Advanced architecture for INTER-domain quality of service MONitoring, modelling and visualisation

INTERMON-IST-2001-34123

Integration of the Inter-Domain Modelling and Simulation Toolkit

(Deliverable 11)

Deliverable

Work-package No. / Title	WP5 – Modelling and Simulation
Planned Issueing Date	30-06-30
Distribution	WP5 Partners
Document Identifier	im-wp5-v100-D11-pf
File name	im-wp5-v100-D11-pf.doc
Version	V 1.00
Editor/Author	**University of Bern**
Contact Person(s)	Matthias Scheidegger <mscheid@iam.unibe.ch >
Authors	Maurizio Bartoli (TILAB) Florian Baumgartner (UniBe) Christof Brandauer (SAR) Roberto Canonico (CINI) Pedro A. Aranda Gutierrez (TID) Tamas Mahr (BUTE) Maurizio Molina (NEC) Fabrizio Orlandi (TILAB) Matthias Scheidegger (UniBe) Carsten Schmoll (FHG) Joern Seger (UniDo)

Change History

V0.01	Empty document with TOC
V0.02	Partial integration of contributions and slightly revised TOC
V0.03	Included virtually all contributions. Back to partners for revision.
V0.04	Updated references and author list. Additional FHG contributions. Some graphics fixed. BUTE's final fixes included. TILAB's changes merged. Moved basic xml model structure to own section (proposal by TILAB). SAR v2 merged. Included a section about integration levels and status.
V0.10	Renamed and restructured parts of Chapters 1 and 2. Did the layout for text and XML / source code.
V1.00	Christof's corrections included. Fixed page headers. Renamed adapter and tool manager to simulation manager (Paul's suggestion). Carsten's fixes added. Included Ilka's picture and fixed layout after that.

Table of Contents

EXECUTIVE SUMMARY 6

1 TOOLKIT STRUCTURE 7

1.1 INTRODUCTION 7
1.2 ROLES AND STATUS OF THE SIMULATION APPROACHES 8
1.2.1 Hybrid Simulation 8
1.2.2 Time series simulator 8
1.2.3 RTC-FSIM 8
1.2.4 Inter-IP 9
1.3 RELATIONS TO THE GENERAL ARCHITECTURE 9

2 COMMUNICATION 11

2.1 INTERACTION WITH THE GENERAL ARCHITECTURE 11
2.1.1 Starting a simulation process 11
2.1.2 Querying a simulation process 12
2.1.3 Simulations Results Handling 12
2.1.4 Visualisation of Results 12
2.2 INTERFACE TO THE VISUAL DATA MINING MODULE 13
2.3 USING THE BGP-4 TOPOLOGY DESCRIPTION IN SIMULATION PROCESSES 14
2.3.1 View of the Internet delivered by the BGP-4 tools 14
2.3.2 The End to End QoS scenario 14
2.3.3 Static use of BGP4 data 15
2.3.4 Dynamic use of BGP4 data 15

3 TOOLKIT IMPLEMENTATION 16

3.1 HYBRID SIMULATOR 16
3.1.1 General Integration 16
3.1.2 Model Implementation 17
3.1.3 NS-2 Plug-In Interface 21
3.2 TIME SERIES SIMULATOR 24
3.2.1 The GUI module 24
3.2.2 The GC component 25
3.2.3 Simulation manager and simulator 25
3.2.4 Simulator Architecture 25
3.3 FLUID SIMULATOR 27
3.3.1 Introduction 27
3.3.2 Integration of IPFIX Measurements 27
3.3.3 RTC-FSIM Implementation 30
3.4 INTER-IP 30
3.4.1 Interd 31
3.4.2 Intrad 31
3.4.3 INTER-IP Integration 32
3.5 GENERIC SIMULATION CONTROL MODULE FOR THE CLIENT GUI 32

4 DATA FORMATS AND APIS 34

4.1 SIMULATOR-INTERNAL FORMATS 34
4.1.1 Formats and APIs used in Hybrid Simulation 34
4.1.2 Formats used in Time Series Simulation 42
4.1.3 Formats used in INTER-IP 45
4.2 INTRA-TOOLKIT FORMATS AND APIS 47
4.2.1 Basic XML Format for Models 47
4.2.2 General Time Series 48
4.2.3 Question 48
4.3 EXTRA-TOOLKIT FORMATS AND APIS 49
4.3.1 VDM Interface Formats and APIs 49
4.3.2 BGP-4 Topology Description 53

5 APPENDIX 56

5.1 IMPORT FILTER EXAMPLE SOURCE CODE 56
5.2 REFERENCES 59

List of Figures

Figure 1.1 - The two stages of integration 7
Figure 1.2 – Architecture view of simulation toolkit integration 10
Figure 2.1 - Starting a simulation 11
Figure 2.2 - Passing simulation results 12
Figure 2.3: Visual Data Mining Module 13
Figure 2.4 - View of the Internet as provided by the BGP tools 14
Figure 3.1 - Structure of the interface between NS-2 and the MDModel plug-in 17
Figure 3.2 - Multi-domain model topology structure 18
Figure 3.3 - Multi-domain model list structure 19
Figure 3.4 - Mapping of ns-2 and plug-in internal Ids 20
Figure 3.5 – Structure of the plug-in mechanism 22
Figure 3.6 - Time series simulator integration 24
Figure 3.7 - ON/OFF states of 10 flows 28
Figure 3.8 - Rates of flow #1 28
Figure 3.9 - Superposition of the 10 ON/OFF flows 29
Figure 3.10 – Inter-IP integration 32
Figure 3.11 - Simulation control GUI 33
Figure 4.1 - Example multi-domain configuration 35

List of Tables

Table 4.1 – Intra-domain entities 45
Table 4.2 – Inter-domain entities 46

Executive Summary

This document describes the implementation and integration work done by the InterMON work package 5 partners based on the specifications from Deliverable 6, “Modelling and Simulation Toolkit Specification”. D6 described several modelling approaches and, based on these, a number of simulation approaches. The common basis for all these modelling and simulation approaches is the inter-domain point of view. Traditional simulation approaches try to model the network as exactly as possible (some even simulate MAC layer behaviour), making simulation scenarios of inter-domain networks virtually impossible due to scalability problems. Choosing a suitable abstraction reduces this scalability problem and enables the simulation of large scale inter-domain scenarios, at the cost of reduced exactitude. Four of these approaches were implemented, hybrid simulation by the University of Bern, time series simulation by Budapest University of Technology, RTC-FSIM by Salzburg Research and Inter-IP (called “Planisfero” in D6) by Telecom Italia Lab. Additional work was done by NEC (http traffic generator), University of Dortmund (VoIP generator, VDM interface) and CINI (integration work).

One part of the integration task was the implementation and integration of the statistical and analytical models into simulators. Also in some cases third party software and frameworks had to be integrated to build a simulator framework. RTC-FSIM is an good example for that. This part of integration is complete for most of the simulator approaches. Very little issues remain in this respect. The detailed implementation and integration reports for each simulator approach can be found in Section 3, the implementation and integration status of each approach in Section 1.2.

At a higher level, the simulators have to be integrated into the global InterMON architecture. This involves a customized GUI interface to build simulation scenarios and to select the appropriate simulators to evaluate them. The topology visualisation tool from work package 6 is reused here and can be customized to the simulators’ needs. Further needed is a mechanism to pass simulation requests to the simulator’s simulation manager over the global controller, done using the Java Messaging System JMS, which also allows persistent storing of simulation requests and their results for later reuse. Finally, a standardized way to return the simulation results back to the system is required, so the user can examine them using the tools supplied by the visual data mining toolkit. This is done by writing an import filter for every simulator approach that converts the simulators output (event traces, time series, etc.) to the VDM format used in the visual data mining module.

Section 1.3 gives an overview over this part of the integration architecture. Section 2 contains more detailed information about the involved procedures and signalling and Section 3.5 specifies the generic simulation control module for the GUI. While the simulators have not been fully integrated using this design, the services themselves are already implemented and ready to use.

Because of the great number of tools, modules and interfaces in the work package, and more so in the project, an important focus was the definition of data formats. In the InterMON project the use of XML as a basis for data formats is mandated. However, especially third party tools like measurement programs and simulator environments use their own proprietary formats specialized to their purpose. Thus, a wrapper program is necessary in many cases. Moreover, modules often use similar, but not exactly the same, kinds of information. This is the case with simulators for example: Every simulator has some notion of topology and traffic models, but the details differ from simulator to simulator. A format with as little redundancy, while retaining the simulators particularities, had to be found.

The descriptions of data formats involved in the modelling and simulation toolkit are described in Section 4, which also cover the interface definitions between the several parts of the toolkit and the remaining architecture.

In the appendix the reader can find example source code for interfaces specified in earlier sections.

1 Toolkit Structure

1.1 Introduction

Work package 5 of the InterMON project investigates and develops novel approaches to inter-domain modelling and simulation, which were described in Deliverable 6, "Modelling and Simulation Specification" [D6]. There, several different approaches were proposed, each specialized for a set of questions the simulator user may want to find answers for. Of the simulation approaches, four were finally implemented: The Hybrid Simulator originally described in Section 4.2 of D6, Time Series simulation as described in Section 4.4 of D6, the fluid simulator RTC-FSIM from Section 4.3 of D6, and Inter-IP, the implementation of the models shown in Section 3.7 of D6. Section 1.2 briefly describes the ideas behind all the implemented approaches and their main areas of application.

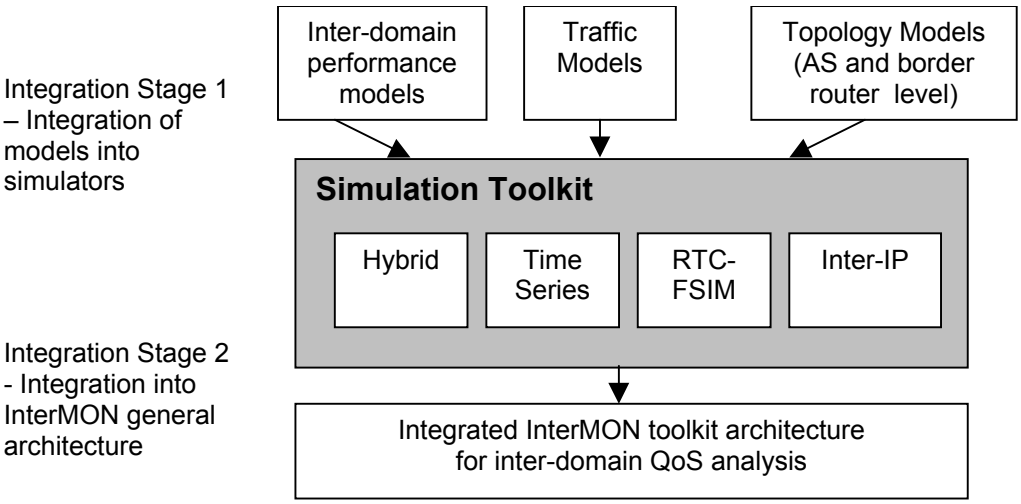


Figure 1.1 - The two stages of integration

Integration of the modelling and simulation toolkit consists of two stages (as shown in Figure 1.1): The first stage is the integration of the various developed models into simulators. An example for this stage is the development of the model plug-in mechanism in ns-2 described in Section 3.1.3. For the time series and the RTC-FSIM simulators this also included the integration of third party tools such as Matlab and Simulink. This stage of integration is described in the simulators' implementation reports located in Section 3.

Stage two is the integration of the modelling and simulation toolkit into the global controller architecture of the InterMON project, where a common way to build and start simulation runs had to be specified and implemented. Subtopics of this are the integration into the GUI, message passing through the GC, persistence of the generated results and interfacing towards the visual data mining module, which gives the user the ability to examine the simulators' output. Section 2 describes the processes in detail. All parts of this global integration design that are not simulator dependent have been implemented. However, the simulators themselves do not fully comply with this design yet. See Section 1.2 for details on their integration level.

1.2 Roles and Status of the Simulation Approaches

1.2.1 Hybrid Simulation

Role

The hybrid simulation module combines packet-based simulation of ns-2 with analytical models by using a hot-plug mechanism, which makes single ns-2 nodes behave like whole networks (e.g. autonomous systems). This abstraction allows the user to simulate large topologies in a fraction of the time a full scale packet-based simulation would take.

Suggested application areas for the hybrid simulator include

- End-to-end QoS evaluation of single flows – simulated using traditional packet-based models – over a complex backbone network.
- The effect of changes in a backbone network (e.g. addition/removal of links, capacity changes, big changes of network load due to new SLAs, etc.) on flows traversing the domain.

Integration Status

Stage one of the hybrid simulator integration is complete. The module loading mechanism in ns-2 as well as a general purpose multi-domain model module have been implemented, integrated and tested. Integration stage two has only been specified so far. No implementation on this level has been done yet.

1.2.2 Time series simulator

Role

The time series simulator evaluates the QoS state of the network in terms of throughput, drops, delay and jitter. Since the simulator is based on aggregate load information the QoS values represent an average for the traffic as a whole.

Suggested applications:

- Analysing the effect of additional traffic on the QoS state of the network.
- Analysing the effect of rerouting a part of the traffic on the QoS state of the network.

Integration Status

The time series simulator has nearly completed stage one: The simulator engine runs and can parse the input XML files. However, the code generating formatted simulator output has not been implemented yet. Also, integration stage two has only been specified at present. No implementation exists.

1.2.3 RTC-FSIM

Role

The Rate and Time Continuous Fluid Simulation (RTC-FSIM) is a novel fluid simulation approach developed within the InterMON project. In contrast to other fluid-based modelling techniques which require an event-based simulator for the execution of the model, the RTC-FSIM approach models all traffic as continuous signals and describes the signal transformations by means of differential equations.

As there are no events in the RTC-FSIM approach it does not suffer from the performance drawbacks that may arise from the ripple effect [Liu] in event-based fluid simulators. The performance of RTC-FSIM basically depends on the implementation for solving the differential equations.

As RTC-FSIM performance is completely independent of the link speeds and amount of traffic it is particularly well suited for scenarios with high link speeds and large amounts of aggregated traffic. Another key feature of RTC-FSIM is the ability to increase the simulation speed by allowing for less accurate results.

It is therefore suggested to employ RTC-FSIM in large inter-domain scenarios. The simulator produces queuing delay, loss, and throughput results. As the simulator is fully integrated with IPFIX measurements it is very useful for evaluating alternative routes for portions of the observed traffic. For the definition of that "portion" the full flexibility of the IPFIX meter rules is available. As an example, the "portion" could be defined as all traffic going to destination domain X; it could be all IP telephony flows; it could be all Web traffic coming from domain Y.

RTC-FSIM can also be used to simulate the effect of additional traffic beside the measured traffic. The addition can be a general load increase by some factor, a previous IPFIX measurement or a general traffic model (e.g. of Web traffic).

Integration Status

The RTC-FSIM approach is completely implemented in a Matlab / Simulink environment. However, running simulations still requires human interaction, which prevents automated simulation processes. Use of IPFIX measurements is fully integrated into RTC-FSIM, even if so far only artificial data have been used. Stage two integration has been planned and some simple, preliminary integration was already done earlier.

1.2.4 Inter-IP

Role

The INTER-IP module evaluates the end-to-end packet or volume transfer delay performance for a traffic relation (i.e. a flow identified by the source and destination IP address and by the service class if differentiated service is used) that crosses multiple domains in a Differentiated Services context.

A suggested application in the InterMON context could be the rapid comparison (on the base of the delay metric) of alternative routes for the same flow. The tool is able to compute the end-to-end delay performance in a small amount of time (if compared to simulation) because is based on a pure analytical model resolved in a closed form.

Integration Status

Stage one of integration is complete. Stage two has been specified, and some preliminary integration has already been done. Users can already specify the name of a file containing the predefined example scenarios and a simulation type, which are sent to the simulator via the global controller and the simulation manager. The results are them sent back to the GUI.

1.3 Relations to the General Architecture

The simulation and modelling toolkit makes use of the main architecture components GUI and Global Controller (GC) and presents the simulation specific simulation managers for integrating the simulation software into the InterMON system. Figure 1.2 shows the part of the InterMON architecture that is involved in the task of modelling and simulation.

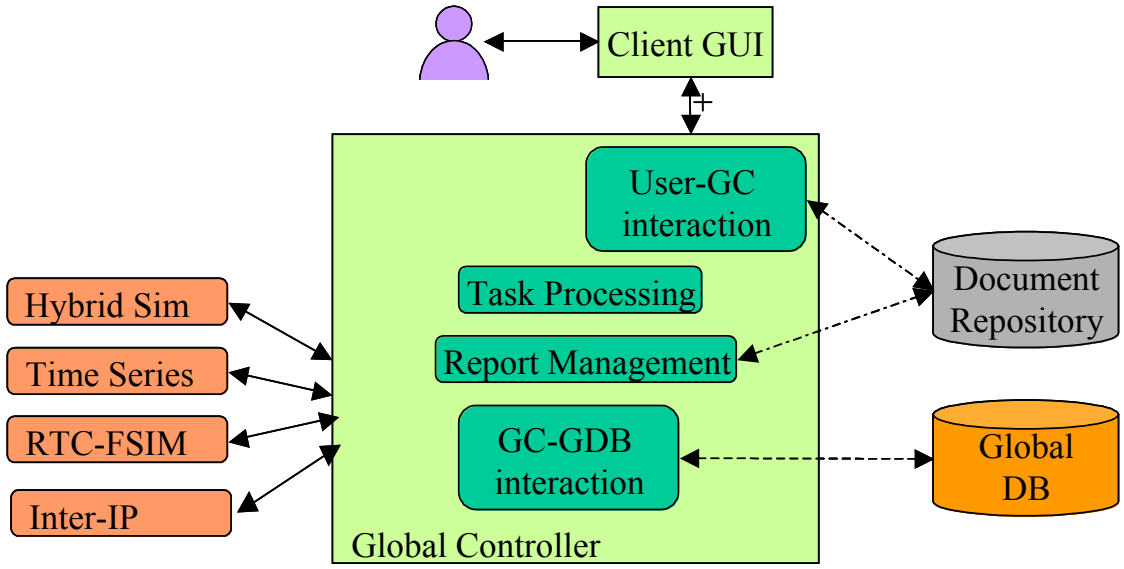


Figure 1.2 – Architecture view of simulation toolkit integration

The Client GUI is extended with an application module to start and control simulation tasks. This module is described in detail in section 3.5. Messages between an instance of the Client GUI and the Simulators always pass the GC. Inside the GC functions handle the forwarding of simulation requests to the targeted simulators. These functions may check and/or modify the messages. They can also access the data base attached to the GC and make use of Repository functions.

