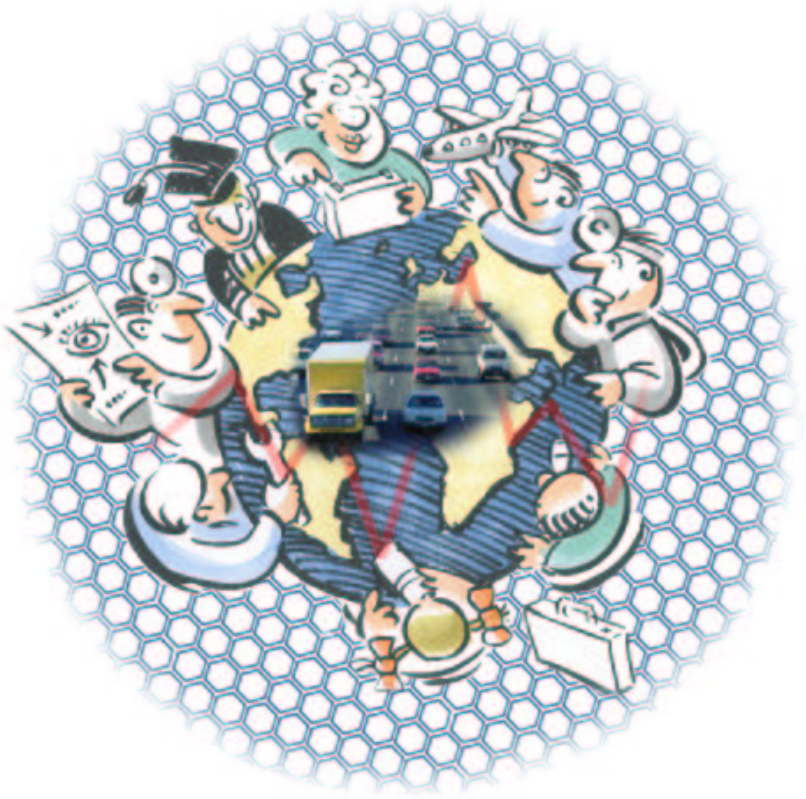
The Virtual Traffic Lab



An exploration of an environment
to experiment with traffic applications

author:	J.G. Zoetebier 9603824
study:	Business information systems
specialization:	Technical information systems
group:	Computer Architecture and Parallel Systems
faculty:	Faculty of Science
	Faculty of Economics and Econometrics
university:	University of Amsterdam
supervisor:	Drs. A. Visser
date:	August 2002

Abstract

As the computing capacity grows steadily and continuously, there is a growing need for the right tools to aid scientists in using the available computing resources. The past few years the Grid has become the standard in this area. Although the Grid offers an unprecedented amount of resources, right now it lacks tools that aid the user in handling, analyzing and visualizing information flows. The University of Amsterdam tries to bridge this gap with the Gridbased Virtual Laboratory AMsterdam (VLAM-G). It offers a distributed analysis platform for applied experimental science and provides science portals for several domains.
In this thesis we describe such a portal for the traffic domain called the Virtual Traffic Lab. We explore the possibilities of such an environment, where scientists can experiment with traffic applications. We have charted the requirements for a Virtual Traffic Lab based on the traffic tools that are currently in use. Several several interfaces have been created between MatLab, an environment for mathematical computing, analysis and visualization, and VLAM-G to enable the integration of ADSSIM, a discrete event simulation environment for Automatic Debiting System simulations. Using these interfaces, we have designed and partially implemented a prototype traffic application called ADSSIM-VLAM. This prototype can execute a distributed traffic simulation based on user supplied parameters.
Although our prototype application shows the possibilities of VLAM-G for the traffic domain, the user still has to cope with low level details that ideally will be hidden in the future.

keywords: parallel computing, Grid, collaborative environment, Virtual Laboratory, traffic applications, simulation, MatLab, Simulink.

Acknowledgements

I love deadlines. I like that swooshing sound they make when they pass by.

- Douglas Adams

It took a bit more time than expected, but here it is lying before you: my masters thesis. When I first discussed this project with Arnoud we came to the conclusion that it would offer interesting but also uncertain opportunities. It has proven to offer both. This thesis could not have been realized without the efforts and support of a number of people.
First of all I would like to thank Arnoud. His patience, knowledge and insights kept me going and helped me to find my way.
Discussions with and feedback from various members of the VLAM-G team, in particular Adam Belloum and Hakan Yakali, have been a great help. Hakan also assisted me with useful MatLab and Unix tips.
My parents and brother have supported me throughout the years, offered a listening ear and also put up with me while I was programming or writing late at night.
Several friends provided the necessary distraction as well as valuable advice. I can not mention all of them, but I would like to name two special friends.
Hans has been in the same boat for quite some time now. Our exchange of ideas, frustrations, tips, quotes and spam has been a priceless experience for me.
Finally, Els has been kind of a "mental coach". She has helped me with countless encouragements and with finding the discipline that is required for completing a thesis.

Joost Zoetebier
August 2002

Table of Contents

List of Figures

Chapter 1

Introduction

> Glaucus Proteomics BV announced on April 24th 2002 that it has entered into agreements with SARA, one of Europes largest supercomputing facilities, and GigaPort, a next generation Internet initiative which provides a state-of-the-art broadband network, both of the Netherlands. These agreements are expected to provide the biocomputing capacity and connectivity to help with the development of novel tools and technologies for high throughput proteomic analysis and the rapid screening of antibody and small molecule drug candidates for improved specificity. [...] "Genomics and proteomics will transform the search for new medicines into an information driven science. In order to understand the causes of human disease, high performance computing is essential and must, therefore, be an integral part of the business strategy of Glaucus Proteomics," stated Prof. Ian Humphery-Smith, Chief Scientific Officer at Glaucus Proteomics.

The rapid increase in computer capacity and bandwidth has fundamental impact on the way science is practiced. Until recently, we relied heavily on theory and experiment to find answers to scientific questions. More and more, we have the capability to simulate and model systems considered too complicated to characterize experimentally with previously available technology [8]. This leads to huge information flows which have to be managed and analyzed. Supercomputing and mass storage systems can aid us in controlling these information flows.

As the computing capacity grows steadily and continuously [21], there is a growing need for the right tools to aid scientists in using the available computing resources. One of the advancements that is made in this area is the Grid [3]. Grids serve as a scalable computing platform for executing large-scale computational and data intensive applications in parallel through the aggregation of geographically distributed computational resources [13]. Although the Grid offers an unprecedented amount of resources, right now it lacks tools that aid the user in handling, analyzing and visualizing information flows.

The University of Amsterdam tries to bridge this gap with the Gridbased Virtual Laboratory AMsterdam (VLAM-G). It offers a distributed analysis platform for applied experimental science and provides science portals for several

domains: chemo-physical analysis of material surfaces, simulated bio-medical vascular reconstruction using immersive visualization techniques, correlation of gene expression data from heterogeneous databases, and a simulation/analysis environment for road-traffic measurement data.

The Dutch roads are equipped with an extensive set of devices, which can measure a large and diverse set of features of the Dutch traffic. The measurements can be simple vehicle counters, or complex systems that track individual vehicles over longer distances based on license plate numbers. A scientist who wants to analyze these type of measurements, for instance to compare it with simulations, has to be supported by a number of tools to be able to manage large datasets and to visualize certain aspects. We are interested in the tools users in this area of research need, and how they can use the power and possibilities of the Grid that are offered in a userfriendly way through VLAM-G.

1.1 Aims of this thesis

At this moment there exist few real VLAM-G applications. For the further development of VLAM-G, it would be useful to develop a prototype of an application, in our case for the traffic domain. Eventually, VLAM-G wants to offer a traffic science portal consisting of several applications. Traffic scientists currently use several tools to perform analyses. Ideally, in the future there will exist a set of modules that are specifically written for the analysis and visualization of traffic data. Using these modules scientists will able to perform distributed experiments that previously had to be conducted with improvised tools on standalone computers.
We want to explore the possibilities of an environment in which scientists can experiment with traffic applications. In this thesis we want to give insight into a Virtual Traffic Laboratory by designing and partially implementing a prototype traffic application.

Before we continue we need to clarify some terms and make some assumptions. The *Virtual Traffic Laboratory* denotes the traffic science portal of VLAM-G. When we say we want to *give insight* into such a laboratory, we mean two things. We want to give this insight by describing the functionality such a Virtual Traffic Laboratory should possess, and by describing our hands-on experience during the design and implementation of a prototype of a traffic application.
We will follow the evolutionary approach of systems development [11] for the *design and implementation* of this prototype. This approach breaks up a project into separate parts, and then, one by one, each of the parts is taken through the systems development process. The choice for this approach is based on our assertion that the specification and implementation of parts of VLAM-G might change while we are designing and implementing our prototype, because VLAM-G is still in development. Another reason is the expectation that hands-on experience with VLAM-G might ask for a different design or implementation of our prototype. The evolutionary systems development approach minimizes the costs, time and efforts of a possible reiteration of the systems development cycle.

The realization of the entire Virtual Traffic Lab is too extensive for the purposes of this thesis, we therefore limit the scope to partially implementing a prototype traffic application. By *partially* implementing we mean that we will select one of the current applications and integrate this into VLAM-G. The result will be a *prototype*, a VLAM-G application for the traffic domain which will prove the functionality and benefits of the Grid and VLAM-G for traffic applications.
Traffic applications are defined as applications which enable the user to perform experiments and analyses on data that is generated by real or simulated traffic flows.

Now that we have defined the aims of this thesis and the terms in which these aims are expressed, we can describe the approach we have followed to reach these aims.

1.2 Approach

We have followed a bottom-up approach in this thesis. The Grid is the foundation beneath VLAM-G, so we start with background information on the Grid. When the features of the Grid are known, we can take a look at VLAM-G and explain how it relates to the Grid on the one side, and to scientific users on the other side. We will describe the context of VLAM-G, i.e. discuss the past and the future, and compare it to other Virtual Laboratories. The next step is to narrow our focus to a particular instance of VLAM-G: the Virtual Traffic Lab. In order to discuss traffic applications, we first have to establish the domain specific information and terminology used by traffic scientists. Using this knowledge we can start designing the Virtual Traffic Lab. This design starts with the requirements determination which consists of several phases: problem definition, requirements acquisition and requirements analysis. These phases are described in detail and eventually lead to a requirements specification for the Virtual Traffic Lab.

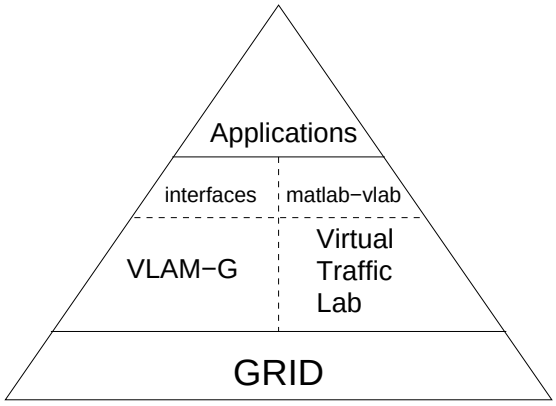


Figure 1.1: Framework for the approach

Because we will not develop the entire Virtual Traffic Lab, and want to prove the functionality and benefits of the Grid and VLAM-G for traffic applications, we decided to try to integrate a current application into VLAM-G. To realize this integration we have created an interface. Before we actually implement this interface, we first describe which environments and languages we will use.
We have developed several interfaces in order to integrate VLAM-G with ADS-SIM, the traffic simulator we selected to use for our prototype (the reason for selecting this application will be discussed in the final chapter). Besides describing the implementation of these interfaces, we also evaluate their design and implementation.
The prototype application for the Virtual Traffic Lab is the last step. We show how we envision a traffic scientist using our application through a real-life example of a topic that is investigated by traffic scientists at the University of Amsterdam. This example is followed by the design, implementation and evaluation of the prototype.
We conclude the thesis by summarizing the aims that we intended to reach, describing our findings and the implications of these findings, and identify topics that require further study or analysis.
The approach we followed is summarized in the framework of figure 1.1.

1.3 Overview

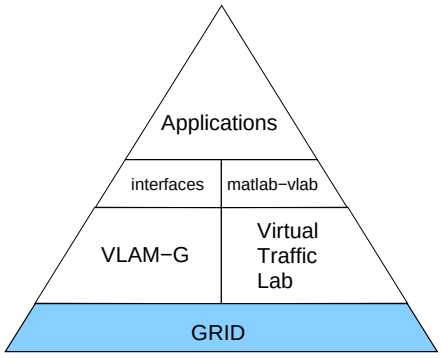
The thesis reflects the approach that was followed.
Chapter 2 describes the Grid, its architecture, strengths and weaknesses. We discuss the future of the Grid and make a comparison between the Grid and ASP, and between the Grid and Web Services.
Chapter 3 shows the Gridbased Virtual Laboratory AMsterdam, its users and architecture. We review the development, compare it to other Virtual Laboratories in order to determine the added value of VLAM-G, and take a look at the future of VLAM-G.
Chapter 4 outlines the Virtual Traffic Lab. After supplying the necessary background information on traffic research, we choose a systems development method, and perform the requirements determination. One of the phases of this process involves requirements acquisition, where we take a look at the tools that are currently used by traffic scientists to perform their analyses and visualizations. We analyze these tools and tasks and distill wishes and requirements.
Chapter 5 describes environments (MatLab, Simulink) and languages (MDL, XML) that will be used for the creation of our interfaces.
Chapter 6 continues with the description of several interfaces we created for the Virtual Traffic Lab. Small code examples will be given to explain their functionality, accompanied by screenshots and small experiment descriptions. We evaluate each interface to show to what degree they can be used or can be adapted.
Chapter 7 starts with the selection of the application that will be converted to a prototype for the Virtual Traffic Lab. A case example is used to describe the way we imagine this prototype could be used. Based on this example is our design and implementation, which is evaluated at the end of this chapter.
Chapter 8 sums up the conclusions based on our findings in the previous chapters and on our initial aims. We conclude with topics that require further study or analysis.

Chapter 2

The GRID

2.1 Introduction

Simulations play an essential role in evaluating different aspects of traffic systems. As in most application areas, the available computing power is one the determining factors with respect to the level of detail that can be simulated [1] and. The availability of datasets that can validate the results is equally important [10]. Consequently, lack of these factors leads to more abstract models [2]. Since we focus on the use of computational power as part of an experimentation platform and want to be able to afford more detailed simulations, we looked how we could use the resources provided by the Grid.



2.2 The Grid

The term "the Grid" was invented in the mid 1990s to denote a proposed distributed computing infrastructure for advanced science and engineering [3]. The vision behind this idea is that users can "plug into" the Grid to get access to resources and data similar to the ease of use of electricity, water or the telephony system. The main problem in creating this Grid infrastructure is the coordination of resource sharing and problem solving in dynamic, multi-institutional virtual organizations [9]. Foster defines a virtual organization as a set of individuals and/or institutions which act as resource providers and/or consumers, who have clearly and carefully defined what is shared, who is allowed to share, and the conditions under which sharing occurs. Up till then the available distributed computing technologies were unable to provide these features. The Grid addresses these issues by creating an architecture which consists of several (standards based) protocols, Application Programming Interfaces (API's) and Software Development Kits (SDKs). This open and extensible architecture acts

as middleware to enable the sharing and accounting of resources.
Grids serve as a scalable computing platform for executing large-scale computational and data intensive applications in parallel through the aggregation of geographically distributed computational resources. They enable exploration of large problems in science, engineering, and business with huge data sets, which is essential for creating new insights into the problem [13].
The Grid architecture is composed of several layers. Each layer provides different capabilities and at the same time utilizes the functionality of the lower layers. We will briefly describe these layers.

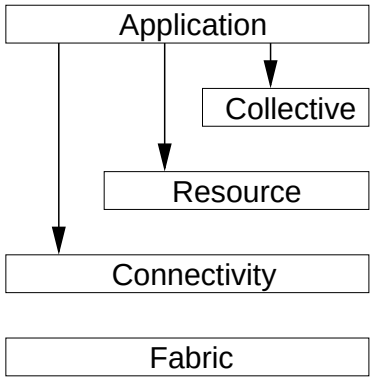


Figure 2.1: Grid Protocol Architecture

Fabric layer provides the resources to which shared access is mediated by Grid protocols: for example computational resources, storage systems, catalogs, network resources and sensors.
Connectivity layer defines core communication and authentication protocols required for Grid-specific network transactions. Communication protocols enable the exchange of data between Fabric layer resources. Authentication protocols build on communication services to provide cryptographically secure mechanisms for verifying the identity of users and resources.
Resource layer builds on Connectivity layer communication and authentication protocols to define protocols (and APIs and SDKs) for the secure negotiation, initiation, monitoring, control, accounting and payment of sharing operations on individual resources.
Collective layer contains protocols and services (and APIs and SDKs) that are not associated with any one specific resource but rather are global in nature and capture interactions across collections of resources. Examples are directory services, co-allocation, scheduling and brokering services, monitoring and diagnostics services, Grid-enabled programming systems and community accounting and payment services.
Application layer is constructed in terms of and calls upon services defined at any layer. Each underlying layer consists of protocols (and possibly APIs and/or SDKs) that provide access to specific services and that perform desired actions. The Globus Toolkit[1] is the de facto standard in the area of Grid computing. It

[1] *http://www.globus.org*

offers a set of services and software libraries to support Grids and Grid applications. The Toolkit includes software for security, information infrastructure, resource management, data management, communication, fault detection, and portability.

Wide-area distributed computing, or "grid" technologies, provide the foundation to a number of large-scale efforts utilizing the global Internet to build distributed computing and communications infrastructures. As common Grid services and interoperable components emerge, the difficulty in undertaking these large-scale efforts will be greatly reduced and, as importantly, the resulting systems will better support interoperation.

Our department participates, via the DutchGrid, in the European DataGrid project. This makes seamless resource sharing by all participants both within the Netherlands and in Europe possible. The actual sharing of resources remains of course subject to bilateral usage agreements, but in principal this promises users unprecedented computational power.

Two important factors facilitate the existence of the Grid. Number one is bandwidth. A significant amount of bandwidth is needed for the communication between computers and the exchange of data. Without this bandwidth the response time of applications would be unacceptable, and the quality of the service would be perceived as poor by the end users. The fact that bandwidth is becoming increasingly plentiful and increasingly inexpensive, with the promise of that trend continuing, is a major reason why Grid applications are able to function.
Another important factor is the wide acceptance and use of standards. Because of the heterogeneous nature of networks, computers and data that form the Grid, there has to be a set of common languages and protocols to get things to work together. Open standards such as TCP/IP, XML and SOAP provide this interoperability.

We have described the basic outline of the Grid. But what makes it so special?

2.3 Strengths and weaknesses of the Grid

The Grid has several strengths that can be noted.
Maintenance. Software and data do not have to run locally, so the system operators can decide which software will be installed and maintained on a chosen platform.
Scale of operation. More data can be processed with more computer capacity in a smaller amount of time with more detailed information as a result.
Higher reliability. A different part of the Grid can be used in case of a malfunction of hardware or software, without the end user even knowing.
Transparency. Details of the lower levels of the Grid are hidden for the end user. Ideally, a Grid application can be run as if it runs on one computer.
Security. Access to data and resources is restricted. This is a necessary condition for trust (in the integrity) of the Grid.
Less redundancy. Because the Grid offers transparency and security, several currently separated information systems can be integrated into one (virtual) Grid

system. Example: sharing of data between hospitals, medical organizations, drugstores and family doctors.
Accounting. The use of every resource in a particular time frame is logged. This is essential for a fair exchange of resources, or for converting the use of resources to an amount of money.

Of course there also exist some drawbacks and weaknesses to the Grid.
Applications. A lot of work has been done on the layers that form the basis of the Grid. Accounting, security, transparency, etcetera are all taken care of. At first, the top level which consists of applications did not receive as much attention as the lower levels of the Grid. The majority of the users of the Grid at this moment therefore consists of computer scientists. Applications are mostly centered around scientific problems which demand lots of storage and/or computational capacity. Examples are GridLab[2] (gravitational wave detection and analysis, numerical relativity), NASA's Information Power Grid (IPG)[3] (numerical propulsion system simulations, distributed/collaborative scientific visualizations), Astrophysics Simulation Collaboratory [14] and Molecular Modelling for Virtual Drug Design [13].
Now that the possibilities of the Grid become more and more obvious and within reach of (application) developers, other application domains such as analysis of medical data are explored, but few Grid applications have been realized for those domains.
One possible solution to lower the threshold for Grid applications in other domains could be the the integration of ASP concepts into the Grid.

