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networks Devices: Morgan Kaufmann: Copyright Elsevier, 2005.

[Wang07] Pi-Chung Wang, Chun-Liang Lee, Chia-Tai Chan, and Hung-Yi Chang: Performance Improvement of Two-Dimensional Packet Classification by Filter Rephrasing. In IEEE/ACM Transactions on Networking, Vol. 15, No.4, August 2007. DOI: 10.1109/TNET.2007.893872.

[**Watson03**] D. Watson, G. R. Malan and F. Jahanian: An extensible probe architecture for network protocol performance measurement. SOFTWARE—PRACTICE AND EXPERIENCE Softw. Pract. Exper. 2004; 34:47–67 (DOI: 10.1002/spe.557). Copyright c _2003 John Wiley & Sons, Ltd.

[Weyuker00] Elaine J. Weyuker, and Filippos I. Vokolos: Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study: In IEEE Transactions on Software Engineering, Vol. 26, No. 12, December 2000.

[**XML**] The XML Language: http://www.w3.org/XML/

[Zhang10] Haiyan Zhang, Mincheng Zhao, Wendong Wang, Xiangyang Gong, Xirong Que: A Novel Autonomic Architecture for QoS Management in Wired Network: In proceedings of the IEEE MENS Workshop at IEEE Globecom 2010: 6-10 December 2010, Miami, Florida, USA.

Terminology And Definitions

11.3 Abbreviations

ANM – Autonomic Node Manager

ASN.1 – Abstract Syntax Notation One

ATM – Asynchronous Transfer Mode

CMDL - Capability Model Description Language

DE – Decision Element (synonymous with DME)

DME – Decision Making Element (synonymous with DE)

EDBSLang – Event Description and Behaviour Specification Language

EDCE – Event Detection & Computation Engine

FCAPS – Framework for **F**ault-Management, **C**onfiguration-Management, **A**ccounting-Management, **P**erformance-Management, **S**ecurity Management

GANA – Generic Autonomic Network Architecture

GPRS – General Packet Radio Service

IP – Internet Protocol

ME – Managed Entity

MIB – Management Information Base

MPAE – Multi-Protocol Analyzer Engine

MPLS – Multi-Protocol Label Switching

NMS – Network Management System

NOC – Network Operations Center

ODM – On-Demand Monitoring

odmRHACC– ODM Request Handler & Admission Control Component

OID – Object Identifier

PBNM – Policy Based Network Management

RSVP – Resource Reservation Protocol

SDL – Specification and Description Language

SMI – Structure of Management Information

SNMP – Simple Network Management Protocol

TCP – Transmission Control Protocol

TG –Traffic Generator

TMN – Telecommunications Management Network

TS – Traffic Sinker (for sink on request i.e. absorb and drop the traffic so it does not pass the point)

TTL – Time-to-Live

UDP – User Datagram Protocol

UMTS – Universal Mobile Telecommunications System

11.4 Definitions

Active Monitoring — A monitoring technique (also known as active probing) that involves the generation of intrusive traffic whose characteristics are then determined or for which measurements are taken at some points of traffic observation in the network [PassiveActiveMonit].

Automated Task — An *Automated Task* (in an abstract sense) is an executable system function or network function that has a start-time and stop-time, some logic (a set of ordered steps and decisions executed to achieve its intended goal(s)) and may require some monitoring at some point in *space* (location) and *time* during its execution lifetime in order to use monitoring information in taking some decisions during its execution lifetime.

Autonomic System — It is a system that is designed to exhibit the so-called self-* functions or properties triggered as a result of information flow within *control-loop structures* that underline the *design for autonomicity* of the system [Autonomic_Computing] [Autonomic_Networking]. Examples of self-* functions or properties are *self-configuring, self-healing, self-adaptation, self-optimisation, etc*, all of which are collectively called *self-managing* properties of such a system [Autonomic_Computing] [Autonomic_Networking]. Autonomic systems are systems designed to cope with those management complexities and tasks that are otherwise difficult or daunting to be handled by humans i.e. systems and network operations personnel. Refer to **Chapter 3** for more information on *autonomic systems engineering*.

CMDL (*Capability Model Description Language*) — It is a meta-language that can be used by ODM-capable monitoring components to self-describe their *Monitoring Capability Models* and publish them to the network. For more information on this subject, including the nature of such capability models, refer to **Chapter 6** as well as **Appendix B**.

EDBSLang (*Event Description and Behaviour Specification Language*) — It is a Composition *Language for specifying monitoring behaviours* that includes the possibility to *specify reactions* of a monitoring-behaviour that can be *instantiated* on an ODM-capable component. The specification of a monitoring-behaviour can be created programmatically by an Automated Task and uploaded into a target monitoring component for execution or can be created by network monitoring engineers and uploaded (instrumented) into ODM-capable components of the network so that automated tasks of the network can request for the execution of a selected monitoring-behaviour-specification without creating and uploading the specifications by themselves.

Managed Monitoring-Session — It is a monitoring session that is directly managed by an automated task via session-management primitives issued to the monitoring component running the session. It can be contrasted to a **Query-driven non-managed monitoring-session.** Refer to **Chapter 3-section 3.2** for more information on this concept.

Monitorable Flow — A "Monitorable Flow" is a kind of flow that a component can monitor, either by the way it is configured to monitor traffic AND/OR by virtue of its point of attachment to the network as part of the picture on "network observation (i.e. *observability*)". By "flow", we mean a "traffic flow" (see definition of a traffic flow).

Monitoring Component — a component designed to offer some dynamic monitoring services and provides interfaces through which monitoring services can be requested for and terminated.

Monitoring Context— A *monitoring-context* is the instantaneous aggregation of all the monitoring needs of diverse automated tasks being served by a monitoring component in terms of the executing monitoring functions or services (include all running threads and processes). Refer to **Chapter 3-section 3.2** for more information on this subject.

Monitoring Function — We define it as a piece of code or even a thread or process that can be invoked at run-time, serving a particular monitoring purpose e.g. a packet capturing function.

Monitoring Point — It is a point on a network topology, on which a *network element*$_{re\text{-}defined}$ associated with that point of attachment to the network has capabilities to monitor traffic flow.

Monitoring Probe — It is a specially designed software component or a device for monitoring purposes and can be instrumented/embedded into systems or installed at some vantage point(s) in the network. An example of a hardware-based monitoring probe is a RMON probe.

Monitoring-Behaviour — It is the behaviour of a monitoring service requested for by an *Automated Task,* and is meant to be dynamically offered by a targeted monitoring component. Refer to **Chapter 3-section 3.2** for more information on this subject.

Monitoring-Behaviour-Specification — It is a specification of the behaviour of the requested monitoring service, specified using an appropriate language and passed to the monitoring component along with a monitoring-request. Refer to **Chapter 3-section 3.2** for more information on this subject.

Monitoring-Request — A monitoring request is a request issued by an *Automated Task* to a monitoring component and conveys the monitoring-behaviour-specification requested for execution or to be used as a new binding monitoring-behaviour-specification to an already running monitoring service, or conveys a Monitoring Query. Refer to **Chapter 3-section 3.2** for more information on this subject. In this document, **Monitoring-Request** is synonymous with **ODM-Request**.

Monitoring-Session — It is a view of a monitoring service offered to an automated task by a monitoring component. A monitoring component provides the means to identify multiple sessions running on the component using *session-identifiers*, and also associates individual sessions with their session-owners i.e. the automated tasks (*session-creators/requesters/owners*) that created the sessions on the component. Session-owners receive their assigned *session-identifiers* for use in subsequent session-management on a target component. A *monitoring-session* is viewed by the associated session-owner as a monitoring-behaviour belonging the *session-owner* and has a lifetime i.e. Time-To-Live (TTL) determined by the session-owner. Refer to **Chapter 3-section 3.2** for more information on this subject. **Monitoring-session** is synonymous with **ODM-Session.**

Monitoring-Session Management — The management of a monitoring-session by an automated task on a targeted component via the use of *primitives like:* **`create-session`**,

`pause-session`, **`resume-session`** and **`stop/terminate-session`** for *managed-monitoring-sessions.* The support for such session-management primitives by a monitoring component is useful for allowing an automated task to create and manage its monitoring-session on a target monitoring component. Refer to **Chapter 3-section 3.2** for more information on this concept.

Monitoring-Session View — According to an *automated task* that is considered the *session-owner* by a monitoring component, it is the view that the monitoring component provides a *session-identifier* that distinguishes the session from other sessions running on the component, and allows the session-owner to use the *session-identifier* in subsequent session-management (see definition of self-management) requests issued by the session-owner. The *session-view* according to the monitoring component is that it has a session-identifier and the monitoring-session may be sharing resources on the system with other sessions, such as thread-specific monitoring functions e.g. packet capturing function (thread or process). Refer to **Chapter 3-section 3.2** for more information on this concept.

Multi-Service Network — A ***multi-service network*** is either a network consisting of a number of connected networks using different protocols of the same layer of either the TCP/IP model or the OSI model, and requiring translation (gateways) in order to seamlessly provide end-to-end services across the differing networks or is a network characterized by multiple services in terms of the type of user information or user data transferred end-to-end e.g. voice, data or video transport services i.e. the case of converged networks [IEC]. A ***multi-service network*** may also be a network characterized by a number of schemes or protocols such as diverse routing or transport schemes that are used in different situations in which it pays to use a particular scheme. Refer to **chapter 2—section 2.1** for more information on this subject.

Multi-Service Self-Managing Network — We talk of a **multi-service self-managing network** when the multi-service network implements self-managing functions across all the FCAPS [ITU-T Rec. M.3400] management planes for **F**ault-management, **C**onfiguration-management, **A**ccounting-management, **P**erformance-management and **S**ecurity-management.