The four simulators shown in Figure 1.2 depict four different classes of simulators (i.e. hybrid simulation, time series simulation, RTC-FSIM, and Inter-IP). A number of different simulation types are used by the InterMON system (see chapter 3). This requires the presence of an simulation manager component near to the actual simulator instance to allow for remote configuration and results transfer in a way that is understandable by InterMON's GC.

2 Communication

This section presents the communication entities, processes, and messages that are involved in using the modelling and simulation toolkit within the InterMON architecture.

2.1 Interaction with the General Architecture

Communication processes inside the simulation toolkit exchange messages between GUI, GC, simulation manager and database. The Global Controller acts as a communication centre (server). The GUI and the Simulation manager (clients) connect to it on startup and build a communication channel.

Communication between GUI and GC, as well as Simulation manager and GC, uses Java Message Services (JMS). This allows for asynchronous transfer of notification messages and task requests. A synchronous mode, where the sending client is blocked until a result is returned, is also possible.

The database access uses JDBC (Java data base connectivity) methods to store results and query the data base (mysql).

2.1.1 Starting a simulation process

To configure and run the simulation a user needs to start the InterMON GUI and to login to the system. He then selects 'Modelling and Simulation' from the Tools menu and chooses 'Start Simulation' from that section. Input data for the simulation process in queried from the user (details in section 3.5). When the Submit-Button is clicked the GUI generates a start simulation request from the user's choices and input parameters. Values in the request message determine the targeted simulation type and simulator instance. The request is send directly to the Global Controller which in turn hands it down to the selected simulator instance. The GC itself will store the request so that other requests (e.g. for the status) and result messages can be matched to the request and handled properly. Upon receiving the “start simulation” request the simulation manager parses and checks the request. It also tests whether or not it can start the task directly or has to queue it (in case of too much current workload). The manager finally constructs a reply for the user which it sends to the GC. The GC determines the source of the matching request and tells the user the reply from the simulator.

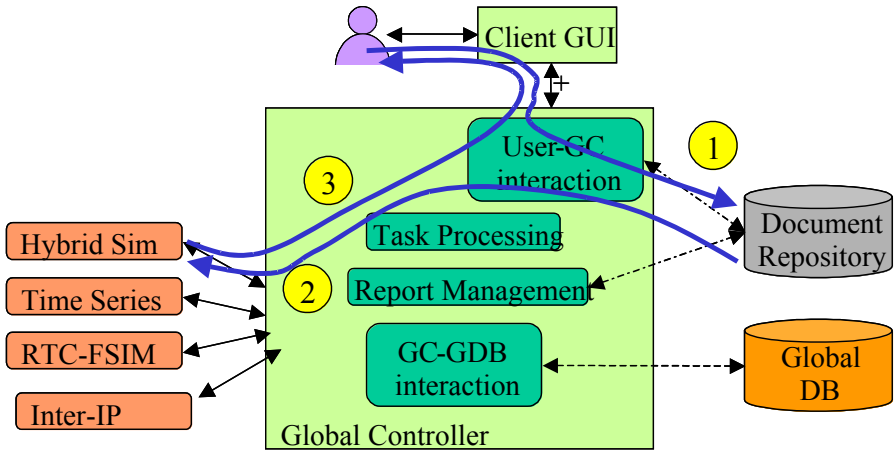


Figure 2.1 - Starting a simulation

The numbers in Figure 2.1 denote the order of performed communication processes.

2.1.2 Querying a simulation process

The process for querying the status for a previously started simulation request is very similar to starting the simulation itself. The only difference is that the request only needs to specify 'status' as the requested command and include some reference parameter which uniquely identifies the started simulation task.

2.1.3 Simulations Results Handling

Simulation result data is pushed from the simulation manager into the InterMON system, i.e. to the GC. Whenever a simulation process has finished it can tell its manager about that fact using a signalling function. The manager will in turn collect all the results from the simulator (often stored by the simulator in some files) and format these accordingly to the output formats. This data is extended with information from the original simulation start request (at least a unique task ID). A message of type “simulation result” is built that is send directly to the Global Controller. The GC stores that reply in conjunction to the original request. Based on this message a notification message is sent to the GUI. This message does not contain the (potentially huge) simulation result data sets. The user may then request the system to perform some kind of data analysis on the simulation results. Figure 2.2 shows the process. Afterwards the user may request the system to perform some kind of data analysis and visualisation on the simulation results. He can refer to these data sets using a unique ID for the simulation task.

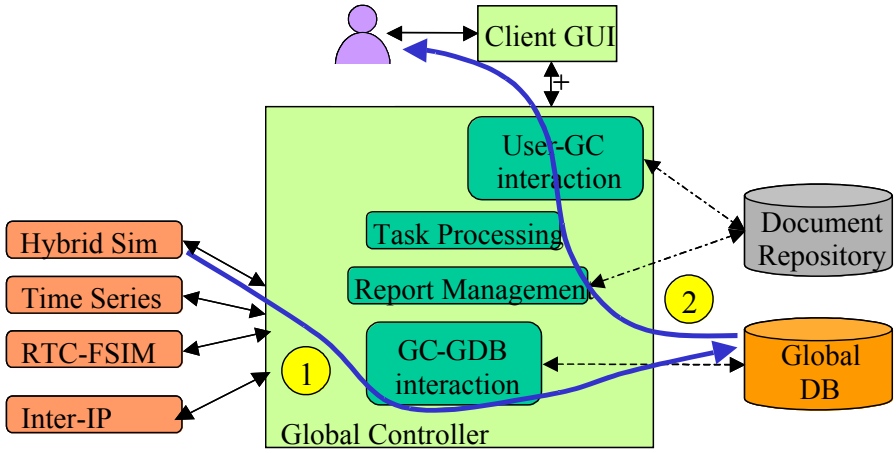


Figure 2.2 - Passing simulation results

2.1.4 Visualisation of Results

When a simulation has finished and the results are transferred to the GC and stored there, the user can request further analysis (e.g. statistical functions) to be applied to the results. Often this involves the preparation of images from the processed data sets. This is the task of the visual data mining toolkit. GUI components for selecting and starting visualisation tasks can base their input on the data sets obtained from the simulation processes (in addition to direct evaluation of captured traffic traces). In that case a visualisation request is issued to the GC which refers to the results of a specific simulation task. This request is enriched by the GC with the referred data and passed to the VDM component (review architecture overview in Deliverable 4). The detailed process of the visualisation and the transfer of image information to the user is not in the scope of this deliverable.

2.2 Interface to the Visual Data Mining Module

The Visual Data Mining (VDM) module will be used as the interconnection path between data production and data representation. It contains three sub-modules, which are called Filtering/Data Mining, Mapping and Rendering (please refer to [D7] and [SM02]).

The most important component is the Filtering/Data Mining module, which imports the proprietor data into the VDM-Module, formats this data and does the required Data Mining.

The VDM Module looks as seen in Figure 2.3.

To be able to define even complex data mining algorithms, the data mining filters could be chained, as seen in Figure 2.3. Therefore the import-filter will create a Filter Data Item (FDI) which is able to be transferred from one filter to another. The FDI, which was produced by one filter, could also be used within multiple other filters as an input FDI. The Class Reference of the Filter Data Item is given in Section 4.3.1.2.

The schemes will define what the filter will expect to get as an input item and variation item (which will be defined below) and it will define what data will be produced for the output. The function of a scheme is to give a detailed description of the communication data in a kind, a VDM module can understand. So the VDM is able to detect “wrong” connections within chains.

The conversion between two filters is needed to be able to define “general” filters, which are easy to reuse within different filter chain pipelines. The conversion module will also be called “Mapper” because it maps data from one (input) scheme to an (output) scheme.

The variation is the definition of variables within the filtering process. One example could be an interval length for an aggregation or a threshold value.

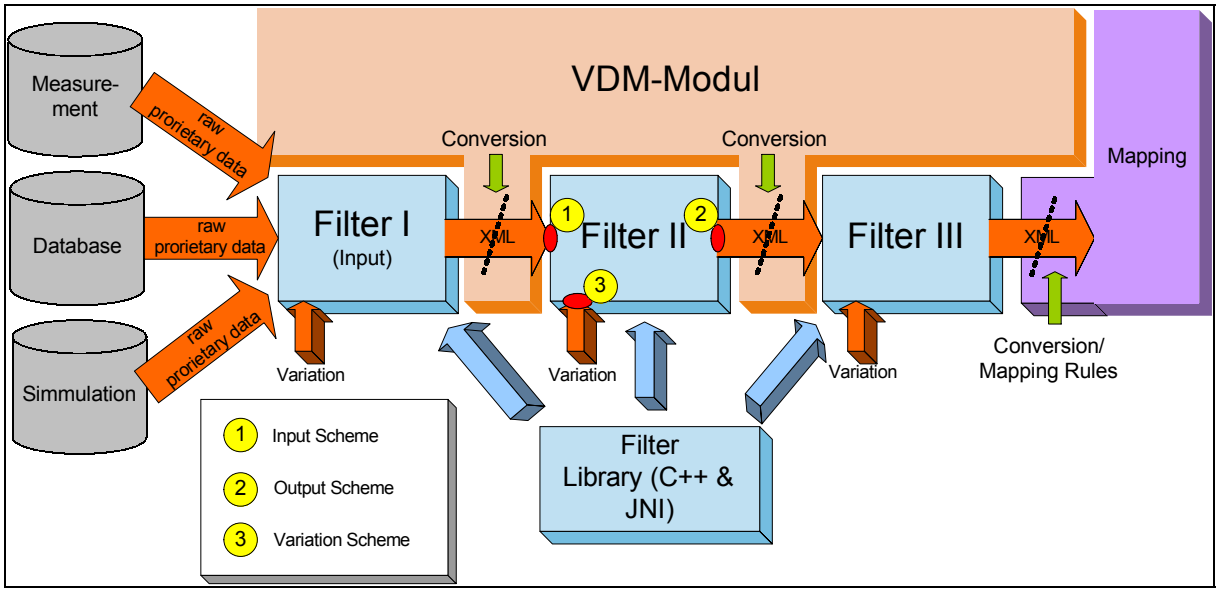


Figure 2.3: Visual Data Mining Module

2.3 Using the BGP-4 topology description in simulation processes

2.3.1 View of the Internet delivered by the BGP-4 tools

The data collected by the BGP4 tools yields the following view of the Internet for each autonomous system they are deployed in:

a) border routers of the autonomous system where the tools are deployed

b) AS-PATHs: The collection of possible paths to all destination prefixes in the Internet as seen from the autonomous system where the tools have been deployed.

c) Routing events which reach the autonomous system during the observation period.

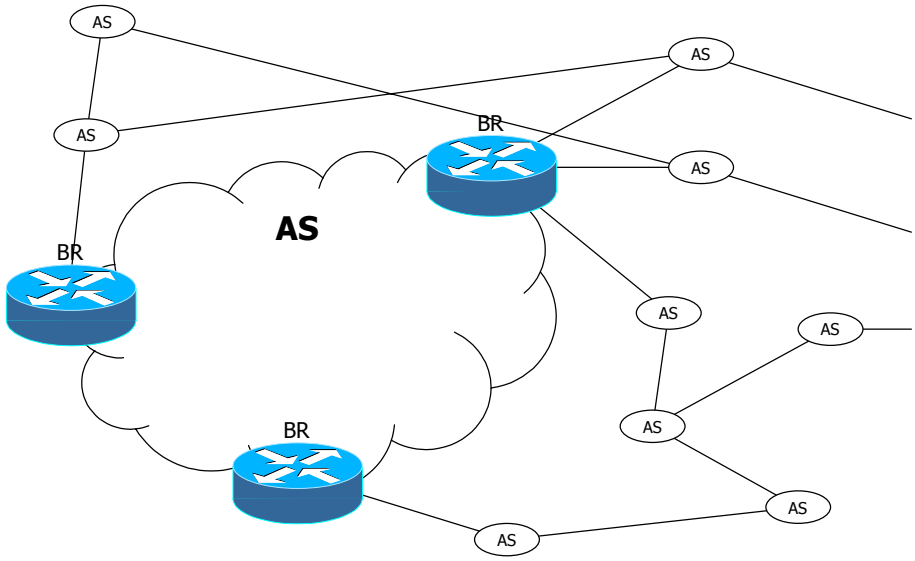


Figure 2.4 - View of the Internet as provided by the BGP tools

Common practice design rules mandate that the border routers appear as the next hop for external networks in the BGP4 data. Routes injected by the AS into the Internet (AS-path = ^$) will behave differently, because the router injecting the route into the BGP4 protocol doesn't necessarily have to be the router which has the network directly connected to it. This behaviour is compatible with the black box model and the proposed traffic matrix with entries reserved to AS internal traffic sources and sinks.

Additional information delivered by the BGP-4 tools includes topology events. Topology events are updates, either announcements of new or existing routes or withdrawals, which can be used to generate a dynamic description for all paths involved in the end to end QoS scenario which is being simulated.

2.3.2 The End to End QoS scenario

The End to End QoS scenario is bound by the IP address of two end systems which have to be interconnected through the Internet. The topology information for this scenario is restricted to routes and updates for the most specific prefixes to these IP addresses, since routing is done by directing the packets to the Interface in the router which leads to the next hop for the most specific prefix for the corresponding IP address.

2.3.3 Static use of BGP4 data

The first way to use the topology data in simulation is to use the static components of the BGPTopologyTree, i.e. the AS's, border routers and links, mapping them as elements in ns2 scripts (or other simulator's configurations), on top of which the end to end traffic flows are sent.

2.3.4 Dynamic use of BGP4 data

2.3.4.1 Including dynamic creation and destruction of links in the topology

Changes in the AS policies might lead to changes in the paths followed by specific traffic flows. These changes result in modifications in the topology which are stored as announcements and/or withdrawals of routes. This information can be integrated into ns2 scripts to simulate the effect of creation and destruction of links in the topology.

2.3.4.2 Simulating the BGP-4 protocol

The BGPTopologyTree may include different levels of dynamic behavioural information. The first level is information on how the BGP-4 protocol behaves. It is thinkable that the network layer simulation scripts might include simulations of the BGP-4 protocol behaviour, which could be fed with this kind of information.

3 Toolkit Implementation

3.1 Hybrid Simulator

3.1.1 General Integration

3.1.1.1 GUI Interaction

Like the other simulation approaches, the hybrid simulator follows the general scheme for issuing simulation requests. Using the topology visualisation tool and a topology derived from BGP messages the simulation scenario is set up. Callback methods supplied to the GUI application pop up dialogs when a link or domain is (right-)clicked in the editor. The basic functionality of these pop up dialogs allow the user to define a set of pre-selected properties for the inter-domain link or the autonomous system.

For links, the user must be able to select a bandwidth from a set of predefined bandwidths of typical link media. Alternatively, non-standard link bandwidths are configurable by manually typing in the desired bandwidth value. A special value for link bandwidth is zero, which can be used to simulate link failure. If known, the queue capacity (parameter *K* in the hybrid simulation inter-domain link model) should also be configurable from the pop up dialog, to allow the user to fine-tune the scenario.

Clicking on autonomous systems' symbols also pops up a dialog, which allows to the user to choose from a number of delay models and to further configure them as necessary. In an advances case, the list of available delay models would be derived from a runtime query of the database and a set of fixed, artificial delay models.

By handling the callbacks appropriately the user can be given the possibility to define a from-to relationship between ASs. This can be used to provide a third kind of dialog with a choice a flows that can be configured between the two ASs. Again, the list of available flows could depend on the available flow measurements in the InterMON database.

When the user has finished editing the scenario, the topology and the list of changes done to it is sent back to the appropriate GC component.

3.1.1.2 GC Component

When the request from the GUI reaches the simulator-specific GC component it is usually not complete. Many parameters will be missing from the graph, such as the actual link bandwidths and AS delay models of nodes and links that were not explicitly configured by the user. These missing values must now be inserted into the request. Depending on the availability of measurement data for the network entities, default values, e.g. infinite bandwidth for inter-domain links and zero delay for ASs, or values distilled from measurements performed on the network entities' real world counterparts are used. Once this process is completed, the request is sent on to the simulation manager.

3.1.1.3 Simulation manager and Simulator

The simulation manager converts the request coming from the global controller into the request format specified in 4.1.1, i.e. an ns-2 script and a set of internal XML model descriptions, and sends it to the simulator. After the simulation run, the resulting trace file is converted to VDM format using a simulator-specific input filter to the data mining module (see also Section 4.3.1.1 for more information about this step). Finally, the converted results are stored as a persistent java bean together with the original request. The user can then further examine the results using the visual data mining module or store the results for later use.