2.4 Grid and ASP

The core idea behind Application Service Providers (ASP's) is that applications can be made available as a service rather than an installation. Businesses or consumers pay a monthly or yearly fee to have access to and make use of the latest applications supplied by the ASP. Its core competencies are to produce and maintain the service. This is beneficial to both parties: the ASP can specialize in this service, receives a continuous amount of money and builds a lasting relationship with the customer. The business or consumer that purchases this service does not have to worry anymore about the time, energy and costs of keeping the application(s) up to date.
The application can be accessed from every trusted host or trusted user within or outside of the company. The application itself and the data however are stored at one central location.
The Grid takes this concept further by removing the need for the application to live in a particular location or on specific hardware. A Grid application can obtain resources from anywhere on the network. An application can get disk space from here, computing cycles from there, specialized application services from over there, and network bandwidth from yet another place.
Because sensitive data can be compromised, ASP's rather keep this data within the protected network that they provide. They can offer encrypted transfer of data or the encrypted storage of data to ensure the data privacy [23]. The Grid

[2] *http://www.gridlab.org*
[3] *http://www.ipg.nasa.gov/*

also supports the encrypted interchange and storage of data [24], so it would be a logical step to offer Grid-based ASP solutions.
IBM is one of the first major IT companies to have embraced Grid technology and Grid computing. They are involved in several projects, one of which is the Electronic Medical Record (EMR) data grid and repository. This is a patient-centric medical record system that can capture from any location the full range of healthcare files including high-fidelity patient medical images (CT, MRI, mammograms), records, and clinical history. In collaboration with the University of Pennsylvania they have built a Grid that delivers computing resources as a utility-like service over a secure Internet connection. Enabling up to thousands of hospitals to store mammograms (X-ray examination of breasts for detection of tumors) in digital form, it gives authorized medical personnel near-instantaneous access to patient records and reduces the need for expensive X-ray films. Hospitals are connected to the grid via secure Internet portals that allow authorized physicians to upload, download, and analyze digitized X-ray data to identify potential tumors and other problems. Sophisticated algorithms can uncover patterns that appear in the population, such as cancer "clusters", or abnormal concentrations of disease in a particular community.
Several other applications are in development, such as analytical tools to help physicians diagnose identify cancer, and educational tools for training medical students, interns and radiologists [25].
Related to the idea of ASP is the concept of "web services".

2.5 Grid and Web Services

The Grid is far from finished. One of the topics that is heavily influencing the (direction of the) development of the Grid is the concept of services. The current Grid architecture as described in the beginning of this chapter is structured in terms of protocols. Instead of this protocol-centered view, the Grid is now seen from a functional point of view. To meet the needs of a (virtual) organization, the Grid has to supply a set of services [26]. One increasingly popular type of services is known as "web services".
Each vendor, standards organization, or marketing research firm defines Web Services in a different way. Gartner, for instance, defines web services as "loosely coupled software components that interact with one another dynamically via standard Internet technologies." Forrester Research takes a more open approach to web services as "automated connections between people, systems and applications that expose elements of business functionality as a software service and create new business value." [27]. The more technical definitions however agree on the fact that a web service is a network-accessible application based on open standards, plus a formal description of how to connect to and use the service[4].
Several aspects of web services are desirable when heading to a functional oriented Grid. Examples of these aspects of web services are service description and discovery; automatic generation of client and server code from service descriptions; binding of service descriptions to interoperable network protocols; compatibility with emerging higher-level open standards, services and tools; and broad commercial support.

[4]See *http://www.jeckle.de/webServices/* for more than ten different definitions of the term web service.

The majority of the current Grid applications has been developed for the scientific domain. Generally speaking the technology should be useful for several domains, including the commercial one. This domain has some specific demands which can not be fulfilled using the current Grid technology, such as seamless integration with existing resources and applications, tools for workload, resource, security, network Quality of Service, and availability management.
One important difference between web services and (future) Grid services is the state of the service. Web services offer an interface to a persistent, network accessible application. Grid services on the other hand can be created and destroyed dynamically, just like the allocation of resources. These types of services are called *transient*. This dynamic state has significant implications on the management, naming, discovering and usage of the Grid services. The Open Grid Services Architecture (OGSA), a framework which is currently in development and builds on the foundation of the Globus Toolkit and on the existing web services framework, addresses these issues. Ultimately the Globus Toolkit will be integrated and conform to the OGSA.

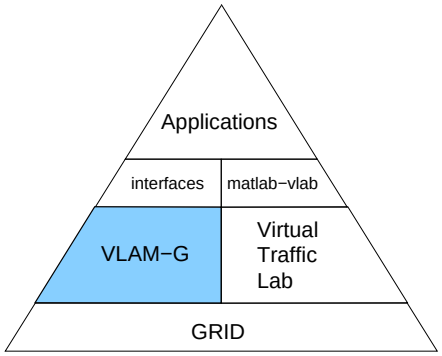
2.6 Summary

We have outlined the concept and architecture of the Grid. We have shown the strengths and weaknesses, and discussed possible ways to solve these weaknesses. In the next chapter we describe middleware for the Grid, that tries to bridge the gap between the distributed computing of the Grid and the application layers above: the Gridbased Virtual Laboratory AMsterdam (VLAM-G).

Chapter 3

VLAM-G

3.1 Introduction



Simulation and real world experimentation both generate huge amount of data. Much of the effort in the computer sciences groups is directed into giving scientists smooth access to storage and visualization resources; the so called middle-ware on top of the Grid-technology. In this chapter we will describe one particular type of middle-ware called VLAM-G, short for Grid-based Virtual Laboratory AMsterdam.

3.2 The Virtual Laboratory

VLAM-G wants to hide resource access details from scientific users, and allows scientific programmers to build scientific portals. These portals give access to the user interfaces to scientific studies: combinations of information gathering, processing, visualization, interpretation and documentation. A typical application is a large instrument, gathering so much data, that the analysis has to be off-line.

The Virtual Laboratory facilitates the software and hardware-control for working on large studies in distributed environments. Yet, it is important not to forget the human factor. Large studies involve multiple scientist to cooperate. A Virtual Laboratory has to facilitate the means to communicate between those scientist, for instance when they can not be in the same room due to time constraints or the geographical distance.

VLAM-G offers a distributed analysis platform for applied experimental science. Summarized in one sentence, VLAM-G is *a science portal for remote experiment control and collaborative, Grid-based distributed analysis in applied sciences, using cross-institutional integration of heterogeneous information and resources* [4].

This laboratory is called "virtual" for several reasons. The allocation of resources is invisible for the user, when the user performs an experiment it looks as if it runs on one virtual supercomputer. Besides that, VLAM-G aims to actually *be* a Virtual Lab, in the sense that scientists that are geographically dispersed can work together on experiments as if they were in the same laboratory. This requires collaborational tools to support the scientists, tools that will be described in more detail in paragraph 3.5.

3.3 Users

The Virtual Lab is meant for users who want to perform scientific experiments. These experiments consist of the analysis and/or visualization of large quantities of (sensory) data. Prerequisite is that they are computationally intensive and can be decomposed well. If these experiments are relatively simple, it would probably be quicker and more efficient to use an alternative of the Virtual Lab. If the problem can not be decomposed the advantages of the Virtual Lab can not be fully utilized.
For somebody with a desktop application there is no real use of the Virtual Lab. However, a demanding application which needs lots of computational power has to be executed on a powerful system. One option is for instance to use SARA[1]. In that case, the user has to setup (software) connections to SARA, send and receive data, monitor the progress, etcetera. This is unnecessary and time-consuming work from the point of view of the user, who just wants to get his work done and see the results. The alternative which saves him this extra work is the Virtual Lab.
Besides those advantages, the Virtual Lab offers a set of predefined modules, for instance hundreds of different visualization modules that can display a vector, display 3D data, rotate 3D data and perform other visualization routines. This offers the scientist options that were previously unavailable.
Another option is for instance to go to CERN in Switzerland to perform some physics experiment. The use of the Virtual Lab saves time that otherwise would be spent on traveling and on adapting the software to prepare it for execution at CERN.

If we assume that the problem is suitable for the Virtual Lab, then it is interesting to see what type of users the Virtual Lab has and at what level they operate. We discern three sorts of actors: the resource manager, the scientific programmer and the scientist.

At the bottom of figure 3.1 we see the Resource Manager. The Resource Manager is responsible for the Grid infrastructure: he grants access to the local resources, configures the environment to make sure that VLAM-G modules can run, provides Scientific Programmers with some general VLAM-G modules, and monitors the resource usage of the VLAM-G studies by the Scientists.

The Scientific Programmer (in the middle of figure 3.1) creates study templates. Study templates are a decomposition of the study in a number of logical steps, represented by a process-flow graph. Some of the steps are forms, that

[1]SARA Computing and Networking Services supplies a complete package of High Performance Computing- and infrastructure services, based on state-of-the-art information technology, and is located at the Wetenschap & Technologie Centrum Watergraafsmeer (WTCW)

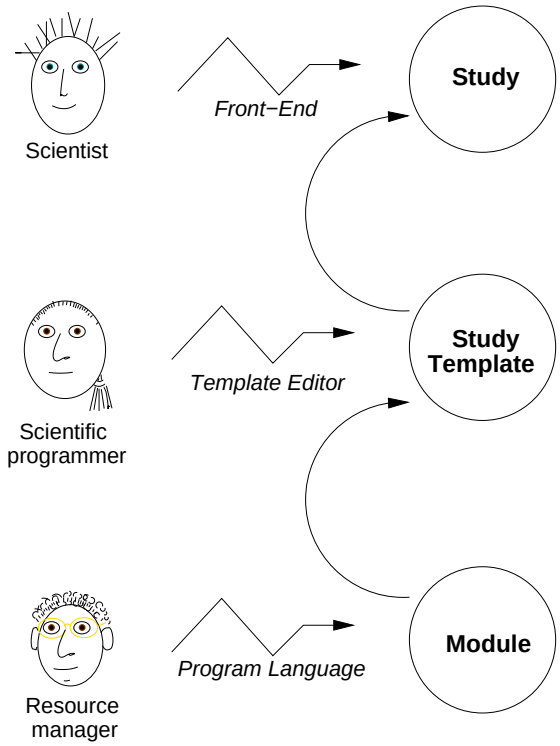


Figure 3.1: Three different actors for a study

force the scientist to document the experiment by providing meta-data (sample-description for instance). Other steps are real operations, some manual, some by real devices, and some by the Grid. The Grid operations are defined as experiments: a number of computational modules coupled via ports. The Scientific Programmer takes care that there exists a template with those steps and experiments, that is nearly ready to execute, so that the Scientist only has to fill in the details specific for this sample.

The Scientist is the one that actually is performing the studies. He first prepares a study by filling in the required meta-data, and adjusts some of the parameter-settings to his own personal preferences. He then starts the experiment, which steps through the different operations specified in the template. Some steps can be quite time-consuming, so the Scientist can log-out from this study, prepare another study, and come back to inspect the intermediate results and perform another step of the study.
So, when the Scientist starts working with VLAM-G, the Resource Manager and Scientific Programmer have done their job, and do not have to be present. Their roles are taken over by two systems: the information management system VIMCO (Virtual laboratory Information Management for COoperation) and the Run Time System (RTS).

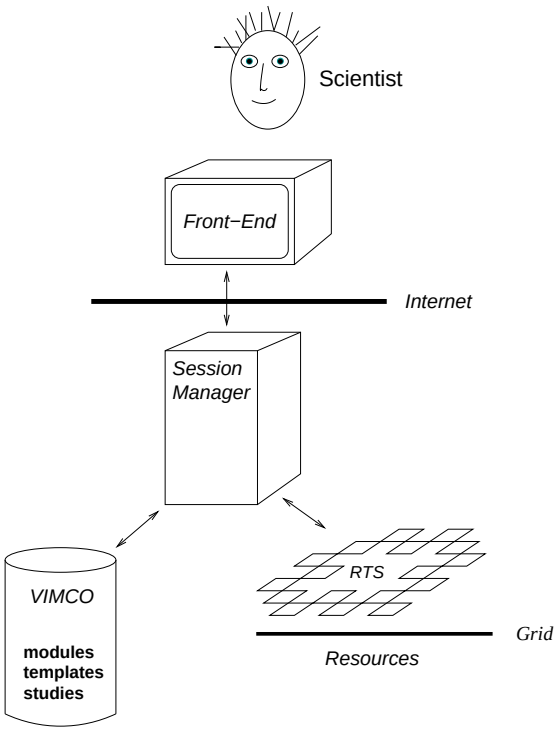


Figure 3.2: The different systems for a study

3.4 Architecture

The description of the different actors for a study already mentions some of the subsystems that are a part of VLAM-G. Seven principal components can be identified: the Graphical User Interface (GUI), Session Manager, Collaboration, PFT Assistant, Module Repository, Run Time System (RTS) and VIMCO.
The GUI is the only part of VLAM-G that the user directly interacts with. He can select an existing Process Flow Template or choose to create a new experiment. The assistant can aid the user during this process, by suggesting appropriate modules or experiment topologies. When the experiment is ready to be executed, it is passed in XML form via the Session Manager to the RTS. The Session Manager makes it possible for a user to work on multiple studies, execute one study while preparing another study. The systems are coupled via Session Manager, which provides a single access point to VIMCO (the preparation) and RTS (the execution). VIMCO provides access to study and experiment descriptions that are stored in application specific databases. The RTS takes care of scheduling, instantiating and monitoring the computational modules of an experiment. It makes extensive use of Globus services to perform these tasks. The Collaboration System will offer audio and video communication, an electronic whiteboard and possibly other collaborative aspects to enable cooperation between scientists that participate in an experiment.

The entire architecture of VLAM-G can be seen in figure 3.3. The dotted boxes located on resource A and B represent examples of modules instantiated by the RTS as part of a specific experiment.

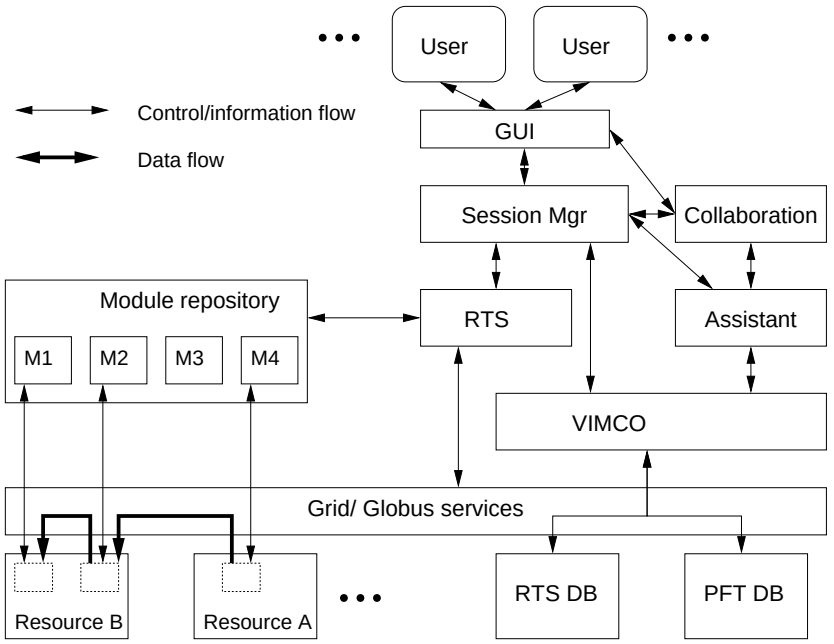


Figure 3.3: The architecture of VLAM-G

How did this architecture develop, which aspects are realized and which are still work in progress?

3.5 Development of VLAM-G

In the fall of 2001 the University of Amsterdam started the development of the Virtual Laboratory. It was decided that there were several objectives that had to be met. An incremental systems development process was followed to gradually achieve these objectives.

First milestone was the development of the Run Time System. The role of the RTS is described in the previous paragraph.

Second milestone was to provide APIs for each component of the Virtual Lab. This ensures interoperability between components while they can be developed concurrently. Each component is implemented stand-alone, while integration tests take place at fixed moments in time. During this phase, the GUI which enables users to compose experiments has been developed. The user can drag & drop modules from the module repository, and can supply meta-data about the experiment by filling in the template. The VIMCO and session manager have also been implemented. Their function is described above.

Integrating security aspects is the third milestone. (Grid) Authentication at the lower levels, making the system *hacker proof* against possible intruders from the outside world and enabling fault tolerance are important security aspects

that are taken care of in this phase. "Fault tolerance" involves the migration of processes to other computers in case of a malfunction (i.e. hardware failure) of one of the currently used computers.

One of the objectives that is yet to be reached is the implementation of collaborational aspects. In principal it should not be a big step to implement these features. The major part of the required functionality is already available for one person to operate the Virtual Lab. Aspects that need attention are typical multi-user problems, such as the "protection" of an experiment when multiple persons are modifying parameters or modules, keeping the view up to date for all participants and possibly integrating third party software for video conferencing and/or which provide a multi-user electronic whiteboard. A (commercial) example of such a groupware application is DOLPHIN [29], which is suitable for face-to-face and remote meetings, which provides an electronic whiteboard, audio/video-connectivity and which supports the creation and manipulation of informal structures (e.g. free hand drawings, handwritten scribbles) as well as formal structures (e.g. hypermedia documents with typed nodes and links). An Open Source example of a collaborative computing environment is the Collaborative Virtual Workspace (CVW) [30]. This project uses the metaphor of virtual rooms where different users can gather. To a user, a CVW is a building that is divided into floors and rooms, where each room provides a context for communication and document sharing. CVW allows people to gather in rooms to talk through chat or audio/video conferencing and to share text and URLs with one another. Document types include whiteboards, URLs, notes and other documents edited through the user's local applications (e.g., word processor, spreadsheet). Documents that can be edited through local applications are managed through a document server within CVW, which provides a universally available file space.

Because VLAM-G is still in development, some of the features have a lower priority than others and have been delayed until a later moment. The assistant is one of these features. Ideally, some sort of intelligent agent should support the (non computer) scientist when preparing (and possibly while executing) an experiment. This assistant may suggest module definitions or study templates of previous experiments. Decisions of the assistant are planned to be supported by knowledge gathered from the RTS and the application databases under VIMCO.

One of the strengths of VLAM-G is the existence of a large library of generic modules which perform a specific task or computation. This does require a module repository however that has to be filled, for several different domains (e.g. biology, chemistry, physics). A one time effort has to be made to adapt or convert current blocks of software to the module specification of VLAM-G [39]. Other modules will only be added or developed when actual users perform experiments, and can give feedback about frequently used operations (modules) that are not yet supplied by VLAM-G.

To summarize, VLAM-G provides a GUI, structures to encapsulate and interface to experiment specific software and hardware, data storage and visualization systems. Standardized Grid services are used to execute experiments in a transparent manner. In the future it will offer an assistant to guide a scientist through the process of structuring his study, by giving a unified view and intelligent advice. It will also have a collaboration system which will enable scientists to jointly perform experiments while they and their instruments are located at different locations.

3.6 Comparison of Virtual Labs

We are not the only group that wants to provide a Virtual Laboratory. Several other universities are developing systems that are somewhat similar to our Virtual Laboratory. We will give a short description of these projects and will point out what the main similarities and differences are.

Molecular Modelling for Drug Design, Monash University Australia [31].
"The Virtual Laboratory project is engaged in research, design, and development of Grid technologies that help in solving large-scale compute and data intensive science applications in the area of molecular biology. The virtual laboratory environment provides software tools and resource brokers that facilitate large-scale molecular studies on geographically distributed computational and data grid resources. This helps in examining/screening millions of chemical compounds (molecules) in the Protein Data Bank (PDB) to identify those having potential use in drug design."
This project shares the use of Grid technology to solve a computationally heavy problem. The most important difference between these two projects is the application domain. This example focuses on the specific (sub) domain of molecular modelling in chemistry. The explicit starting point of our Virtual Lab however is its generic infrastructure. Domain specific knowledge is put into (reusable) modules, but not into the Virtual Lab itself. This way the Virtual Lab can (in theory) be used as a laboratory for every imaginable domain.
The Virtual Lab from Monash University also lacks tools to support collaboration within the Virtual Laboratory.