Networking Function — It is an abstraction of network functionality (network function) that is implemented by a single protocol, or diverse protocols that employ diverse algorithmic schemes or policies of operation on a *network element*$_{re\text{-}defined}$, such that we are able to reason about networking functions of a *network element*$_{re\text{-}defined}$ like *its routing function*, *its forwarding function*, *its mobility management function*, *QoS management function*, etc. Refer to **Chapter 3** for more information on abstractions for network functionality.

Network Element$_{\mathbf{re\text{-}defined}}$ — In this document, we define a ***network element*** as meaning both, a managed physical "network device" such as switch, router, end-system, gateway, server-hardware, etc, as well as a managed "network node" instantiated in a physical network device

(box) that supports instantiation of multiple nodes in the same physical device (box). In the same network, the case of "node-management" for such kind of instantiated nodes may exist together with the case of pure "device-management" pertaining to the management of hardware devices that do not support the creation of nodes.

ODM-Capability Model — It is a descriptive model describing the monitoring capabilities of an ODM-Capable Component. The ODM-Capable component creates (at boot-up time) and self-publishes the model to the network, by pushing it into the Capability Models Database of the network. It is updated to the network whenever the monitoring capabilities of the component have changed. For more information on this subject, including the nature of such capability models and their potential dynamics, refer to **Chapter 6** as well as **Appendix B**.

ODM-Capable Component (i.e. an ODM-Supporting Component) — It is a component that supports the ODM-Principles (**Principle-P1 to Principle-P7**) defined and described in **Chapter 4**. Such an ODM-Capable Component may be implemented as forming the **traffic monitoring subsystem** of system or device. A system embedding such an ODM-Capable component is an ODM-Capable system. It may be a router, a host, a switch, a signaling gateway *or* a special traffic monitoring probe or a component embedded in such kind of systems.

ODM-MIB — It is an On-Demand SNMP MIB Data Model (*an in-memory on-demand data model*) that is created (if specified in a monitoring-behaviour-specification), and is destroyed by the monitoring component upon the expiration of the binding monitoring-session(s).

ODM-MIB Tree — It is an SNMP MIB tree that is created (if specified in a monitoring-behaviour-specification) by an ODM-Capable Component according to the OIDs requested in the monitoring-behaviour-specification being executed, and is created as an On-Demand SNMP MIB Data Model (*an in-memory on-demand data model*) that is destroyed by the monitoring component upon the expiration of the binding monitoring-session(s).

ODM-Platform — A collection of all the ODM-Capable Components (or systems) of a network, which form a traffic monitoring platform consisting of *traffic monitoring components* distributed throughout the network.

ODM-Probe — It is an ODM-Capable prototypical system i.e. an ODM-Capable Component that that supports the ODM-Principles (**Principle-P1 to Principle-P7**) defined and described in **Chapter 4**, whose functional specification is given in **Chapter 7**.

ODM–Request — It is a **Monitoring-Request** (see corresponding definition) that is issued to an ODM-Capable Component by an ODM-Session-Creator (see corresponding definition).

ODM-Session — It is **a Monitoring-Session** (see corresponding definition) created by an ODM-Capable Component and has an associated ODM-Session-Owner (see corresponding definition).

ODM-Session-Creator — It is an *Automated Task* whose Monitoring-Request (ODM-Request) previously issued to an ODM-Capable Component has been accepted/admitted, resulting in an ODM-Session being created by the component as a result, and the *Automated Task* becoming an **ODM-Session-Owner (**see corresponding definition).

ODM-Session-Owner — A session-owner is an *Automated Task* e.g. a protocol, a task automation application such as a troubleshooter or an entity acting on behalf of other entities e.g. a Network Management System (NMS) that invoked a monitoring-behaviour (session-behaviour) on a monitoring component at a point in the network. The invoked monitoring-behaviour should be specific to the needs and/or the control of the session-owner and can be terminated by this session-owner. The session-owner can be remote or local to the system hosting the monitoring component (i.e. the ODM-Capable Component). Refer to **Chapter 3-section 3.2** for more information on this concept.

ODM-Session-Requester — It is an Automated Task that issues a Monitoring-Request (ODM-Request) to an ODM-Capable Component.

ODM-Traffic-Filter — A traffic filter specified in monitoring-behaviour-specification or in a Monitoring-Query conveyed by a Monitoring-Request (ODM-Request).

Passive Monitoring — A monitoring technique that involves capturing or sniffing traffic in a non-intrusive manner at some point of traffic observation in the network, and then analysing, inferring, or determining some traffic measurements or traffic characteristics of interest from the captured traffic.

Query-driven non-managed monitoring-session — It is a monitoring-session that is not directly managed by an automated task but is rather managed internally by the monitoring component. It can be contrasted to a **Managed monitoring-session.** Refer to **Chapter 3-section 3.2** for more information on this concept.

Self-Managing Network — It is a network whose network network elements (see definition of *network element*$_{re\text{-}defined}$) are designed/engineered in such a way that all the traditionally so-called network management functions defined by the FCAPS management framework, as well as the fundamental network functions such as routing, forwarding, monitoring, supervision, fault-detection and fault-removal, etc, are made to automatically feed each other with information (knowledge) such as events, in order to effect feedback processes among the diverse functions,

thereby enabling reactions in individual diverse functions of the network and of individual network elements (see definition of *network element*$_{re\text{-}defined}$), in order to achieve and strive to maintain some well defined goals of the network. As such, even the FCAPS functions become diffused into *network element*$_{re\text{-}defined}$ architectures and network architectures. Refer to **Chapter 3** for more information how self-managing networks can be engineered.

Traffic Filtering — A process of capturing traffic whose characteristics match a filter expression according to a particular filter language used. Refer to sources like: (chapter 8 of [Varghese05]) [Ethereal] [tcpdump] [CoralReef] [libpcap] [Watson03] [Bos04] for traffic filter types and languages.

Traffic Flow (or simply a flow) — A *traffic flow* is a stream of transferable units of information e.g. packets that share some properties. Without talking about whether the units of information are in the process of being transferred from one point to another, we can allow some space to think of a flow in an imaginary sense and be able to reason about the absence or presence of a flow at some point of observation.

Traffic Flow(s)-Generator Capability Model — An ODM-Capable component may have the capability of generating and/or sinking of a diagnostic/test flow(s) of specific properties and/or possibly the participation of the system hosting the component in active measurements of traffic characteristics. Such capability should be described using a *Flow(s)-Generator Capability Model.* For more information on this subject, including the nature of such a capability model, refer to **Chapter 6** as well as **Appendix B**.

Traffic Flow(s)-Sinker Capability Model — An ODM-Capable component may have the capability of sinking or dumping (on request) certain types of flows e.g. diagnostic flows injected by some automated task in the network, such that the flow's packets are not forwarded further via the egress interface. Such capability should be described using a *Flow(s)-Sinker Capability Model*. For more information on this subject, including the nature of such capability model, refer to **Chapter 6** as well as **Appendix B**.

12 Appendix A

In this **Appendix A**, we present the XML-Schema of the *EDBSLang Language*—a composition language for specifying monitoring-behaviour-sepecification.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
	<xs:element name="ODM_Behaviour_Specification">
		<xs:complexType>
			<xs:sequence>
				<xs:element name="Nth_packet">
					<xs:complexType>
						<xs:sequence>
							<xs:element name="number" type="xs:integer" />
							<xs:element name="layer" type="xs:integer" />
							<xs:element name="field">
								<xs:complexType>
									<xs:sequence>
										<xs:element name="number" type="xs:integer" />
										<xs:element name="value" type="xs:string" />
										<xs:element name="event_notification" type="xs:boolean" />
										<xs:element name="dest" type="xs:string" />
										<xs:element name="event_description" type="xs:string" />
										<xs:element name="pause_session" type="xs:boolean" />
										<xs:element name="terminate_session" type="xs:boolean" />
										<xs:element name="event_name" type="xs:string" />
										<xs:element name="goto_statement" type="xs:string" />
										<xs:element name="notification_sink_action" type="xs:string"/>
									</xs:sequence>
								</xs:complexType>
							</xs:element>
							<xs:element name="byte">
								<xs:complexType>
									<xs:sequence>
										<xs:element name="number" type="xs:integer" />
										<xs:element name="value" type="xs:string" />
										<xs:element name="event_notification" type="xs:boolean" />
										<xs:element name="dest" type="xs:string" />
										<xs:element name="event_description" type="xs:string" />
```

```
                                        <xs:element name="pause_session" type="xs:boolean" />
                                        <xs:element name="terminate_session" type="xs:boolean" />
                                        <xs:element name="event_name" type="xs:string" />
                                        <xs:element name="goto_statement" type="xs:string" />
                                        <xs:element name="notification_sink_action" type="xs:string"/>
                                    </xs:sequence>
                                </xs:complexType>
                            </xs:element>
                            <xs:element name="Label" type="xs:string" />
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
        <xs:element name="no_packet_captured">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element name="seconds" type="xs:integer" />
                                <xs:element name="event_notification" type="xs:boolean" />
                                <xs:element name="dest" type="xs:string" />
                                <xs:element name="event_description" type="xs:string" />
                                <xs:element name="own_action" type="xs:string" />
                                <xs:element name="pause_session" type="xs:boolean" />
                                <xs:element name="terminate_session" type="xs:boolean" />
                                <xs:element name="goto_statement" type="xs:string" />
                                <xs:element name="Label" type="xs:string" />
                                <xs:element name="notification_sink_action" type="xs:string" />
                                <xs:element name="reference" type="xs:string" />
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
        <xs:element name="packet_captured">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element name="reference" type="xs:string" />
                                <xs:element name="seconds" type="xs:integer" />
                                <xs:element name="event_notification" type="xs:boolean" />
                                <xs:element name="dest" type="xs:string" />
                                <xs:element name="event_description" type="xs:string" />
                                <xs:element name="notification_sink_action" type="xs:string" />
                                <xs:element name="pause_session" type="xs:boolean" />
                                <xs:element name="terminate_session" type="xs:boolean" />
                                <xs:element name="goto_statement" type="xs:string" />
                                <xs:element name="Label" type="xs:string" />
                                <xs:element name="own_action" type="xs:string" />
                        </xs:sequence>
                </xs:complexType>
```