3.1.2 Model Implementation

The MDModel plug-in is a somewhat simplified implementation of the analytical model introduced in the section about hybrid simulation of Deliverable 6 (“Specification of the Inter-Domain Modelling and Simulation Toolkit”). The simplifications mainly concern the lack of support for multiple traffic classes as used in differentiated services networks, for example. Also, the implementation of inter-domain link models is restricted to the queuing model approach since the alternative hierarchical function trees are mainly useful in the context of multi-class traffic and queuing models have been researched more thoroughly. This section presents the structure and the main concepts of the implementation, i.e. interfacing to ns-2, calculating network load distribution and simulating the dropping delay behaviour of single paths in the multi-domain model. Some major changes had to be done to the mathematical model of load distribution and will be discussed below.

The plug-in was written in C++ (approx. 3000 lines of code) and was confirmed to be compiling and running on Linux (Intel), Solaris (SPARC) and MacOS X (PowerPC). Only the GNU C++ compiler has been used to compile it, so using other compilers might introduce minor problems.

3.1.2.1 Plug-In Structure

To avoid confusing the reader, we will investigate the structure of the MDModel plug-in in a top-down way, starting with the interface to ns-2 and proceeding to the structure of the various subsystems. From the perspective of ns-2 the plug-in has the simplified structure shown in Figure 3.1.

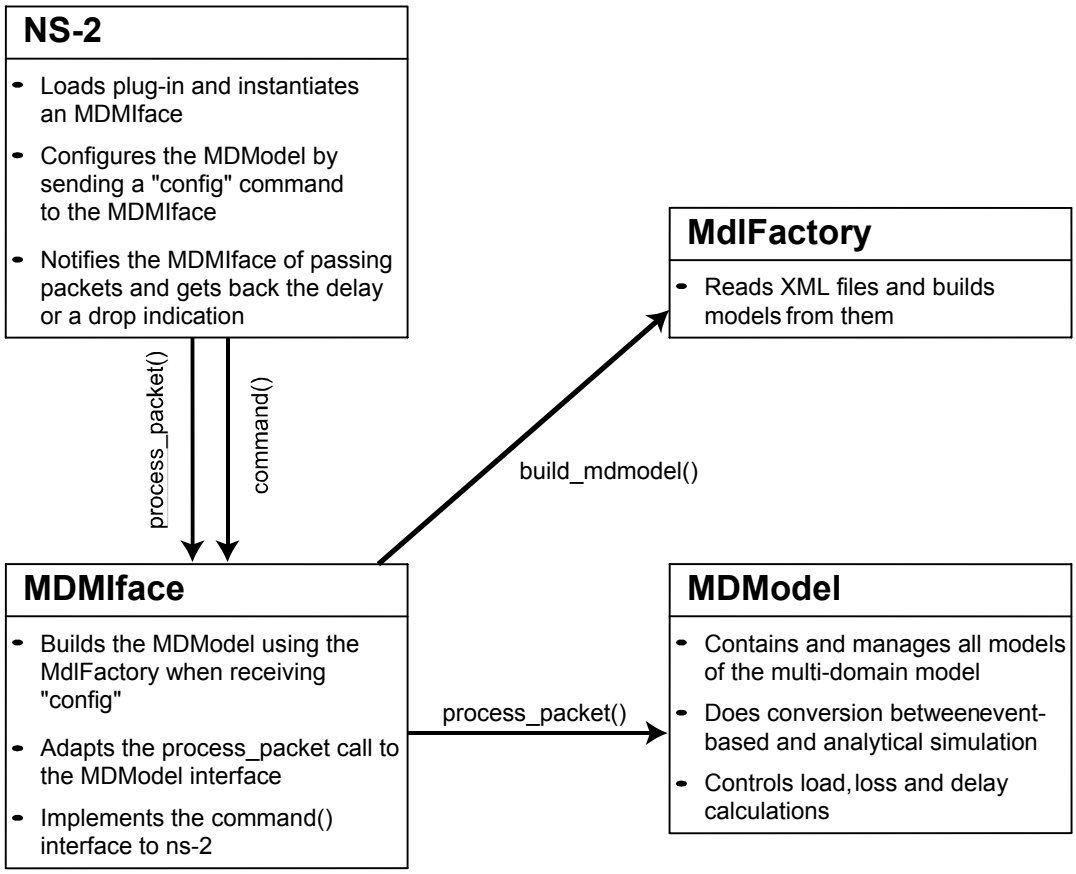


Figure 3.1 - Structure of the interface between NS-2 and the MDModel plug-in

When the plug-in module is loaded only an MDMIface object is created, which serves as the interface (and potentially and adaptor) between the actual multi-domain model code and ns-2. It also interprets generic command strings sent from ns-2 using the `command()` method. Once the plug-in module has been loaded it has to be configured before it can be used. To do so, NS-2 sends a “config” command

containing the filename of the XML multi-domain model description to the MDMIface, which delegates the actual construction and configuration of the model to the factory method `build_mdmodel()` of the class MdlFactory (which is static and thus does not have to be instantiated). The MdlFactory class implements the XML parsers for all types of models (multi-domain model, domain models, inter-domain link models and traffic generators) and can recursively construct the submodels of the multi-domain model. Once the MDModel object has been successfully built it is returned to the MDMIface, which stores a pointer to it and subsequently forwards all event notifications from NS-2 to it.

Inside the MDModel there are two separate data structures managing the submodel objects at the same time in order to provide efficient access for algorithms with different requirements. One is the *linked graph structure* shown in Figure 3.2: All submodels have pointers to their successors. Additionally, IDLinks store a type identifier and a connector index for their successors since they serve as graph edges and can be connected to any other type of submodels, to a several connectors in the case of a domain[1] model. Some of the models may only be predecessors (InLinks, TrafficGens), others may only be successors (OutLinks). Moreover, connections crossing the logical border to ns-2 can connect to InLink and OutLink objects. They implement the conversion between analytical simulation and the "event-based world". A discussion of the algorithms used in that context can be found further below. This structure is useful when implementing graph traversal algorithms like the calculation of the delay distribution of a path.

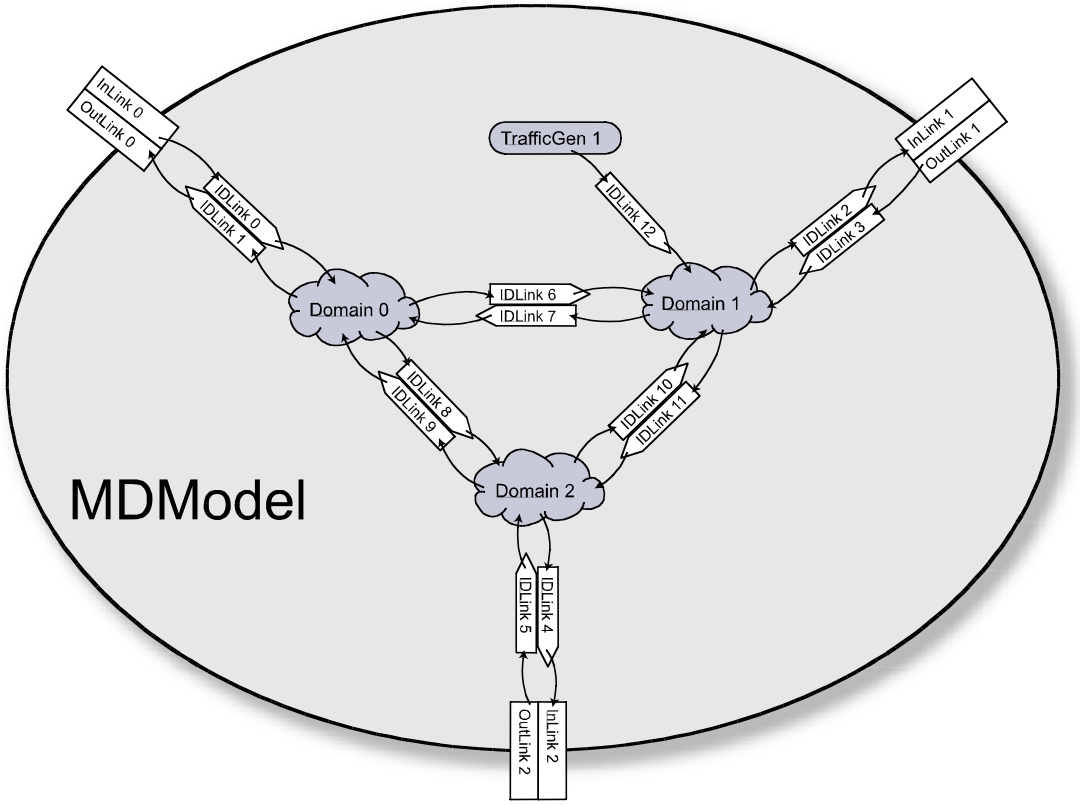


Figure 3.2 - Multi-domain model topology structure

To better manage the models it contains, MDModel also stores lists of pointers to them. This is not only useful for direct access to the submodels. Routing inside the multi-domain model also depends on lists of pointers to IDLinks, one for each possible pair of inbound and outbound links and one for each traffic generator. Figure 3.3 shows the situation from Figure 3.2 from this perspective.

[1] The name domain models is somewhat historic. AS would better reflect the actual use.

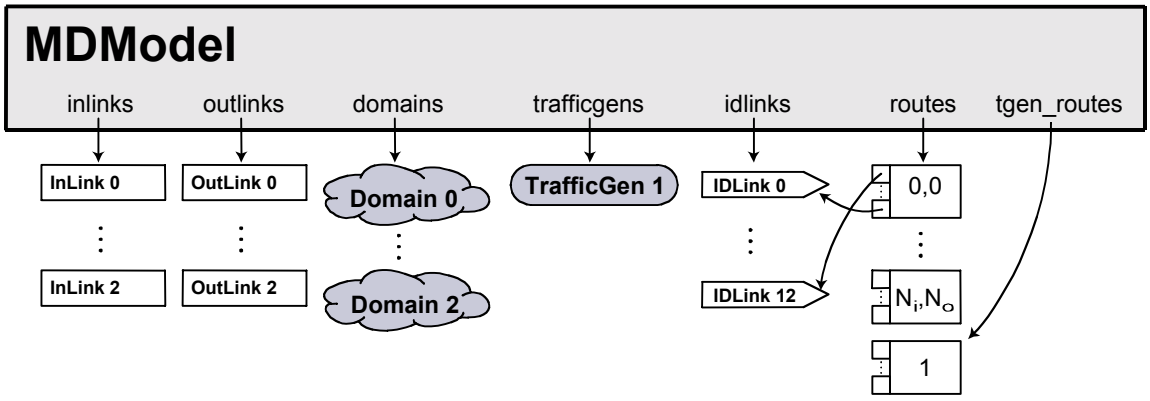


Figure 3.3 - Multi-domain model list structure

The lists of border links (InLinks and OutLinks), domains and traffic generators are trivial. The inter-domain links of the multi-domain model are linked in a more complex way, however, in order to maintain information about the routing paths. `routes` points to a $n \times m$ matrix, where n is the number of inbound links into the multi-domain model and m is the number of outbound links, whose elements in turn point to route. Routes are simply stored as lists of pointers to IDLinks, which suffices to uniquely define them. There is also a list of routes related to the traffic generators, `tgen_routes`, storing only a single path per traffic generator. Its purpose is to describe the path the generated traffic takes from the generator itself to the model border.

There is still one open issue with traffic generators, however. It would be very convenient for the user if InLinks could be assigned traffic generator objects whose traffic load values would be added to the bandwidth estimates. With this feature it would be possible to easily and efficiently create background traffic while keeping detailed simulation of single flows in the packet-based space.

3.1.2.2 Interfacing to NS-2

Interfacing between ns-2 and the plug-in modules was somewhat enhanced to allow for a more elegant solution of node identifier translation. The problem with node identifiers is that because ns-2 generates them at runtime they are not known during the configuration phase of a simulation scenario. However, the plug-in module has to know them in order to be able to find the inbound and outbound links a packet passes and so to distinguish between the paths a packet can take inside the model. Both ns-2 and MDModel use integer identifiers for nodes and border links, respectively. With proper configuration a solution with lookup tables like shown in Figure 3.4 solves this problem.

When a packet passes an enhanced node the plug-in receives a message carrying the previous and next hop identifiers. Using a simple lookup table it can then determine, which path the packet would take and thus what loss probability and delay distribution should be applied to it.

The remaining problem with this solution is the impossibility to configure this mapping before the simulation runs. We solve this by giving nodes additional names (ASCII strings) in the ns-2 script and refer to these names within the module configuration file. Once the simulation starts, a mapping between these names and the integer runtime identifiers inside ns-2 can be created. This mapping is stored in the file “nodenames”. The plug-in module can then read this file and replace the configured names in its mapping with the runtime integer identifiers.

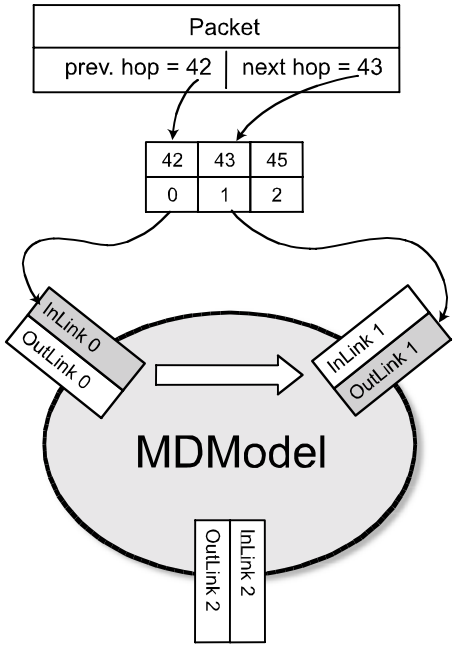


Figure 3.4 - Mapping of ns-2 and plug-in internal Ids

3.1.2.3 Load Distribution

In Deliverable 6 the calculation of load distribution in multi-domain models was specified using so called transit matrices. While implementing the approach we found that even if the approach works well for isolated network domain models it is problematic for multi-domain models. Calculating the equilibria of such systems is not a problem, but due to the missing routing information in the models they tend to be too high. For example, a system with of 2Mbit/s offered load could have an equilibrium outbound load of 2.3Mbit/s without any internal sources! Introducing restraints on the parameters and changes to the algorithm calculating the load distribution ultimately also failed because of this absence of routing information.

The solution now implemented is therefore directly based on routing information (which has to be present in multi-domain models anyway). At simulation time t we have the bandwidth estimates $b_{i,e}$ between all border node pairs i and e of the multi-domain model. A simple algorithm using knowledge of routing paths is now used to calculate the bandwidth distribution. Let $R_{i,e}$ be the routing path (i.e. a sequence of inter-domain links) between i and e. The outgoing load of an inter-domain link l depending on the offered load L is written as $l(L)$.

Set the offered loads on all inter-domain links to 0
FOREACH i,e DO
    $\Delta L \leftarrow b_{i,e}$
    $R \leftarrow R_{i,e}$
    FOREACH l in R DO
        $L \leftarrow$ current offered load on l
        $\Delta L \leftarrow l(L + \Delta L) - l(L)$
        Set current offered load on l to $L + \Delta L$
    END
END

When the algorithm has completed, the offered and outgoing load of every inter-domain link in the multi-domain model is known (Note that if a link is not a member of any routing path its load value remains zero, which is what would be expected).

Depending on the number of links and routing paths in a multi-domain model the above algorithm can take a rather long time to complete. Therefore, it should not be executed whenever a bandwidth estimate changes. MDModel updates the system in regular intervals unless the bandwidth estimate on an InLink changes noticeably. In these cases we don't have to recalculate the loads for the whole system. It suffices to take the load difference at the ingress router and update only the affected path using the inner loop of the above algorithm.

3.1.2.4 Packet Loss and Delay Simulation

Based on the load distribution calculations presented in the previous section, a multi-domain model's delay distributions and loss ratios can be calculated. Both of these are found by traversing the inter-domain link and domain models along a routing path through the multi-domain model.

The packet loss ratio along the routing path $R_{i,e} = l_{r_1}, \mathrm{K}\ , l_{r_n}$ is simply given by

$$1 - \prod_{l \in R_{i,e}} p_f(l)$$

where $p_f(l)$ is the packet forwarding probability of link *l*. This is the probability that the link's queue (actually the preceding router's queue) is not full when a packet arrives.

Delays along a path can be simulated analogously. Let D_l be the random variable of the delay caused by link *l*, and let $D_{l,k}$ be the random variable of delay on the corresponding path in the domain between the inter-domain links *l* and *k*. Then the delay on the path between ingress router *i* and egress router *e* is given by

$$D_{i,e} = \sum_{l \in R_{i,e}} D_l + \sum_{k=r_1}^{r_n - 1} D(l_k, l_{k+1})$$

These values can be simulated by traversing all inter-domain link and domain models along the path, generate a random value for each of them and taking the sum of the results. For large multi-domain models this calculation might take too long. In such cases an alternative approach would be to combine the delay distributions of all the models into a single probability distribution. This can be by convolving the discrete distributions of all random variables along the path. Having a single discrete delay distribution of a path further allows to easily calculate statistical moments like the mean delay or the path's jitter, which would be Var($D_{i,e}$) (if jitter is interpreted as delay variation).