Materials Microcharacterization Collaboratory, Argonne National Laboratory, Lawrence Berkeley National Laboratory, National Institute of Standards and Technology, Oak Ridge National Laboratory, University of Illinois [32].
"The Materials Microcharacterization Collaboratory links several laboratories through videoconferencing, shared data-viewing, and collaborative analysis. Materials Science is a blend of a multitude of disciplines ranging from basic science to applied engineering, from physics and chemistry through metallurgy and ceramics, in which researchers combine their expertise with state-of-the-art instrumentation to push forward the frontiers of materials technology. The team members of this project collectively house virtually every characterization technique which employs electrons, ions, photons, x-rays, neutrons, mechanical and/or electromagnetic radiation to elucidate the microstructure matter. [...] The ultimate goal of the program is to create a virtual laboratory where all sites can interact with one another and share information and expertise. [...] Another large component of the MMC is its educational aspect. By putting these resources on the internet, it will allow teachers and students from all educational levels to access the virtual lab. Additionally, with proper security, these people will be able to interact and control electron microscopes themselves, and even perform experiments."
This laboratory has multiple application domains just like our lab. It has interfaces to and remote control over several different instruments. Our lab aims to offer similar access, but this has not been realized yet. The MMC does not utilize the Grid or Grid services to perform analysis or visualization, in contrast to our laboratory which used the Grid as a foundation for everything else.

Virtual Lab, FernUniversität Hagen (Germany) [33].
"This contribution presents a collaborative virtual environment for a remote laboratory. Students have access to the remote laboratory via Internet from anywhere at any time. The remote laboratory is based on a client/server architecture, which is mainly implemented in the Java programming language. [...] The collaborative environment allows the experimentation in a team. The group is able to interact and to discuss the results of their work."
The focus in this lab is on the collaborative aspects and the universal access provided by the platform independence. Our lab is also platform independent, it eventually will include the same collaborative tools as are used in this case.

We can conclude that VLAM-G is the only project that combines all of the mentioned aspects; it is Grid-based, platform independent and suitable for different (scientific) domains. Add to this the fact that VLAM-G will be extended in the future with a collaborative environment and interfaces to real instruments, and it is clear that it comprises the most ambitious and comprehensive Virtual Laboratory to date.

3.7 The future of the VLAM-G

While the first official release of VLAM-G is imminent (approximately September 2002), this does not mean that is finished. Besides the obvious bug fixing that will take place based on errors encountered by the first users, they will (hopefully) supply comments, suggestions and feature requests based on their experiences with the Virtual Lab. We attempt to provide feedback on our experiences in this thesis.
The first release of VLAM-G will not conform to the ideal vision of the Virtual Laboratory. The current GUI for instance is made so that it works. It is build using the logic of computer scientists, which is (probably) not equal to the logic of other (non-computer) scientists. It is pretty straight forward with tables and forms that contain data about modules, there is no stepwise procedure that assists the user while creating or editing an experiment. The GUI still implicitly requires an understanding of the internal working of the Virtual Lab, because it does not hide details like internal data structures of modules and process flow templates. Creating a user friendlier GUI which operates on a conceptually higher level is one of the challenges for the future. The planned assistant is a step in the right direction trying to achieve this goal.
Another direction that can be pursued is adapting the GUI to (the users of) the different domains. One can imagine that the type of user or the type of domain requires a different GUI. By "a different GUI" we do not just mean the look and feel of the Virtual Lab, but for instance the amount of information that is presented on the screen. It could be that a physicist wants to control every detail of an experiment, while a biologist just wants to have a few buttons like "Go" or "Auto-select optimal scenario". The GUI has to support the user and has to comply with the way the user thinks and acts in the Virtual Lab. In order to determine what the preferences of the users are, it would be advisable to conduct interviews or to use prototyping to develop the GUI.

The credentials system is one of the components that also has to be improved

in the future. Right now the credentials for the Grid are stored locally on each computer. A much more elegant solution would be something like a chipcard reader and a PIN-code as an authentication method. The same ease of use should be available for the Virtual Lab.

Right now access to the Virtual Lab is still limited to computers that are located on desks. Wireless access on all sorts of devices would offer more freedom to the users and would increase the virtuality of the lab (just imagine checking out intermediate results of an experiment while traveling from one meeting to another).

VLAM-G is aimed at scientists and uses scientific resources. The architecture however is very generic, and could also be used for domains that have not been integrated into VLAM-G. One could imagine that certain types of businesses would be very interested to have access to an environment such as VLAM-G to perform analysis on large data sets. Possible examples are telecom providers who want to analyze their GPS data or large companies in general that want to search their client data for interesting patterns. Although the architecture could in theory support these new domains, VLAM-G currently does not embody an important aspect: expressing the computational services in terms of money. How much it costs for the end user to use a developed module is unknown. It is also unknown how much it costs to perform an entire experiment. One could imagine that the user would like to have the option to choose between performing the experiment in one day for a certain amount of money (say ten thousand dollar), in three days for a quarter of this amount or in one week for one thousand dollar.

The examples above show that considerable work has to be done before VLAM-G will fully reach the goal of providing a science portal for remote experiment control and collaborative, Grid-based distributed analysis in applied sciences.

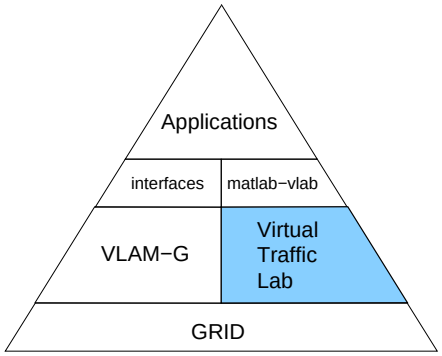
3.8 Summary

VLAM-G offers a distributed analysis platform for applied experimental science. It provides a GUI, structures to encapsulate and interface to experiment specific software and hardware, data storage and visualization systems. Standardized Grid services are used to execute experiments in a transparent manner. In the future it will offer an assistant to guide a scientist through the process of structuring his study, by giving a unified view and intelligent advice. It will also have a collaboration system which will enable scientists to jointly perform experiments while they and their instruments are located at different locations. VLAM-G is the only project that is Grid-based, platform independent and has a generic infrastructure that is suitable for different (scientific) domains.
In the next chapter we will describe the traffic domain, for which we will design the Virtual Traffic Lab.

Chapter 4

The Virtual Traffic Lab

4.1 Introduction



The Organisation for Economic Co-operation and Development (OECD), an international organisation with over thirty member countries helping governments tackle the economic, social and governance challenges of a globalised economy, states the following in an overview document about their Road Transport Research Programme [35] :

> "[...] as a general rule road investments lag behind what would be necessary to properly address infrastructure limitations. However, even if sufficient funding were available, it is likely that the problems would still exist because the conventional approach of building more roads is hampered for political, financial, social, and environmental reasons. In addition, there is more and more resistance to building new roads because it is believed that it has proven to often compound the problem by simply inducing a more rapid rate of travel growth.
>
> The challenge [...] is to identify or develop the ways and means to alleviate traffic-related problems without building new roads. The two principal ways in which these problems can be addressed without new roads is through the application of better traffic management measures and the development of new technologies."

In the previous chapter we have taken a look at the Virtual Lab. We stated that VLAM-G will offer several science portals that can support scientists. In this chapter we focus on one of these portals: the Virtual Traffic Lab. Purpose of this portal is to support researchers to aid them in their quest to design, analyze, evaluate and improve new technologies in the area of traffic management.
Traffic research covers a wide span of areas, ranging from traffic psychology to road safety, from the planning, design and evaluation of infrastructure to

analysis and solutions for negative aspects of traffic such as pollution, accidents and congestion of the roads. Great expectations are placed on advanced road transport technologies to control traffic congestion. This control should lead to a more efficient road use and hopefully to less (serious) accidents. Other research efforts focus on alternative transportation methods instead of automobiles to reduce the amount of traffic and therefore road congestion and the amount of pollution.
Not all of these topics are suited to be incorporated into the Virtual Traffic Lab. Since experiments consist of a number of modules that each perform a (set of) calculation(s), only those areas of traffic research that require that kind of tools are candidates for the Virtual Traffic Lab. Research areas that evolve around systems engineering, simulations and analysis of large quantities of data are particularly suitable, because they are likely to fit into the topology of the Virtual Lab.

4.2 Background information on traffic research

In this section we outline some basic terms and abbreviations that are common in the field of traffic. We will refer to these concepts in the rest of this chapter.

The past few years the Dutch government has been contemplating and investigating the use of road pricing as a means to try to control the increasing amount of traffic and congestions. The payment for the use of a transport service without any action from the user at the moment of the use of the service (in this case a road), is denoted by several terms: Automatic Debiting, Electronic Fee Collection (EFC), road pricing or in Dutch *Rekening Rijden*.
The systems are not allowed to impose any constraints on the traffic flow, which means that the techniques have to be used which can cope with cars performing lane changes while traveling with high speeds.

The Dutch Government organized a tendering procedure in 1999 to test and select an Electronic Fee Collection system, which will be used for road pricing in the future. A twofold approach was taken: field tests were used to demonstrate the performance of several EFC systems under real life conditions on a part of the A12 near Utrecht in The Netherlands, and simulations were used to investigate the systems for a large amount of traffic under different circumstances.

Figure 4.1: Photograph of gantries over the A12 near Utrecht

Electronic Fee Collection systems consist, in principle, of a group of technical components, which together perform the functions necessary for automatic fee collection. The systems are not allowed to impose any constraints on the traffic flow, which means that techniques have to be used which can cope with cars performing lane changes while traveling with high speeds. Gantries are therefore positioned over the road at fixed locations. Attached on top of these gantries are cameras that monitor and register traffic on the road, and antennas that perform communication with equipment at the road-side and with vehicles on the road. Every vehicle has a device mounted inside the car behind the front window called an On-Board Unit (OBU). It also contains a smart card which keeps the data required to allow the transaction and (possibly) acts like an electronic purse. The EFC communicates with the OBU using the antennas above the road. This (local) exchange of messages over a short distance is called Dedicated Short Range Communication (DSRC) [40].
Figure 4.2 schematically displays the way an EFC system works [42].

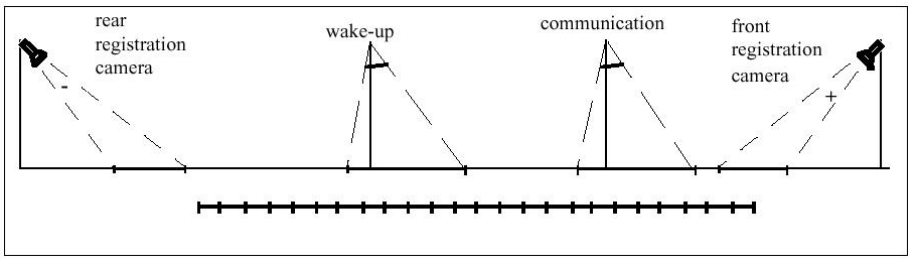


Figure 4.2: An Electronic Fee Collection System

This figure represents a fictitious EFC, i.e. it describes functionality an EFC should possess, but it is not an actual EFC that is positioned on a highway (the cameras are for instance facing each other, which would not be the case in a real EFC). The functions contained in this figure can be described as follows (partly based on [41]) :

Entering ADS zone - When the vehicle has entered the automatic debiting zone, a photo is taken from the rear license plate.

Vehicle tracking - An Inductive Loop System (ILS) in the road tracks the movements of the vehicle during his passage of the ADS zone. A laser curtain can also be used to detect the vehicles.

Initialization of communication - The charging procedure starts with waking up the OBU through a signal broadcasted by a DSRC beacon on top of a gantry.

Read OBU data - The OBU replies with data relevant for the charging, such as its ID.

Determination of class - The class to be applied for the determination of the fee will be determined according to a classification scheme.

The input for this determination can either come from
- the data read from the OBU or
- a measurement of vehicle characteristics or
- data collected from some available database (using the number plate)

Determination of fee - The due fee will be calculated on the basis of the determined class.

Debit of charging account - The due fee is put as a debt on the users electronic purse. If the user does not have an electronic purse or if the communication with the OBU fails, the bill will be sent to the address corresponding with the license plate.

Registration - The complete charging transaction is registered centrally. If the transaction has succeeded, the photograph of the rear license plate will be removed from the system. If the transaction failed, a photo of the front license plate will be taken and is sent to the back office for further processing.

Q-Free is one of the consortia that participated in Dutch tendering procedure. Q-Free's system[1] uses gantries that are located above the road to measure passing traffic on multiple lanes. It registers the width and height of the vehicles and exchanges messages with the OBU through antennas that are located on top of the gantries.
Part of the QFree system is OBU localisation. The OBU localisation is a functionality performed by the DSRC equipment. Every uplink transmission from the OBU to one of the antennas can be used to estimate the position of the OBU within an accuracy that is limited by the antenna setup. The unique OBU identifier needs to be kept together with the localisation data and the transmission time. With this data a set of all position estimates belonging to one OBU can be generated, and can be compared to the result of a completed EFC transaction. The system can now check whether a specific vehicle (with a given OBU) has paid.

Experimenting with real EFC systems is not something that can be done everyday. Besides the fact that there exist few operational EFC systems, not all of the desired conditions can be controlled, such as different types of weather and different traffic volumes. This is where traffic simulators come in.
A simulation is the execution of a model, represented by a computer program that gives information about the system being investigated. The decision whether to use a discrete or a continuous model for a particular system depends on the specific objectives of the study. For example, a model of traffic flow on a freeway would be discrete if the characteristics and movement of individual cars are important. Alternatively, if the cars can be treated "in the aggregate," the flow of traffic can be described by differential equations in a continuous model [43]. In case of a discrete model, the activities of the model consist of events, which are activated at certain points in time and in this way affect the overall state of the system. Events exist autonomously and are discrete, so between the execution of two events nothing happens.

[1] *http://www.qfree.com/References/tolling/rekning_rijden/rekning_rijden.html*

Traffic simulators offer the possibility to (re)view several scenarios in a very controlled environment. One parameter, such as the traffic density, can be changed, and the effects can be seen in the simulation. Once the simulator is properly calibrated and validated (we will discuss this in more detail in paragraph 4.3.1), predictions can be made about interesting questions such as under which certain circumstances congestion forming takes place.

4.3 Requirements determination

During the development of the Virtual Lab, it was decided that an analysis workbench for electronic fee collection would be one of the application domains. This decision leads to the development of a (sub) system within the framework of the Virtual Lab. The logical step following this decision is to choose the appropriate systems development process.

We are dealing with a changing environment, the specifications of the Virtual Lab can change because it is still in development. The (traffic) user requirements can also change, because they become more aware of the features the Virtual Lab (will) offer. Because of these relative uncertain circumstances, we have chosen the evolutionary approach of systems development. This approach breaks up a project into separate parts, and then, one by one, each of the parts is taken through the systems development process (requirements - design - implementation - testing). Each part either adds to the functionality of one of the earlier parts or integrates into the system with other parts. The emphasis is on a learning process, whereby users and developers refine the requirements or learn more about the possibilities of the technology, from the experience of developing and testing a given part, and then use this knowledge to shape the development of the next part [11]. After every part has been integrated the final phase of systems development is reached, the maintenance phase.

The development of the entire Virtual Traffic Lab is much too comprehensive for this thesis. We will perform the requirements determination phase, which in turn consists of several other phases.
Problem definition is the first step. We take a look at the problems in traffic research at the University of Amsterdam and explain how we think these can be solved by and will from VLAM-G.
Normally a *feasibility study* will be conducted. During this phase it is ascertained whether the proposed system will help to attain organization objectives (i.e. does it fit in the organization strategy?) and what the economic (i.e. costs versus the expected benefits), technical (i.e. hard- and software requirements) and operational feasibility (i.e. acceptation by the users) of the proposed system is. In our case the decision to develop the Virtual Traffic Lab as part of VLAM-G has already been taken. The manpower and funds are allocated as part of this project, the required hardware and software is available, and since the traffic researchers actually are developing the system themselves, the acceptation should not be a problem. Because of these premises we assume that a further extensive study into these factors is not necessary.
Requirements acquisition involves collecting and analyzing information sources that describe the tasks that are performed. This can range from observing or interviewing the current users to the analysis of existing system documents. We have made an inventory of tools that are currently used to perform traffic anal-

yses, based on manuals and reports that describe these tools. Conversations with and demonstrations by the end users took place to explain unclear parts of the manuals or tools.
Finally during the *requirements analysis* we translated the descriptions of the tools into a more formal and structured list of requirements. We make a distinction between *functional* and *technical* requirements. Functional requirements describe what the system should do. These requirements are based on the goals users want to reach and the tasks they intend to perform with the system. Technical requirements describe technological solutions or the types of technology components for the functional requirements.

Many studies show that the lack of, or inadequate requirements, are a major cause of system failure, where "failure" is defined as not meeting predefined expectations by the contractors, or even actual system bugs or crashes. Boehm [12] states that the cost difference to correct an error in the early phases of system development as opposed to following phases is in the order of one to one hundred. In other words: we need to pay attention to the requirements, otherwise we will regret it in the next phases of systems development.
Requirements are statements of *what* a proposed system is supposed to do. This is separate from *how* the function is to be accomplished: the *how* is a design issue, not a requirements issue. For example: "The system shall allow users to select from a set of predefined modules", is a requirement. On the other hand, "Module descriptions will be stored in a Matisse database", is a statement of how something might be accomplished, and is therefore not a requirement. The design of the system can only start after completion of the requirements phase. We start with explaining the problems that are present in traffic research.

4.3.1 Problem definition

Two elements are essential for experimental traffic research at the University of Amsterdam: traffic simulators, and files or databases that contain logs from real or simulated traffic. These two factors consume most of the time when performing or analyzing traffic experiments.
It might seem obvious to think that simulating specific traffic situations is the most important issue and challenge in traffic research. What is less known, is the fact that the preparation and interpretation of data that is produced by real or simulated traffic is far from trivial.
A lot of meta data about a simulation or about a conducted traffic experiment has to be registered in order to provide a solid basis for hypotheses and their proof. An example is the accuracy of the sensors in or above the road that are used to detect traffic. The way they are configured has to be known, if not, a certain configuration of the sensor may for instance lead to the situation where trucks pass the sensor and are (wrongfully) classified as "regular traffic" instead of "heavy traffic". This has severe consequences for the interpretation of the recorded traffic data.
Another problem is occlusion, or in other words: visual obstruction. In the context of EFC systems "occlusion" is used to denote the situation where a specific aspect of a vehicle (i.e. the OBU or license plate) is blocked from sight by another vehicle, and therefore making it impossible to establish communication or to make a photograph. EFC systems have to comply with several System

Quality Factors such as preventing free rides (where a violator, through an error in the sensor system or coordination system, is not identified as such and therefore is not registered). The Dutch government demands the chance for this event to happen to be smaller than one in a million.
The calibration of (real or virtual) sensors therefore needs attention. This is especially difficult for simulators, because it requires large and various amounts of traffic data. Calibration of a traffic simulator involves the process of fine-tuning model parameters with the objective of reproducing specific traffic flows [36]. Once it is calibrated, it also needs to be validated. Validation is the process of checking whether a model (including fine-tuned parameters) reproduces traffic flows for which it was *not* calibrated. By definition, since traffic flows differ with each situation, validation results will not be as good as calibration results. However, validation allows application of the model for other traffic flows than the flow(s) for which the model is calibrated. This is of course one of the major strengths of traffic simulators, although one still has to keep in mind that it presents a possible outcome and not necessarily the one and only outcome.
One of the related problems that traffic researchers face is the lack of registered and available traffic flows. It takes a lot of careful planning and organization to produce valid and interesting data from real traffic flows. Our group uses data that is derived from RDW's Test Centre Lelystad[2] and from an electronic fee collection test that was held in the beginning of 2000 at the A12 near Utrecht. The problem is that there is hardly any reference material, benchmarking data that can be used to validate the experiments. In some cases this data is available, but it lacks information about the specific conditions (about the weather, the vehicles, the sensors) surrounding the registration of the traffic flow. In the worst case this renders the data useless, or requires lots of time and energy for manual inspection and editing of the data.
Traffic scientists also encounter problems that are not directly related to their area of expertise but that have to be solved anyway. An example is the use of overhead cameras above the road; the pictures or movies of traffic that are recorded show perspective and radial distortion, caused by the angle and by the lens of the camera. Computer graphics techniques have to be used to try to compensate for these distortions.