```
</xs:element>
<xs:element name="ComputationAndChecks">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="computation">
                                <xs:complexType>
                                        <xs:sequence>
                                                <xs:element name="computation" type="xs:string" />
                                                <xs:element name="duration" type="xs:long" />
                                        </xs:sequence>
                                </xs:complexType>
                        </xs:element>
                        <xs:element name="computationStep" type="xs:integer" />
                        <xs:element name="storeComputation" type="xs:boolean" />
                        <xs:element name="propagateComputedValue" type="xs:boolean" />
                        <xs:element name="computed_event_check">
                                <xs:complexType>
                                        <xs:sequence>
                                                <xs:element name="computation">
                                                        <xs:complexType>
                                                                <xs:sequence>
                                                                        <xs:element name="computation"
                                                                        type="xs:string" />
                                                                        <xs:element name="duration"
                                                                        type="xs:long" />
                                                                </xs:sequence>
                                                        </xs:complexType>
                                                </xs:element>
                                                <xs:element name="threshold" type="xs:boolean" />
                                                <xs:element name="booleanCheck" type="xs:boolean" />
                                                <xs:element name="event_notification" type="xs:boolean" />
                                                <xs:element name="dest" type="xs:string" />
                                                <xs:element name="event_description" type="xs:string" />
                                                <xs:element name="own_action" type="xs:string" />
                                                <xs:element name="pause_session" type="xs:boolean" />
                                                <xs:element name="terminate_session" type="xs:boolean" />
                                                <xs:element name="goto_statement" type="xs:string" />
                                                <xs:element name="event_name" type="xs:string" />
                                        </xs:sequence>
                                </xs:complexType>
                        </xs:element>
                        <xs:element name="Label" type="xs:string" />
                </xs:sequence>
        </xs:complexType>
</xs:element>
```

```
<xs:element name="FiniteStateMachine">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="m_state">
                                <xs:complexType>
                                        <xs:sequence>
                                                <xs:element name="TrafficStreamSource" type="xs:string" />
                                                <xs:element name="StateName" type="xs:string" />
                                                <xs:element name="Filter" type="xs:string" />
                                                <xs:element name="Relative_time" type="xs:long" />
                                                <xs:element name="NextStateMatch" type="xs:string" />
                                                <xs:element name="NextStateNoMatch" type="xs:string" />
                                                <xs:element name="own_action" type="xs:string" />
                                                <xs:element name="PacketCount" type="xs:string" />
                                                <xs:element name="computation" type="xs:string" />
                                                <xs:element name="propagateComputedValue" type="xs:boolean" />
                                                <xs:element name="event_description" type="xs:string" />
                                                <xs:element name="dest" type="xs:string" />
                                                <xs:element name="notification_sink_action" type="xs:string"/>
                                                <xs:element name="pause_session" type="xs:boolean" />
                                                <xs:element name="terminate_session" type="xs:boolean" />
                                                <xs:element name="event_name" type="xs:string" />
                                                <xs:element name="goto_stament" type="xs:string" />
                                                <xs:element name="TraceFile">
                                                        <xs:complexType>
                                                                <xs:sequence>
                                                                        <xs:element name="StartTime"
                                                                        type="xs:string" />
                                                                        <xs:element name="CaptureDuration"
                                                                        type="xs:long" />
                                                                        <xs:element name="CreateInterval"
                                                                        type="xs:integer" />
                                                                        <xs:element name="dest"
                                                                        type="xs:string"/>
                                                                        <xs:element name="format"
                                                                        type="xs:string" />
                                                                        <xs:element name="TraceName"
                                                                        type="xs:string" />
                                                                        <xs:element name="Label"
                                                                        type="xs:string" />
                                                                        <xs:element name="DisseminationMethod"
                                                                        type="xs:string" />
                                                                        <xs:element name="goto_statement"
                                                                        type="xs:string" />
                                                                </xs:sequence>
```

```
                                                        </xs:complexType>
                                                </xs:element>
                                        </xs:sequence>
                                </xs:complexType>
                        </xs:element>
                        <xs:element name="Label" type="xs:string" />
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="ODM_Traffic_Filter" type="xs:string" />
<xs:element name="ODM_MIB">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="Table_OID" type="xs:string" />
                        <xs:element name="Scalar_OID" type="xs:string" />
                        <xs:element name="Label" type="xs:string" />
                        <xs:element name="goto_statement" type="xs:string" />
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="ForLoop">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="NumberOfPackets" type="xs:long" />
                        <xs:element name="seconds" type="xs:integer" />
                        <xs:element name="own_action" type="xs:string" />
                        <xs:element name="Label" type="xs:string" />
                        <xs:element name="goto_statement" type="xs:string" />
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="Decision">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="expression" type="xs:string" />
                        <xs:element name="Label" type="xs:string" />
                        <xs:element name="EvaluationTrue">
                                <xs:complexType>
                                        <xs:sequence>
                                                <xs:element name="own_action" type="xs:string" />
                                                <xs:element name="goto_statement" type="xs:string" />
                                                <xs:element name="event_notification" type="xs:boolean" />
                                                <xs:element name="event_description" type="xs:string" />
                                                <xs:element name="event_name" type="xs:string" />
                                                <xs:element name="dest" type="xs:string" />
```

```
                                                <xs:element name="notification_sink_action" type="xs:string"/>
                                                <xs:element name="pause_session" type="xs:boolean" />
                                                <xs:element name="terminate_session" type="xs:boolean" />
                                        </xs:sequence>
                                </xs:complexType>
                        </xs:element>
                        <xs:element name="EvaluationFalse">
                                <xs:complexType>
                                        <xs:sequence>
                                                <xs:element name="own_action" type="xs:string" />
                                                <xs:element name="goto_statement" type="xs:string" />
                                                <xs:element name="event_notification" type="xs:boolean" />
                                                <xs:element name="event_description" type="xs:string" />
                                                <xs:element name="event_name" type="xs:string" />
                                                <xs:element name="dest" type="xs:string" />
                                                <xs:element name="notification_sink_action" type="xs:string"/>
                                                <xs:element name="pause_session" type="xs:boolean" />
                                                <xs:element name="terminate_session" type="xs:boolean" />
                                        </xs:sequence>
                                </xs:complexType>
                        </xs:element>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="SessionStartTime" type="xs:string" />
<xs:element name="SessionStopTime" type="xs:string" />
<xs:element name="TraceFile">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="StartTime" type="xs:string" />
                        <xs:element name="CaptureDuration" type="xs:long" />
                        <xs:element name="CreateInterval" type="xs:integer" />
                        <xs:element name="dest" type="xs:string" />
                        <xs:element name="format" type="xs:string" />
                        <xs:element name="TraceName" type="xs:string" />
                        <xs:element name="Label" type="xs:string" />
                        <xs:element name="DisseminationMethod" type="xs:string" />
                        <xs:element name="goto_statement" type="xs:string" />
                </xs:sequence>
        </xs:complexType>
</xs:element>
        </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

13 Appendix B

In this **Appendix B**, we present the *Capability Model Description Language* (CMDL) for the description of capabilities of an ODM-Capable component. CMDL can still be further evolved as research in this subject of capability models continues. CMDL can be used by ODM-capable monitoring components to self-describe their *Capability Models* and publish them to the network. CMDL is a meta-language (i.e. an XML Schema) for describing capability models of ODM-capable components or devices i.e. *ODM supporting devices (routers, switches, hosts, signalling gateways, special probes instrumented in a network, etc), diagnostic/test traffic flow generators and sinkers.* The *self-description* and *self-publishing* of capability models by a system/component should be subject to security policies. The issue of security in the descriptions and publishing of capability models by monitoring components was not covered in this research and would amount to a subject for further research.

In this appendix, first the overall structure of CMDL is presented, and then the descriptions and the significance of the Tags of the CMDL language are presented after the overall structure.