3.1.3 NS-2 Plug-In Interface

3.1.3.1 Introduction

The base of the ns extension, to delay packets within a node, is implemented rather straightforward. An new object ISPDelay is available, which can be attached to ns nodes and provides interfaces for the external modules. The command syntax is something like:

```
set c1 [new ISPDelay]
$n1 attach-isp-module $c1
```

This object provides the following mechanisms:

- It provides an API to have an external module attached. The external module provides information about packet delay and whether a packet has to be dropped. Information about the last visited node and the next node as well as information about the packet itself are provided to the external module.
- It updates some header information, required by external modules of other nodes.

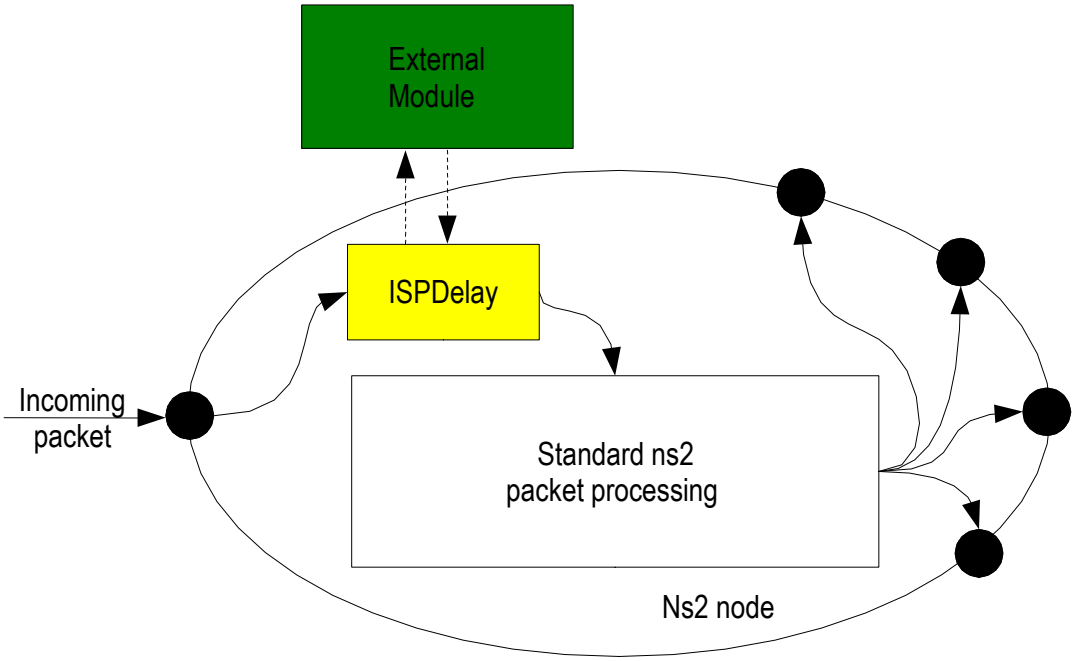


Figure 3.5 – Structure of the plug-in mechanism

Figure 3.5 shows the relationship between the ns2 node, the ISPDelay object, and the external module. An ISPDelay object, which is part of the ns2 node does not alter the simulator behavior unless an external module is attached. However, if a simulation has to be set up using external modules, it is important to attach the ISPDelay object to all nodes, even if no external modules are attached. This is necessary for the other external modules to work properly.

3.1.3.2 Topology Setup

The simplest way is to source "ispns.tcl" and use the new commands IM-duplex-link, IM-simplex-link and IM-node, to create links and nodes. This commands behave almost exactly like their ns2 counterparts, but take care of some additional administrative work. The ISPDelay object is placed at the entry point of a ns2 node. To attach an external module, a command like

```
[$n1 entry] attach-module /tmp/isp_test.mod
```

has to be used. [$n1 entry] returns a reference to the object at $n1's entry point: the ISPDelay module. The attach-module command of the ISPDelay object attaches the external module in the file isp_test.mod to the node. As an example, the following script sets up a topology with three nodes in a row with simplex links between and external modules attached to the node in the middle:

```
set ns [new Simulator]

set n1 [$ns IM-node "Knoten-0"]
set n2 [$ns IM-node "Knoten-1"]
set n3 [$ns IM-node "Knoten-2"]

[$n1 entry] attach-module /home/baumgart/ns2-extension/modules/isp_test.mod
[$n1 entry] modcmd delay 0.1

$ns IM-simplex-link $n0 $n1 1Mb 19ms DropTail
$ns IM-simplex-link $n1 $n2 1Mb 19ms DropTail
```

The additional parameter to the IM-node method "Knoten-0" is used to label nodes with names. After the initialization of the last node, a file "nodenames" is written, which lists the "names" of the node and the numbers, which were assigned by ns. This numbers are used by ns in it's trace and log-files and are also used by the external module API to reference the previous, the next and the local node. Therefore this file can be used throughout the analysis of simulation results and also by the external modules. Reading this file during module initialization provides information to the external module, which node is references by which number.

3.1.3.2.1 New ns Commands

Only few additional ns commands have to be used to set up a topology with external modules.

- `$ns IM-node <label>`
  This command instantiates a new node and returns the object reference to the new node. It behaves similar to "$ns node" , but requires an additional parameter to label the node with an unique label. This labelling results in the "nodenames" file, which allows an external module to lookup the label for an specific node id.
- `$ns IM-simplex-link <src> <dest> <bw> <delay> <queue>`
  This command behaves identically to the simplex-link command, but automatically updates the object reference tables within of the ISPDelay object.
- *$ns IM-duplex-link <src> <dest> <bw> <delay> <queue>*
  This commands behaves identically to the duplex-link command, but automatically updates the object reference tables.

3.1.3.2.2 Module Commands

There are some commands to configure the ISPDelay object or to query information. Usually these commands are not used directly, but called by the high level ns commands IM-node, IM-simplex-link, IM-duplex-link. The following commands are currently implemented:

- `$c attach-module <modulefile>`
  This command is used to attach an external module to a node. After the module is attached it is called to provide information about delay and loss for each single packet entering that node. <modulefile> is simply the fully qualified pathname of the file containing the external modulefile.
- `$c obid $ref $id`
  `$c obid me $id`
  The ISPDelay module provides information about the previous and the next node to the external modules. Unfortunately on the C++ level, a ns node does not maintain a reference to a neighbouring node or a link to that node, but only a reference to a component, the link head, within that link. To provide information about the previous and the next node to the external modules, a reference to a link head has to be mapped to the node-id of the target node of that link. Therefore each ISPDelay module maintains a table with (object-reference,id) pairs, which

allow to map the object reference of a link-head to the id of the link's target node. This command usually does not have to be directly, but is called automatically by the IM-simplex-link and IM-duplex-link commands.
A special syntax is to use the string *me* instead of the object reference. In that case the ISPDelay module will assume that *$id* is the id of the node it is attached to. This command is automatically called during node instantiation by the IM-node command.

- `$c trace [file]`
  This command enables tracing if packets. Once this command is executed, a trace file is for this external module is written. For each packet the trace file contains a line like:

  1.1 last: 1 this: 2 next: 3 size: 1000 type: cbr src: 0 dst: 3 ttl: 30 dly: 0.2

  The first token is the simulation time, the other fields provide information about the packet and the involved nodes. If *[file]* is omitted, the trace file is simply written to stdout.

- `$c stat`
  This command returns some statistics in a simple text form like:

  mod: /tmp/isp_test.mod in: 0 out: 0 loss: 0.0 avg-delay: 0.0

  This lists some simple some statistics about the packets received and transmitted by the module, the packet loss and the average delay. The average delay is simply the sum of all delays calculated by the module, divided by the number of transmitted packets.

3.2 Time Series Simulator

This integrated simulation tool consists of three parts. A GUI module that drives creation of a simulation request, a java bean that runs inside the global controller and modifies and completes the request it received from the GUI module and forwards it to the simulation engine, to the third part, that receiving the completed request executes it. In the following we discuss the three parts.

3.2.1 The GUI module

The *Requester* module in the GUI is not a monolithic module but consists of as many parts as many simulators exists in the system as depicted in Figure 3.6. When a simulation is to start, a visualization application is initiated to visualise the topology in any way and to handle user requests. Each sub module provides a call-back function to other GUI elements to let them delegate GUI events such as a user click on a link or on a router:

```
void processGUIEvent(GUIEvent e)
```

This call-back method can decide then, how to handle the event based on its type. If it recognises the event type, then it should pop up a window and let the user make changes specific to the scenario he want to run.

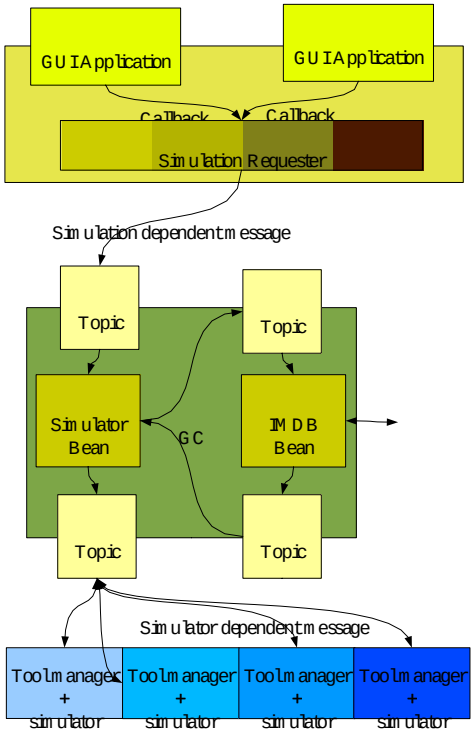


Figure 3.6 - Time series simulator integration

After the user finished the configuration the GUI application invokes the s*etInput()* method of the *Requestor* (sub)module that is used to send the topology data to the *Requestor*. Then the simulation specific module can pop up a window to ask any information it still requires to complete the request.

In the end the module creates a simulation specific scenario description based on the topology data received from the GUI application and the introduced changes and send it to the *Global Controller.*

3.2.2 The GC component

The java bean in the *Global Controller*, just like the module in the GUI, processes the message in a simulator specific way. That means that for every simulator type there may exist a bean in the GUI. However, it is only needed if any post-processing of the message from the GUI module is needed. It may alter the data in the request it received. It can retrieve any traffic or node model information regarding to the defined simulation and simulation time from the InterMON database as depicted in Figure 3.6.

In case of the time series simulator this component has the important task of retrieving data from the InterMON database. On the GUI the user specifies the network topology and may introduce some changes. This component has the task then to retrieve load information generated by one of the traffic metering tools (eg. IPFIX). For every router specified by the user it has to find, retrieve, optionally resample and convert the traffic to the XML format specified in 4.1.2.8 and additionally it has to define the traffic distributions (see 4.1.2.9) too. All this information can be derived from measurements of traffic in an AS between each pair of border routers. So, we measure the traversing traffic based on its entry and exit point in the AS.

Certainly, all this information should be present in the database before a user initiates a simulation. This requires proper setup of the measurement tools well in advance, to make them measure the traversing traffic.

Finally it should send all the configuration information in a simulation specific format that enables the given simulator to run the simulation scenario.

3.2.3 Simulation manager and simulator

The simulation manager could only be a wrapper interface between the java *Global Controller* and the C/C++ simulators. It receives the simulation description from the *GC* and starts the simulation engine parameterised by a file name that contains this description.

Additionally it can execute some control logic (eg. a number of iterations) defined by the message received from the *GC*. This logic should be defined again in a simulator specific way so every simulation manager should understand only those instructions that are described in its 'language'.

As the simulator engine finished, the simulation manager collects the results from certain files generated by the engine and converts the data to VDM format (see 4.3.1.1). This format is then serialized and sent back to the GC.

3.2.4 Simulator Architecture

In the following sections the main components of the simulator are introduced as *Scheduler*, *Node*, *Model*.

3.2.4.1 Scheduler

An object of the *Scheduler* class is responsible for running a simulation scenario. After the scenario is built up based on the configuration file, the execution is carried out in a loop of the following steps.

1. Pick up a node.
2. Initialise the node.

3. Execute the node.
4. Propagate the results of the execution

By calling the *init* method of the node the scheduler let it prepare for the execution. Execution means that the node (more precisely the (input/output)model) processes the time series describing the load and produces time series describing load for the successive nodes. Propagation means that these produced time series get set as input at the successive nodes.

A node should be executed only if the time series of load is present. This is either defined in the configuration file or computed from the output of other nodes. This reveals that prior execution the *Scheduler* should queue the nodes into an appropriate order to let those time series be computed that are not available in the beginning.

The ordering algorithm starts from nodes whose inputs are defined. These will be the first nodes to execute. In the next turn those nodes will be executed that has already got their inputs from the nodes in the first turn. And so on. The simulation cannot cope with circles in traffic but since this would not be a realistic scenario we can exclude this case.

After all the nodes was executed the *Scheduler* gathers the state information of nodes that were listed in the question section of the configuration file, and compiles the output XML.

3.2.4.2 Node

The *Node* class is a framework for the traffic handling models. Each node contains two models, one to handle the traffic that enters to the domain and one to handle the one that leaves it. The models are dynamically loaded as the node is created.

Each node has two registers for neighbouring nodes. One for nodes in the same domain and one for nodes in neighbouring domains. The class provides two different method to register the different neighbours. If any of these registers contain more than one neighbour, then a traffic distribution file should be specified to that register. These traffic distributions tell the node how to divide the outgoing or the incoming traffic among the appropriate set of neighbours.

After all the topology and routing (traffic distribution) information is specified (at configuration time), the *Scheduler* invokes the *init*, *process* and *propagate* methods. The *init* method initialise the appropriate (input/output) model. The *process* method opens all the input files for the model that should be executed and executes is. In the end the *propagate* method sets the output files as input to the appropriate neighbours.

3.2.4.3 Model

This class is the abstract base class of any node model. It defines a common functionality like setting the input (*setInput()*) for this model, initialising (*init()*) and executing the model (*process()*).

Since the load (what is the input for a node model) is stored in temporary files, setting the input means that the file name of the appropriate time series has to be handed to the model.

On initialization the model opens an output and a state file. These files will be filled during the execution.

The derived classes of this class differ mainly in the executing method. This is where the model behaviour should be defined. To pass model parameters the *process* method defines a string parameter. This string can be parsed by the model specific code to set the parameters. The rest of this method is usually a loop that consumes the input file and in every step computes and outputs the QoS state of the node to the state and output files. Every model fills as many output files as many inter-/intra domain neighbours the containing node has. (Input models – intra-domain, output models – inter-domain neighbours.) The distribution of the module output among the output files are defined in the traffic distributions.

Since the time series of traffic and traffic distribution is processed only by models the format and the contained information of these files can vary from model to model.

The implemented models:

- NullModel: This simply copies the input to the output introducing only a static delay and distributing the output among the neighbours. This is used as input model since the inside of a domain is supposed to be ideal except a static mean delay.
- LeakyBucket: This models a simple FIFO queue with limited server capacity and queue length. It handles aggregate load information and produces throughput, drop, delay and jitter state information (the latter two at low accuracy). Parameters for the model are: server capacity, maximal queue length, actual queue length.
- Class4LeakyBucket: This models the same queue as the LeakyBucket model but this handles aggregate and packet level load information too. It produces the same QoS state as the LeakyBucket model but at a higher accuracy. Basically it implements the same queuing model as the LeakyBucket but while packets are queued and served as expected, the aggregate traffic information is used to modify the model parameters such as service rate, total queue length. The concept is that the traffic is modelled as aggregate background traffic and the (probably important) long packets [Hof03]. This queuing model will queue the long packets and serve them as a normal queuing system, while the aggregate traffic information will decrease the service rate and increase the queue level, hence increase the drops and the delay. The model parameters are: server capacity, maximal queue length, actual queue length, background traffic mean packet length.

3.3 Fluid Simulator

3.3.1 Introduction

The Rate and Time Continuous Fluid Simulation (RTC-FSIM) is a novel fluid simulation approach developed within the InterMON project. In contrast to other fluid-based modelling techniques which require an event-based simulator for the execution of the model, the RTC-FSIM approach models all traffic as continuous signals and describes the signal transformations by means of differential equations. The function of the RTC-FSIM simulator is to solve these differential equations that model the system behaviour. A detailed description of the RTC-FSIM approach can be found in [D6], [Ber02a], [Ber02b], [Ber02c].

3.3.2 Integration of IPFIX Measurements

One of the main tasks concerning the integration of the simulator into the InterMON platform is the automated generation of the simulation model and its traffic inputs. The traffic inputs are derived from the main passive monitoring functionality implemented in InterMON, namely the IPFIX flow meter [D8]. It enables the measurement of flow volumes within a configurable time interval. IPFIX provides a very flexible flow definition [IPFIX] which allows for traffic measurements at very different granularities. A flow is simply defined as a set of packets that have a common set of properties, e.g. packet header fields. In this sense, a flow pattern (i.e. the set of common properties) can be defined such that it matches the total traffic of a link; on the other hand, the flow pattern can be very specific and match only some packets of an application level flow (e.g. only the SYN packets of a TCP connection between two applications).

At the end of a reporting interval, the IPFIX meter sends the measured flow information to a data collector. The data collector then writes the received information into the InterMON database. The IPFIX traffic flow table is specified in [D10, Section 3.5]. It is important to note that at the end of the following reporting interval the new measurements of a flow are stored as a new record in the flow's data base table, i.e. the previous measurement is not overwritten with the new one. The number of bytes / packets (table entries: NUM_BYTES / NUM_PACKETS) are stored in a cumulative way, i.e. summed up from the beginning of the measurement task.