Some tools have been developed through the course of the last few years to support researchers in coping with these problems. These tools are not generic however, they are specifically engineered to perform one task and are not designed to cooperate with other tools.
The users have expressed the wish to have an analysis workbench at their disposal. VLAM-G offers a generic approach for problem solving and offers reusability of (parts of) tools. It promises to deliver a more integrated set of tools than the current disjoint collection.
The cooperation and sharing between traffic researchers might be stimulated by the collaborative aspects that VLAM-G will offer. Multiple users from different countries who are also doing research on electronic fee collection, or traffic management in general, would be able to perform analyses in the Virtual Traffic Lab, and could learn and benefit from each others experiments. This could also lead to an increase of validation data.

[2] *http://www.rdw.nl/eng/diensten/ondernemingen/tcl/index.htm*

Because VLAM-G utilizes the Grid, the time it takes to perform analyses can be reduced significantly. This time benefit can lead to a faster progress or to a more detailed study.
To (re)develop all the routines that are now available for VLAM-G would take too much time and effort for the purposes of this thesis (which aims to give an exploration of the Virtual Traffic Lab and not an entire implementation). We therefore limit the scope of the solution to the problems to the integration of the current tools into VLAM-G.

To summarize, several problems exist that have to do with traffic simulators, and with data generated by these simulators or by real traffic flows. While most of these problems can be solved right now with the current tools, the expectation is that VLAM-G will offer, besides the distributed computational resources, more flexibility through its module repository, which will have to be filled with specific traffic operations. Additional (indirect) benefits are expected, caused by the collaborative aspects of VLAM-G. Development of the Virtual Traffic Lab will first be directed towards the integration of the current tools into VLAM-G.

4.3.2 Requirements acquisition

In order to be able to integrate the current tools into VLAM-G, we need to investigate two things. First, we need to make an inventory of the tools that are currently used to aid researchers in preparing or performing analyses, and in general, which tasks the users perform. Second, we need to analyze these tools and tasks and distill wishes and requirements. If we split them up and can distinguish atomic (sub) tasks, we have the basic decomposition in modules for VLAM-G.
In this paragraph we will give a short description about each tool and will summarize its functionality. This inventory will offer us an overview of tasks the traffic researchers perform at the University of Amsterdam.

ADSSIM

ADSSIM is short for Automatic Debiting System SIMulator, and is a joint development of CMG and the University of Amsterdam. ADSSIM is a discrete event simulation environment for Automatic Debiting System simulations. ADSSIM is one of the components of a traffic simulator. The entire simulation starts with a Traffic Generator, which generates amongst other things several types of vehicles, their arrival time and their intended speed. These variables can be based on injection files containing real traffic flow data or on a stochastic process. The generated traffic is handed over to the Road Traffic Simulator. This simulator determines the trajectory of each vehicle through the ADS section of the road. It also determines longitudinal and lateral behaviour of each vehicle on the road. This behaviour is based on several parameters such as the intended headway (in case of a car in front) and the intended speed for longitudinal behaviour, and the "wish" to make a lane change and small in-lane changes for lateral behaviour. Events are scheduled which represent all activities of the ADS. These events represent e.g. a start-up of communication, OBU activity, a sensor activity or a registration activity. For each vehicle all ADS activity is logged. Next, after the vehicle leaves the ADS, it enters an analysis module, which generates estimates

of the desired System Quality Factors. These factors are requirements that are specified by the contractor of the ADS, the Dutch government in case of the Rekening Rijden project. The System Quality Factors can be seen on the right side of figure 4.3.

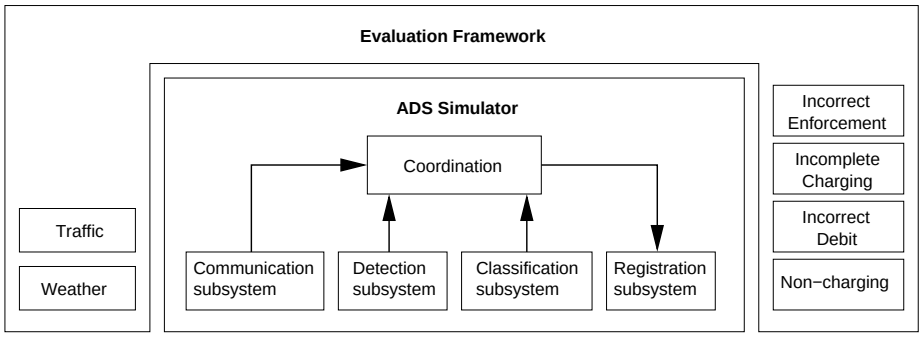


Figure 4.3: ADSSIM system overview

This structure closely resembles the functionality and steps which are described in the background information on traffic research (section 4.2). The communication subsystem exchanges messages with the OBU inside the vehicle. The detection subsystem detects the vehicles with sensors when they are passing gantries. The classification system determines the type of vehicle. When these readings are combined, almost every vehicle can be uniquely identified (besides a few exceptional cases which will have to be investigated further).
The evaluation framework outlines interesting scenarios that can be examined, such as different types and amounts of traffic, and different types of weather. On the right side the requirements for the ADS which have to be evaluated after the simulation run has finished.

VideoRecorder Tool

This tool can capture and display footage that was recorded on video using an overhead camera on a gantry. A text file containing the desired start time, end time and filename is given to the application. It digitizes and compresses the selected analog video signal and saves it on disk.
The application also enables the user to view a selected movie. The interface offers the ability to select a movie, play it forwards and backwards at regular and high speed, pause the movie and to copy the current frame to the clipboard. The videotool offers an API built in ActiveX, a standard that enables software components to interact with one another in a networked environment, which means that other environments such as MatLab are able to control the videotool.

Detection of occluded vehicles

Purpose of this tool is to seek occlusions of specific vehicle locations from a sensor by other vehicles, based on a given ADS configuration file, a traffic file and trigger condition of the sensor. Three possible situations exist where occlusion can take place: occlusion of a sensor pointing in the traffic direction, occlusion of

a sensor pointing *against* the traffic direction and lateral (side-ways) occlusion.

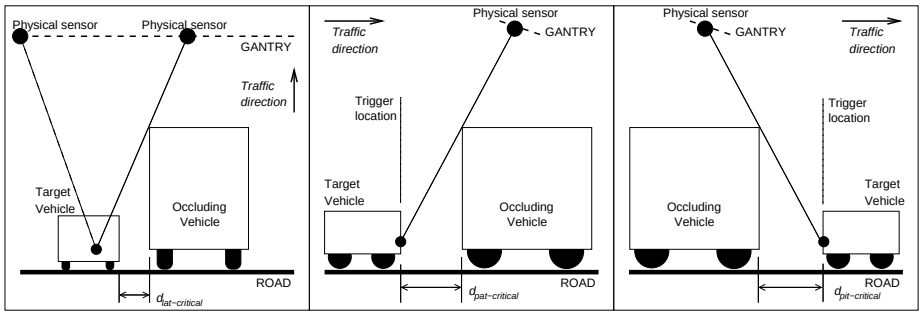


Figure 4.4: Three types of occlusion. On the left lateral occlusion, in the middle occlusion of a sensor pointing against the traffic direction, on the right occlusion of a sensor pointing in the traffic direction.

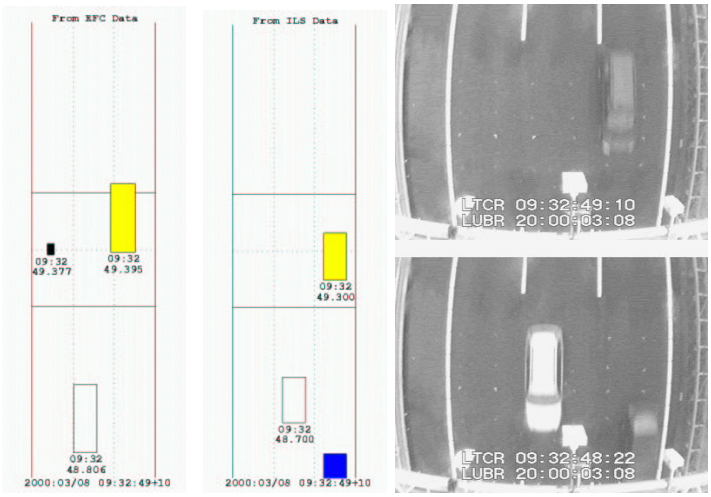
In each of these situations there exists a critical distance between the vehicle that has to be registered by the sensor and the vehicle that causes the occlusion (typically a truck or lorry). These three distances first have to be calculated for a given setup (in a simulation). Once these distances are calculated, their occurrence in the traffic files has to be found. Traffic files do not contain distances between vehicles, but the inter-vehicle distance can be calculated using the inter-arrival time and velocity of two vehicles. The resulting distance can then be compared to the critical occlusion distance(s).

Detection data analysis tool

Purpose of this tool is to give visual insight into communication messages that are used to detect vehicles. The communication logs contain thousands of lines, each consisting of seventeen separated numbers that indicate the OBU ID, the communication zone, etcetera. For a human reader it is difficult to interpret these messages. The analysis tool can filter detection and communication messages from a logfile. A listbox is presented to the user in which he can select a date, and a half hour or hour interval on that date. The logfile is indexed dynamically in order to display the dates and the time intervals, this intermediate result is stored in a temporary file. The file is then used for visualization of the data in the selected interval.
Another option is to use a listbox which displays the test cases that were performed. The user selects one of these cases, and all the available data for this test (usually several hours in one day) is displayed in a plot. After selecting a test, OBU data needs to be matched to the detection data. This process is performed by a validation tool, which calculates and validates the matches. Afterwards the user is able to manually remove or add a match.
The results that are displayed in the plot can offer a starting point for further analysis and give an indication of the detection quality.

DataVisTools

This tool gives a (symbolic) visual representation of the location of vehicles as recorded by laser curtains and the ILS of the EFC. It also displays an AVI which contains the actual footage of the traffic on the road taken by the overhead cameras. This tool was used to (manually) determine and locate discrepancies in the measurements by the ILS and overhead cameras.
Two lanes are shown vertically, with cars represented by blocks moving on this road. The user interface offers the scientists the possibility to step through the data on two scales: time based and frame based. The user can move forward and backward on a logarithmic scale.



(a) EFC data (b) ILS data (c) still frames

Figure 4.5: DataVis tools

We selected this figure on purpose, to show one of the exceptions that can be determined using this tool. Several things can be noticed in the example figure. The symbolic representations and still frame show an example of a situation where the detected data is slightly offset; the time, location and length of a detected vehicle differs when viewing the ILS and overhead camera data.
Figure 4.5a also shows a small black square. This denotes a detected vehicle, while the still frames do not show a vehicle in the left lane. Detailed analysis of the (moving) frames showed a drop of water falling down from the gantry on the road, with light reflecting on it. This probably caused the (incorrect) detection of a non-existent vehicle.

ComVis

During the development and testing of Q-Free's Rekeningrijden Electronic Fee Collection system a tool called ComVis was created. ComVis (short for Communication Visualization) is a graphical tool that presents the Dedicated Short Range Communication in an easy to understand fashion. This tool is similar to

the detection data analysis tool, but ComVis is able to visualize different types of communication in several plots.
The program runs under MatLab 5.3 and is written as a collection of function scripts. ComVis extracts data from logfiles which contain time-stamped lines. It only extracts lines that contain specific markers which denote interesting messages. Based on these messages, several visualizations are presented to the user. ComVis displays plots which show the number of messages involved in each passage, the type of activity (transmission or reception) and the transmissions and receptions per zone. These plots show data of all the registered OBUs, but the user also has the possibility to view the data of an individual OBU. Appendix B contains an example of the plots made by ComVis.

OBU Localisation Comparison Tool

An EFC system has to comply with a specified accuracy of OBU localisation. Analysis of the accuracy can be used to evaluate this aspect of a EFC system, but it can also be used as input for this aspect in a simulation.
This tool compares the lateral OBU localisations (as seen by the communication system of Q-Free) to the position as seen from the overhead camera.

Figure 4.6: Still frame of traffic passing through a gantry

The communication logs are filtered for localisation messages. The user specifies a date and time, a transaction number and an OBU number, which results in a matrix containing the position messages. This matrix is converted to a script file which contains commands for the VideoRecorder utility which is described in section 4.3.2. A MatLab routine reads the AVI-file that is created by the VideoRecorder utility and enables the user to scroll interactively through the frames of the movie. On the frame where the OBU is above the detection line, the user clicks on the position of the OBU. This frame and the location (image coordinates) are saved.
The final step is the conversion of the location of the OBU from image coordinates (i.e. pixels) to real world coordinates (i.e. meters). A pinhole camera model [38] is used to compensate the perspective and radial distortion caused by the camera (which can be seen in figure 4.6).
The OBU localisation as detected by the overhead camera can now be compared to the OBU localisation that is based on communication messages. The mean,

minimum and maximum error and the standard deviation are calculated for the specified OBU.

Observations

While studying the manuals and tools, we made some observations and had some conversations about the way the traffic researchers are used to work.

Traffic users are demanding, they will not accept one single button which promises to solve all problems. They expect the ability to view and (if they see that as necessary) add, edit, or remove intermediate data. This requirement has influence on the level of detail of the tools. A more fine grained set of tools has to be created to fulfill this requirement.

The traffic scientists perform two roles. On the one hand they develop traffic tools themselves, on the other hand they use these tools to verify hypotheses and to perform analysis and visualization on traffic data. The first role requires a thorough understanding of topics in computer science. It is not surprising to see that the traffic scientists have a background in computer science. The second role requires domain specific knowledge, in this case knowledge about traffic flows, Automatic Debiting Systems, OBUs, etcetera. Because traffic scientists combine the computer science and traffic domain, we make two assumptions.
The first assumption is that a high level or abstract user interface is not absolutely necessary. If some of the details of the underlying implementation are shown to the user, the traffic scientists probably would not mind because they are used to it.
The second assumption is that we expect more feedback from traffic scientists about applications then from users from other domains. This assumption is based on the fact that we think that traffic scientists have a better understanding of the underlying techniques and implementation than other users (such as biologists or chemists). We expect that this will result in more accurate feedback and in more feedback in general. This does not mean that feedback from other users will not be appreciated, on the contrary, users with a non-computer science background should be able to supply valuable feedback exactly *because* of this reason. They can provide feedback about the usability that the developers did not think of because they are "too close" to their application.

The majority of the tools are MatLab-based. This environment is used because of the powerful data analysis and visualization tools it offers. After we have summarized the requirements, we will explain what consequences this has.

4.3.3 Requirements analysis

In the previous paragraphs we have described several tools that are used to perform specific tasks. We analyzed these tasks and have translated them into a more formal description. Before we specify these requirements, we first summarize frequently used abbreviations.

Abbreviations

ADS	Automatic Debiting System
ADSSIM	Automatic Debiting System SIMulator
DSRC	Dedicated Short Range Communication
EFC	Electronic Fee Collection
ILS	Inductive Loop System
OBU	On-Board Unit

Requirements specification

Functional requirements

FUNC01	The system shall be able to visualize DSRC communication of an EFC in an easy to understand fashion.
FUNC02	The system shall provide access to ADSSIM.
FUNC03	The system shall provide processing algorithms on ADSSIM- or EFC-logfiles.
FUNC04	The system shall provide the option to select part of an ADSSIM- or EFC-logfile based on user specified characteristics.
FUNC05	The system shall provide access to the VideoRecorder Tool.
FUNC06	The system shall be able to compare two sets of localisation data.
FUNC07	The system shall be able to give a symbolic visualization of a road with traffic.
FUNC08	The system shall offer the user the possibility to view, add, edit, or remove intermediate data.
FUNC09	The system shall be able to detect occurrences of occlusion in ADSSIM- or EFC-logfiles.

Technical requirements

TECH01	The system shall be able to provide input to ADSSIM in the form of a runtime specification defined in several files.
TECH02	The system shall be able to read logfiles from an EFC and from ADSSIM.
TECH03	The system shall give the user the option to specify one or more communication markers, which will result in a cleaned up EFC- or ADSSIM-logfile containing only those messages.
TECH04	The system shall give the user the option to specify which of the available parameters of the logfile should be selected.
TECH05	The system shall give the user the option to specify a time and date, or a time and date interval, which will result in a cleaned up EFC- or ADSSIM-logfile containing only data of this time and date (range).
TECH06	The system shall give the user the option to view a list of all of the performed real life or simulated traffic runs and will register the selection.

Technical requirements (continued)

TECH07	The system shall be able to graphically display the location of detected vehicles at a certain moment in time, based on a given (selection of a) logfile. Desired display: x-axis: time (hh:mm), y-axis: y-position (m).
TECH08	The system shall be able to draw plots of DSRC, for every OBU that was logged and for each individual OBU: Number of messages in each passage: x-axis: time (s), y-axis: number of messages. Type of activity: x-axis: time (s), y-axis: transmission and reception. Transmission / reception per zone: x-axis: time (s), y-axis: zone (zone number), separate markers for transmission and reception.
TECH09	The system shall be able to give commands to the VideoRecorder Tool via an ActiveX-interface
TECH10	The system shall be able to create a scriptfile suitable for the VideoRecorder Tool based on a given date, time, transaction number and OBU number.
TECH11	The system shall be able to read and display an AVI-file.
TECH12	The system shall be able to retrieve the image coordinates based on the position where a user clicked in a still frame of an AVI-file.
TECH13	The system shall be able to convert image coordinates to real world coordinates, based on a pinhole camera model that compensates for perspective and radial distortion caused by the camera. Necessary inputs: image coordinates (x, y) and camera position (x, y, z, ω, φ, κ).
TECH14	The system shall be able to calculate the mean, standard deviation, minimum and maximum error for two given sets of (lateral) positions during a time interval (y_1, t_1, y_2, t_2).
TECH15	The system shall be able to compare two sets of location data (ID, y, t) and shall give the matches as a result.
TECH16	The system shall give the user the option to view a set of location data (ID, y, t) and manually add or remove rows.
TECH17	The system shall provide a top view of a road with vehicles represented as blocks, based on EFC and ILS data contained in files. This display shall be accompanied by a user interface which enables the user to move seconds or frames forward or backward, based on a logarithmical scale.
TECH18	The system shall calculate the critical distance between two vehicles so that one does not occlude the other vehicle (as seen from the sensor point of view). Necessary input: type of occlusion (lateral, sensor pointing in traffic direction, sensor pointing in traffic direction), ADS configuration file, traffic file, trigger condition of the sensor. Output: occlusion distance (m).

Technical requirements (continued)	
TECH19	The system shall be able to find occurrences of inter-vehicle distance smaller than a given distance (i.e. critical occlusion distance). Necessary input: sensor passage time of a vehicle (s), vehicle velocity (m/s). Output: occurrence of inter-vehicle distance smaller than specified (true/false), inter-vehicle distance (m)

The requirements that have been determined form a starting point for the design and implementation of the Virtual Traffic Lab. The development of the entire Virtual Traffic Lab is too comprehensive for this thesis. We want to prove the functionality and benefits of the Virtual Traffic Lab by developing a prototype. This prototype will not demonstrate the full possibilities of the Virtual Traffic Lab, because that would require the development of over a dozen separate modules. The majority of these tools runs in MatLab. We have chosen to use the currently available traffic tools as much as possible. In order to use these tools in combination with the Virtual Lab, an interface is needed between these two.

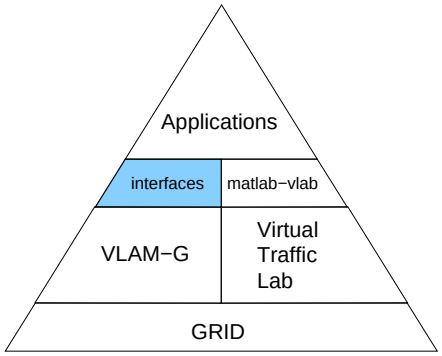
4.4 Summary

We have discussed several problems which exist for the traffic domain, that have to do with traffic simulators and with data generated by these simulators or by real traffic flows. While most of these problems can be solved right now with the current tools, the expectation is that VLAM-G will offer a significant time reduction to run these tools. The distributed computational resources are expected to decrease the time spent on waiting for an analysis to complete. The resources of the Grid also provide far more storage capacity than currently available.
Another expectation is an increased flexibility through VLAM-G's module repository. Additional (indirect) benefits are expected, caused by the collaborative aspects of VLAM-G. We have analyzed the current tools and have specified the functional and technical requirements for the Virtual Traffic Lab based on these tools. The requirements can form the starting point for the development of several traffic modules.
Because traffic scientists are demanding, a fine grained set of modules will have to be created to satisfy their needs. The development of the Virtual Traffic Lab will first be directed towards the integration of the current (MatLab based) tools into VLAM-G.
In the next chapter we discuss some common terms and concepts such as MatLab and XML that will be used to create an interface between VLAM-G and the current traffic applications.