13.1 The XML-Schema of the Capability Model Description Language (CMDL)

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  - <xs:element name="ODMCapabilityModel">
    - <xs:complexType>
      - <xs:sequence>
        - <xs:element name="KnownSessionBehavioursSpecificationsTree">
          - <xs:complexType>
            - <xs:sequence>
              - <xs:element
                  name="InstanceOfEDBSLangBasedMonitoringBehaviourSpecification">
                - <xs:complexType>
                  - <xs:sequence>
                    - <xs:element
                        name="TheUniquelyAssignedBehaviourIDofTheMonitorigBehaviourSpec">
                      - <xs:complexType>
                          <xs:sequence />
                        </xs:complexType>
                      </xs:element>
                    - <xs:element name="TheSequentialPositionInTheSpecification">
                      - <xs:complexType>
                        - <xs:sequence>
                            <xs:element name="number"
                              type="xs:unsignedLong" />
                          - <xs:element name="EDBSLangTag">
                            - <xs:complexType>
                              - <xs:sequence>
                                  <xs:element
                                    name="TheAssignedIDofThisTagInstance"
                                    type="xs:unsignedLong" />
                                  <xs:element name="TagName"
                                    type="xs:string" />
                                  <xs:element name="AttributeValue"
                                    type="xs:string" />
                                - <xs:element name="element">
                                  - <xs:complexType>
                                    - <xs:sequence>
                                        <xs:element name="name"
                                          type="xs:string" />
                                        <xs:element name="value"
                                          type="xs:string" />
                                      - <xs:element name="element">
                                        - <xs:complexType>
                                          - <xs:sequence>
                                              <xs:element
                                                name="name"
                                                type="xs:string" />
                                              <xs:element
                                                name="value"
                                                type="xs:string" />
                                              <xs:element
                                                name="element"
                                                type="_x0028_element_x0029_" />
                                            </xs:sequence>
                                          </xs:complexType>
                                        </xs:element>
                                      </xs:sequence>
                                    </xs:complexType>
                                  </xs:element>
                                </xs:sequence>
                              </xs:complexType>
                            </xs:element>
                          </xs:sequence>
                        </xs:complexType>
```

```
                </xs:element>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  - <xs:element name="MonitorableFlows">
    - <xs:complexType>
      - <xs:sequence>
        - <xs:element name="MonitorableFlowTypeA">
          - <xs:complexType>
            - <xs:sequence>
                <xs:element name="TrafficFilterExpression" type="xs:string" />
                <xs:element name="protocol" type="xs:string" />
                <xs:element name="layer" type="xs:unsignedShort" />
                <xs:element name="lowerLayerProtocol" type="xs:string" />
              - <xs:element name="source">
                - <xs:complexType>
                  - <xs:sequence>
                      <xs:element name="srcAddr" type="xs:string" />
                      <xs:element name="srcPort"
                        type="xs:unsignedLong" />
                    </xs:sequence>
                  </xs:complexType>
                </xs:element>
              - <xs:element name="destination">
                - <xs:complexType>
                  - <xs:sequence>
                      <xs:element name="dstAddr" type="xs:string" />
                      <xs:element name="dstPort"
                        type="xs:unsignedLong" />
                    </xs:sequence>
                  </xs:complexType>
                </xs:element>
                <xs:element name="SourceNetwork" type="xs:string" />
                <xs:element name="DestinationNetwork" type="xs:string" />
              - <xs:element name="FlowDirection">
                - <xs:complexType>
                  - <xs:sequence>
                    - <xs:element name="PrevHopForThisFlow">
                      - <xs:complexType>
                        - <xs:sequence>
                            <xs:element name="NetworkLayerAddr"
                              type="xs:string" />
                            <xs:element
                              name="PhysicalAddressOfNeighborInterface"
                              type="xs:string" />
                          </xs:sequence>
                        </xs:complexType>
                      </xs:element>
                    - <xs:element name="NextHopForThisFlow">
                      - <xs:complexType>
                        - <xs:sequence>
                            <xs:element name="NetworkLayerAddr"
                              type="xs:string" />
                            <xs:element
                              name="PhysicalAddressOfNeighborInterface"
                              type="xs:string" />
                          </xs:sequence>
                        </xs:complexType>
                      </xs:element>
```

```
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="NetworkPrefixOfIngressInterface"
      type="xs:string" />
    <xs:element name="NetworkPrefixOfTheMonitoringInterface"
      type="xs:string" />
  - <xs:element name="PerformableActions">
    - <xs:complexType>
      - <xs:sequence>
        - <xs:element name="BitModification">
          - <xs:complexType>
            - <xs:sequence>
                <xs:element name="TheDataUnit"
                  type="xs:string" />
                <xs:element name="BitNumber"
                  type="xs:positiveInteger" />
                <xs:element name="BitValue"
                  type="xs:hexBinary" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        - <xs:element name="ByteModification">
          - <xs:complexType>
            - <xs:sequence>
                <xs:element name="TheDataUnit"
                  type="xs:string" />
                <xs:element name="ByteNumber"
                  type="xs:positiveInteger" />
                <xs:element name="ByteValue"
                  type="xs:hexBinary" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        - <xs:element name="ProtoFieldModification">
          - <xs:complexType>
            - <xs:sequence>
                <xs:element name="Protocol"
                  type="xs:string" />
                <xs:element name="FieldName"
                  type="xs:string" />
                <xs:element name="Value"
                  type="xs:hexBinary" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        - <xs:element name="Re_Marking">
          - <xs:complexType>
            - <xs:sequence>
                <xs:element name="ipv4TOS"
                  type="xs:hexBinary" />
                <xs:element name="ipv6TrafficClass"
                  type="xs:hexBinary" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element name="Shaping" type="xs:string" />
          <xs:element name="Blocking" type="xs:boolean" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="MaximumSizeOfObservationWindow"
      type="xs:duration" />
```

```
sKept"                              <xs:element name="NumberOfPacketsKept"
                                      type="xs:unsignedLong" />
ite"                                <xs:element name="BufferSizeForPa
                                      type="xs:unsignedLong" />
Supported">                       - <xs:element name="FilterLanguages
                                  - <xs:complexType>
                                    - <xs:sequence>
>                                     - <xs:element name="Language"
                                        - <xs:complexType>
                                          - <xs:sequence>
uageName"                                     <xs:element name="Lang
                                                type="xs:string" />
                                            </xs:sequence>
                                          </xs:complexType>
                                        </xs:element>
nguage">                              - <xs:element name="preferedL
                                        - <xs:complexType>
                                          - <xs:sequence>
uageName"                                     <xs:element name="Lang
                                                type="xs:string" />
                                            </xs:sequence>
                                          </xs:complexType>
                                        </xs:element>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        - <xs:element name="PacketTimeStampsForMonitorableFlow">
          - <xs:complexType>
            - <xs:sequence>
                <xs:element name="FirstTimeSeen"
                  type="xs:unsignedLong" />
                <xs:element name="LastTimeSeen"
                  type="xs:unsignedLong" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        - <xs:element name="Sampling">
          - <xs:complexType>
            - <xs:sequence>
                <xs:element name="SamplingRate"
                  type="xs:unsignedLong" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        - <xs:element name="ClassOfODMCapableSystem">
          - <xs:complexType>
            - <xs:sequence>
                <xs:element name="Identifier" type="xs:string" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  - <xs:element name="MonitorableFlowTypeB">
    - <xs:complexType>
      - <xs:sequence>
          <xs:element name="TrafficFilterExpression" type="xs:string" />
          <xs:element name="protocol" type="xs:string" />
          <xs:element name="layer" type="xs:unsignedShort" />
          <xs:element name="lowerLayerProtocol" type="xs:string" />
        - <xs:element name="FlowDirection">
```

```
- <xs:complexType>
  - <xs:sequence>
    - <xs:element name="PrevHopForThisFlow">
      - <xs:complexType>
        - <xs:sequence>
            <xs:element name="NetworkLayerAddr"
              type="xs:string" />
            <xs:element
              name="PhysicalAddressOfNeighborInterface"
              type="xs:string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    - <xs:element name="NextHopForThisFlow">
      - <xs:complexType>
        - <xs:sequence>
            <xs:element name="NetworkLayerAddr"
              type="xs:string" />
            <xs:element
              name="PhysicalAddressOfNeighborInterface"
              type="xs:string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="NetworkPrefixOfIngressInterface"
  type="xs:string" />
- <xs:element
  name="FlowConsumingHighestBwdthOnParticularInterface">
  - <xs:complexType>
    - <xs:sequence>
      - <xs:element name="interface">
        - <xs:complexType>
          - <xs:sequence>
              <xs:element
                name="NetworkPrefixOfTheMonitoringInterface"
                type="xs:string" />
              <xs:element
                name="PhysicalAddressOfIface"
                type="xs:string" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="MaximumSizeOfObservationWindow"
    type="xs:duration" />
  <xs:element name="NumberOfDataUnitsKept"
    type="xs:unsignedLong" />
  <xs:element name="BufferSizeForDataUnits"
    type="xs:unsignedLong" />
- <xs:element name="PacketTimeStampsForMonitorableFlow">
  - <xs:complexType>
    - <xs:sequence>
        <xs:element name="FirstTimeSeen"
          type="xs:unsignedLong" />
        <xs:element name="LastTimeSeen"
          type="xs:unsignedLong" />
      </xs:sequence>
    </xs:complexType>
```

```
            </xs:element>
          - <xs:element name="Sampling">
            - <xs:complexType>
              - <xs:sequence>
                  <xs:element name="SamplingRate"
                    type="xs:unsignedLong" />
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          - <xs:element name="ClassOfODMCapableSystem">
            - <xs:complexType>
              - <xs:sequence>
                  <xs:element name="Identifier" type="xs:string" />
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
- <xs:element name="FlowSinkerCapabilityModel">
  - <xs:complexType>
    - <xs:sequence>
      - <xs:element name="FlowToSink">
        - <xs:complexType>
          - <xs:sequence>
              <xs:element name="flowID" type="xs:unsignedLong" />
              <xs:element name="CanGenerateAndSink" type="xs:boolean" />
              <xs:element name="TrafficFilterExpression" type="xs:string" />
              <xs:element name="protocol" type="xs:string" />
              <xs:element name="layer" type="xs:unsignedShort" />
              <xs:element name="lowerLayerProtocol" type="xs:string" />
            - <xs:element name="source">
              - <xs:complexType>
                - <xs:sequence>
                    <xs:element name="srcAddr" type="xs:string" />
                    <xs:element name="srcPort"
                      type="xs:unsignedLong" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            - <xs:element name="destination">
              - <xs:complexType>
                - <xs:sequence>
                    <xs:element name="dstAddr" type="xs:string" />
                    <xs:element name="dstPort"
                      type="xs:unsignedLong" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element name="SourceNetwork" type="xs:string" />
              <xs:element name="DestinationNetwork" type="xs:string" />
            - <xs:element name="FlowDirection">
              - <xs:complexType>
                - <xs:sequence>
                  - <xs:element name="PrevHopForThisFlow">
                    - <xs:complexType>
                      - <xs:sequence>
                          <xs:element name="NetworkLayerAddr"
                            type="xs:string" />
```