In the following we describe how the IPFIX traffic measurements can be used for the generation of the RTC-FSIM input signal(s).

Basically, each IPFIX flow is subdivided on the time axis into the reporting intervals. For each reporting interval, the number of bytes (packets) is measured. Our approach is to interpret this measurement series as the output of an ON/OFF traffic source. In each reporting interval, the flow can be either in OFF state (i.e. no packets were measured) or in ON state. From the number of bytes sent in an ON interval, an average rate of flow *i* in reporting interval *j* can be derived as

$$rate_{i,j} = \frac{NUM\_BYTES_{i,j} - NUM\_BYTES_{i,j-1}}{TIME\_LAST\_PACKET\_OFFSET_{i,j} - T_j}.$$

The variable names are equal to those used in the database table description of an IPFIX data flow from [D10, section 3.5]. Additionally, T_j denotes the time instant when the reporting interval *j* started. For the first reporting interval, i.e. j = 1, T_j should be set as TIME_FIRST_PACKET_OFFSET.

In Figure 3.7 the approach is illustrated. On the y-axis the different flows are shown: if a flow sent data during a reporting interval (denoted by the dashed red vertical lines) then a solid black horizontal line is drawn according to the duration of the ON time. The gaps between the black lines represent the OFF periods for that flow.

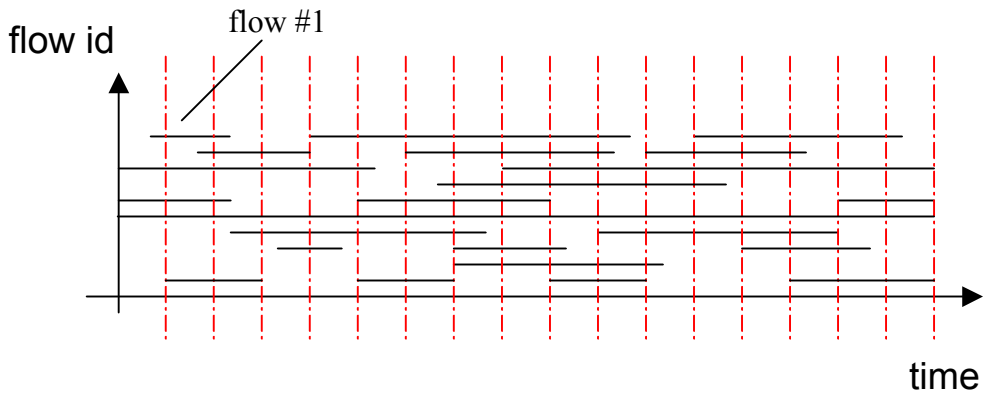


Figure 3.7 - ON/OFF states of 10 flows

If we select one of these flows, e.g. flow #1, and look at the rates during each reporting interval, the result could look like Figure 3.8.

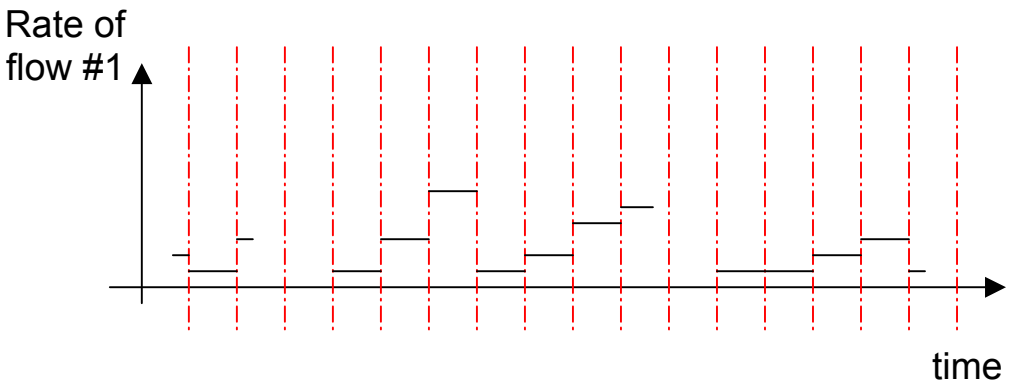


Figure 3.8 - Rates of flow #1

Such a time series is created from each flow measurement. In order to generate a signal that represents the total load of the measured interface, a superposition of these ON/OFF sources is built. The resulting sum could look like Figure 3.9.

It is described below in 3.3.3 how this time series is used as an input signal for the fluid simulator.

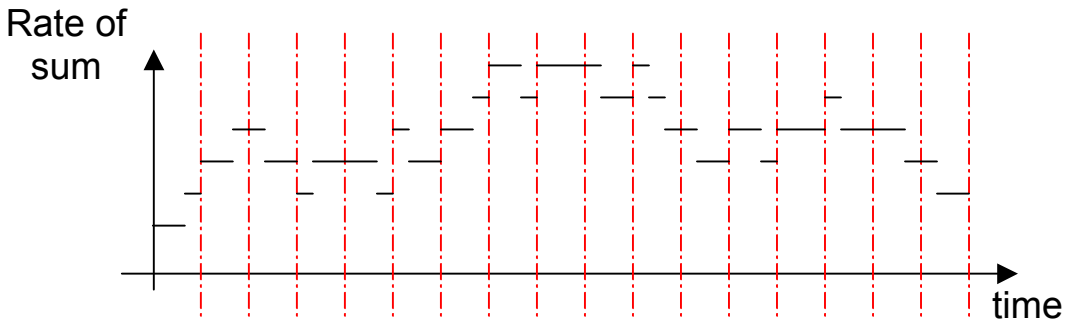


Figure 3.9 - Superposition of the 10 ON/OFF flows

3.3.2.1 Measurement Requirement

The correctness of the above sketched approach can only be guaranteed if an important condition is fulfilled: each packet must be accounted for exactly once.

In the simplest case, a useful set of patterns could consist of a single flow pattern that matches on all traffic passing the interface. This would result in a single ON/OFF flow.

If there are packets that do not match on any flow pattern then the IPFIX flow measurement captures only a portion smaller than 100% of the total traffic. On the other hand, if a packets are accounted for more than once then this would be a distortion of the real traffic passing the interface.

It may, nevertheless, be necessary to define the flow patterns such that a single packet matches more than 1 flow pattern. As an example, assume that a user is interested in a question like: “What consequences would it have (in terms of QoS parameters like delay, packet loss) if all traffic going to destination X would be routed over AS B instead of AS A. In that case a useful set of IPFIX flow patterns would consist of 2 filters, one that matches on all packets with a destination IP address of X and a second filter that matches on all other packets. This would require the ability to define a filter with a logical “NOT” rule, e.g., destination IP address NOT equal X. However, the current implementation of the IPFIX meter does not allow such a rule. As a workaround, the filters could be defined such that:

- Filter 1 that matches on all destination IP addresses
- Filter 2 that matches only on destination IP address X

In that case all packets with a destination of X are accounted for in both filters. If, in each reporting interval, the number of bytes measured in Filter 2 is subtracted from the number of bytes measured in Filter 1, the desired results can be obtained. Note that it is necessary to a) start the measurement rules at the same time and to b) use the same reporting interval for both rules.

While it is possible to perform this subtraction at the collecting station it is not strictly necessary to do so. The collector may simply save the measured values of each flow to the database. It is sufficient to perform the subtraction after the user defined the simulation scenario and before the traffic measurements are passed to the simulation.

3.3.3 RTC-FSIM Implementation

3.3.3.1 Current status

The RTC-FSIM approach is completely implemented in a Matlab / Simulink environment. The implementation provides components for a multi-class server station with and without priorities. These components are provided in a separate file as a Simulink library from where they can be copied into a new simulation scenario. Currently, the transformation of a topological description into the Simulink object graph has to be done by hand. The IPFIX measurements are already fully integrated into the simulation environment. Note that so far we used artificial values as no real measurement data was available until now.

In the Brussels demo version the RTC-FSIM was integrated into the InterMON toolkit with a simple measurement-based interface.

3.3.3.2 Planned integration

After a simulation scenario has been defined via the GUI, a simulation request is sent from the simulation manager to a java wrapper object that's responsible for the fluid simulator. The RTC-FSIM wrapper transforms the XML-based description of the simulation task into a sequence of commands that can be executed by simulator.

In case of the RTC-FSIM, the simulator is implemented in Matlab / Simulink [MATHW]. Therefore, the output of the transformation is a Matlab script.

3.3.3.2.1 Traffic

The scenario description contains a section on the input traffic which is the superposition of ON/OFF flows as described above. The wrapper transforms this time series into a simple ASCII file with two columns: a time value in the first column and a rate value in the second column. If the scenario contains descriptions of several flow aggregates then this step is repeated for each flow superposition.

Besides the creation of the text file(s) containing the time series, commands that read from these text files are added to the Matlab script. Basically, the time series is read into a Matlab internal array which is then used as the input signal to a server station.

3.3.3.2.2 Topology

The topology information that is sent in the simulation request is automatically transformed into the Simulink object chart. The already available components for a server station are used as the main building block. In the transformation process, the topology information is mapped to Matlab commands that build up the Simulink model in a non-interactive way.

3.4 INTER-IP

The INTER-IP tool evaluates the end-to-end packet or volume transfer delay performance for a traffic relation (i.e. a flow identified by the source and destination IP address and by the service class if differentiated service is used) that crosses multiple domains in a Differentiated Services context.

A suggested application in the InterMON context could be the rapid comparison (on the base of the delay metric) of alternative routes for the same flow.

The delay calculation is totally analytical and based on queuing models for the links. Each link is modelled by a one server, multi-class queue with a strict priority class and up to five classes serviced in a round-robin way (this model is similar to the MDRR scheduling discipline implemented on CISCO giga-routers [CISCO]).

The queue model is derived from the M/G/1 priority queue [Kle76] model with some modifications for the classes serviced in a round-robin way. The packet delay is computed only for the traffic in the strict priority class; it is assumed that in this class are mapped delay-sensitive applications like Voice-Over-IP or interactive applications. The traffic mapped on the other classes is supposed to be of type "bulk" (i.e. applications based on large volume transfer, generally using TCP) and the model allows for computing the volume transfer delay. The model takes into account that the M/G/1 priority queue is an inaccurate model for GPS scheduling because the absolute priority doesn't reflect the scheduler behaviour. The model considers also the nature of TCP traffic neglecting the packet level because the end-user perceives the network performance relative to the flow level.

The formula resolving this model is in closed form and had been validated using simulations [P1115]. This allows for a practically real-time performance evaluation.

The end-to-end delay is derived from the composition of the delays experienced on each link crossed. The mean delay is computed using the additive property for the average delay.

The tool has two components that are described in the paragraphs below.

3.4.1 Interd

This module has in charge the computation of the end-to-end delay performance for a certain number of traffic paths (identified by a list of couples AS/Border Routers crossed).

The average end-to-end delay is obtained as the sum of the average delays evaluated for all the hops belonging to the path that are:

- inter-domain links: in this case the delay performance is evaluated using the queue model described above; each class of traffic on these links is characterized by the average length of the packets for the strict priority class and of the files for the other classes[2];
- intra-domain hop: in this case the delay is directly available, as the delay of each class for each BR-to-BR traffic relations (per-class delay matrix) is directly supplied to the module. The intra-domain delay is considered constant and independent from the load of the network crossed. This assumption has been made because in an inter-domain context the bottlenecks, due to the large cost of the bandwidth in the inter-domain connections, are located on the links interconnecting two different domains. As a consequence, the intra-domain delays are probably negligible, relatively to the end-to-end delay, and the error made by assuming these delays as independent from the load of the network crossed is not too large.

A particular case of intra-domain hop regards two components of the end-to-end delay: the delay from the source to the egress router in the originating domain and the delay from the ingress router in the destination domain to the destination of the traffic. The problem with these components is that a precise evaluation needs information about internal topology, routing, link loads and scheduler weights for the two domains. If this information is available, the module Intrad described in the next section could be used. But for the same reasons seen above it is possible to neglect these components or to use a fake value calculated from the delay-matrix in such a way to be representative of a generic "internal" (to the source or destination AS) delay.

The intra-domain delay matrix can be measured (using the MonRes or Ipfix modules features) or calculated using the module described below.

3.4.2 Intrad

This module has in charge the per-class delay matrix computation. The per-class delay matrix is also needed as input of the module Interd described above.

[2] The assumption is that it is possible that the mapping of the traffic on the classes is unique for all the domains crossed

The input data are listed in 4.1.3.1; since the Autonomous System administrators will probably not share a part or even all of them, it should be possible to have them stored in a dedicated section of the local database.

3.4.3 INTER-IP Integration

The Interd and Intrad module are binaries executables (compiled C language) called by a Java class, which is driven by the Simulation manager via XML-RPC. The communication with the GUI and the Global (or Local) Controller is performed by the JMS messaging system. The design is depicted in Figure 3.10.

From the user's perspective, the simulator is configured in the GUI inside the INTER-IP requester module, which pops-up when a simulation request is performed from the "Modelling and Simulation" menu.

The Simulation manager can retrieve the missing data of the scenario specified by the requester directly from the InterMON database. In case such data were not available, it is planned to achieve a level of integration such that features of other InterMON modules can be used; for example:

- the list of alternative routes for a traffic relation could be supplied by the topology visualization tool;
- information about the average volume size transferred in a connection (traffic class characterization) could be supplied by the Ipfix tool;
- information about link loads could be supplied by the MonRes and/or Ipfix tools

The Simulation manager can also configure a what-if analysis by elaborating the actual data and computing a new set of inputs resulting from the changes requested by the user in the INTER-IP requester module.

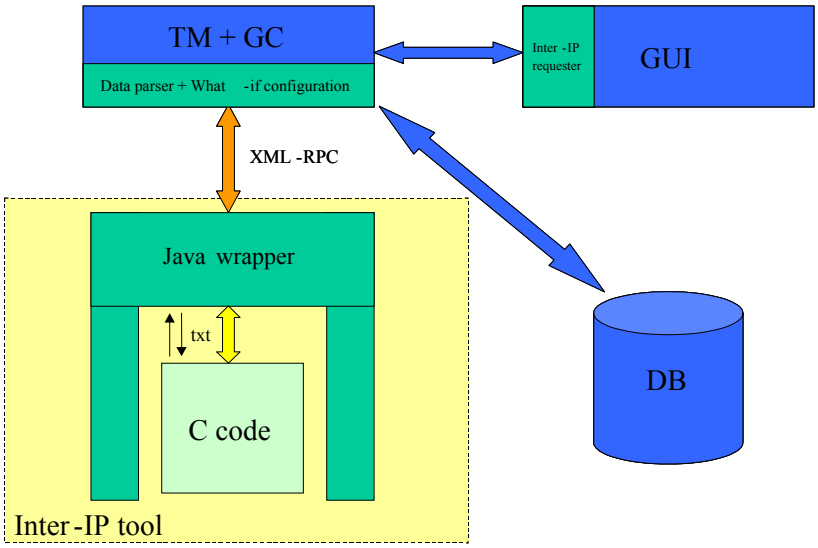


Figure 3.10 – Inter-IP integration

3.5 Generic Simulation Control Module for the Client GUI

In order to configure and start new simulation tasks a common GUI component is implemented. This component is build in the form of an application plug-in module[3] for the InterMON GUI. It is available in the 'Tools/Modelling and Simulation' section of the GUI menus. This module implements a common front end for the different simulations yet it is flexible enough to allow specifying additional parameters specific to each type of simulation.

[3] technically: a class derived from BaseApplication

The Simulation configuration front end has three main dialogs which are organised into three tabbed panes:

1) a generic dialog to enter common parameters like task ID, task description and start time and also select the type of simulation which shall be performed (packet based, flow based, analytical, etc.)
2) a dialog where the user can choose from a list of stored network topologies (collected by the topology toolkit) and also attach configuration changes to elements in the network by clicking on elements inside the network view (e.g. change link bandwidth or delay, configure router attributes)
3) a simulation specific dialog in which the user can enter additional parameters specific to the currently selected simulation type, e.g. simulation granularity or exported items. This dialog is supplied by the integrator of that simulation type.

Figure 3.11 shows a screen shot of this module (first dialog shown):

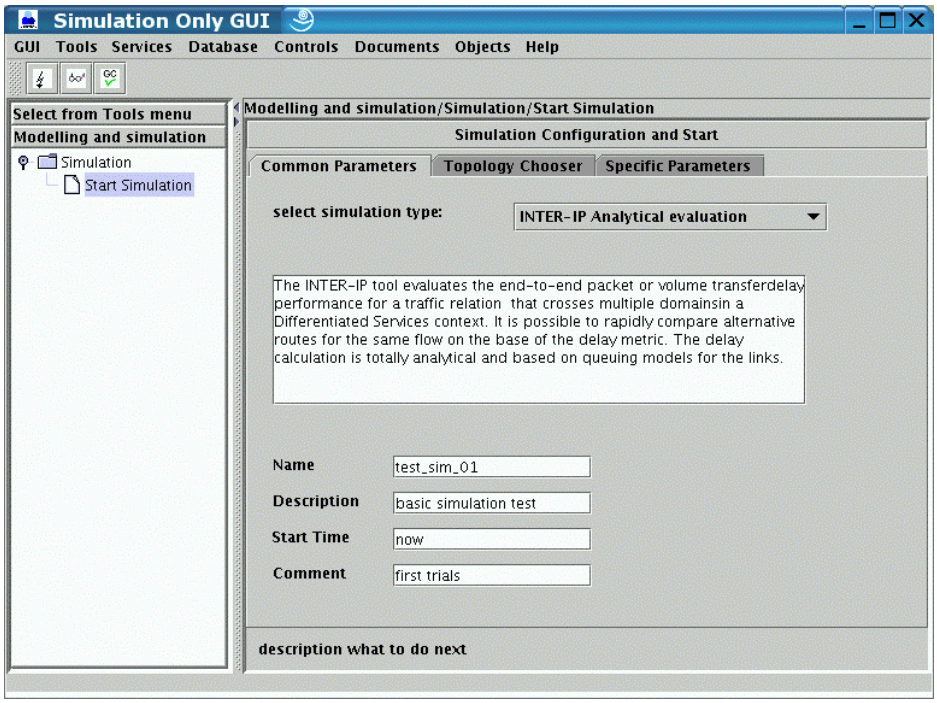


Figure 3.11 - Simulation control GUI

The user first needs to select a type of simulation from the drop down box. He can also view a description of the selected simulation type then to see if this type fits his simulation requirements. when a simulation has been selected, generic parameters are queried. The user's selection determines which parameters can be given to the simulation on the third tab and which attributes can be set on network elements in the second tab.