Chapter 5

Interfaces

5.1 Introduction



In order to realize mobile applications, an interface is needed between the Virtual Lab and an environment in which the mobile applications can run. This environment has to comply with certain requirements.
First of all, general knowledge and acceptation of the environment by the (end) users is a prerequisite.
Earlier on we already described that the analysis and visualization of (sensory) data are essential tasks as part of experiments. So in the second place, the selected environment has to be able to perform these tasks.
Thirdly, the environment has to support various options for modelling and simulation. And finally, the environment must offer interfaces to external systems, such as databases and hardware interfaces (sensors) but also other programming environments like C, C++ and Fortran.

5.2 MatLab

Environments that meet these requirements are MatLab and Mathematica. MatLab is the de facto standard for scientific computing, enjoying wide use in industry and universities. MatLab already offers a database interface to Matisse, a (multimedia) database which will be used in the near future to store the descriptions of the Virtual Lab-modules. Considering these arguments we favor the Matlab environment for complex system engineering as traffic systems.

The increase of computational power the past few decades has been largely influenced by the introduction of multiple processors and parallel processing architectures. These systems offer a significant performance advantage over conventional (single processor) systems. High Performance Computing has matured over the decades, and the availability of key applications (such as database

applications) make these systems interesting not only for the academic world but also for commercial customers [21]. MatLab however is an application that is still based on a single processor. In the past, MathWorks has developed a few experimental versions of MatLab for parallel computers. They found three difficulties while evaluating these versions, which lead to the conclusion that a fully functional MatLab running on parallel computers was not viable [20]. The three reasons for this decision are:
the memory model; it takes longer to distribute the data than to do the actual computation.
granularity; MatLab spends only a small portion of its time in routines that can be parallelized, like the ones in the math library. It spends much more time in places like the parser, the interpreter and the graphic routines, where parallelism is difficult to find.
business situation; there are not enough potential customers with parallel machines to justify fundamental changes in the MatLab architecture.

In principle the first two reasons could also apply to the Grid. The expectation however is that these factors will play a small role, since the Grid will be used for computations of large blocks of data.

Despite MathWorks decision not to develop a parallel MatLab, there have been a number of initiatives to circumvent this situation. We can distinguish four general approaches to providing parallel functionalities to Matlab:

1. Provide message passing routines (like PVM / MPI) in Matlab.
   The user can issue PVM- or MPI-calls inside MatLab, can start MatLab processes on other machines and then pass commands and data between between these various processes [15]. Examples of implementations are MultiMATLAB, Cornell Multitasking Toolbox for Matlab, DP-Toolbox, MPITB/PVMTB, MATmarks, Parallel Toolbox for MATLAB and MatlabMPI.

2. Provide routines to split up work among multiple Matlab processes.
   A tool which enables the user to start MATLAB processes on remote machines or on multiprocessor machines. The tool allows the user to spawn a set of slave Matlab sessions "underneath" the current session forming a Virtual Machine [16]. Loops where the individual loop iterations can be performed independent of one another are executed on the different machines. The amount of inter process communication is minimal [17]. These types of problems are commonly referred to as embarrassingly parallel. Examples of implementations are the MULTI Toolbox, Paralize, PMI, PLab, Parmatlab and the MATLAB Parallelization Toolkit.

3. Translate (sequential) MATLAB code for parallel platforms.
   A compiler translates ordinary MatLab scripts into (for instance) C programs, which invoke parallel function libraries such as ScaLAPACK. Main advantage besides the parallelism is the fact that the original code (MatLab scripts) does not have to be adapted to create the parallel version [19]. Examples of implementations are Otter, RTExpress, ParAL, FALCON, CONLAB Compiler, MATCH and Menhir.

4. Enable parallel processing inside MATLAB.
   As explained before, MathWorks has decided not to pursue a parallel MatLab because they estimated this would not be viable at that moment in time (1995). However, if the communication bandwidth is no longer a bottleneck and if there is enough demand for a parallel MatLab, they might reconsider and re-evaluate the possibilities for parallelism within MatLab.

There exist some crossovers which combine the message passing routines of the first approach with the spawning of multiple MatLab processes, such as PPServer / MATLAB*P [18], Netsolve, DLab, Matpar, PLAPACK and Paramat.

As Dongarra states in his article "High Performance Computing Today", two things remain consistent in the realm of computer science: I) there is always a need for more computational power than we have at any given point, and II) we always want the simplest, yet most complete and easy to use interface to our resources. In recent years, much attention has been given to the area of Grid Computing. We want to combine the ease of use of MatLab with the computational power that is provided by the Grid. We have chosen to try to re-use as much of the available tools as possible.

Ideally we do not want to alter the existing MatLab code and want to apply a wrapper around the relevant parts so we can use it in the Virtual Lab. MatLab offers the option to create a standalone application based on MatLab-files. The idea is to develop a new or use an existing algorithm in MatLab to perform a (set of) operation(s), and use the MatLab Compiler to create a C shared library or a C++ static library. The algorithm can then be integrated into another application, in our case the Virtual Lab, by loading the library.
A different approach is to execute external C-code using the so-called MEX-construction. This construction consists of two distinct parts [34]:

1. A computational routine that contains the code for performing the computations that you want implemented in the MEX-file. Computations can be numerical computations as well as inputting and outputting data.

2. A gateway routine that interfaces the computational routine with MATLAB. The interface is provided by the function *mexFunction* which requires a number of right-hand input parameters and a number of left-hand output parameters. The gateway calls the computational routine as a subroutine.

The figure 5.1 shows how inputs enter a MEX-file, what steps the gateway function performs, and how outputs return to MATLAB.

MEX-files run in the same process space as the MatLab interpreter. When the user invokes a MEX-file, the MATLAB interpreter dynamically links in the MEX-file. This is different from the first approach, because stand-alone C or C++ applications run independently of MATLAB.
In the next chapter we describe the implementation of the MatLab-VLAM-interface, and we also discuss the choice between these two approaches.

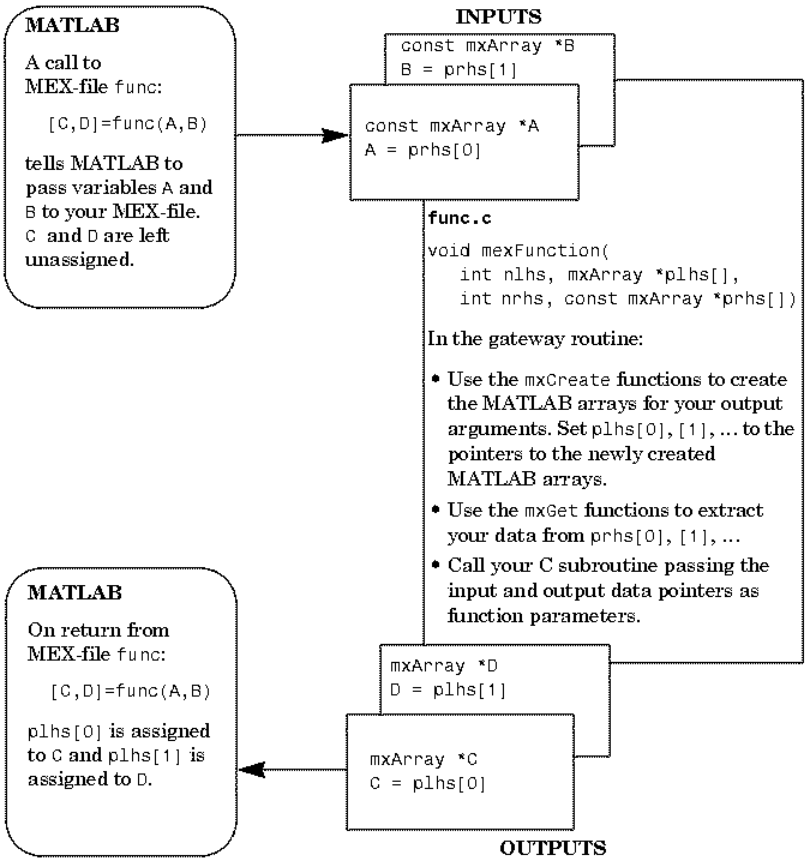


Figure 5.1: The MEX-cycle

One of the advantages of MatLab is the amount of toolboxes and additional software that is available. One of MathWorks main products is Simulink.

5.3 Simulink

Simulink is a simulation and prototyping environment for modelling, simulating, and analyzing real-world, dynamic systems. Simulink provides a block diagram interface that is built on the core MATLAB numeric, graphics, and programming functionality. It provides a graphical simulation environment called StateFlow for modelling and designing event-driven systems. It makes extensive use of blocksets, collections of application-specific blocks that support multiple design areas. Examples are electrical power-system modelling and digital signal processing. These blocks can be incorporated directly into Simulink models. Each block has one or multiple inputs, outputs and mathematical functions that together form the state of that particular block at a given moment in time.
One of Simulinks predefined blocks is the S-Function. An S-function (system-function) is a custom code module that defines the behavior of a Simulink block. Simulink provides templates for creating custom S-functions using existing or

newly-developed code (C, Ada, Fortran, or MATLAB). Simulink also provides a masking capability. This allows us to create a custom user interface, called a mask, for any subsystem or S-function block. The mask can include a custom icon, parameter dialog, online help, and initialization script. Using this mask we can alter a block's appearance and user interface. These features will be very useful for creating our interface with VLAM-G.
Simulink blocks and models are saved in a format called MDL.

5.4 MDL

MDL is short for Model Definition Language. Using Simulink, one can create a model which consists of several connected blocks. The constructed model, its connected lines and specified parameters are stored in an ASCII-file with the .mdl-extension. The internal format used is MDL. Each system or subsystem is described by the corresponding name followed by curly brackets. Inside these brackets the system's parameters are specified. Parameters are stored using the construction *name (tab) "value"*.
MatLab and Simulink run on various platforms (Windows, Macintosh and several Unix and Linux platforms such as Solaris, IRIX and HP-UX) which means that users can exchange their Simulink models by exchanging MDL-files (models and possibly libraries). Nonetheless MDL is a MatLab specific data format. If one wants to execute or validate a certain model in another simulation environment, the only option is to rebuild it manually which takes a lot of time and effort. The possibility that errors are introduced during this process is very likely. A more generic data structure would circumvent these problems and would offer a more universal exchangeable format. XML is an example of a generic language that could be used instead of MDL.

5.5 XML

XML is short for e*X*tensible *M*arkup *L*anguage. It is a common syntax for expressing structure in data. XML originated in 1996, as a result of frustration with the deployment of SGML on the Internet.
Key concept underlying SGML (Standard Generalized Markup Language) is separating the representation of information structure and content from information processing specifications. Information objects modelled through an SGML markup language are named and described (using attributes and sub elements) in terms of what they are, not in terms of how they are to be displayed or otherwise processed.
The SGML family of standards that include SGML (the modelling framework), DSSSL (the transformation framework for presentation) and HyTime (the linking and timing framework) are ISO standards that proved difficult to implement and aroused little interest outside of specialist fields of expertise. XML simplified the requirements for implementation, with the specific intention of enabling deployment of markup applications on the Internet. XML is a dialect of SGML that is designed to enable 'generic SGML' to be served, received, and processed on the World Wide Web.

XML has several characteristics that make it a powerful and flexible lan-

guage. It separates the content (within XML) from the structure (schema) and from the presentation (eXtensible Stylesheet Language - XSL).
XML Schemas describe the structure and meaning of the information within an XML document. The schema determines the permissible tags for the document as well as defining what type of data they can contain. XML Schemas are described in Document Type Definition documents (DTDs) or XML Schema - Data Reduced documents (XDRs). DTDs are the older way of describing the structure and meaning of an XML document. Because the syntax for DTD's was difficult to comprehend and limited, XDR, a new, more flexible mechanism, was developed.
Using an XSL document you can transform the look of an XML document from one format into another. One of the key uses of this approach is to transform an XML document into an HTML document.
Because the structure and meaning of XML document content are known, a semantic search is possible. This means that not only the content data can be searched, but also the tag names and tag attributes, which makes it more accurate than regular (data content-only) searches.
XML is an open standard. Because it separates the content from the presentation, it is ideal for the interchange of documents between users and applications. Each person or program can choose its own output format, such as HTML, PDF or PS, while the exchange of the document(s) takes place in XML-form.
The *X* in XML stands for "extensible". This means that the person writing the XML-document can create new tags. The generic framework of XML makes it very flexible, everybody can modify and extend documents to their own needs.
XML handles relatively sparse data compactly. In contrast to for instance relational databases, where every row must contain data and where non-existent data is noted by a "null"-value, this tag can just be omitted in XML.

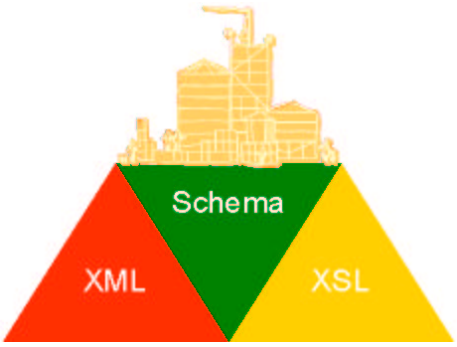


Figure 5.2: The relationship between XML, XSL and the schema

Earlier we stated one of the conditions to make sure that the Grid can operate correctly and fluently: a set of common languages and protocols. XML and derived protocols such as SOAP perform a key role in this area, because they make data portable. SOAP ("Simple Object Access Protocol") is a lightweight protocol for exchange of information in a decentralized, distributed environment [28]. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a

set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses. SOAP is an example of a standard that is able to send XML over HTTP.
All communication in VLAM-G is performed using XML. Topologies of experiments are for instance fetched from the VIMCO database, converted to XML and sent over HTTP to the RTS. Because all of the communication takes place in the form of XML, this offers the opportunity to interface other applications with (parts of) VLAM-G, as long as they comply with the definitions that are specified for VLAM-G.

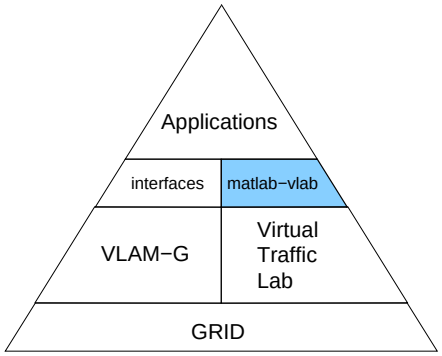
5.6 Summary

We have given background information on MatLab, which provides analysis, visualization modelling and simulation possibilities. MatLab however is an application that is still based on a single processor. For complex problems we would like to use distributed computing. We identified four general approaches to providing parallel functionalities to Matlab. For the Virtual Traffic Lab and VLAM-G we want to combine the ease of use of MatLab with the computational power that is provided by the Grid.
Simulink, MatLab's modelling and simulation environment, offers the possibility to create custom blocks. We believe we can use this functionality for creating our interface with VLAM-G. Simulink models are stored in the MDL format. An example of a generic language that could be used instead of MDL is XML. XML is also used in VLAM-G for the communication between subsystems.
Based on these elements, we can create and describe some interfaces for the Virtual Lab.

Chapter 6

MatLab-VLAM interfaces

6.1 Introduction



In this chapter we show several interfaces we have built, for the prototype of the Virtual Lab and for the actual implementation of VLAM-G. We will evaluate the pros and cons of these interfaces. The experience and tools gathered by us during this process can be used as input for the (future) development of the Virtual Traffic Laboratory.
In order to understand the interfaces and the way the Virtual Lab is implemented, we will first describe an experiment which demonstrates the basic elements. This case will be used through out this chapter.

6.2 CASE: histogram experiment

An experiment is composed of a number of modules, coupled by streams. As an example, the RTS prototype demonstrated an histogram experiment. This experiment consists of three modules:

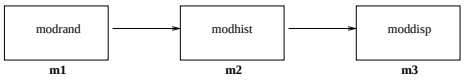


Figure 6.1: Topology of the histogram experiment

m1 modrand generates random numbers and sends these through its output port.

m2 modhist moves the numbers supplied at its input port into predefined databins. The bins are published on the output port.

m3 moddisp counts the numbers in the bins, and when reaching a predefined threshold, send it to the display.

A prototype of the RTS was build, with an interface to Tcl [6]. Building the experiment in Tcl looks like this:

```
# Loading all modules...
vlab load modrand.so mod_rand
vlab load modhist.so mod_hist
vlab load moddisp.so mod_disp

proc create {} {
    global range
    # Creating experiment...
    vlab create e
    # Adding modules...
    e add m1 rand
    e add m2 hist
    e add m3 disp
    # Setting modules...
    m1 set range $range
    m2 set range $range
    m3 set range $range
    # Connecting modules..
    e connect m1 out m2 in
    e connect m2 out m3 in
}

proc start () {
    e start
}

proc quit () {
    destroy .
}
```

Added to this code are some lines that create a simple GUI that is capable of creating, starting and modifying the experiment.The result of this simple example is a running histogram, for the RTS prototype visualised in the TCL-environment. Figure 6.2 shows a screenshot of the histogram. The height of each of the bars is updated multiple times per second. The maximum height can be controlled using the slider, which sets this parameter of the *mod_hist* module.

6.3 Implementation method

In the previous chapter we described two possible ways of implementing the MatLab-VLAM-interface: creating a standalone application based on MatLab-files, or executing external C-code from the MatLab prompt (from now called the MEX approach).

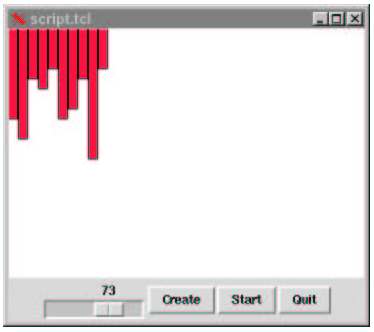


Figure 6.2: The histogram experiment executed by the prototype RTS

The standalone approach offers more advantages than the MEX approach, because of several reasons. First, it offers the analytical and visualization capabilities of MatLab without requiring an installation of MatLab. The standalone approach bundles the necessary MatLab routines together with the application code that needs to be ported from MatLab to C or C++. Because MatLab is not required in order for these "MatLab" modules to run, the threshold for potential other scientists to use the Virtual Traffic Lab is lowered.
Second, the MatLab code that has been written for the application needs little or no modification. In order for the code to work as a module in VLAM-G, it has to comply with the concept of input and output ports. If the MatLab code expects several inputs and produces several outputs, then the application can be seamlessly integrated into a VLAM-G module.
This approach can provide much added value to VLAM-G, the module repository can be greatly extended using these standalone MatLab applications.

The MEX approach works the other way around. Using this MEX construction, we want to call VLAM-G from MatLab. The idea is to utilize the Grid through VLAM-G for heavy processing (e.g. Fast Fourier Transformations). The resulting data will then be returned to MatLab where the final visualization can take place. Of course, if the visualization has to occur immediately, this method requires a continuously running MatLab while performing the experiment. Another option is to temporarily store the results, so the scientist can choose the moment and way of visualization in MatLab.
Downside to this approach is the fact that the GUI of VLAM-G cannot be used, MatLab will use a direct interface to the RTS of VLAM-G to be able to access the Grid.

Because the standalone approach seemed more promising, we tried to convert some of our code to C using the MatLab compiler, and compiled this code to a standalone executable. While this was succesful, the actual execution was not. The conversion and compilation of several MatLab scripts had unwanted side effects which prevented a correct execution. We ran into fundamental limits of the compiler, which can not cope with nested functions.
Due to these problems, we decided to persue the other implementation method: the MEX approach. The first part of this chapter is based on this approach. In the second part, we will describe an alternative GUI in MatLab to create experiments and to interface with the RTS of VLAM-G.

6.4 Prototype interface

The functionality of the interface to the RTS prototype is based on the API of the prototype Virtual Lab [5]. This API specifies the operators which are valid in the prototype Virtual Lab. The user creates an experiment in the Virtual Lab. Every experiment consists of a number of modules with one or more input- and output ports. These modules are shared libraries which are added to the experiment. The output ports of a module are connected to the input ports of the following module through a pipe (stream). When the creation of the experiment is completed, it can be started. One can give commands to each module. *set* is used to set the value of a certain argument before the experiment is started. *get* is used to retrieve the state of a certain argument. The operator *put* is used to set the value of an argument while the experiment is running. In the end an experiment can be stopped as well. The interface between MatLab and vlab must support all these operators.