```
                        <xs:element
                          name="PhysicalAddressOfNeighborInterface"
                          type="xs:string" />
                      </xs:sequence>
                    </xs:complexType>
                  </xs:element>
                - <xs:element name="NextHopForThisFlow">
                  - <xs:complexType>
                    - <xs:sequence>
                        <xs:element name="NetworkLayerAddr"
                          type="xs:string" />
                        <xs:element
                          name="PhysicalAddressOfNeighborInterface"
                          type="xs:string" />
                      </xs:sequence>
                    </xs:complexType>
                  </xs:element>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
            <xs:element name="NetworkPrefixOfIngressInterface"
              type="xs:string" />
            <xs:element name="MeasurementsPerformableOnTheFlow1"
              type="xs:string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
- <xs:element name="FlowGeneratorCapabilityModel">
  - <xs:complexType>
    - <xs:sequence>
      - <xs:element name="FlowToGenerate">
        - <xs:complexType>
          - <xs:sequence>
              <xs:element name="flowID" type="xs:unsignedLong" />
              <xs:element name="CanGenerateAndSink" type="xs:boolean" />
              <xs:element name="MaxNumOfFlows" type="xs:unsignedLong" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="TrafficFilterExpression" type="xs:string" />
        <xs:element name="protocol" type="xs:string" />
        <xs:element name="layer" type="xs:string" />
        <xs:element name="lowerLayerProtocol" type="xs:string" />
      - <xs:element name="source">
        - <xs:complexType>
          - <xs:sequence>
              <xs:element name="srcAddr" type="xs:string" />
              <xs:element name="srcPort" type="xs:unsignedLong" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      - <xs:element name="destination">
        - <xs:complexType>
          - <xs:sequence>
              <xs:element name="dstAddr" type="xs:string" />
              <xs:element name="dstPort" type="xs:unsignedLong" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
```

```
        <xs:element name="DestinationNetwork" type="xs:string" />
      - <xs:element name="NextHopForThisFlow">
        - <xs:complexType>
          - <xs:sequence>
              <xs:element name="NetworkLayerAddr" type="xs:string" />
              <xs:element name="PhysicalAddressOfNeighborInterface"
                type="xs:string" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="NetworkPrefixOfEgressInterface" type="xs:string" />
      - <xs:element name="FlowCharacteristics">
        - <xs:complexType>
          - <xs:sequence>
            - <xs:element name="TrafficShape">
              - <xs:complexType>
                - <xs:sequence>
                    <xs:element name="description" type="xs:string" />
                    <xs:element name="window"
                      type="xs:unsignedLong" />
                  - <xs:element name="Burst">
                    - <xs:complexType>
                      - <xs:sequence>
                          <xs:element name="parameterDescription"
                            type="xs:string" />
                        </xs:sequence>
                      </xs:complexType>
                    </xs:element>
                    <xs:element name="ConstantBitRate"
                      type="xs:unsignedLong" />
                    <xs:element name="Poison" type="xs:string" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
- <xs:element name="AdditionalCoreODMRequirementsSupported">
  - <xs:complexType>
    - <xs:sequence>
      - <xs:element name="EventNotificationProtocolsSupported">
        - <xs:complexType>
          - <xs:sequence>
              <xs:element name="EventNotificationProtocolName"
                type="xs:string" />
              <xs:element name="PreferedNotificationProtocol"
                type="xs:string" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      - <xs:element name="ODMDataModelsSupported">
        - <xs:complexType>
          - <xs:sequence>
              <xs:element name="OnDemandSNMPtypeOfMIBs"
                type="xs:boolean" />
              <xs:element name="PacketTraces" type="xs:boolean" />
              <xs:element name="FlowTraces" type="xs:boolean" />
            </xs:sequence>
          </xs:complexType>
```

```
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

13.2 The descriptions and the importance of the Tags of the CMDL language

<**ODMCapabilityModel**>
This is the root Tag that marks the beginning of the description of the Capability Model of an ODM-Capable Component. The ODM-Capable component creates (at boot-up time) and self-publishes the model to the network, by pushing it into the Capability Models Database of the network. The model can also be solicited for by automated tasks of the network. The *self-description* and *self-publishing* of capability models by a system/component should be subject to security policies.

<**KnownSessionBehavioursSpecificationsTree**>
This Tag marks the beginning of the description of the Monitoring-Behaviour-Specifications already known by the ODM-Capable Component i.e. the instances of Monitoring-Behaviour-Specifications stored in the **"The Repository for storing Monitoring-Behaviour-Specifications"** maintained by the component.

<**InstanceOfEDBSLangBasedMonitoringBehaviourSpecification**>
This Tag marks the beginning of the description of an instance of an EDBSLang based Monitoring-Behaviour-Specification.

<**TheUniquelyAssignedBehaviourIDofTheMonitorigBehaviourSpec**>
This Tag provides the description of the uniquely assigned **"behaviour-identifier (*Behaviour-Id*)"** for a particular instance of an **EDBSLang** based Monitoring-Behaviour-Specification.

<**TheSequentialPositionInTheSpecification**>
This Tag provides the description of the sequential position of the appearance of a particular <**EDBSLangTag**> in a particular instance of an **EDBSLang** based Monitoring-Behaviour-Specification. This is described using the Tag: <**number**>. The associated <**EDBSLangTag**> also has the following Tag elements associated with its description:

```
<TheAssignedIDofThisTagInstance> //The assigned Tag Identifier
<TagName> // The name of the <EDBSLangTag>
<AttributeValue> //For each attribute of the <EDBSLangTag>
<element> //Sub-Tags of the <EDBSLangTag>
      <name></name> // The name of the Sub-Tag (element)
      <value></value> // The value of the "element"
      <element> ….</element> //Any sub-elements that may exist
</element>
```

<**MonitorableFlows**>
This Tag marks the beginning of the descriptions of what we call "Monitoring Flows" that the ODM-Capable component can be requested to monitor via an **EDBSLang**

based Monitoring-Behaviour-Specification. A "Monitor-able Flow" is a kind of flow that a component can monitor, either by the way it is configured to monitor traffic AND/OR by virtue of its point of attachment to the network as part of the picture on "network observation (i.e. observability)".

<**MonitorableFlowTypeA**>
This Tag describes what we call **MonitorableFlowTypeA**. An ODM-Capable component describes the characteristics of a Flow it is **able to monitor** upon receiving an ODM-Request, either by virtue of its point of attachment to the network or its configuration and its configured policies. A number of "monitorable" flows of this type may be described by the ODM-Capable component. **MonitorableFlowTypeA** (as opposed to **MonitorableFlowTypeB**), is defined by the following characteristics or properties that the ODM-Capable component describes using special Tags:

<**TrafficFilterExpression**> //A traffic filter expression e.g. "IP proto UDP" may be used to describe the monitorable Flow.

<**protocol**> //A protocol name identifier e.g. "IP" may be used to describe the monitorable Flow. If a <**TrafficFilterExpression**> or any other type of <"***TAG***"> has been included in the description and is meant to mean a CONSTRAINT to the Monitorable Flow, i.e. not meaning an "alternate case", it must be ensured that there are no ambiguities with the inclusion of a <**protocol**> description.

<**layer**> // In addition, a layer view e.g. "layer 3" of the <**protocol**> may be added to the description by the ODM-Capable component. If a <**TrafficFilterExpression**> or any other type of <"***TAG***"> has been included in the description and is meant to mean a CONSTRAINT to the Monitorable Flow, i.e. not meaning an "alternate case", it must be ensured that there are no ambiguities resulting from the inclusion of a <**layer**> description.

<**lowerLayerProtocol**> //In addition, a lower layer protocol e.g. "Ethernet" of the <**protocol**> may be added to the description by the ODM-Capable component. If a <**TrafficFilterExpression**> or any other type of <"***TAG***"> has been included in the description and is meant to mean a CONSTRAINT to the Monitorable Flow, i.e. not meaning an "alternate case", it must be ensured that there are no ambiguities resulting from the inclusion of a <**lowerLayerProtocol**> description.

<**source**> // The Monitorable Flow may be described using the description of the source node address information for such a Flow as indicated via *source address* and *source port* pairs:<**srcAddr**>:<**srcPort**>. If a <**TrafficFilterExpression**> or any other type of <"***TAG***"> has been included in the description and is meant to mean a CONSTRAINT to the Monitorable Flow, i.e. not meaning an "alternate case", it must be ensured that there are no ambiguities resulting from the inclusion of a <**source**> description. "**Wildcards**" can also be supported, and the use of "**Ranges**" or special characters like "**0**" in the descriptions can also be supported where appropriate, to mean special cases. For example "**0**" could mean "**Any**".

<**destination**> // The Monitorable Flow may be described using the description of the destination node address information for such a Flow as indicated via *destination address* and *destination port* pairs: <**dstAddr**>:<**dstPort**>. If a <**TrafficFilterExpression**> or any other type of <"***TAG***"> has been included in the description and is meant to mean a CONSTRAINT to the Monitorable Flow, i.e. not meaning an "alternate case", it must be ensured that there are no ambiguities resulting from the inclusion of a <**destination**> description. "**Wildcards**" can also

be supported, and the use of **"Ranges"** or special characters like **"0"** in the descriptions can also be supported where appropriate, to mean special cases. For example **"0"** could mean **"Any"**.