When the user selects 'submit simulation task' (on the third tab) all entered parameters are collected and put into one simulation start request message (written in XML). This message is then forwarded to the Global Controller where the GUI has registered. The GC will give the request to the appropriate simulation then (for details see Chapter 2).

4 Data Formats and APIs

4.1 Simulator-Internal Formats

4.1.1 Formats and APIs used in Hybrid Simulation

4.1.1.1 Format of Hybrid Simulation Model XML Descriptions

The plug-in modules for the ns-2 network simulator used in the hybrid simulation approach can all be configured using an XML file, or a collection of XML files in the case of a multi-domain model. This section describes the detailed format of all currently implemented models and submodels from a top-down perspective.

4.1.1.1.1 Basic structure

Every hybrid approach model description follows a basic XML format described in 4.2.1 to allow for better handling in databases. The remaining model definitions in the hybrid family have the following structure:

```
<modinfo>
      <name> Model name </name>
      <comment> Optional comment </comment>
</modinfo>
<model category=(domain|idlink|trafficgen|multi) type=?>
      […]
</model>
```

The fields under the `modinfo` tag are used to identify the described models in repositories, databases or graphs, for example. Model names should be short but descriptive so they can be used in automatically generated graphs. Examples for good names are “AS51423” and “Domain A”. Comments may contain further information about the described model, for example its creation date, the kind of measurements used to parameterize it or more detailed information about the modeled network entity. We deliberately leave this for the users to decide.

The actual model description is stored under the model tag, which is characterized by the attributes `category` and `type`. `category` designates which kind of model is used (either a domain model, an inter-domain link model, a traffic generator or a multi-domain model) and `type` selects a specific model in a category. In the case of traffic generators this chooses between different kinds of modeled traffic aggregates, for example. If a category only has one type of model this parameter may be omitted.

4.1.1.1.2 Multi-Domain Models

Multi-domain models are containers for all other types of models (we speak of a multi-domain model’s submodels). From the perspective of the ns-2 plug-in mechanism they do the interfacing between ns-2 and model space. Name translation is an important part of this task. From the perspective of model space they control calculations concerning more than a simple model, like load distribution over more than one domain model or cumulative delay along a path.

There are four parts (and thus four main tags) to a multi-domain model description, one to configure the interface towards ns-2 (`border` tag), one to include the submodels (`submodels` tag), one to connect them all together (`topology` tag) and a fourth one defining the routing paths along the models (`routing` tag). The following sections each describe one of these parts.

Border Section

Interfacing between the event-based world of ns-2 and the analytical models used in the plug-ins requires some configuration. To be able to map a packet event of ns-2 to a path in the multi-domain model we need to specify the number of inbound and outbound links, as well as their counterpart nodes in ns-2. Unfortunately, node names in ns-2 can only be determined at runtime. Therefore, we need two mappings, one between the model's link indices and symbolic names and one between these symbolic names and ns-2 runtime names. The first mapping is configured in the XML model description. The corresponding section in the XML file takes the following form:

```
<border inlinks=number outlinks=number>
      <inbound link=number> Symbolic name </inbound>
      …
      <outbound link=number> Symbolic name </outbound>
      …
</border>
```

The attributes `inlinks` and `outlinks` of the `border` tag specify the number of inbound and outbound links, respectively. Then a list of symbolic names follows, one for every link. The `inbound` tag maps the given symbolic name to the inbound link specified with the `link` attribute. The `outbound` tag works analogously for outbound links. Note that missing mappings can either trigger an error or cause all packets on related paths to be dropped. Also, the same symbolic names can be used with multiple link numbers.

Submodels Section

In this section, surrounded by the "`submodels`" tag, the models contained in the multi-domain model are listed. Each list entry has the form `<type>`*filename*`</type>`, where type is either `domain`, `idlink` or `trafficgen`, for domain models, inter-domain link models and traffic generators, respectively. The models of each type are numbered internally according to their appearance in the list, starting with zero. Entries of different types can be freely mixed. The file names can also be absolute or relative path names. In fact, submodels of a multi-domain model "mdmodel.xml" will be usually stored in a subdirectory "mdmodel". A typical submodel specification of a file "MDM" would look like this:

```
<submodels>
      <domain> MDM/domain_A.xml </domain>
      <domain> MDM/domain_B.xml </domain>
      <idlink> MDM/idlink_AB.xml </idlink>
      <idlink> MDM/idlink_BA.xml </idlink>
      <trafficgen> MDM/tgen_1.xml</trafficgen>
      <idlink> MDM/idlink_1A.xml </idlink>
</submodels>
```

This would yield the submodels of the multi-domain model shown in Figure 4.1. The file pointed to by a list entry must contain a model description of the correct type.

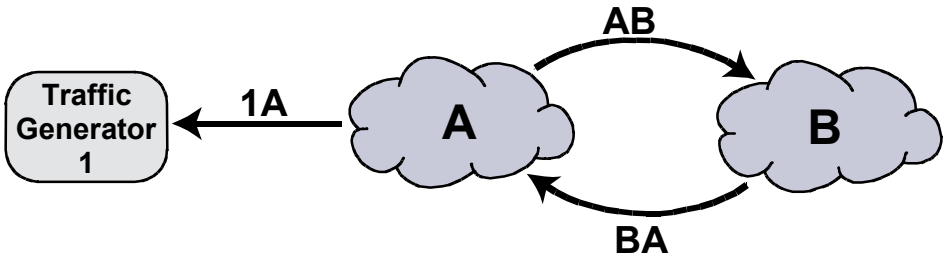


Figure 4.1 - Example multi-domain configuration

Topology Section

The topology section of multi-domain models specifies the connections between the models loaded in the submodels section and between these models and the inbound and outbound links of the multi-domain model. Connections are defined by specifying a source connector, a destination connector and the identifier of the inter-domain link between. A special format is used to refer to connectors:

D:<id>:<c>	The connector <c> of the domain model with index <id>. These can be specified as source or destination connectors.
T:<id>	The traffic generator with index <id>. Traffic generators can only be used as source connectors.
I:<c>	The inbound link <c> of the multi-domain model. Inbound links can only be source connectors.
O:<c>	The outbound link <c> of the multi-domain model. Outbound links can only be destination connectors.

These connector selectors are used on the `connect` tag, which look like this:

`<connect from=`*source connector* `to=`*destination connector*`>`*inter-domain link ID*`</connect>`

This connects two models (or inbound/outbound links) using the inter-domain link model specified by its numeric identifier. The topology section of a multi-domain model, below the `topology` tag, consists of a complete list of `connect` entries.

Routing Section

While the topology section defines the connections between the models contained in a multi-domain model, the paths packets take along these connections must be configured separately. This is done with a list of `path` entries below a `routing` tag. These take the form

`<path from=`*number* `to=`*number*`>` *IDLINK_ID* … *IDLINK_ID* `</path>`

to specify the path between inbound and outbound links given by their numeric identifiers, or

`<path type="tgen" from=`*number* `to=`*number*`>` *IDLINK_ID* … *IDLINK_ID* `</path>`

to specify the path between a traffic generator and an outbound link, again given by their numeric identifiers. The path itself is defined by a list of numeric inter-domain link identifiers. This section must include all possible inbound/outbound combinations and a path for every traffic generator of the multi-domain model.

4.1.1.1.3 Domain Models

Domain models represent autonomous systems or generic network clouds inside the analytical model. In contrast to earlier work done in the InterMON project they don't play a significant role in the simulation of load distribution anymore. They now merely serve to connect a number of inbound inter-domain link models to a number of outbound ones. Also, delay specifications have changed from earlier versions. Delays are now mainly simulated using empirical cumulative distribution functions (ECDFs), which can be directly derived from delay measurements over actual network domains. There are three levels of detail for delay simulation depending on the quality of measurement data. If only a single delay value can be obtained – when using `traceroute` for example – the domain model can be configured to have a fixed delay on all its paths. In the case of a single time series the corresponding ECDF can be calculated and the domain model will yield equally distributed delays on

all its paths. The most detailed representation of domain delay is the case where delay measurements of every path are available. Then, an ECDF will be calculated for each one of these paths.

After the common header for InterMON models of the hybrid family, the `model` tag starts the domain model description with the `category` attribute set to "`domain`" and the `type` attribute being ignored, because there currently is only one type of domain model. Next, the two tags `inbound` and `outbound` are expected, each containing positive integer values and giving the number of inbound and outbound inter-domain links connected to the. The main part of a domain description is placed under the `depred` tag (delay predictor tag), which includes a `type` attribute to select one of the three delay models. This attribute can take the values `"fixed_value"`, `"single_distr"` or `"multi_distr"`. In the fixed value case the value of the `depred` tag is a single floating point value. In the case of a single ECDF a white-space separated list of floating point values is expected, which are then used to parameterize the distribution. The most complicated case is the one where several distributions are concerned. It looks like this:

```
<depred type="multi_distr">
      <path from=number to=number>
            FLOAT … FLOAT
      </path>
      …
</depred>
```

The `path` tag has the two attributes `from` and `to` that select the concerned inbound and outbound links and thus uniquely define the path across the domain. Again, a sequence of floating point values is used to specify the distribution. To be valid, a multiple distribution delay predictor has to contain distributions for every path, i.e. a `path` entry for every from/to combination.

4.1.1.1.4 Inter-Domain Links

Inter-domain links play an important role in multi-domain models. They serve as logical connections between other submodels (configured in the topology section of multi-domain model descriptions), as basis for the calculation of load distribution and packet loss ratios and as simulators of the "dynamic part" of delay.

Of the several types described theoretically in earlier work only one (the default queuing model) is currently implemented. It can be configured using a remarkably small set of parameters, namely the capacity *K* of the simulated queuing system.

`<K>`*positive integer*`</K>`

which influences the loss ratio and the maximum delay caused by the link (bigger values of *K* yield smaller loss and greater delay), and the link rate

`<rate>`*bandwidth spec*`</rate>`

whose value has the format (<INT>K | <INT>M | <INT>G | INF) for kbit/s, Mbit/s, Gbit/s capacities or infinite capacities (upper-/lowercase does not matter). Infinite bandwidth links serve as purely logical links without packet loss or delay. They are mainly used between the inbound and outbound links of a multi-domain model and the internal submodels, since the behaviour of these links is already simulated by ns-2. Internally, bandwidths are stored in floating point kbit/s values.

4.1.1.1.5 Traffic Generators

It is very difficult to define a meaningful common subset of configuration parameters other than the one already included in multi-domain model specifications. Consequently, the only common part of traffic generator specifications is the root tag

```
<model category="trafficgen" type=?>
      …
</model>
```

The `type` attribute selects one of the available traffic generators. Currently, there are only two values to choose from, `"tgen_null"` for the Null traffic generator, which has no further tags or attributes, and `"tgen_http"` for the http traffic generator . There are two parameters to the http generator, the number of sources and the direction of simulated traffic, which can be either "client to server" to generate the traffic behaviour of client requests, or "server to client" for the generator to model the http server's replies to the clients. These parameters can be configured using the following tags:

`<sources>`*number*`</sources>`

for the number of clients modeled and either

```
<direction>cli->ser</direction>
```

or

```
<direction>ser->cli</direction>
```

to select the direction of the modeled traffic.

4.1.1.2 NS-2 External Module API

This section described the format of external modules and how to implement them. External modules are executable object files. They have to be written in C++ and are derived from a class "ISP_module". These modules then can be loaded dynamically into ns and attached to single nodes. Within a single ns topology setup, different nodes may have different modules attached. For the loading the dlopen mechanism is used.

4.1.1.2.1 Loading of External Modules

The concept of loading executable objects is simple and follows these steps:

1. The file with the executable object is loaded via the dlopen() function. The object has to be derived from the ISP_module class.
2. A pointer to an initialization function within the loaded object is obtained by performing a symbol lookup with the dlsym() function. This initialization function simply calls the constructor of the object and returns the reference to the new instance. The symbol required for the lookup is defined with the *`ISP_MODULE_INIT(<classname>)`* macro.
3. The init function is called, constructing an instance of the obejct defined in the external module file.

The base-class ISP_module looks like:

```
class ISP_module
{
  public:
     ISP_module(const int n);
     virtual ~ISP_module();
     virtual double process_packet(int prev_node_id,
        int next_node_id, double ti, struct ISP_pinfo *p);
     virtual int command(const int argc,
        const char*const* argv);

 protected:
     int node_id;
};
```

Since all functions are virtual, an derived class only has to provide the constructor. Of course that a external module will have no further functionality. The *process_packet(...)* function, is called for each packet received by a node. This function decides, for how much time a packet is delayed or whether a packet is dropped. Any positive-zero value causes a packet delay (in secs), a negative value causes a packet drop.

The ISPDelay object provides information about the last visited node, the next node, the simulation time and the information about the packet. The nodes are specified by their ns-number (see also the nodenames file). The simulation time is in secs. The ISP_pinfo structure will be explained later.

4.1.1.2.2 Passing Commands to external modules

The command function is used to pass commands from the ns tcl frontend to the external module. A simple argc-argv mechanism is used to pass the parameters to a *command(...)* function, which can provided by the external module. Argc is the number of tokens on the command line and argv is a vector of char pointers to the single token strings. The *command(...)* function should return 0, if the command was successful. Any other value is interpreted as an error code. To configure an external module the command

```
$o modcmd [t1] [t2] [t3] [t4]
```

is used. $o is the reference to the ISPDelay object, which has the module attached.

4.1.1.2.3 The "struct ISP_pinfo" type

This structure is used to pass information about the packet to the external module. The structure is defined as follows:

```
struct ISP_pinfo
{
    union {              // if ipv4, address is in [0]
        int32_t v4;
        int32_t v6[4];
    } src;

    union {
        int32_t v4;
        int32_t v6[4];
```

```
        } dest;

    int32_t size;
    int32_t ttl;
    int32_t protocol;           // not yet implemented
    int32_t pkt_type;           // the ns packet type

    int32_t flowid;             // not yet implemented
    int32_t prio;               // not yet implemented
};
```

Even if the ISP_pinfo structure does support IPv6 and IPv4, the packet handling for IPv6 and IPv4 packets within ns is rather similar. The src and dest fields, which are capable to store either a IPv4 or IPv6 address are used in a ns manner and simply store the number of the source and destination node as a simple integer. The size and ttl fields have the same meaning as in the original IPv4/v6 headers. The protocol field is currently not used, since ns does not maintain such a field in its internal IP packet header. As an alternative ns provides a pkt_type field, which provides information about the packet type. As can be seen the pkt_type is used much more general than the protocol id in the IP header. The following table lists the values defined within ns.

#no	Traffic-type	#no	Traffic-type
1	TCP	26	REJECT
2	UDP	27	TELNET
3	CBR	28	FTP
4	AUDIO	29	PARETO
5	VIDEO	30	EXP
6	ACK	31	INVAL
7	START	32	HTTP
8	STOP	33	ENCAPSULATED
9	PRUNE	34	MFTP
10	GRAFT	35	ARP
11	GRAFTACK	36	MAC
12	JOIN	37	TORA
13	ASSERT	38	DSR
14	MESSAGE	39	AODV
15	RTCP	40	IMEP
16	RTP	41	RAP_DATA
17	RTPROTO_DV	42	RAP_ACK
18	CtrMcast_Encap	43	TFTC
19	CtrMcast_Decap	44	TFRC_ACK
20	SRM	45	PING
21	Request	46	DIFF
22	ACCEPT	47	RTPROTO_LS
23	CONFIRM	48	LDP
24	TEARDOWN	49	GAF
25	LIVE	50	REALAUDIO
		51	PUSHBACK

Most of the values defined within ns do only make sense in a special environment or in combination with very specific ns2 extensions. However, packet or traffic types like TCP, UDP or CBR can be used to differentiate between packets.

4.1.1.2.4 The "nodenames" file

The node ids, which are used within the trace files and also by the API to the external module are set by ns. Since the external module has to know, which node got which ids a "nodenames" file is written, simply listing line by the line, the id of a node with the label provided to the IM-node command. A typical nodenames file looks like:

```
0 Knoten-0
1 Knoten-1
2 Knoten-2
3 Knoten-3
```

The first column represents the node id the second column the label of the node.