6.4.1 MEX interface

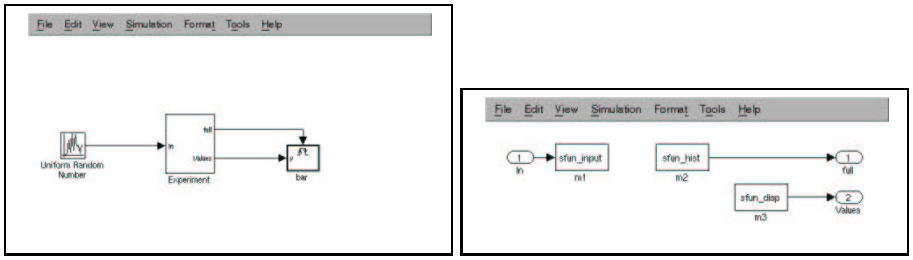
We have coupled a prototype of the Run Time System with the Matlab environment. Here the RTS is used for the heavy computational tasks, while the Matlab environment is used for analysis and visualization of the results. To be able to demonstrate the possibilities of Matlab as front-end, we have implemented a gateway routine as described in paragraph 5.2. Our interface between the RTS prototype and MatLab utilizes this gateway routine. The C subroutine that is run consists of the main function *vlab* which is based on the API-description as described above.
This gateway routine allows the user to access the Virtual Lab prototype from MatLab: he can load modules from the commandline, couple them, configure these modules and start the experiment. Our histogram experiment can be created from the command line using the following statements:

```
\% Loading all modules...
vlab('load','modrand.so','mod_rand');
vlab('load','modhist.so','mod_hist');
vlab('load','moddisp.so','mod_disp');
\% Creating experiment...
vlab('create','e');
\% Adding modules...
vlab('e.add','m1','rand');
vlab('e.add','m2','hist');
vlab('e.add','m3','disp');
\% Setting modules...
vlab('e.m1.set','range','10');
vlab('e.m2.set','range','20');
vlab('e.m3.set','range','30');
\% Connecting modules...
vlab('e.connect','m1','out','m2','in');
vlab('e.connect','m2','out','m3','in');
```

Although a powerful approach, such a commandline interface is not that attractive for unexperienced users. We therefore adapted our interface and made it more user friendly.

6.4.2 Simulink interface

Here the power of using a commercial analysis and visualisation environment as Matlab pops up. We have hidden our gateway routine inside a user-defined block of Simulink. Figure 6.3a shows an example of three graphical block diagrams. The first block generates random numbers, as module modrand. The last block displays the generated bins when triggered that the bins are full. The middle block represents a experiment that is sent to the RTS. The details of this experiment can be seen in 6.3b.



(a) top level

(b) bottom level

Figure 6.3: The histogram experiment in Simulink

Note at the bottom level that there is no line between the block sfun_input and sfun_hist. The connection between those two modules is made by the run time system RTS. The block sfun_input is a general utility to convert the information flowing over the line to the input port of a VLAM-G module. The block sfun_disp does the reverse, it converts the information flowing out of a port of a VLAM-G module to a line.

6.4.3 Evaluation

We have shown three different ways to access the prototype RTS. We will shortly review each of these and describe the different interaction levels.

Tcl prototype

We have extended Tcl so that you can give Virtual Lab commands on the Tcl-prompt. With Tcl, you can easily develop user interfaces. Yet, you have to implement the actual drawing routines for each user interface. This gives you low level control, but standard visualisation routines have to be developed from scratch.

Matlab command prompt

We have coupled a prototype of the RTS with the Matlab environment. A user can give arbitrary Virtual Lab commands on the Matlab prompt. Besides proving the basic functionality of the Virtual Lab, the coupling with MatLab gives the scientist the possibility for extensive analysis and visualization by using the predefined tools that Matlab provides. In our example the data that

is produced by the third module moddisp, is sent to Matlab which calls its bar-function to display the output. This shows the strenght of our Matlab interface: we could just as well have chosen another way of displaying the data by using another visualization function of Matlab, or by inserting it into a database.

Simulink

Simulink is a part of the Matlab suite. It uses graphical block diagrams which are created by selecting components from an extensive library of predefined blocks. The user can also define its own blocks that can incorporate existing C, Ada, MATLAB, and Fortran code. The Virtual Lab concept of re-usable modules can be fulfilled by using the extensive library of functions that Simulink has to offer.

In our example we used Simulinks standard uniform random generator as the the first block and used a threshold function for the last block, replacing the custom made modules of the virtual laboratory. When the standalone approach is operational, we can also convert the standard blocks from Simulink to standalone modules, thereby extending the module repository of VLAM-G.

Comparison of interaction levels

The RTS makes the actual execution of the experiment possible. The way of operating and the flexibility (or lack thereof) makes this interface difficult to use for inexperienced users. The Matlab command prompt interface is more flexible because Matlab provides the user with predefined functions for analysis, visualization and storage. While the possibilities are extensive, the user interface is still text-based and limited.
The Simulink interface offers a graphical user interface and at the same time a large repository of components.

While we were experimenting with the prototype, interfacing it with Matlab and Simulink, the VLAM-G group put their efforts into the Gridbased VLAM. This VLAM-G includes a Run Time System that accepts XML-files as input and spawns Grid-jobs.
The experiences gathered in creating the different interfaces to the RTL prototype helped us in creating a new interface for the RTS of VLAM-G.

6.5 RTS interface

Since the standalone approach is not (yet) working, we want to call VLAM-G from MatLab. The execution of an experiment in VLAM-G is handled by the Run Time System, which expects and accepts an experiment (and module) description in XML. This means that we have to arrange a way to create these descriptions in MatLab, so we can call VLAM-G.
One option could be a command line program which asks for several things: the number of modules, the name of every module, the number of input and output ports, the associated datatype, etcetera. This is not a very userfriendly approach and possible typing errors would require the input process to start all over again.
The other option is to use an environment within MatLab that is able to create

process flows and that allows click, drag & drop operations on components. Earlier we described Simulink, which matches these requirements. If we want to use Simulink as a modelling environment for VLAM-G in MatLab, we need to realize two steps:

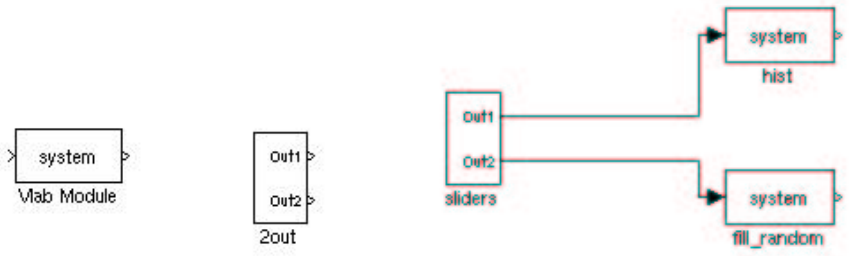
- Create one or more Simulink blocks that represent a VLAM-G module
- Create a convertor that translates MDL to XML

Before we can translate the experiment to XML, we first need the ability to create this experiment in Simulink. We designed a Simulink block to make this possible.

6.5.1 VLAM-G module in Simulink

We used one of Simulinks predefined blocks called "S-Function" to create an outline for a VLAM-G module. We used Simulinks masking capability to create custom parameters for this module. Appendix D.2 displays the generic XML description of an experiment, a module and the connection between the modules. We added all the necessary parameters that are mentioned in the XML module description to the masked system, so we can map these parameters later on to their appropriate location in the XML file.
The module that we have created is a basic one, containing one input and one output port. Several different experiment setups can be thought of that require modules that have multiple input and/or output ports. Our VLAM module will not suffice in those situations. To solve this issue, we have developed a different kind of VLAM module.
One of the basic building blocks of Simulink are input and output ports. These ports do not perform any operation, their function is to offer the opportunity to connect. When two or more of these ports are selected, they can be grouped into a subsystem. This subsystem can then be masked just like we have previously done with the S-function block. Added to this mask are the necessary parameters for the module. Using this approach we have created a VLAM module consisting of zero input and two output ports, called "2out". Appendix C describes how a custom VLAM module can be created, with an arbitrary number of parameters or input/output ports. As long as the module contains the required module parameters, i.e. specifies ports, the platform, datatype, etcetera, it will work as a VLAM module.
This generic VLAM module design offers several advantages. First of all, every possible port configuration can be made for a module, whether it has zero or ten input or output ports. Second, if the module definition of VLAM-G ever changes, we just need to add or edit this parameter to the masked module, and add this parameter to the list that has to be converted to XML (more about this in the next paragraph).

We added the (dummy) VLAM module based on the S-function and the 2out-module to a custom Simulink library 6.4(a).



(a) VLAM module library (b) experiment

Figure 6.4: The module library and an experiment based on this library

When a user wants to create an experiment in MatLab, all he has to do is start Simulink, open our custom Simulink library which contains the VLAM modules, and drag multiple instances of this module to a new model. He can connect the modules by dragging a line from the input to the output ports, and can specify the necessary parameters of each module. The masked modules present a simple screen which the user has to fill in (see figure 6.5).

Figure 6.5: The parameter screen of a masked module

The finished experiment now exists in Simulink and is stored in MDL. In order to execute the experiment, this description has to be converted to XML.

6.5.2 MDL2XML

We first performed a survey to determine whether there exists a tool that can convert MDL to XML, to prevent re-inventing the wheel. We discovered that

the Vanderbilt University in Nashville just recently released a MDL2XML tool as part of a framework called Unified Data Model, that will be capable of generating automatic semantic translators given the data model of the source and destination tools, and the translation/mapping specifications [37]. They have described the data models of Simulink and Stateflow as an UML Class diagram, accompanied by a description of the classes, attributes and relationships used. The UML diagram is converted by their framework into a Document Type Definition (DTD) of Simulink and Stateflow models. The DTD is used to validate XML that is generated by their MDL2XML tool.
Although their tool is not so much aimed at the result of the translation but at the actual translator(s) itself, the result looks promising. A generic XML file is created that contains the blocks, lines and parameters of the Simulink model.
We use Simulink for a specific purpose, i.e. to compose an experiment that can be handed over to the RTS of VLAM-G. We could therefore choose to use their tool, but would then have to create a convertor which transforms the generic XML description of the experiment to an XML definition that complies with the DTD of VLAM-G. Because it is relatively easy to retrieve the Simulink model parameters in MatLab, we decided to create our own MDL2XML, which translates the MDL directly to the XML description of VLAM-G.

We have developed a set of functions which each write a part of the XML file that describes the experiment topology. These functions write the header, modules, ports, instances and connections to a file. Each of these functions is called one or multiple times using the values which have been filled in by the user and which are extracted using the following MatLab commands:

```
blks = find_system(gcs, 'Type', 'block');
moduleNames = get_param(blks, 'moduleName');
moduleIds = get_param(blks, 'moduleId');
platforms = get_param(blks, 'platform');
locations = get_param(blks, 'proglocation');
datatypes = get_param(blks, 'datatype');
directions = get_param(blks, 'direction');
replications = get_param(blks, 'replication');
hosts = get_param(blks, 'host');
dns = get_param(blks, 'dn');
```

`find_system` access the currently opened experiment model in Simulink, and finds the blocks. The model (experiment) consists of blocks (modules), so every module is added to the `blks`-list. For every module in this list the module parameters are requested using `get_param`. These values can be passed directly to the functions which write parts of the XML file.
For determining which module has which input and output ports, which name is coupled to these ports and to which other module this module is connected, some counting and calculating has to be done. The name of each port can be retrieved the same way as the other parameter values, by using `get_param`. Modules that have more than one input and/or output port are subsystems, based on multiple in and out ports. This feature is used to detect the difference between a module that only has one input and output port, and a module that has multiple input and output ports.

```
blocktypes = get_param(blks, 'BlockType');
if findstr( cell2mat(blocktypes(i)), 'SubSystem')
```

For these subsystem modules each of the portnames is retrieved instead of just one portname. The portnames are stored in a matrix together with the corresponding moduleName and instanceID.
We now know the portnames, but we do not know which name corresponds to which port, and to which other port this module is connected. We use the lines that connect the modules to determine this information. Lines connect the output ports of modules to input ports of other modules. We retrieve all the line information, and determine for the start (source) and for the end (destination) of each line the name of the module it is connected to, and the portnumber that is used in Simulink.

```
lines = get_param(gcs, 'Lines');
j = 1;
for i = 1:length(lines)
    sources{j} = get_param(lines(i).SrcBlock, 'Name');
    destinations{j} = get_param(lines(i).DstBlock, 'Name');
    j=j+1;
    sources{j} = lines(i).SrcPort;
    destinations{j} = lines(i).DstPort;
    j=j+1;
end
```

The information about the ports is now extracted, we just need to combine it. To determine the mapping of a portname with the (internal) portnumber, we combine two corresponding matrices. For the output ports we combine the matrix `outputports` which contains the combinations of `portname`, `moduleName`, `instanceID`, and the matrix `sources` which contains the combinations of `moduleName`, `srcPortNumber`.
For the input ports we combine the matrix `inputports` which contains the combinations of `portname`, `moduleName`, `instanceID`, and the matrix `destinations` which contains the combinations of `moduleName`, `dstPortNumber`.
We have written a function `portnr2portname` which performs this combination for a given portnumber and returns the corresponding portname. This function is called when the connection between modules is written to XML .
We refer to appendix D.3 for the complete source code of the MDL2XML implementation.

6.5.3 Evaluation

We have created two VLAM modules for Simulink. The most common one consists of one input and one output port. The other is a subsystem based on two output ports. Both contain the required parameters to function as a VLAM module. We have written a manual which describes how to create a custom VLAM module, consisting of an arbitrary number of input and output ports. This manual can be found in appendix C.
During the design and implementation of MDL2XML, we have given attention to the fact that the XML description of a port, module or experiment might

change. After all, VLAM-G is still in development so changes in the description can not be ruled out. To minimize the impact of a possible change, we created modularized functions which write XML parts. Only the relevant XML function would have to be changed, editing or adding one extra line to this function would suffice.
If a module parameter has to be added, a small number of steps have to be taken to integrate this parameter into MDL2XML.

1. Add the parameter as a variable to the mask of the VLAM module subsystem.
2. Edit MDL2XML.M to retrieve the parameter and parameter value using `get_param` .
3. Edit MDL2XML.M to add the parameter to the appropriate XML function, which will write the new parameter to the XML file.

After completing these steps, the new parameter is found and converted to the XML file generated by MDL2XML.

6.6 Summary

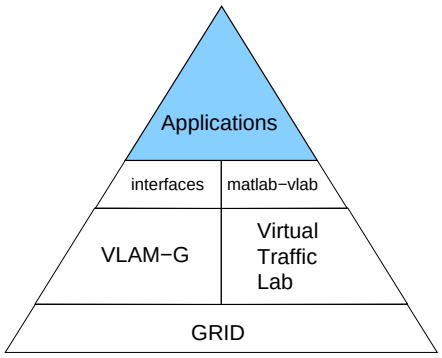
We have created several interfaces to the Virtual Lab. First we interfaced MatLab with a prototype of the RTS, we made it accessible from the MatLab prompt and from Simulink. The Matlab command prompt interface provides the user with predefined functions for analysis, visualization and storage. While the possibilities are extensive, the user interface is still text-based and limited. The Simulink interface offers a graphical user interface and at the same time a large repository of components.
For the RTS of VLAM-G we created a VLAM module library in Simulink, which enables the user to create experiments. We also created the utility MDL2XML, which can be used to convert the experiment from MDL to XML. During the design and implementation we have given attention to the fact that both tools have to be easily adaptable and extensible, and have to be generic.
The required interface between MatLab and the Virtual Lab is now available. This interface can be used during the development of applications for the Virtual Traffic Lab. In the next chapter, we will describe an example application.

Chapter 7

Application: ADSSIM

7.1 Introduction

We reach the highest level of abstraction on top of the Grid in this chapter, and discuss a traffic application for the Virtual Traffic Lab.



The Grid is already being used to solve specific type of problems that require large processing power and can be decomposed into smaller sub problems, such as Monte Carlo simulations and parameter sweep applications. Several initiatives have emerged to bridge the gap between the power and possibilities of the Grid and the ease of use of current applications. Examples of these initiatives are the European Data Grid[1] and the Particle Physics Data Grid[2], which both aim to develop and provide end to end integration and deployment of experiment applications using existing and emerging Grid services.
We stated earlier on in paragraph 2.3 about the Grid that most of the applications are designed for users with a background in computer science. We have designed and partially developed a traffic application that is more user oriented and aligns with the philosophy of VLAM-G.

7.2 Selection of the traffic application

We have specified a set of requirements for the Virtual Traffic Lab in paragraph 4.3.3. Based on these requirements several modules can be developed for VLAM-G. Not of all the desired modules consume huge amounts of computational power, such as *TECH09* which creates a scriptfile for the VideoRecorder Tool. To (fully) demonstrate the functionality of the Virtual Traffic Lab a resource intensive application (i.e. set of modules) has to be selected. Two applications fit into this category: camera calibration (part of the OBU localisation tool from 4.3.2), and traffic simulations. Initially we decided to select

[1] *http://web.datagrid.cnr.it*
[2] *http://www.ppdg.net*

the camera calibration application because we already have an implementation in MatLab to solve this problem. This implementation involves (amongst other steps) performing a parameter sweep which requires considerable computational resources [7]. However, since the standalone version of MatLab is not operational yet, we do not have the possibility to convert these scripts to an executable that can be wrapped into a VLAM module. Because ADSSIM *can* be called from a module, we have chosen to integrate ADSSIM into the Virtual Traffic Lab.

The scientist that starts a simulation is normally interested in (re)viewing multiple scenarios. Right now the execution of simulation runs happens sequentially, which can take a lot of time. The execution of one simulation run by ADSSIM on a standard configuration (Ultra Sparc 250 MHz / Solaris 5.6-5.8) for example takes between 3 minutes for a simple model, and over 5 hours for a complex one. Roughly speaking it takes one minute to simulate one thousand cars using a model of average complexity. The use of distributed computing to reduce the execution time of simulating multiple scenarios is an obvious but not a simple solution. Although significant amounts of time will be saved during the execution of the simulation runs, the scientist still wastes time adapting his simulation setup for the distributed environment, and still has to prepare each of the individual simulation runs manually. These two tasks will have to be performed all over again if the scientists wants to perform a different type of simulation. It would be a great advantage for the scientist if he would not have to set up the distributed environment and all of the simulation runs each time he wants to analyze certain aspects using a simulator. Ideally, the scientist would only have to specify the necessary input parameters and the desired output or desired way of displaying the output. This exactly resembles the philosophy of VLAM-G.

We will use a real example of a topic that is investigated by traffic scientists at the University of Amsterdam to demonstrate the benefits from the integration of ADSSIM into the Virtual Traffic Lab, and to show how we want to realize this implementation.

7.3 CASE: influence of heavy traffic on average speed distribution

Driving behaviour can be investigated by looking at the intensity of the traffic and by looking at their interactions. One of the indicators for the intensity is the average speed distribution. Given a certain capacity of the road, the traffic reaches a certain average speed based on the current traffic intensity. The term "traffic" in this case denotes both passenger cars as heavy traffic, i.e. trucks. This relation can be expressed as *(heavy) traffic* $= I / C$. Practically speaking this assumption states that 1% more (heavy) traffic leads to 1% less road capacity. Traffic simulators such as ADSSIM have these kinds of relationships built in. The scientist in our example has doubts about the scalability this relationship, based on the fact that 100 % heavy traffic does not mean 0 % capacity, and based on the fact that the length of heavy traffic compared to passenger cars is roughly 3:1, which should result in a lower limit of the inverse capacity (i.e. at one third of the current estimate).

To investigate this relationship, the scientist wants to visualize low, medium and high amounts of heavy traffic (as a percentage of the total amount of vehicles on the road) on one axis and the intensity of the road that is being used (as a percentage of the maximum theoretical intensity) on another axis, with the average speed on the road as a result. Before the scientist starts the experiment, he has an expectation about the way the plot will look. The expected results can be seen in figure 7.1.

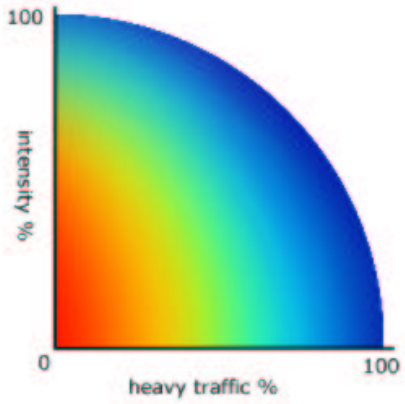


Figure 7.1: The expected relationship between the heavy traffic%, intensity% and the resulting speed distribution. Blue denotes slow speed (80 km/h), green denotes medium speed (100 km/h) and red denotes high speeds (120 km/h).