<**SourceNetwork**> // The Monitorable Flow may be described using the description of the source network information for such a Flow. If a <**TrafficFilterExpression**> or any other type of <***"TAG"***> has been included in the description and is meant to mean a CONSTRAINT to the Monitorable Flow, i.e. not meaning an "alternate case", it must be ensured that there are no ambiguities resulting from the inclusion of a <**SourceNetwork**> description.

<**DestinationNetwork**> // The Monitorable Flow may be described using the description of the destination network information for such a Flow. If a <**TrafficFilterExpression**> or any other type of <***"TAG"***> has been included in the description and is meant to mean a CONSTRAINT to the Monitorable Flow, i.e. not meaning an "alternate case", it must be ensured that there are no ambiguities with resulting from the inclusion of a <**DestinationNetwork**> description.

<**FlowDirection**> // The Monitorable Flow may be described using the description of the "flow direction" information for such a Flow. If a <**TrafficFilterExpression**> or any other type of <***"TAG"***> has been included in the description and is meant to mean a CONSTRAINT to the Monitorable Flow, i.e. not meaning an "alternate case", it must be ensured that there are no ambiguities resulting from the inclusion of a <**FlowDirection**> description. The <**FlowDirection**> should be described using the following information sub-Tags: A Tag describing the previous hop of the Flow and/or a Tag describing the next hop for the Flow, according to the ODM-Capable component or the system embedding the component. The "flow direction" information may be included in the Monitorable Flow description by the following Tags:

<**PrevHopForThisFlow**> //Previous hop info.
<**NetworkLayerAddr**></**NetworkLayerAddr**>
<**PhysicalAddressOfNeighborInterface**>
</**PhysicalAddressOfNeighborInterface**>
</**PrevHopForThisFlow**>
<**NextHopForThisFlow**> //Next hop info.
<**NetworkLayerAddr**></**NetworkLayerAddr**>
<**PhysicalAddressOfNeighborInterface**>
</**PhysicalAddressOfNeighborInterface**>
</**NextHopForThisFlow**>

<**NetworkPrefixOfIngressInterface**> //The network prefix information of the interface that is considered as the ingress interface of the Monitorable Flow, by the system embedding the ODM-Capable component.

<**NetworkPrefixOfTheMonitoringInterface**> //The network prefix information of the interface that is considered as the monitoring interface for the Monitorable Flow, by the system embedding the ODM-Capable component. This is the same as the <**NetworkPrefixOfIngressInterface**>, but possibly there may be cases where the interfaces associated with the two prefixes may not be one and the same interface.

<**PerformableActions**> // This Tag marks the beginning of the descriptions of the "Actions" that the ODM-Capable component can perform on the Monitorable Flow (subject to requests to do so, as may be indicated in an **EDBSLang** based Monitoring-Behaviour-Specification). Performable actions are especially useful for cases where an automated task such as an automated troubleshooter requires that an injected diagnostic flow get modified or acted upon at certain points in the network in order to observe the reaction of the network or certain entities of the network by the way the diagnostic flow is then treated threafter.

<**BitModification**> // The kind of modification that can be performed on a particular Bit of the **DataUnits** of the Monitorable Flow, whereby the <**TheDataUnit**> can be a *packet*, a *cell* (like an ATM-cell in ATM networks), a *frame*, or datagram, or *transport layer units*. The sub-Tags of the <**BitModification**> Tag are as follows: <**TheDataUnit**> describing the DataUnit being referred to, and being the unit of the Monitorable Flow; the <**BitNumber**>; and the <**BitValue**>. The <**BitValue**> may be specified as "Any" so that an automated task intending to request for the modification to be performed can specify the "value" to be used for the modification by the ODM-Capable component. The <**BitValue**> may be specified with a "concrete value" that the ODM-Capable component is configured to use for the modifications.

<**ByteModification**> // The flow modification may be on the basis of Byte modification instead, or in combination with Bit modification. The <**ByteModification**> Tag also includes a <**TheDataUnit**> describing the DataUnit being referred to, and being the unit of the Monitorable Flow; the <**ByteNumber**>; and the <**ByteValue**>. The <**ByteValue**> may be specified as "Any" so that an automated task intending to request for the modification to be performed can specify the "value" to be used for the modification by the ODM-Capable component. The <**ByteValue**> may be specified with a "concrete value" that the ODM-Capable component is configured to use for the modifications. The use of both the <**BitModification**> and the <**ByteModification**> should be subject to avoiding ambiguities.

<**ProtoFieldModification**> // The flow modification may be on the basis of modification of a particular protocol field of the Monitorable Flow instead, or in combination with Bit modification and Byte modifications. The use of <**ProtoFieldModification**>, the <**BitModification**> and the <**ByteModification**>, together, should be subject to avoiding ambiguities. The <**ProtoFieldModification**> is described using the following sub-Tags:

<**Protocol**> //The protocol, of which is part of the description of the Monitorable Flow.

<**FieldName**> // The "Field" name of the protocol's message(s) e.g. the "Traffic Class" field of an IPv6 packet.

<**Value**> // The <**Value**> may be specified with a "concrete value" that the ODM-Capable component is configured to use for the modifications.

<**Re_Marking**> // Another "performable action" that the ODM-Capable component can perform (subject to a request expressed through the EDBSLang monitoring-behaviour-specification), is "Re_Marking" of packets. This is especially useful for automated tasks such as an automated troubleshooter that may require that an injected diagnostic flow (i.e. diagnostic packets) get "re_marked" at certain points in the network in order to observe the reaction of the network or certain entities of the network by the way the diagnostic flow is then treated thereafter. The following are examples of the kind of re_marking that can be

performed on a Monitorable Flow: <**ipv4TOS**> and <**ipv6TrafficClass**>. Ambiguities resulting from the use of the <**BitModification**>, <**ByteModification**> or the <**ProtoFieldModification**> should be avoided.

<**Shaping**> // The ODM-Capable may also specify the "traffic shaping characteristics" that it can introduce into the Monitorable Flow (upon request of-course, as described earlier).

<**Blocking**> // The ODM-Capable may also indicate that it can "block" the Monitorable Flow (upon request of-course, as described earlier), from going further into the network i.e. it can block the packets of the flow from being forwarded further to their tentative next hop. Both the <**Shaping**> and <**Blocking**> capabilities are especially useful for automated tasks such as an automated troubleshooter that may require that an injected diagnostic flow (i.e. diagnostic packets) get "re_shaped" or "blocked" at certain points in the network in order to observe the reaction of the network or certain entities of the network by the way the diagnostic flow is then treated thereafter.

Apart from the performable <**PerformableActions**>, the following items may also need to be described by the ODM-Capable component.

<**MaximumSizeOfObservationWindow**> // The maximum time for which the ODM-Capable component can commit resources required for monitoring the Monitorable Flow and performing actions it claims is able to perform on the flow (should those actions be requested).

<**NumberOfDataUnitsKept**> //The number of Data-Units of the Monitorable Flow e.g. packets, cells, frames, datagrams, etc, which can be stored by the ODM-Capable component, subject to resource availability. The Data Units can potentially be requested for by an entity (an automated task).

<**BufferSizeForDataUnits**> //The buffer size allocated for storing the <**NumberOfDataUnitsKept**>.

<**FilterLanguagesSupported**> // The Traffic Filtering languages supported by the ODM-Capable component. Examples are *BPF filters*, *pcap type of filters*, *windmill filters* [Watson03], etc. By describing the languages supported, an automated task in the network can use the knowledge of the languages supported by a ODM target, and compose the appropriate syntax of the **ODM-Traffic Filter** that is required in the **EDBSLang** based monitoring-behaviour-specification to be executed on a target component. The languages can be listed using the following sub-Tags:

<**Language**> //An instance of a Traffic Filtering language
<**LanguageName**></**LanguageName**>
</**Language**>

<**preferedLanguage**>//The preferred traffic filtering language to be used by automated tasks in composing the **ODM-Traffic Filter** in monitoring requests. This is indicated by the associated <**LanguageName**> of the <**preferedLanguage**>.

<**PacketTimeStampsForMonitorableFlow**> //This is used for indicating the kind of packet timestamps that can be stored

(remembered) by the ODM-Capable component. The timestamps can then be accessed by an automated task (if interested).

<**FirstTimeSeen**> //This is used for indicating that the timestamp of the first captured packet of the Flow can be stored.

<**LastTimeSeen**> //This is used for indicating that the timestamp of the last captured packet of the Flow can be stored.

<**Sampling**> //This is used for indicating that Flow Sampling can be performed. The associated <**SamplingRate**> is used for indicating the Sampling Rate or Sampling Rates supported.

<**ClassOfODMCapableSystem**> // The ODM-Capable component must describe the class of the functional role being played by the system hosting the component, relative to the Monitorable Flow being described. Examples of the class of the functional role of the system hosting the ODM-Capable component are: a **Host, Router, Switch, Gateway, Monitoring Probe**. This knowledge may be required by automated tasks of the network, which select systems and request for the execution of a monitoring-behaviour-specification. The class is indicated via the <**Identifier**> Tag, which is meant to specify a textual identifier of the class of the system e.g. **"Router"**.

-------- End of <**MonitorableFlowTypeA**> Definition ------------

The previous Tags given after the <**MonitorableFlowTypeA**> are meant for describing the <**MonitorableFlowTypeA**>, which differs slightly from <**MonitorableFlowTypeB**> given below but both have some elements which are common to both flow types.