4.1.1.2.5 Compiling external modules

There are a few special compiler and linker commands necessary to compile the external modules. Most of the flags are rather standard. The *-fPIC* flag is required to compile position independent code, the *-shared flag* takes care of linking the executable object. The following table shows the Makefile used to compile the sample module, described in the next section.

```
INCLUDE = -I../ns-2.26
CFLAGS = $(INCLUDE) -O6 -Wall -fPIC
LD  = g++ -shared
CPP = g++ -c
all: isp_test.mod
isp_test.o: isp_test.cc isp_test.h
        $(CPP) $(CFLAGS) isp_test.cc -o isp_test.o
%.mod: %.o
        $(LD) $< -o $@
        cp $@ /tmp
clean:
        rm -f *.mod *.o *~ nodenames
```

4.1.1.3 A Sample Module

The next paragraphs show the header and the source files of a simple external module.

```
// The header file of the sample module

#include "../ns2/ispmodule/isp_module.h"

class ISP_test : public virtual ISP_module
{
  public:
    ISP_test(int n);
    double process_packet(int prev, int next, double ti, struct ISP_pinfo *p);
    int command(const int argc, const char*const* argv);

  private:
    double delay;
};
```

```
// The source file of the sample module
#include <stdio.h>
#include "isp_test.h"

ISP_MODULE_INIT(ISP_test)

ISP_test::ISP_test(int n) : ISP_module(n)
{
    delay=0.1;          // default delay
    return;
}

double ISP_test::process_packet(int prev, int next, double ti, struct ISP_pinfo *p)
{
    return delay;       // just return the delay value
}

int ISP_test::command(const int argc, const char*const* argv)
{
    if(argc==2 && !strcmp("delay",argv[0]) && atof(argv[1])!=0) {
        delay=atof(argv[1]);    // convert string to float and set new delay
        return 0;               // okay
       }
    return 1;           // error
}
```

ISP_MODULE_INIT(ISP_test) is a call of the macro, which is necessary to provide the symbols for the dlopen mechanism used during loading the module. The argument has to be the classname of the derived class. The example above simply adds a static delay of 0.1 seconds to each packet passing the node. This delay can be altered by a command "delay". Assuming this module is attached to a node $n, the command to set the delay to 0.2 seconds would be something like:

[$n entry] modcmd delay 0.2

The module commands (everything after the modcmd token) only depends on the external module.

4.1.2 Formats used in Time Series Simulation

The simulation engine is configured by an XML file. This file describes the scenario the simulation engine should run. The description of the XML scheme follows.

4.1.2.1 Scenario

A scenario tag defines the topology *(NetworkDescription)*, the traffic *(Source)* and the routing *(TrafficDistrib)* tags. Inside these tags are defined the set of nodes and links that forms the simulated network, the load on these nodes and links and the distribution of the traffic inside the domains respectively.

```
<xs:element name="scenario">
        <xs:complexType>
            <xs:sequence>
                <xs:element maxOccurs="1" minOccurs="0"
                 name="networkDescription" type="NetworkDescription"/>
                <xs:element maxOccurs="unbounded" minOccurs="0"
                 name="source" type="source"/>
                <xs:element maxOccurs="unbounded" minOccurs="0"
```

```
                name="trafficDistrib" type="trafficDistrib"/>
               <xs:element maxOccurs="unbounded" minOccurs="0"
                name="question" type="question"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
```

4.1.2.2 NetworkDescripition

The *NetworkDescription* tag describes the network as a set of autonomous systems *(tsismAS)* and links between these systems *(iDContact).*

```
<complexType name="NetworkDescription">
    <sequence>
        <element name="description" type="xs:string"/>
        <element maxOccurs="unbounded" minOccurs="0" name="as"
         type="tssimAS"/>
        <element maxOccurs="unbounded" minOccurs="0" name="iDContact"
         type="iDContact"/>
    </sequence>
</complexType>
```

4.1.2.3 TssimAS

All autonomous system is identified by an *id* attribute. Autonomous systems are modelled by border models.

```
<complexType name="tssimAS">
    <sequence>
        <element name="description" type="string"/>
        <element maxOccurs="1" minOccurs="0" name="borderModel"
         type="borderModel">
    </sequence>
    <attribute name="id" type="int" use="required"/>
</complexType>
```

4.1.2.4 BorderModel

Each autonomous system is modelled by its border routers and a static delay that is a mean value of the internal delays of the autonomous system.

```
<complexType name="borderModel">
  <sequence>
    <element maxOccurs="unbounded" minOccurs="1" name="brNode"
     type="brNode"/>
  </sequence>
  <attribute name="meanDelay" type="float" />
</complextype>
```

4.1.2.5 BrNode

A border router in a simulation is a framework for the traffic handling models. Each border router consists of two model. One handles the traffic that enters the autonomous system, the other handles the traffic that leaves the autonomous system. Border routers are identified by a unique *id* attribute.

```
<complexType name="brNode">
  <sequence>
    <element name="inputModel" type="model" />
    <element name="outputModel" type="model" />
  </sequence>
  <attribute name="id" type="string" use="required"/>
</complexType>
```

4.1.2.6 LeakyBucket Model

This tag defines the parameters of a LeakyBucket model in the time series simulator. The parameters are the service rate, the time scale, the actual and the maximal queue length.

```
<xs:complexType name="LeakyBucket">
            <xs:all>
                <xs:element maxOccurs="1" minOccurs="1" name="rate"
                 type="xs:unsignedLong"/>
                <xs:element maxOccurs="1" minOccurs="1" name="timeScale"
                 type="xs:unsignedLong"/>
                <xs:element maxOccurs="1" minOccurs="1" name="queueLen"
                 type="xs:unsignedLong"/>
                <xs:element maxOccurs="1" minOccurs="1" name="queueMax"
                 type="xs:unsignedLong"/>
            </xs:all>
        </xs:complexType>
```

4.1.2.7 IDContact

An *iDContact* tag defines a link between two border nodes of two autonomous systems. It can define point-to-point links by listing the two ends, or multiple access links by listing more than two ends.

```
<element name="iDContact">
  <complexType>
    <sequence>
      <element name="brNodeId" type="string" minOccurs="0"
       maxOccurs="unbounded">
      </element>
    </sequence>
  </complexType>
</element>
```

4.1.2.8 Source

Every *<source>* tag is bounded to a border router. It defines the traffic that enters the router's autonomous system in form of time series. This definition does not make any restriction on the format of the time series. It is up to the simulation models to interpret the values defined in the time series.

```
<element name="source">
  <complexType>
    <sequence>
      <element name="traffic" type="ts" />
    </sequence>
    <attribute name="brNodeId" type="string" use="required" />
  </complexType>
</element>
```

4.1.2.9 TrafficDistrib

This tag defines the routing information for the simulator models. Since the models handle aggregate traffic values, the usual routing information is not interpretable for them. Instead for every router model there is a traffic distribution defined that describes how the traffic should be distributed among the successive routers (see architectural section on models). This information is coded, similarly to the traffic, in time series.

<element name="trafficDistrib">

```
  <complexType>
    <sequence>
      <element name="traffic" type="ts" />
    </sequence>
    <attribute name="brNodeId" type="string"/>
  </complexType>
</element>
```

4.1.3 Formats used in INTER-IP

There are two different set of data for the INTER-IP tool, one for the intra-domain evaluation, and another one for the inter-domain case. The what-if analyses are handled by the Simulation manager, which takes care of computing different sets of input for multiple instances of the inter-domain module.

4.1.3.1 Intra-domain Evaluation

The complete internal topology of the domain is required, plus the statistical characterization of the traffic along with load information:

- List of AS internal routers with specific functionality: border, access or core;
- List of AS internal links, with their associated capacity and OSPF metrics. For a DiffServ network, we also need the scheduling parameters;
- Per-class traffic load;
- Per-class traffic model: the average length of the packets for the strict priority class and of the files for the other classes[4];

These data are synthetically represented in Table 4.1.

Table 4.1 – Intra-domain entities

Network Entity	Attribute	Type	Description
Node	ID	string	Identifier of the node
Node	Function	string, the value can be: core, access, border router	Node functionality
Node	Description	string	A complete description of the node
Link	ID	integer	Identifier of the link
Link	headend	string	Identifier of the source node[5]
Link	tailend	string	Identifier of the destination node
Link	bandwidth	double	Bandwidth of the link expressed in bits per second
Link	OSPF weight	integer	OSPF weight of the link
Link	Class code	integer	A class identifier. For strict priority and Best Effort standard code (EF and BE) are used
Link	MDRR weight	integer	MDRR scheduler weight specified for each class except for the strict

[4] The mapping of the traffic on the classes has to be unique for all the domains crossed
[5] The links are modelled as uni-directional. A bi-directional link is the combination of two uni-directional links

			priority one. Best Effort is mandatory. If Best Effort is the only round robin class the weight can be zero
Link	Traffic	double	The per-class traffic of the link expressed in bits per second
Class	Class code	integer	A class identifier. For strict priority and Best Effort standard codes (EF and BE) are used. Only Best Effort is mandatory.
Class	Class description	string	A complete description of the class
Class	Average length	integer	For each class the mean length of the packet (if the class is strict priority) or the mean volume size transferred (for the round robin classes) is specified. Both values are expressed in bits

4.1.3.2 Inter-domain Evaluation

The topological and statistical characterization of the end-to-end connection must be specified:

- List of Autonomous Systems crossed by the end-to-end connection;
- List of the Border Routers connected to the above links;
- The internal Delay Matrix for each of the above ASs;
- List of the inter-domain links traversed by the flows;
- Per-class traffic load for each of the above links.

These data are synthetically represented in Table 4.2.

Table 4.2 – Inter-domain entities

Network Entity	Attribute	Type	Description
AS	ID	Integer	Identifier of the AS
Border Router	BR code	string	Border router identifier[6]
Border router	BR description	string	Text description of the border router
Link	ID	integer	Identifier of the link
Link	Headend_id	string	Identifier of the source border router[7]
Link	Headend_AS_id	string	Identifier of the AS of the source border router
Link	Tailend_id	string	Identifier of the destination border router
Link	Tailend_AS_id	string	Identifier of the AS of the

[6] At least one border router for AS is mandatory.
[7] The links are modelled as uni-directional. A bi-directional link is the combination of two uni-directional links

			destination border router
Link	Bandwidth	double	Bandwidth of the link expressed in bits per second
Link	Class code	integer	A class identifier. For strict priority and Best Effort standard codes (EF and BE) are used
Link	MDRR weight	integer	MDRR scheduler weight specified for each class except for the strict priority one. Best Effort is mandatory. If Best Effort is the only round robin class the weight can be zero
Link	traffic	double	The per-class traffic of the link expressed in bits per second
Delay_matrix	BR_to_BR delay	double	The per-class delay (in seconds) expressed for each couple of Border Router. The crossing delay of a single[8] Border Router is conventionally assumed to be zero
Traffic Relation	TR_id	string	Identifier of a traffic relation
Traffic Relation	TR_list	string	A list of couples AS,BR crossed by the traffic relation
Class	Class code	integer	A class identifier. For strict priority and Best Effort standard codes (EF and BE) are used. Only Best Effort is mandatory.
Class	Class description	string	Text description of the class
Class	Average length	integer	For each class the mean length of the packet (if the class is strict priority) or the mean volume size transferred (for the round robin classes) is specified. Both values are expressed in bits

4.2 Intra-Toolkit Formats and APIs

4.2.1 Basic XML Format for Models

A very simple XML structure for descriptions of models from all simulation approaches has been defined. It is mainly useful for handling in a database. The actual content was left to be defined in a simulator-specific way.

Every model description follows a basic structure: The top level tag is `imonmodel`, which identifies the XML file as a model description belonging to one the InterMON simulation approaches. Exactly which approach is meant is further selected by the `family` attribute. Possible values of `family` can be `hybrid` for the approach described here, `fluid` for the fluid flow approach done by Salzburg

[8] for example for non-collaborative AS we don't know anything about BRs and we can collapse thye whole AS to a fake BR

Research, `ts` for the time series simulator done by BUTE, and `analytical` for the “Inter-IP” approach by TILAB.

4.2.2 General Time Series

A general time series consists of a sequence of tuples. Each tuple has two attributes: a *time* that defines the independent variable in the time series; and a *unit* that defines the measurement unit of the *time* variable.

Each tuple in the time series consists of a sequence of values, each of that has a name. In this way arbitrary number of dimensions can be defined for the time series and it is up to the user of the time series to interpret the values and names.

```
<complexType name="ts">
 <sequence>
  <element name="tuple" minOccurs="0" maxOccurs="unbounded">
   <complexType>
    <sequence>
     <element name="value" minOccurs="0" maxOccurs="unbounded">
      <complexType>
       <simpleContent>
        <extension base="string">
         <attribute name="name" type="string"/>
        </extension>
       </simpleContent>
      </complexType>
     </element>
    </sequence>
    <attribute name="time" type="float" />
    <attribute name="unit" type="string" />
   </complexType>
  </element>
 </sequence>
</complexType>
```

4.2.3 Question

The question tag is a sequence of node ids. In this section the user expresses his interest in the specified nodes. At the end of the simulation state information of only these node will be retrieved. If no node is listed, then all node's state information will be returned.

```
<element name=”question”>
 <complexType>
  <sequence>
   <element name=”brNodeId” minOccurs=”0” maxOccurs=”unbounded”>
    <complexType>
     <simpleContent>
      <extension base=”string”>
      </extension>
     </simpleContent>
    </complexType>
   </element>
  </sequence>
 </complexType>
</element>
```

4.3 Extra-Toolkit Formats and APIs

4.3.1 VDM Interface Formats and APIs

4.3.1.1 Writing an Import Filter

Every tool, which wants to display the produced data within the visualisation process, has to implement an *import filter*. This filter has to transfer this data from the proprietary format of the tool to the well defined Filter Data Item which could be transferred via xml from one host to another (this will be done by using the Global Controller), or via handle, if the next filter resides on the same host.

To write an import filter, the programmer only has to derive the abstract c++ Class **`importFilter`** and fill the virtual method **`_import(string& inputInfo).`** The **`inputInfo`** string is not restricted and could be used in any way. It is recommended to use this string to define the import file, or to define a configuration file (please use xml), in which the import of data is described.

Within this method, the Output Filter Data Item has to be filled. But before this step, the programmer should think seriously about the data she/he wants to provide to the module:

1. Is the data multi dimensional?
2. what parameters are independent, what are dependent
3. what type is the data

The most important thing is to identify the independent and dependent variables, therefore here is a small definition for this:

Independent variables

Independent variables exists without any other knowledge. The only thing, which has to be taken into account, is that the data should be inserted into the independent list in any kind of order. In most cases, the time is an independent parameter and will be taken as the first dimension. The implied order here is the order of time itself (8pm is earlier than 10pm). The independent variable itself does not imply this order and even does not imply any kind of constant intervals between two timestamps.

The independent list (which contains the values of one independent variable) could easily be accessed via the []-operator. The abstract value, which will be given back, is the base class for the five VDM-Types: integer, non aggregateable integer, floating point, time and string. An reference to this base class could be found under section 4.3.1.3.

Example:

```
unsigned int pos = 42;
abtractValue& item = independentValueList[pos];
```

Dependent variables

Dependent variables could not exist without the knowledge of the independent variables. This means, one dependent variable is always defined, if every independent variable within one table is set to a specific value.

Example:

```
valueList::pos_identifier pos;
      pos["timeline"] = 5;
      pos["router"]   = 12;
      abtractValue& item = ValueList->at(pos);
```

If the programmer has a clear vision of the data structure, she/he will have the following methods to create tables and scalars:

```
OutputItem = ItemHandle(new FilterDataItem("coolFilter"));
scalar* newScalar = OutputItem.getItem()->createNewScalar("coolName", vdminteger);
table* newTable   = OutputItem.getItem()->createNewTable("coolerName");
newScalar = integer("23"); // strings will work as input value
independentList* newIndepList = newTable->createNewIndepList("timeline",
                                                              vdmtime);

/* fill independent list */

newTable->postInit(); // no more independent variables
dependentList* newDepList = newTable->createNewDepList("coolvalues",
                                                       vdmfloatpoint);

/* fill dependent list */

for (int i=0; i<size; i++) {
  newIndepList.insertItem(indepvalueXY);
  /* this is a "fast hack" for dependent values and only works in one
     dimensional data arrays */
  newDepItem.insertItem(depvalueXY);
}
```

In this example, the data will be inserted into the independent and into the dependent lists in parallel by just inserting the values. Normally more dimensional data will be filled one after another (first al independent variables, then the dependent.

This will be done in the following way:

```
OutputItem        = ItemHandle(new FilterDataItem("coolFilter"));
table* newTable   = OutputItem.getItem()->createNewTable("coolerName");

independentList* newIndepList1 =
       newTable->createNewIndepList("timeline", vdmtime);
independentList* newIndepList2 =
       newTable->createNewIndepList("router", vdmnagint);

/* fill independent list */
for (int i=0; i<size; i++) {
  newIndepList1.insertItem(indepvalue_1_XY);
  newIndepList2.insertItem(indepvalue_2_XY);
}
newTable->postInit(); // no more independent variables

dependentList* newDepList =
       newTable->createNewDepList("coolvalues", vdmfloatpoint, true);

/* fill dependent list */
for(int j=0; j<newIndepList1.size(); j++)
  for(int j=0; j<newIndepList1.size(); j++){
    pos["timeline"]=j;
    pos["router"]=k;
    newDepItem.insertItemAt(depvalueTojAndk, pos);
  }
```

A complete example of parsing a comma separated file will be given in 5.1.