The scientist expects a plot which will show areas of the same average speed which we will call "isovelocities". The distance between the isovelocities will increase when the amount of heavy traffic increases. The lower left corner will show high average speeds due to little heavy traffic and an almost empty road, while the plot will show low average speeds where there are lots of trucks and the road is crowded. In reality it rarely happens that three lanes are filled with trucks, that is why the upper right corner is empty; this situation does not occur.
The scientist has to specify the (relative) amounts of heavy traffic and traffic intensity. Depending on the required granularity of the simulation this could result in tens (e.g. 5 heavy traffic percentages vs. 5 intensity percentages), hundreds (e.g. 10 x 10) or even thousands (e.g. 50 x 50) of simulation runs.

Using this case, we can describe the kind of information the scientist has to supply ADSSIM in order to perform the simulations.

7.4 ADSSIM-VLAM

Ideally we would like to offer the scientist an ADSSIM-module. He can specify the number of simulation runs that ADSSIM has to perform, and the parameter ranges that have to be simulated. When the user has specified the start, end and step values for every parameter, he will select a storage or visualization module. VLAM-G checks whether the selected number of parameters and the required inputs for the storage or visualization module correspond, and will suggest a

different module if the selected one is incompatible.
In our example the scientist specifies a heavy traffic range between 0 and 60% with a step value of 2%, and an intensity volume between 0 and 100% with a step value of 2%. He selects a visualization module which displays a 2D-plot, with the heavy traffic on the x-axis and the intensity volume on the y-axis. The blocks in the plot will represent the average speed, ranging from low speed (80 km/h - dark blue) to high speed (120 km/h - dark red). These two modules will suffice in order to run the simulation. The resulting plot is shown in figure 7.2. Several things can be seen in this plot. Low amounts of heavy traffic

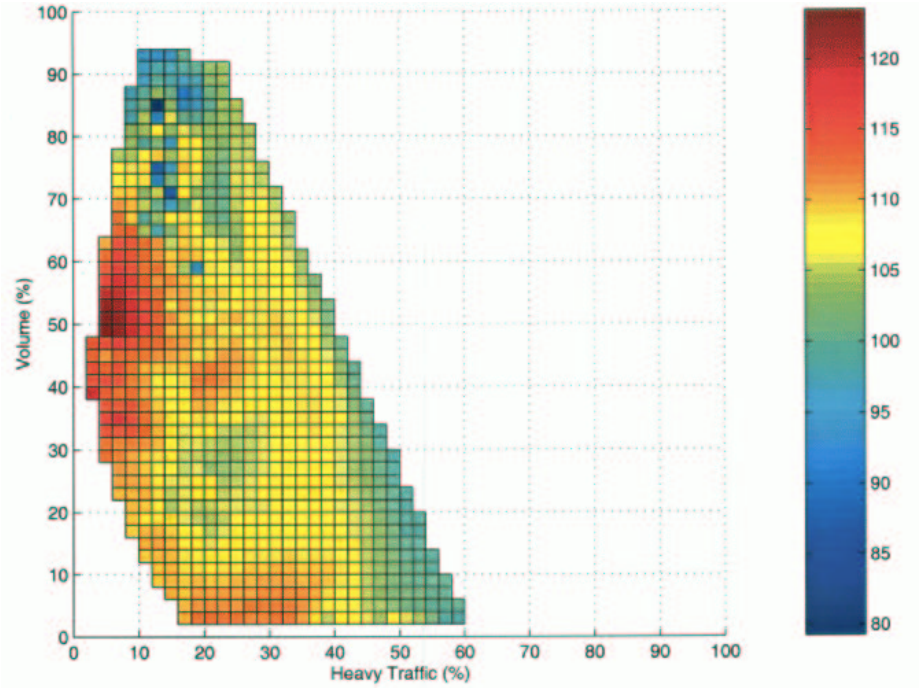


Figure 7.2: Average speed on the road for flowing traffic

generally lead to a high average speed, shown by the (dark) reds near the left hand side of the plot. The edges on right hand side show a drop in the average speed (the green colours) when the amount of heavy traffic increases. When the capacity of the road is almost fully utilized (top of the plot), the average speed is significantly lower (denoted by the blue squares). It's up to the scientist to analyze the outcome of the plot and to come up with possible explanations, it is outside the scope of this thesis and this particular example to elaborate on this subject.
If the scientist thinks the current visualization does not offer the insight he would have expected, he can choose to select a different type of visualization. He could for instance select a 3D-plot instead of a 2D, because the 3D-plot will nicely show gaps where sudden changes in average speed occur. He can also decide that the results of the ADSSIM-module will first be stored in a database and then be visualized. Advantage of this approach is the fact that he will only have to execute the simulation runs once, and can choose the appropriate visualization later (and can experiment as much as he wants to).

We have charted the process flow of this application.

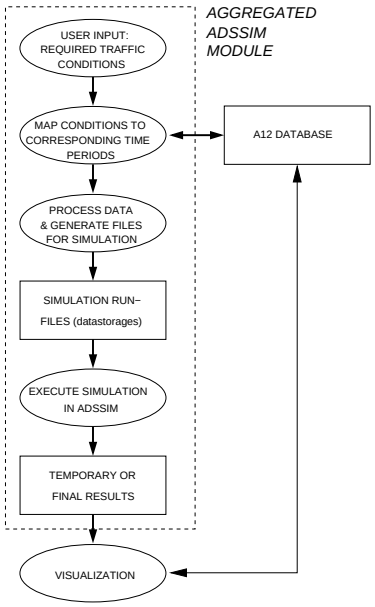


Figure 7.3: Process flow of the ADSSIM-VLAM-application

The majority of the processing occurs within the ADSSIM module, which is in fact an aggregated module of several other modules. The process flow for this application is shown in figure 7.3. Proces steps are indicated by ovals, data storages are indicated by rectangular boxes.

The process flow starts with the user who supplies the required traffic conditions (e.g. heavy traffic% and intensity%) and their start, end and step values. The traffic conditions are mapped to the corresponding time periods where they occurred using the A12 database. The relevant data is first processed and then stored in several datastorage files which will be used by ADSSIM. This intermediate step is important, because the traffic scientist might want to view the datastorage files that have been created after the experiment has completed, in order to find the cause of unexplainable anomalies.
The files are supplied as input for the (distributed) execution of the simulation by ADSSIM. The results are stored in temporary files, or in a file or database if the user wants to keep the results. The final step is visualization, using one of the predefined visualization modules in the repository of VLAM-G.

We have partially implemented these steps. Because our first focus was on enabling the Grid for multiple concurrent simulation runs, we did not implement the process of asking the user for the required traffic conditions. We used the parameters of the CASE example for our simulation and manually extracted the corresponding periods and data from the A12 database.
Two modules have been created, one module generates several ADSSIM files based data from the A12 database, the other module executes ADSSIM based on these files.

The visualization modules of VLAM-G have been developed but have not yet been integrated into the module repository, so we used MatLab to visualize the results of the simulations. We used MDL2XML to generate the XML file which is passed to the RTS of VLAM-G. This XML file contains the module descriptions of the two developed modules.

7.5 Evaluation

As stated before, we want to prove the functionality of the Virtual Traffic Lab and want to show possible applications. Although the design and the actual implementation of our ADSSIM-VLAM application clearly show the gap between how we ideally would want to use an application such as ADSSIM in our Virtual Traffic Lab, and the reality where the user still has to cope with low level details, we have shown several things.
Our application fits the VLAM philosophy of harnessing the power of the Grid and providing it to end users. The design of ADSSIM VLAM is very user oriented, it provides a simple and intuitive interface to a traffic simulator, and hides the low level details within an aggregated module. The implementation shows that the decomposition that was made during the design is correct, the implementation works and generates the expected results.
It has also become clear that the role of module developer, that we fulfilled for this application, is not just a matter of wrapping a current application into a VLAM module specification. Because process steps have to be converted to modules, one application might have to be split up into several parts. This requires a thorough understanding of the application and the ability to make a logical decomposition.
Although VLAM-G promises to be of use for non-computer scientists, the reality for now is that there is still a considerable amount of implementation details that are revealed to the user. This is partly due to the fact that VLAM-G is still under development, eventually these implementation details will be hidden by the only part of VLAM-G that the user interacts with, i.e. the GUI. Another reason is the fact that ADSSIM is not a trivial application to integrate into VLAM-G. Considerable (manual) preparation has to be performed before the actual simulation can be run. The fact that some implementation details are shown is therefore also partly due to the nature of ADSSIM.

7.6 Summary

We have created the prototype application ADSSIM-VLAM for the Virtual Traffic Lab. Based on a case example of the influence of heavy traffic on the average speed distribution, we have described the process flow for this application. We have partially implemented these steps. Because our first focus was on enabling the Grid for multiple concurrent simulation runs, we created two traffic modules. One module generates several ADSSIM files based data from the A12 database, the other module executes ADSSIM based on these files.
The design and implementation of our ADSSIM-VLAM application show the gap between how we ideally would want to use an application such as ADSSIM in our Virtual Traffic Lab, and the reality where the user still has to cope with

low level details. This is partly due to the fact that VLAM-G is still under development, but has also to do with ADSSIM itself which does not hide every implementation detail.

Chapter 8

Conclusion

We will now summarize our findings and the possible implications of these findings, and we identify topics that require further study or analysis.

8.1 Conclusions

We have explored the possibilities of an environment in which scientists can experiment with traffic applications. We set out to give insight into a Virtual Traffic Laboratory. We wanted to give this insight by describing the functionality such a Virtual Traffic Laboratory should possess, and by describing our hands-on experience during the design and implementation of a prototype of a traffic application.

In the requirements determination we concluded that traffic scientists have to cope with several problems that have to do with traffic simulators, and with data generated by these simulators or by real traffic flows. While most of these problems can be solved right now with the current tools, the expectation is that VLAM-G will offer (besides the distributed computational resources) more flexibility and re-usability through its module repository.
We have described the functional and technical requirements for the Virtual Traffic Lab based on the current traffic tools. The complete specification can be found in paragraph 4.3.3. This specification can form the starting point for the (future) development of the required traffic modules.
We have designed and implemented two of these modules as part of our prototype ADSSIM-VLAM application. This prototype can execute a distributed traffic simulation based on user supplied parameters. The experiment topology of this application is created using the generic VLAM modules we developed in Simulink. The MDL2XML utility which we developed is able to convert the experiment topology created in Simulink to a XML description that is passed to the RTS of VLAM-G to execute the experiment.

The majority of the current traffic tools are MatLab based. Initially we wanted to convert these tools to standalone applications using the MatLab compiler, but we have not (yet) succeeded in realizing the conversion using this approach.

The design and implementation of the ADSSIM-VLAM application show the gap between how we ideally would want to use an application such as ADSSIM in our Virtual Traffic Lab, and the reality where the user still has to cope with low level details. This is partly due to the fact that VLAM-G is still under development, but has also to do with ADSSIM itself which does not hide every implementation detail.

8.2 Future Research

We can identify several topics that require further analysis and study. We have split these issues into future research for the Virtual Traffic Lab and for VLAM-G.

8.2.1 Virtual Traffic Lab

Implementing the standalone MatLab approach would be a huge step forward for realizing the Virtual Traffic Lab. We expect that the majority of the current traffic tools can be ported without much adaptation. The traffic module repository would be greatly extended. It would also open up the possibility of porting generic MatLab routines to VLAM-G, which would increase the diversity of modules and thereby the possible experiment configurations. Of course other users would also benefit from these MatLab modules.

Earlier we mentioned an application that would be very suitable for the Virtual Traffic Lab: camera calibration. Overview cameras are typically equipped with wide-angle lens, to have a large field of view. A camera always gives a certain distortion, but for this type of lenses the distortion is clearly visible. This distortion can be calibrated when known objects are placed in the field of view of the camera. It is more important that there are many images of objects in all corners of the field of view, than that the model of the known object is very accurate. This makes it possible to use the passing vehicles to calibrate an overview camera. In order to calibrate the camera, specific points in the image are fitted to their known location in the real world. This sort of fits can use quite some computing resources and therefore this application would be perfect for the Virtual Traffic Lab.

We created our own MDL2XML. It would be interesting to use the MDL2XML utility of the Vanderbilt University [37] to convert an experiment topology to generic XML, and to use XSLT[1] - a language for transforming XML documents into other XML documents - to create a XML description for VLAM-G. This would be particularly interesting when a new VLAM module has to be created in Simulink, with a port configuration that does not exist yet. In our current implementation of MDL2XML the source code would have to be edited in order to be able to convert this port configuration from MDL to XML. Editing the XSLT would probably require less effort.

[1] *http://www.w3.org/TR/xslt/*

8.2.2 VLAM-G

The first release of VLAM-G is imminent, but so far there has been little interaction between the developers and (potential) end users from for instance biology and chemistry. This has lead to the situation where the developers do not exactly know what the end users expect from VLAM-G, and the end users do not have a clear picture about the (im)possibilities of VLAM-G. Topics that require clarification are the exact functionality of VLAM-G, but also Human Computer Interaction; how would the users like to interact with VLAM-G? Do they want fine-grained control like the traffic scientists, or will a few buttons which can load and start an experiment suffice?

If VLAM-G (or derivatives) ever wants to be used as a commercial application, attention will have to be given to aspects that are related to expressing tasks and experiments in terms of money. First a generic study could be performed to analyze methods which can be used to calculate the price of one CPU year. This could be used as the basic unit for expressing the costs of performing an experiment.
In the far future analyses could be performed on VLAM-G databases to gather usage statistics of modules and experiments. This information could be used to determine a price for a module or experiment. Different economic models could be studied and possibly applied to VLAM-G, for instance a licensing model for modules or experiments. Supply and demand models for computing capacity could be studied, and could be coupled with scenarios that offer different price-quality and price-speed ratings.

Bibliography

[1] K. Nagel, M.Rickert, Dynamic traffic assignment on parallel computers in TRANSIMS, in: Future Generation Computer Systems, vol. 17, pp. 637-648, 2001.

[2] A. Visser, H.H. Yakali, A.J. van der Wees, M. Oud, G.A. van der Spek, L.O. Hertzberger, An hierarchical view on modelling the reliability of a DSRC-link for ETC applications, Technical Report CS-99-02, submitted to IEEE Transactions on Intelligent Transportation Systems, 1999.

[3] I. Foster, C. Kesselman, The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, 1999.

[4] H. Afsarmanesh, R. Belleman, A. Belloum, A. Benabdelkader, J.F.J. van den Brand, T.M. Breit, H. Bussemaker, G. Eijkel, A. Frenkel, C. Garita, D.L. Groep, A.W. van Halderen, R.M.A. Heeren, Z.W. Hendrikse, L.O. Hertzberger, J. Kaandorp, E.C. Kaletas, V. Klos, P. Sloot, R.D. Vis, A. Visser, H.H. Yakali, VLAM-G: A Grid-Based Virtual Laboratory, submitted to the Special Issue on Grid Computing (IOS Press), 2001.

[5] B. van Halderen, Virtual Laboratory Abstract Machine Model for Module Writers, Internal Design Document, july 2000. See *http://www.dutchgrid.nl/VLAM-G/colla/proto/berry_running/*

[6] J.K. Ousterhout, Tcl: An Embeddable Command Language, USENIX Conference Proceedings, winter 1990.

[7] G.D. van Albada, A.Visser, J.M. Lagerberg, L.O. Hertzberger, A low-cost pose-measuring system for robot calibration, Robotics and Autonomous Systems, Vol 15., No. 3, pp. 207-227, August 1995.

[8] J.H. Futrell et al., Analytical Instrumentation for the Next Millennium (AINM) (Workshop Report), Orlando, Florida, March, 1999.

[9] I. Foster, C. Kesselman, S. Tuecke, The Anatomy of the Grid, to appear: Intl J. Supercomputer Applications, 2001.

[10] A.E.K. Sahraoui, Issues about Traffic Simulation Models Calibration and Validation, Intelligent transportation systems Congress, 2000.

[11] D.J. Flynn, Information systems requirements: determination and analysis, McGraw-Hill, pp. 118-120, 1992.

[12] B.W. Boehm, Software Engineering Economics, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[13] R. Buyya, Economic-based Distributed Resource Management and Scheduling for Grid Computing, PhD Thesis, Monash University, Melbourne, Australia, April 2002.

[14] M. Russell, G. Allen, G. Daues, I. Foster, E. Seidel, J. Novotny, J. Shalf, G. Von Laszewski, The Astrophysics Simulation Collaboratory: A Science Portal Enabling Community Software Development, Submitted to Journal of Cluster Computing, 2001.

[15] V. Menon, A.E. Trefethen, MultiMATLAB: Integrating MATLAB with highperformance parallel computing, in: Proceedings of Supercomputing '97, 1997.

[16] COMMSIM and MULTI Toolbox for MATLAB 5
http://shay.ecn.purdue.edu/~postal/commsim.html

[17] Matlab Parallization Toolkit
http://hem.passagen.se/einar_heiberg/

[18] C. Isbell, P. Husbands, The Parallel Problems Server: an Interactive Tool for Large Scale Machine Learning, Advances in Neural Information Processing Systems 12, MIT Press, 2000.

[19] M.J. Quinn, A. Malishevsky, N. Seelam, Otter: Bridging the Gap between MATLAB and ScaLAPACK, in: Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing, August 1998.

[20] C. Moler. Why there isn't a parallel MATLAB, MathWorks Newsletter, Spring, 1995.

[21] J. Dongarra, H. Meuer, H. Simon, E. Strohmaier, High Performance Computing Today, to appear in: Foundations of Molecular Modeling and Simulation Conference 2000.

[22] L.F.G. Sarmenta, et al., Bayanihan Computing .NET: Grid Computing with XML Web Services, to appear in the Workshop on Global and Peer-to-Peer Computing at the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid '02), Berlin, Germany, May 2002.

[23] H. Hacigumus, B. Iyer, S. Mehrotra, Providing Database as a Service, 2002 IEEE International Conference on Data Engineering (ICDE), February 2002.

[24] I. Foster, C. Kesselman, G. Tsudik, S. Tuecke, A Security Architecture for Computational Grids, Proc. 5th ACM Conference on Computer and Communications Security Conference, pp. 83-92, 1998.

[25] National Digital Mammography Archive: University of Pennsylvania consortium and IBM develop computing grid for breast cancer screening
http://www-3.ibm.com/solutions/lifesciences/pdf/NDMA_8pg_FINAL.pdf

[26] I. Foster, C. Kesselman, J. Nick, S. Tuecke, The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration (draft), January 2002.

[27] J.M. Myerson, Web Services Architectures, whitepaper at *http://www.webservicesarchitect.com/content/articles/myerson01.asp* January 23, 2002.

[28] Simple Object Access Protocol (SOAP) 1.1, W3C Note, 08 May 2000. *http://www.w3.org/TR/SOAP/*

[29] N.A. Streitz, J. Geißler, J.M. Haake, J. Hol, DOLPHIN: Integrated Meeting Support across LiveBoards, Local and Remote Desktop Environments. In: Proceedings of the 1994 ACM Conference on Computer-Supported Cooperative Work (CSCW'94), pp. 345-358, Chapel Hill, N.C., October 22-26, 1994.

[30] Collaborative Virtual Workspace *http://cvw.sourceforge.net*
Overview: *http://cvw.sourceforge.net/cvw/info/CVWOverview.php3*
Architecture Overview:
http://cvw.sourceforge.net/cvw/info/docs40/ArchOverview.php3

[31] D. Abramson, R. Buyya, K. Branson, J. Giddy, The Virtual Laboratory: A Toolset for Utilising the World-Wide Grid to Design Drugs, 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2002), 21 - 24 May 2002, Berlin, Germany.

[32] Materials Microcharacterization Collaboratory
http://tpm.amc.anl.gov/MMC/

[33] C. Röhrig, A. Jochheim, Java-based Framework for Remote Access to Laboratory Experiments. IFAC/IEEE Symposium on Advances in Control Education, ACE 2000, Gold Coast, Australia, December 2000.

[34] The Mathworks Inc., MATLAB Application Program Interface Guide, revision for 5.2, January 1998.

[35] OECD Road Transport and Intermodal Linkages Research Programme: Outlook 2000, chapter 3, pp. 1-2, Paris, 23/05/1999.
http://www.oecd.org/EN/document/0,,EN-document-51-nodirectorate-no-15-671-25,00.html

[36] C. Tampére, A. Vieveen, M. Droppert-Zilver, Calibration and Validation RTS Module, TNO Inro, Delft, 1998.