<**MonitorableFlowTypeB**> This Tag describes what we call **MonitorableFlowTypeB**. An ODM-Capable component describes the characteristics of a Flow it is **able to monitor** upon receiving an ODM-Request, either by virtue of its point of attachment to the network or its configuration and configured policies. A number of "monitorable" flows of this type may be described by the ODM-Capable component. The main difference between **MonitorableFlowTypeB** and **MonitorableFlowTypeA** is the inclusion of the Tag: <**FlowConsumingHighestBwdthOnParticularInterface**> in **MonitorableFlowTypeB**, which may be specified without any of the other Tags that are common to both types of monitorable flows. Also, **MonitorableFlowTypeB** does not include <**PerformableActions**>. **MonitorableFlowTypeB** is defined by the following characteristics or properties that the ODM-Capable component describes using special Tags:

<**TrafficFilterExpression**> // This Tag carries the same meaning and usage context as described in the case for **MonitorableFlowTypeA.**

<**protocol**> // This Tag carries the same meaning and usage context as described in the case for **MonitorableFlowTypeA.**

<**layer**>// This Tag carries the same meaning and usage context as described in the case for **MonitorableFlowTypeA.**

<**lowerLayerProtocol**>// This Tag carries the same meaning and usage context as described in the case for **MonitorableFlowTypeA.**

<FlowDirection>// This Tag carries the same meaning and usage context as described in the case for **MonitorableFlowTypeA.**

<PrevHopForThisFlow>// This Tag carries the same meaning and usage context as described in the case for **MonitorableFlowTypeA.**

<NetworkLayerAddr>// This Tag carries the same meaning and usage context as described in the case for **MonitorableFlowTypeA.**

<PhysicalAddressOfNeighborInterface>// This Tag carries the same meaning and usage context as described in the case for **MonitorableFlowTypeA.**

<NextHopForThisFlow> // This Tag carries the same meaning and usage context as described in the case for **MonitorableFlowTypeA.**

<NetworkLayerAddr>// This Tag carries the same meaning and usage context as described in the case for **MonitorableFlowTypeA.**

<PhysicalAddressOfNeighborInterface>// This Tag carries the same meaning and usage context as described in the case for **MonitorableFlowTypeA.**

<NetworkPrefixOfIngressInterface>// This Tag carries the same meaning and usage context as described in the case for **MonitorableFlowTypeA.**

<FlowConsumingHighestBwdthOnParticularInterface> //This Tag may be used by an ODM-Capable component to simply indicate that the component can be requested to monitor all flows on the system and create information about the Flow(s) considered as the flow(s) consuming the highest bandwidth share on a particular interface of the system. An ODM-Capable component may have only this capability and not other monitoring capabilities. When used together with other Tags relevant for describing **MonitorableFlowTypeB**, such as:
<**TrafficFilterExpression**>, <**protocol**>, <**FlowDirection**>, etc, it means the <**FlowConsumingHighestBwdthOnParticularInterface**> becomes "constrained" by those Tags, meaning that the constraint may be narrowed to a concrete <**protocol**> (for example).

<interface> //Used to indicate the interface on which <**FlowConsumingHighestBwdthOnParticularInterface**> can be determined.

<NetworkPrefixOfTheMonitoringInterface> // This Tag carries the same meaning and usage context as described in the case for **MonitorableFlowTypeA.**

<PhysicalAddressOfIface> //This is used to indicate the physical address of <**interface**>.

<MaximumSizeOfObservationWindow> // This Tag carries the same meaning and usage context as described in the case for **MonitorableFlowTypeA.**

<NumberOfDataUnitsKept>// This Tag carries the same meaning and usage context as described in the case for **MonitorableFlowTypeA.**

<**BufferSizeForDataUnits**>// This Tag carries the same meaning and usage context as described in the case for **MonitorableFlowTypeA.**

<**PacketTimeStampsForMonitorableFlow**> // This Tag and its associated <**FirstTimeSeen**> and **LastTimeSeen**> Tags carry the same meaning and usage context as described in the case for **MonitorableFlowTypeA.**

<**Sampling**> // This Tag and its associated <**SamplingRate**> carry the same meaning and usage context as described in the case for **MonitorableFlowTypeA.**

<**ClassOfODMCapableSystem**> // This Tag and its associated <**Identifier**> carry the same meaning and usage context as described in the case for **MonitorableFlowTypeA.**

-------- End of <**MonitorableFlowTypeB**> Definition ------------

-------- End of <**MonitorableFlows**> Definition ------------

An ODM-Capable component may have the capability of sinking or dumping (on request) certain types of flows e.g. diagnostic flows injected by some automated task in the network, such that the flow's packets are not forwarded further via the egress interface. Such capability should be described using a Flow*(s)-Sinker Capability Model* of a traffic *flow(s) sinker* (i.e. sink/dump on request), which should include such information as: *flow properties e.g. source and destination address information, previous and next-hop for this flow, traffic measurements that can be performed on the flow*, etc.

<**FlowSinkerCapabilityModel**> // This Tag marks the beginning of the descriptions of the "Flow sinking capabilities" that the ODM-Capable component may have.

<**FlowToSink**> // This Tag describes the kind of Flow the component can "sink or dump" on request. The EDBSLang language needs to be extended to support such kind of requests. One possibility of enabling some limited way of enabling automated tasks to request an ODM-Capable component to "sink or dump" a flow using the current version of **EDBSLang** is to exploit the **"Own_Action"** Tag of the **EDBSLang.**

<**flowID**> //The ODM-Capable component assigns an "identifier" that enables an automated task to indicate the Flow the component should "sink or dump".

<**CanGenerateAndSink**> >// The ODM-Capable component uses this Tag to indicate whether it can "generate" such a Flow to some destination while at the same time being able to "sink or dump" the Flow. This means that the component can be requested to generate a test or diagnostic Flow of certain Flow properties, and can be requested to "sink or dump" a Flow generated from somewhere else, whose properties are similar to the properties of the Flow it is claiming to be able to generate, but with some parameters such as <**srcAddr**>:<**srcPort**> pairs and <**dstAddr**>:<**dstPort**> pairs reversed ("source" becoming "destination" and "destination" becoming "source"). The Flow properties can be described using any or all of the Tags below (subject to avoiding ambiguities).

<**TrafficFilterExpression**>//A traffic filter expression e.g. "IP proto UDP" may be used to describe a kind of Flow that the component can "sink or dump" on-request.

<**protocol**>//A protocol name identifier e.g. "IP" may be used to describe a kind of Flow that the component can "sink or dump" on-request. If a <**TrafficFilterExpression**> or any other type of <"***TAG***"> has been included in the description and is meant to mean a CONSTRAINT to the Flow, i.e. not meaning an "alternate case", it must be ensured that there are no ambiguities with including the inclusion of a <**protocol**> description.

<**layer**>// In addition, a layer view e.g. "layer 3" of the <**protocol**> may be added to the description by the ODM-Capable component. If a <**TrafficFilterExpression**> or any other type of <"***TAG***"> has been included in the description and is meant to mean a CONSTRAINT to the Flow, i.e. not meaning an "alternate case", it must be ensured that there are no ambiguities resulting from the inclusion of a <**layer**> description.

<**lowerLayerProtocol**>//In addition, a lower layer protocol e.g. "Ethernet" of the <**protocol**> may be added to the description by the ODM-Capable component. If a <**TrafficFilterExpression**> or any other type of <"***TAG***"> has been included in the description and is meant to mean a CONSTRAINT to the Flow, i.e. not meaning an "alternate case", it must be ensured that there are no ambiguities resulting from the inclusion of a <**lowerLayerProtocol**> description.

<**source**>// The kind of Flow that the component can "sink or dump" on-request may be described using the description of the source node address information for such a Flow as indicated via *source address* and *source port* pairs:<**srcAddr**>:<**srcPort**>. If a <**TrafficFilterExpression**> or any other type of <"***TAG***"> has been included in the description and is meant to mean a CONSTRAINT to the Flow, i.e. not meaning an "alternate case", it must be ensured that there are no ambiguities resulting from the inclusion of a <**source**> description. "**Wildcards**" can also be supported, and the use of "**Ranges**" or special characters like "**0**" in the descriptions can also be supported where appropriate, to mean special cases. For example "**0**" could mean "**Any**".

<**destination**>// The kind of Flow that the component can "sink or dump" on-request may be described using the description of the destination node address information for such a Flow as indicated via *destination address* and *destination port* pairs: <**dstAddr**>:<**dstPort**>. If a <**TrafficFilterExpression**> or any other type of <"***TAG***"> has been included in the description and is meant to mean a CONSTRAINT to the Flow, i.e. not meaning an "alternate case", it must be ensured that there are no ambiguities resulting from the inclusion of a <**destination**> description. "**Wildcards**" can also be supported, and the use of "**Ranges**" or special characters like "**0**" in the descriptions can also be supported where appropriate, to mean special cases. For example "**0**" could mean "**Any**".

<**SourceNetwork**>// The kind of Flow that the component can "sink or dump" on-request may be described using the description of the source network information for such a Flow (in addition to other necessary Flow description information). If a <**TrafficFilterExpression**> or any other type of <"***TAG***"> has been included in the description and is meant to mean a CONSTRAINT to the Flow, i.e.

not meaning an “alternate case”, it must be ensured that there are no ambiguities resulting from the inclusion of a <**SourceNetwork**> description.