When the **_import** method is written, the filter could be accessed by the following VDM-Code:

```
#include <string>

#include "csvImportFilter.h"
#include "FilterDataItem.h"


int main()
```

```
{
  string Filename("./traces/aa.csv");

  /****** IMPORT FILTER ******/
  /* The csvImportFilter is derived from the ImportFilter and creates
     a system conform data structure */

  /* creating the filter */
  csvImportFilter myfirstFilter;

  /* importing the file defined above and translate it to the FDI */
  myfirstFilter.import(Filename);

  /* get the item handle to transfer the item to the next filter */
  ItemHandle transItem = myfirstFilter.outputItem();

  /* write the imported data to disk in defined XML */
  myfirstFilter.outputFile("traces/aa.xml");

  /* send the transItem to the next filter */

  ....

}
```

4.3.1.2 FilterDataItem Class Reference

The Filter Data Item is the internal transport object to transfer data from one filter to another.

The Filter Data Item class reference was produced by doxygen:

`#include <FilterDataItem.h>`

Detailed Description

Class to transport a filter data item between two filters.

The Filter Data Item contains the data for "inter filter communication". This could be:

1. tables: A **table** is a multi dimensional information storage. It contains independent variables, which opens up the dimensions of the data space and the dependent variables, which are the main data within the multi dimensional array
1. scalars: A **scalar** is one item, that is defined by name, type and value. Scalars could carry maximum/minimum values etc. for example.

every **table** or **scalar** could be accessed by name via hashmaping (name_map is just a typedef of a hashmap for strings, see hashfunc.h).

ItemHandle-Communication

Within the filter communication (from the outside of filter), a new handle has been introduced, to be able to destroy old objects more easy. For further information see **ItemHandle**.

Within filters, these item handles are not used.

Public Methods

- `FilterDataItem` **`(const string &Name=string())`**
  Constructor.

- **~FilterDataItem ()**
  Destructor.

- name_map (table *) **TableList**
  hashmap to access tables.

- name_map (scalar *) **ScalarList**
  hashmap to access scalars.

- **string** getName **()**
  returns the name of this Filter Data Item.

- **void** setName **(const string &Name)**
  sets the name of this Filter Data Item.

- table * createNewTable **(const string &Name)**
  *creates a new **table** with name "Name", puts it into the tablelist.*

- scalar * createNewScalar **(const string &Name, const** VDMtype **type)**
  *creates a new **scalar** and adds it into the scalarlist.*

4.3.1.3 Value Items

Within the Visual Data Mining every value-object is defined in name, type and value. Every type of value-object is derived by the abstract class called **abstractType**.

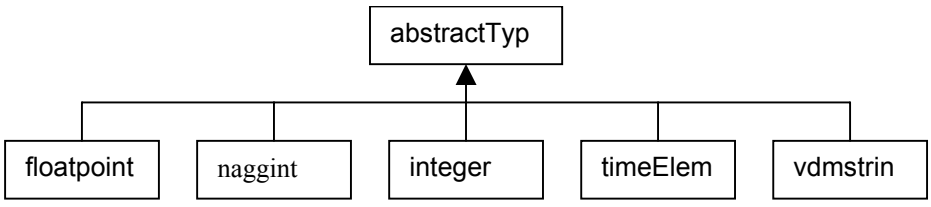
There have been five types defined for data values (integer, non aggregateable integer, floating point, string and time). The abstract Type looks like follows:

abstractType Class Reference

defined as an abstract class for all VDM-types.

`#include <VDMtypes.h>`

Inheritance diagram for abstractType:



Public Methods

- **abstractType ()**
  empty constructor.

- **virtual ~abstractType ()**
  virtual empty destructor.

- **virtual string** str **()=0**
  method to return a string of his own (used for xml writing).

- **virtual void** setValue **(const string &input)=0**
  sets the value by a string (used for xml parsing).

- **VDMtype getType ()**
  returns the Type of this VDMItem.

- **virtual abstractType *** clone **()=0**
  clones the Type without knowing the correct type.

- **virtual abstractType &** operator+= **(abstractType &absType)=0**
  adds two values without knowing the correct type.

- **virtual abstractType &** operator= **(abstractType &absType)=0**
  inserting the value into another object.

- **virtual bool** operator> **(abstractType &absType)=0**

Static Public Methods

- VDMtype getVDMtype **(const string &type)**
  static method to convert a string to a VDMType.

- **abstractType *** createType **(**VDMtype **type, const string *str=0)**
  static method to create a new object with the basetype.

Some operators are still missing and will be implemented as soon as possible.

4.3.2 BGP-4 Topology Description

4.3.2.1 High level description of a general purpose Internet topology object derived from BGP-4 routing information.

The description of a Topology as shown in Figure 2.4 is relatively simple.

A topology consists of an certain number of autonomous systems which are linked together through a given amount of links. If the autonomous system has supplied routing information to the topology tree (i.e. has the BGP-4 Topology Tools deployed) or not, it will include either several border routers or none. Any routing event will be attached to the border router which detected it.

Links will always be point to point. They are the physical representation for the TCP session which carries the BGP-4 routing information between two autonomous systems. The endpoints of a link might be a border router (when the BGP-4 topology tools are being used in the AS) or an AS (when the link was derived from the topology). All three combinations (i.e. BR to BR, AS to BR and AS to AS) are possible.

4.3.2.2 Particularisation of the topology object to the end to end QoS scenario

This general principle used to describe BGP-4 topologies can be refined by including prefix information in the source and destination autonomous system. This information marks both endpoints of the scenario. So an AS may also include an IP prefix if it is one of the endpoints of the scenario.

4.3.2.3 XML Schema

The relations explained in the previous paragraph can be easily translated into following XML Schema.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://BGPTopology" xmlns="http://BGPTopology">
      <!-- definition of simple elements  -->
      <xsd:simpleType name="aspath">
            <xsd:restriction base="xs:string">
                  <xsd:pattern value="([0-9]*|[0-9]+( [0-9]+)+)"/>
            </xsd:restriction>
      </xsd:simpleType>
      <xsd:simpleType name="asid">
            <xsd:restriction base="xs:unsignedShort"/>
      </xsd:simpleType>
      <xsd:simpleType name="timestamp">
            <xsd:restriction base="xs:unsignedLong"/>
      </xsd:simpleType>
      <xsd:simpleType name="updateType">
            <xsd:restriction base="xs:string">
                  <xsd:pattern value="[AW]"/>
            </xsd:restriction>
      </xsd:simpleType>
      <xsd:simpleType name="ipAddress">
            <xsd:restriction base="xs:string">
                  <xsd:pattern value="([0-9]{1,2}|1[0-9]{2}|2[0-5]
                   [0-9]\.){3}([0-9]{1,2}|1[0-9]{2}|2[0-5][0-9])"/>
            </xsd:restriction>
      </xsd:simpleType>
      <xsd:simpleType name="ipPrefix">
            <xsd:restriction base="xs:string">
                  <xsd:pattern value="([0-9]{1,2}|1[0-9]{2}|2[0-5]
                   [0-9]\.){3}([0-9]{1,2}|1[0-9]{2}|2[0-5][0-9])/
                   ([0-9]|[12][0-9]|3[0-2])"/>
            </xsd:restriction>
      </xsd:simpleType>
      <xsd:simpleType name="temperatureType">
            <xsd:restriction base="xs:string">
                  <xsd:pattern value="(cold|warm|hot)"/>
            </xsd:restriction>
      </xsd:simpleType>
      <xsd:simpleType name="linkType">
            <xsd:restriction base="xs:string">
                  <xsd:pattern value="(AS|BR)"/>
```

```
      </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="endPointType">
      <xsd:restriction base="xs:string"/>
      <xsd:attribute name="type" type="linkType" use="required"/>
</xsd:simpleType>
<!-- definition of attributes       -->
<xsd:attribute name="state" type="temperatureType" use="required"/>
<xsd:attribute name="type" type="linkType" use="required"/>
<!-- definition of complex elements -->
<!-- a routing event has
     a type,
     a tiemstamp,
     a prefix
     and may have an ASPATH -->
<xsd:complexType name="update">
      <xsd:sequence>
            <xsd:element name="Type" type="updateType" minOccurs="1"
             maxOccurs="1"/>
            <xsd:element name="Timestamp" type="timestamp"
             minOccurs="1" maxOccurs="1"/>
            <xsd:element name="IPPrefix" type="ipPrefix"
             minOccurs="1" maxOccurs="1"/>
            <xsd:element name="ASPath" type="aspath" minOccurs="0"
             maxOccurs="1"/>
      </xsd:sequence>
</xsd:complexType>
<!-- a routing event list
     has always one or more routing events -->
<xsd:complexType name="eventList">
      <xsd:element name="Update" type="update" minOccurs="1"
       maxOccurs="unbound"/>
</xsd:complexType>
<!-- a border router
     has always an IP address
     and may have a routing event list -->
<xsd:complexType name="borderRouter">
      <xsd:sequence>
            <xsd:element name="IPAddress" type="ipAddress"
             minOccurs="1" maxOccurs="1"/>
            <xsd:element name="EventList" type="eventList"
             minOccurs="0" maxOccurs="unbound"/>
      </xsd:sequence>
      <xsd:attribute name="state" type="temperatureType"
       use="required"/>
</xsd:complexType>
<!-- an AS has always one ID
     and zero or one prefixes
     and zero or more border routers -->
<xsd:complexType name="autonomousSystem">
      <xsd:sequence>
            <xsd:element name="ASid" type="asid" minOccurs="1"
             maxOccurs="1"/>
            <xsd:element name="IPPrefix" type="ipPrefix"
             minOccurs="0" maxOccurs="1"/>
            <xsd:element name="BR" type="borderRouter" minOccurs="0"
             maxOccurs="unbound"/>
      </xsd:sequence>
</xsd:complexType>
<!-- a Link has always two endpoints -->
<xsd:complexType name="link">
      <xsd:sequence>
```

```
                  <xsd:element name="EndPoint" type="endpointType"
                   minOccurs="2" maxOccurs="2"/>
            </xsd:sequence>
      </xsd:complexType>
      <!-- a topologyGraph has at least one AS and one Link -->
      <xsd:complexType name="topologyGraph">
            <xsd:sequence>
                  <xsd:element name="AS" type="autonomousSystem"
                   minOccurs="1" maxOccurs="unbound"/>
                  <xsd:element name="Link" type="link" minOccurs="1"
                   maxOccurs="unbound"/>
            </xsd:sequence>
      </xsd:complexType>
      <!-- Reference to topologyGraph to define BGPTopologyTree -->
      <xsd:element name="BGPTopologyTree" type="topologyGraph"/>
</xsd:schema>
```

5 Appendix

5.1 Import Filter Example Source Code

```
Void csvImportFilter::_import(const char* InputInfo)
{
  if (status == fileLoaded) {
    cdbg << "file is yet loaded\n";
    return;
  }

  /*** create a new Filter Data Item and give it to the handle ***/
  OutputItem = ItemHandle(new FilterDataItem("csvImportItem"));

  /*** create a new table and a new scalar within the FDI ***/
  table* csvTable = OutputItem.getItem()->createNewTable("csvTable");
  scalar* csvScalar =
        OutputItem.getItem()->createNewScalar("NumberOfItems", vdminteger);

  /*** create and fill the independent List ***/
  independentList* indepList = csvTable->createNewIndepList("timeline",
vdmtime);

  char line[255];

  ifstream inputFile(InputInfo);
  if (inputFile.fail())
    throw (filterError("Tracefile not found"));
  int count=0;
  inputFile.getline(line,255); // first line is only description - skip it
  while(!inputFile.eof()) {
    inputFile.getline(line,255);
    if ((line[0] == '#') || (line[0] == ' ') || (line[0] == '\n'))
      continue;
    char* line_ptr = line;
    timeElem time;

    /*** we are using the unix time struct ***/
    tm Time;

    bzero(&Time,sizeof(struct tm));
    Time.tm_mday = atoi(line_ptr);
```

```
    while(line_ptr[0]!='-') ++line_ptr;
    ++line_ptr;

    /**** parsing month ****/
    if (strncmp(line_ptr, "Jan", 3) == 0)
      Time.tm_mon = 0;
    if (strncmp(line_ptr, "Feb", 3) == 0)
      Time.tm_mon = 1;
    if (strncmp(line_ptr, "Mar", 3) == 0)
      Time.tm_mon = 2;
    if (strncmp(line_ptr, "Apr", 3) == 0)
      Time.tm_mon = 3;
    if (strncmp(line_ptr, "May", 3) == 0)
      Time.tm_mon = 4;
    if (strncmp(line_ptr, "Jun", 3) == 0)
      Time.tm_mon = 5;
    if (strncmp(line_ptr, "Jul", 3) == 0)
      Time.tm_mon = 6;
    if (strncmp(line_ptr, "Aug", 3) == 0)
      Time.tm_mon = 7;
    if (strncmp(line_ptr, "Sep", 3) == 0)
      Time.tm_mon = 8;
    if (strncmp(line_ptr, "Oct", 3) == 0)
      Time.tm_mon = 9;
    if (strncmp(line_ptr, "Nov", 3) == 0)
      Time.tm_mon = 10;
    if (strncmp(line_ptr, "Dec", 3) == 0)
      Time.tm_mon = 11;

    while(line_ptr[0]!='-') ++line_ptr;
    ++line_ptr;

    Time.tm_year = atoi(line_ptr);

    /* assume the year */
    if (Time.tm_year < 69)
      Time.tm_year +=100;
    timeElem TimeElement;
    TimeElement.setValue(&Time);

    /* insert the time into the independent list */
    indepList->insertItem(TimeElement);
    count++;
  }
  inputFile.close();

  /*** tell the table that there are no more independent varaibles ***/
  csvTable->postInit();
  ifstream inputFile1(InputInfo);

  /*** create new independent variables ***/
  dependentList* depListOpen =
                        csvTable->createNewDepList("Open", vdmfloatpoint);
  dependentList* depListHigh =
                        csvTable->createNewDepList("High", vdmfloatpoint);

  dependentList* depListLow =
                       csvTable->createNewDepList("Low", vdmfloatpoint);
  dependentList* depListClose =
                       csvTable->createNewDepList("Close", vdmfloatpoint);
  dependentList* depListVolume =
```

```
                        csvTable->createNewDepList("Volume", vdminteger);

  inputFile1.getline(line,255); // first line is only description -
                                // skip it again
  while(!inputFile1.eof()) {
    inputFile1.getline(line,255);
    if ((line[0] == '#') || (line[0] == ' ') || (line[0] == '\n'))
      continue;
    char* line_ptr = line;
    while(line_ptr[0]!=',') ++line_ptr;
    ++line_ptr;

    /**** parsing comma separated values ****/
    floatpoint Open(atof(line_ptr));
    depListOpen->insertItem(Open);
    while(line_ptr[0]!=',') ++line_ptr;
    ++line_ptr;

    floatpoint High(atof(line_ptr));
    depListHigh->insertItem(High);
    while(line_ptr[0]!=',') ++line_ptr;
    ++line_ptr;

    floatpoint Low(atof(line_ptr));
    depListLow->insertItem(Low);
    while(line_ptr[0]!=',') ++line_ptr;
    ++line_ptr;

    floatpoint Close(atof(line_ptr));
    depListClose->insertItem(Close);
    while(line_ptr[0]!=',') ++line_ptr;
    ++line_ptr;

    integer Volume(atoi(line_ptr));
    depListVolume->insertItem(Volume);
  }
  inputFile1.close();

  /*** add the counter as a scalar value to the Filter Data Item ***/
  integer int_count(count);
  (*csvScalar) = int_count;

  cdbg << "file import done\n";
  status = filtered;
}
```

5.2 References

[Ber02a]	G. Bergholz, *Signalflußsimulation für Nachrichtenverkehrsmodelle*, Technical Report SR-ANC-B0I1, September 2002
[Ber02b]	G. Bergholz, *Mehrklassen-Signalflußsimulation für Nachrichtenverkehrsmodelle*, Technical Report SR-ANC-B0I2, December 2002
[Ber02c]	G. Bergholz, U. Hofmann, I. Miloucheva, , P. Haber, C. Brandauer, *InterMON Rate and Time Continuous Fluid Simulation Technology (RTC-SIM) compared with current fluid simulation research*, Technical Report SR-ANC-B0I3, December 2002
[CISCO]	http://www.cisco.com/warp/public/63/mdrr_wred_overview.html#mdrr_overview
[D6]	InterMON partners, *InterMON Deliverable 6: Modelling and Simulation Specification*, December 2002
[D7]	InterMON partners, *InterMON Deliverable 7: Specification of visual data mining and user interface*, 2002
[D8]	InterMON partners, *InterMON Deliverable 8: Prototype of inter-domain QoS monitoring components*, 2002
[D10]	InterMON partners, *InterMON Deliverable 10: Prototype of inter-domain data base with policy-controlled data collection*, 2002
[Hof03]	ABSTRACTTYPE
[Kle76]	L. Kleinrock, *Queuing Systems*, Vol.2: Computer Applications, John Wiley & Sons, 1976
[Liu]	B. Liu et al, *Fluid Simulation of Large Scale Networks: Issues and Tradeoffs*, Las Vegas, NV, June 1999
[MATHW]	http://www.mathworks.com/index.shtml
[P1115]	EURESCOM project P1115 (Saltamontes), *Traffic Engineering in Differentiated Services Networks*, (Volume 3)
[Qui03]	J. Quittek et al, *Requirements for IP Flow Information Export*, June 2003, work in progress
[SM02]	Seger, Michaelis, *Concept of configurable filters for Visual Data Mining Systems*, in Proceedings Inter-domain Performance and Simulation (IPS) Workshop, Salzburg, 2002