<**FlowDirection**>// The kind of Flow that the component can “sink or dump” on-request may be described using the description of the “flow direction” information for such a Flow. If a <**TrafficFilterExpression**> or any other type of <***”TAG”***> has been included in the description and is meant to mean a CONSTRAINT to the Flow, i.e. not meaning an “alternate case”, it must be ensured that there are no ambiguities resulting from the inclusion of a <**FlowDirection**> description. The <**FlowDirection**> should be described using the following information sub-Tags: A Tag describing the previous hop of the Flow and/or a Tag describing the next hop for the Flow, according to the ODM-Capable component or a system embedding the component. The “flow direction” information may be included in the Flow description by the following Tags:

<**PrevHopForThisFlow**> //Previous hop info.
<**NetworkLayerAddr**></**NetworkLayerAddr**>
<**PhysicalAddressOfNeighborInterface**>
</**PhysicalAddressOfNeighborInterface**>
</**PrevHopForThisFlow**>
<**NextHopForThisFlow**> //Next hop info.
<**NetworkLayerAddr**></**NetworkLayerAddr**>
<**PhysicalAddressOfNeighborInterface**>
</**PhysicalAddressOfNeighborInterface**>
</**NextHopForThisFlow**>

<**NetworkPrefixOfIngressInterface**> //The network prefix information of the interface that is considered as the ingress interface of the <**FlowToSink**> by the system embedding the ODM-Capable component.

<**NetworkPrefixOfTheMonitoringInterface**> //The network prefix information of the interface that is considered as the monitoring interface for the <**FlowToSink**>, by the system embedding the ODM-Capable component. This is the same as the <**NetworkPrefixOfIngressInterface**>, but possibly there may be cases where the interfaces associated with the two prefixes may not be one and the same interface.

<**MeasurementsPerformableOnTheFlow**> // The ODM-Capable component describes the kind of measurements that it can perform on the <**FlowToSink**>, such as the metrics or statistics known in **traffic measurements** e.g. packet arrival rate, packet delay, throughput, etc. The measurements can then be made available for the requesting automated task to access.

-------- End of <**FlowToSink**> Definition ------------

-------- End of <**FlowSinkerCapabilityModel**> Definition ------------

An ODM-Capable component may have the capability of generating and/or sinking of a diagnostic/test flow(s) of specific properties and/or possibly the participation of the system in active measurements of traffic characteristics. The *Flow(s) Generator Capability Model* of a traffic flow(s) generator should include such

information as: *flow properties e.g. source and destination address information of a diagnostic/test flow that can be generated on request, next-hop for this flow, traffic characteristics e.g. bit-rate, shape etc, traffic measurements that can be performed on the flow, etc.*

<**FlowGeneratorCapabilityModel**> // This Tag marks the beginning of the descriptions of the "Flow Generation capabilities" that the ODM-Capable component may have.

<**FlowToGenerate**>// This Tag describes the kind of Flow the component can "generate" on request. The EDBSLang language needs to be extended to support such kind of requests. One possibility of enabling some limited way of enabling automated tasks to request an ODM-Capable component to "generate" a flow using the current version of **EDBSLang** is to exploit the "**Own_Action**" Tag of the **EDBSLang.**

<**flowID**>//The ODM-Capable component assigns an "identifier" that enables an automated task to indicate the Flow the component should "generate".

<**CanGenerateAndSink**>// The ODM-Capable component uses this Tag to indicate whether it can "generate" such a Flow to some destination while at the same time being able to "sink or dump" the Flow. This means that the component can be requested to generate a test or diagnostic Flow of certain Flow properties, and can be requested to "sink or dump" a Flow generated from somewhere else, whose properties are similar to the properties of the Flow it is claiming to be able to generate, but with some parameters such as <**srcAddr**>:<**srcPort**> pairs and <**dstAddr**>:<**dstPort**> pairs reversed ("source" becoming "destination" and "destination" becoming "source"). The Flow properties can be described using any or all of the Tags below (subject to avoiding ambiguities).

<**MaxNumOfFlows**> //The Tag is used for indicating the maximum number of Flows that can be generated, subject to the availability of the resources that can be committed for this task with guarantees of precision in the generated flows.

<**TrafficFilterExpression**>//A traffic filter expression e.g. "IP proto UDP" may be used to describe a kind of Flow that the component can "generate" on-request.

<**protocol**>//A protocol name identifier e.g. "IP" may be used to describe a kind of Flow that the component can "generate" on-request. If a <**TrafficFilterExpression**> or any other type of <***"TAG"***> has been included in the description and is meant to mean a CONSTRAINT to the Flow, i.e. not meaning an "alternate case", it must be ensured that there are no ambiguities with including the inclusion of a <**protocol**> description.

<**layer**>// In addition, a layer view e.g. "layer 3" of the <**protocol**> may be added to the description by the ODM-Capable component. If a <**TrafficFilterExpression**> or any other type of <***"TAG"***> has been included in the description and is meant to mean a CONSTRAINT to the Flow, i.e. not meaning an "alternate case", it must be ensured that there are no ambiguities resulting from the inclusion of a <**layer**> description.

<**lowerLayerProtocol**>//In addition, a lower layer protocol e.g. "Ethernet" of the <**protocol**> may be added to the description by the ODM-Capable component. If a <**TrafficFilterExpression**> or any other type of <"***TAG***"> has been included in the description and is meant to mean a CONSTRAINT to the Flow, i.e. not meaning an "alternate case", it must be ensured that there are no ambiguities resulting from the inclusion of a <**lowerLayerProtocol**> description.

<**source**>// The kind of Flow that the component can "generate" on-request may be described using the description of the source node address information for such a Flow as indicated via *source address* and *source port* pairs:<**srcAddr**>:<**srcPort**>. If a <**TrafficFilterExpression**> or any other type of <"***TAG***"> has been included in the description and is meant to mean a CONSTRAINT to the Flow, i.e. not meaning an "alternate case", it must be ensured that there are no ambiguities resulting from the inclusion of a <**source**> description. "**Wildcards**" can also be supported, and the use of "**Ranges**" or special characters like "**0**" in the descriptions can also be supported where appropriate, to mean special cases. For example "**0**" could mean "**Any**".

<**destination**>// The kind of Flow that the component can "generate" on-request may be described using the description of the destination node address information for such a Flow as indicated via *destination address* and *destination port* pairs: <**dstAddr**>:<**dstPort**>. If a <**TrafficFilterExpression**> or any other type of <"***TAG***"> has been included in the description and is meant to mean a CONSTRAINT to the Flow, i.e. not meaning an "alternate case", it must be ensured that there are no ambiguities resulting from the inclusion of a <**destination**> description. "**Wildcards**" can also be supported, and the use of "**Ranges**" or special characters like "**0**" in the descriptions can also be supported where appropriate, to mean special cases. For example "**0**" could mean "**Any**".

<**DestinationNetwork**>// The kind of Flow that the component can "generate" on-request may be described using the description of the source network information for such a Flow (in addition to other necessary Flow description information). If a <**TrafficFilterExpression**> or any other type of <"***TAG***"> has been included in the description and is meant to mean a CONSTRAINT to the Flow, i.e. not meaning an "alternate case", it must be ensured that there are no ambiguities resulting from the inclusion of a <**DestinationNetwork**> description.

<**NextHopForThisFlow**> // The next hop for the Flow, according to the ODM-Capable component or a system embedding the component. The description of the next hop should be made out of the following addresses: <**NetworkLayerAddr**> and <**PhysicalAddressOfNeighborInterface**>.

<**NetworkPrefixOfEgressInterface**> //The network prefix information of the interface that is considered as the egress interface of the <**FlowToGenerate**> by the system embedding the ODM-Capable component.

<**FlowCharacteristics**> // This Tag is required for describing the Flow characteristics for the <**FlowToGenerate**>, such as the following characteristics:

<**TrafficShape**> //This begins the description of the traffic shape of the Flow that can be generated. NOTE: Other types of characteristics apart from the ones described below need to be added to the CMDL meta-language.

<description> // This Tag is required for supplying more information about the Traffic Shape of the flow that can be generated.
<window> //This Tag is required for describing the window size (duration) for which the Flow can be generated.

<Burst> //This Tag is required for describing the "Burst" characteristics of the Flow that can be generated.

<parameterDescription> //Useful for the description of the parameter(s) of the burst.

<ConstantBitRate> //Useful for the description of the parameters describing the "constant bit rate nature" of the Flow (if the Flow is meant to be of this nature).

<Poison> //Useful for the description of the Poison parameters of the Flow (if the Flow is meant to be of this nature).

-------- End of <FlowCharacteristics> Definition ------------

-------- End of <FlowToGenerate> Definition ------------

-------- End of <FlowGeneratorCapabilityModel> Definition ------------

<AdditionalCoreODMRequirementsSupported> //This Tag marks the beginning of the description of the additional core "requirements for On-Demand Monitoring" supported by the ODM-Capable component. These are the kind of requirements that enable automated tasks to know the core "features" supported by the component, in order to either know what to request for monitoring via an **EDBSLang** based Monitoring-Behaviour-Specification OR how to configure themselves.

<EventNotificationProtocolsSupported> // This Tag is useful for describing the Event Notification Protocols supported by the ODM-Capable component. It means the ODM-Capable component can employ any of the protocols to convey event notifications requested for in an **EDBSLang** based Monitoring-Behaviour-Specification. The protocols supported should be listed using the <EventNotificationProtocolName>. The ODM-Capable component should also include the <PreferedNotificationProtocol>.

<ODMDataModelsSupported> //This is used for indicating the kinds of On-Demand Data Models that the component supports. Examples are **On-Demand SNMP MIBs.**

<OnDemandSNMPtypeOfMIBs> // This is used for indicating that the ODM-Capable component supports **On-Demand SNMP MIBs** as On-Demand Data Models.

<PacketTraces> // This is used for indicating that the ODM-Capable component supports the creation and dissemination of **Packet Traces.**
<FlowTraces> This is used for indicating that the ODM-Capable component supports the creation and dissemination of **Flow Traces.**

-------- End of <ODMDataModelsSupported> Definition ------------

-------- End of <AdditionalODMRequirementsSupported> Definition ----------

-------- End of <**ODMCapabilityModel**> Definition ------------