[37] S. Neema, Simulink and Stateflow Data Model (draft), Vanderbilt University, Nashville, Tennessee, 2002.
http://www.isis.vanderbilt.edu/Projects/mobies/techpapers.html

[38] J. Heikkilä,O. Silvén, A Four-step Camera Calibration Procedure with Implicit Image Correction, IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'97), pp. 1106-1112, San Juan, Puerto Rico, 1997.

[39] Module Description File for VLAM-G, manual in preparation.
http://www.dutchgrid.nl/VLAM-G/AM/doclive-vlam.html#sec10

[40] ISIS, BOSCH, TÜV, LISITT, CMG, Interoperable User Requirements, Deliverable A1-WP3-D01-ISI-03, Aachen, December 1998.

[41] K. Stappert, K. Perrett, T.F. Groner, B.B. Orchidnote, Functional Specification for Interoperable EFC, Deliverable D3.2, Köln, March 1999.

[42] A.G. Hoekstra, L. Dorst, M. Bergman, J. Lagerberg, A. Visser, H. Yakali, F. Groen, L.O. Hertzberger, High Performance Discrete Event Simulations to evaluate Complex Industrial Systems - The case of Automatic Debiting Systems for Electronic Toll Collection on Motor Highways, High Performance Computing and Networking, Eds. Bob Hertzberger and Peter Sloot, Lecture Notes in Computer Science 1225 (Springer), pp. 41-50, 1997.

[43] B.J. Overeinder, Distributed Event-driven Simulation: Scheduling Strategies and Resource Management (dissertation), Amsterdam, The Netherlands, November 2000.

Appendix A

Interview Adam Belloum 31-05-2002

May 31st, 2002, we interviewed Adam Belloum, member of the Virtual Lab team.

INTRODUCTION
Purpose & duration of the interview
Can you give a short description about yourself and about your relationship to the Virtual Lab-project?
Could you give me an overview of what you do?
Adam Belloum, 31 years old, postdoc at the University of Amsterdam. For the last two years he has been working as one of the designers (together with David Groep) of the Virtual Lab. He's been doing work on the GUI and the RTS, the heart of the Virtual Lab. The last few months he's also been doing some implementation because of the lack of manpower and time.

GENERAL QUESTIONS
Can you describe the Virtual Lab?
Adam uses the tagline "The easy way to go to the Grid" to explain the Virtual Lab to people who aren't familiar with it. It's an environment which enables non-scientists to perform experiments, while they don't have to worry about resources.
The Virtual Lab is created for doing research. It's more than a proof-of-concept, it should work without major deficiencies. On the other hand, there won't be a phase of six months of extensive testing which is a normal step in commercial systems development.

Can you tell something about the history of the Virtual Lab?
In september 2001 it was decided that there were several objectives that had to be met, in chronological order:

1. run jobs on the Grid using the Virtual Lab (DONE)

2. provide API's for each component of the Virtual Lab (DONE)
3. make the Virtual Lab secure (almost done*)
4. TO DO: collaborational aspects; when the Virtual Lab works for one person, it has to work so that multiple persons can work together on experiments

* this issue is treated later on, when discussing essential elements of the Virtual Lab.

What's "virtual" about it?
The allocation of resources is invisible for the user, when the user performs an experiment it looks as if it runs on one virtual supercomputer.
When the collaborational aspects are built into the Virtual Lab, it can actually *be* a Virtual Lab, i.e. scientists that are geographically dispersed can work together on experiments as if they were in the same laboratory.

Are you familiar with other Virtual Lab-projects, such as Monash University Australia "Molecular Modelling for Drug Design", Oak Ridge National Laboratory's and FernUniversität Hagen's (Germany) Virtual Lab? (see printed summaries)
If so, what are the difference (and hopefully the advantages) of VLAM-G as opposed to those other projects called "Virtual Lab"?
Adam is not familiar with these specific Virtual Labs. Of course he is aware of the general fact that there are related projects at other universities. He points out that there is an important difference between these projects and the Virtual Lab; the other projects focus on a specific (sub) domain, for instance molecular modelling in chemistry. The explicit starting point of the Virtual Lab is its generic infrastructure. Domain specific knowledge is put into (reusable) modules, but not into the Virtual Lab itself. This way the Virtual Lab can (in theory) be used as a laboratory for every imaginable domain.
One of the current problems are the databases, which are still designed and filled per domain. People are trying to achieve a generic data model that can be used for every experiment and every domain, which is not trivial because there exist major differences between application domains (for instance physics or chemistry).

ANALYSIS OF TASKS
Which tasks do you think are suitable for the Virtual Lab?
Any setup that can be put in the form of modules that are connected to each other by their input and output ports using streams, is suitable for the Virtual Lab. What goes on *inside* the modules is not really important for the Virtual Lab, as long as the datatypes between the modules match the data can be transferred from one module to another.

Do specific requirements exist for the input and/or intermediate data?
As said above, the datatypes between modules and ports have to match. This is specified by the user, but an agent that supports the user when supplying the

data can inform the user about a mismatch before he starts the experiment.

Is there a difference if more people are involved in an experiment? Can you tell something about the differences or commonalities?
Since the collaboration aspects of the Virtual Lab are not realized yet, it is difficult to say something about this subject. What *can* be said about the different users, is that there is almost no knowledge about (the wishes of) the actual (or potential) end users. Generally speaking the project group sees two types of users, the module developers and the end users. The module developers are computer scientists which have for instance some C-code which they want to run. Examples are Arnoud Visser who has a computer science background and Gert Eijkel from AMOLF. Input from novice user without a computer science background is not available right now.

Which tools are available at this moment to perform these tasks?
"Regular" supercomputers are available as a means for High Performance Computing. These tools are provided by universities but also by commercial providers such as IBM. They are heavily investing into Grid-applications and Grid-services at this moment and offer multiple options at different prices ("Pay more, get more").

EVALUATION
(The answers to the following question and the next have been grouped because they are closely related.)
Where is the trade-off between executing an experiment locally in some environment or using the Virtual Lab?
Does the Virtual Lab offer added value?
- If so, can you specify this added value?
- If not, how do you perceive the Virtual Lab?
For somebody with a desktop application there is no real use of the Virtual Lab. A demanding application which needs lots of computational power has to be executed on a powerful system. One option is for instance to use SARA[1]. In that case, the user has to setup (software) connections to SARA, send and receive data, monitor the progress, etc etc. This is unnecessary and time-consuming work from the point of view of the user, who just wants to get his work done and see the results. The alternative which saves him this extra work is the Virtual Lab.
Besides those advantages, the Virtual Lab offers a set of predefined modules, for instance three hundred different visualization modules that can display a vector, display 3D-data, rotate 3D-data, ... This offers the scientist options that were previously unavailable.
Another option is to go to CERN in Switzerland to perform some physics experiment. The use of the Virtual Lab saves time that otherwise would be spent on travelling and on adapting the software to prepare it for execution at CERN.

[1]SARA Computing and Networking Services supplies a complete package of High Performance Computing- and infrastructure services, based on state-of-the-art information technology, and is located at the Wetenschap & Technologie Centrum Watergraafsmeer (WTCW)

During the design and implementation of the Virtual Lab the role of expressing tasks in terms of money was ignored. How much it costs (or pays) for the end user to re-use a developed module is unknown. It is also unknown how much it costs to perform an entire experiment, or if there exist several options to execute the experiment. One could imagine the user should have the option to choose between performing the experiment in one day for $10,000 , in three days for $2,500 or in one week for $1,000. It is also not clear if there is any (financial) benefit in sharing computer capacity.

To summarize:
The Virtual Lab saves...

1. time, because...
   (a) modules can be re-used and don't have to be (re) developed.
   (b) software doesn't have to be adapted to the specific setup of a particular supercomputer.
   (c) there is no necessity for travelling to the location where the experiment is executed.
2. money, because of the reasons that are given above.

Which elements of the Virtual Lab are essential for you?
Security is important, especially for the industry. If they ever consider adapting the Virtual Lab, it has to be secure. By "secure" Adam means that the system is *hackerproof* and fault tolerant. The following example explains fault tolerance in the Virtual Lab. It is possible that an experiment that takes three weeks to execute will stop after two weeks and two days because of a malfunction of one of the computers that is used to perform the experiment. The Virtual Lab has to address this issue and has to migrate the process from the malfunctioning computer to another computer so that the experiment can keep on running.

Where is VLAM-G positioned right now and what's in store for the future?
The first official release (alpha or beta) is imminent, it should be ready in the first week of June 2002. This release of the Virtual Lab is capable of running jobs on the Grid. It contains a Java-based GUI which enables the user to create experiments. The structure and data of the experiment is stored in a database. This data is given to the RTS in XML-form.
Companies such as Unilever (research in molecules) and telecom providers (GPS-applications with large amounts of data) have shown their interest in the Virtual Lab.

What will the cooperation between a scientist and VLAM-G look like in (say) 2010?
Adam takes this question as the opportunity to formulate some thoughts about the ideal Virtual Lab.

1. The GUI has to get better
   The GUI has to look different for each user (or each domain). "A lab

is a lab for every scientist. It has four walls, a door and a table. The difference is what is on the table; a chemist has liquids, test tubes, etc. A physicist on the other hand has computers, sensors, etc." This analogy also has to be applied to the GUI, it has to suite the needs of the user. Right now the GUI is made so that it works. It is build using the logic of computer scientists, which is (probably) not equal to the logic of other (non-computer) scientists.

2. The credentials-system has to get better
   Right now the credentials for the Grid are stored locally on each computer. A much more elegant solution would be something like a chipcard reader and a PIN-code as an authentication method. The same ease of use should be available for the Virtual Lab.

3. Make the Virtual Lab accessible everywhere
   Access to the Virtual Lab shouldn't have to be narrowed down to a computer on a desk. Wireless access on all sorts of devices should be available.

4. Before 2010, the issue of fault tolerance has to be solved.

What are questions that you would like to have answered by possible end users? (for instance in the biology or chemistry domain)
Right now there are misunderstandings by these users about what the Virtual Lab is, what it can or cannot do. Interesting questions would then be:
What are you expecting of the Virtual Lab?
How would the GUI look like so that it would be easy for you to use?
How would you like to interact with the Virtual Lab? If we exaggerate, do you want a button "Run it" and don't care how it is run, or do you want to see which machines are available to perform your experiment? Or do you want a proposal which machines are automatically selected to perform the experiment? Or do you want a breakdown of the costs of executing the experiment in several time-frames at different costs?

Appendix B

ComVis example

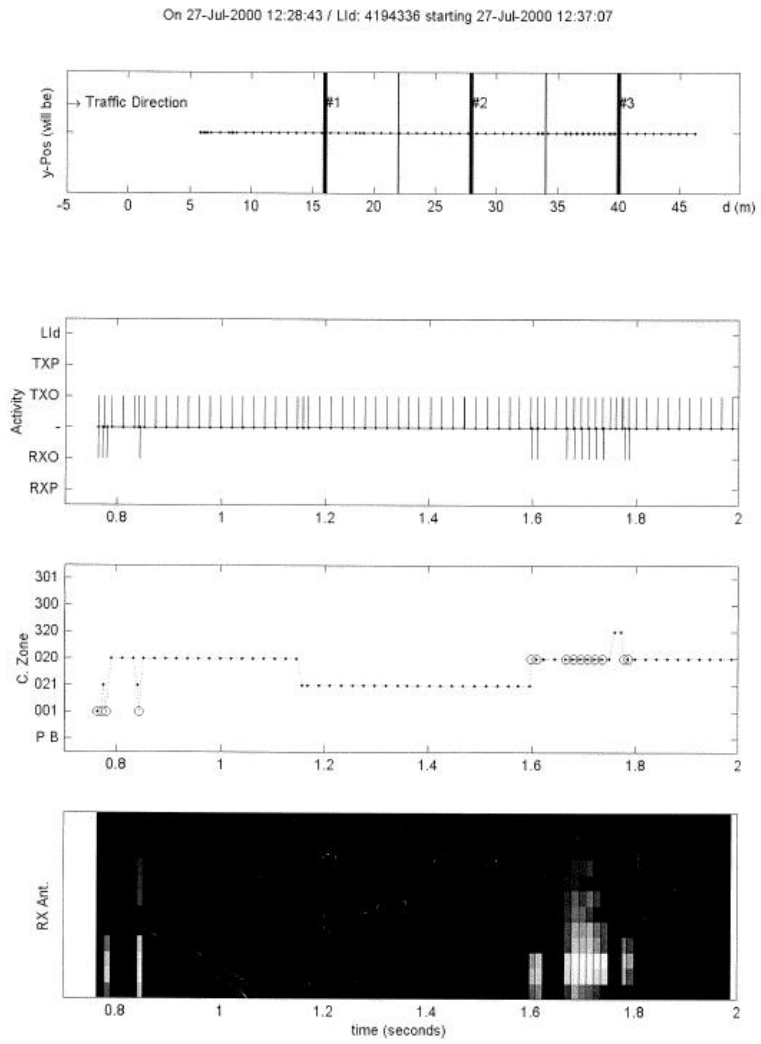


Figure B.1: Example of plots made by ComVis

Appendix C

How to create a custom VLAM module in Simulink

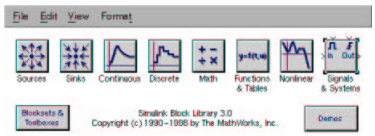
In this appendix we describe how to create a custom VLAM module. We will construct a "2in2out" module as an example, which consists of two input and two output ports. Based on this example one should be able to create any type of VLAM module, no matter the amount of input or output ports, or the number of parameters.

Start MatLab and run Simulink:

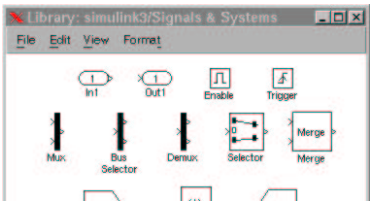
>> *simulink*

(we used MatLab Version 5.3.1.29215a (R11.1), Oct 6 1999, and Simulink version 3.0 to create our modules)

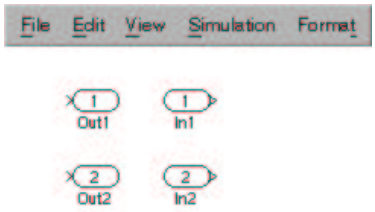
Click on "Signals & Systems":



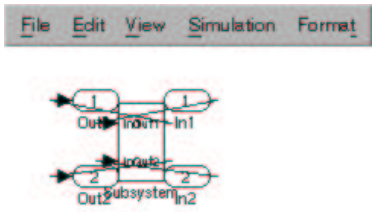
The components we will use are the in and out port.



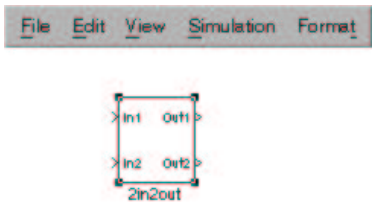
Create a new model (File → New → model). Drag two in ports and two out ports to this new model. Note that the out ports are placed on the *left* and the in ports on the *right*. The symbols attached to the ports show why: the out ports that are placed on the left will *receive* data, the in ports will *send*.



Now select all ports by dragging a square around the four ports using the mouse[1]. We are going to create a subsystem: Edit → Create subsystem (CTRL-G). The result looks pretty messed up:

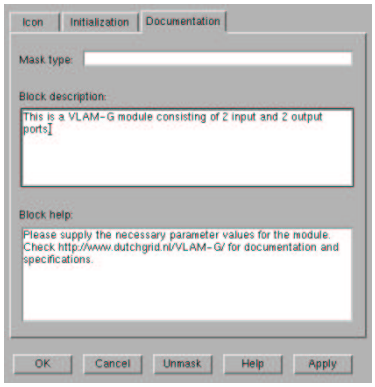


We will now remove the unnecessary and duplicate elements. Remove all the lines and remove the original in and out ports. We now see the clean subsystem. Click once somewhere on the subsystem and enlarge the subsystem by dragging one of the corners. Finally, we change the name from "Subsystem" to "2in2out":



The basic building block is now ready. Because we want to use this block as a VLAM-G module, we have to mask the system and supply the required parameters. Edit → Mask subsystem (CTRL-M). The Mask Editor opens, showing three tabs: Icon, Initialization and Documentation. We start by filling in this last tab:

[1] Apparently there exists a difference between selecting all components by dragging a square around them, and using "Select all" (CTRL-A) or clicking on each component. The last two operations select all components, but Simulink does *not* offer the option to create a subsystem. This option *is* available when using the first "dragging" method.

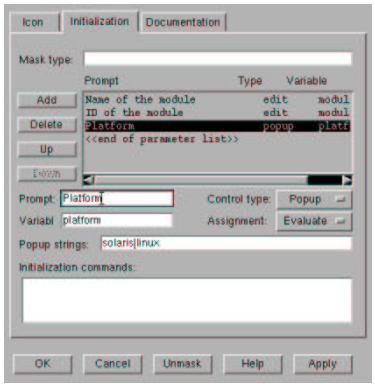


We move to the Initialization tab for the implementation of the mask. The Mask Editor allows us to add parameters. For each parameter we can specify the prompt (the text the user will see), the control type (whether the user sees a form, a checkbox or a list box), the variable name and in case of a listbox the popup strings (more about this option in a moment).
Because we want the user to supply the values for a VLAM-G module, we have to add the required parameters to the mask. Our MDL2XML tool converts the MDL description to XML, and searches for these parameters. It is therefore compulsory to follow the naming convention that is displayed in this table.

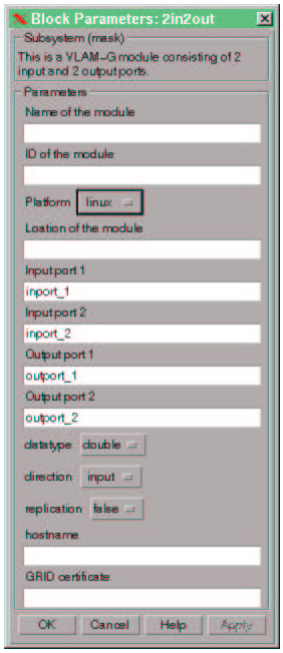
Required variable names for MDL2XML

variable name	type
moduleName	edit
moduleId	edit
platform	popup (solaris|linux)
proglocation	edit
outport1	edit
outport2	edit
datatype	popup (double|integer)
direction	popup (input|output)
replication	popup (false|true)
host	edit
dn	edit

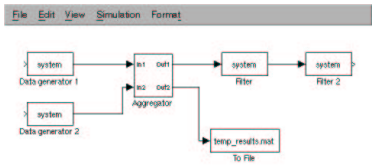
We have added the moduleName, moduleId and platform to the mask. The platform variable shows how a listbox can be created; the control type "popup" is selected, the values that need to be in the popup or listbox are specified in the "popup strings" field. The values are separated by a pipeline (|). The advantage of using a popup is the fact that the input can be controlled. Free text fields are sensitive to errors, which can be prevented using the popup box. The screenshot on the next page shows the Initialization tab with a popup variable selected.



When all of the necessary parameters are defined in the mask, the module is complete. The designer of the module can choose if he wants to predefine a value for certain parameters. If he chooses to do so, he can double click the 2in2out module to open the window that will also be presented to the end user when he uses the module. Values that are entered in the 2in2out module before the module is saved in a (VLAM module) library are stored as the default values for the parameters. An example is shown in the figure below.



Our 2in2out module can now be used as part of an experiment. An example of such an experiment is shown below, this of course just an example to show the possibilities of our new module.



This concludes our HOWTO for the creation of a custom VLAM-G module.

Appendix D

Source code

D.1 Interface to prototype RTS

D.1.1 MEX interface

vlab.h

```
#ifndef _VLAB_H
#define _VLAB_H
/*****************************************************************************/

#define VLAB_OK 0 /* unknown or no error */
#define VLAB_ENOTIMPL (-1) /* function not implemented */
#define VLAB_ENOTAVAIL (-2) /* resource not available */
#define VLAB_ENOTDEF (-3) /* named entity not defined */
#define VLAB_EBADTYPE (-4) /* bad datatype as parameter */
#define VLAB_ENOTSET (-5) /* required attribute not set */

typedef struct module *vlab_module_t;
typedef struct buffer *vlab_buffer_t;
typedef struct port *vlab_port_t;
typedef struct parameter *vlab_parameter_t;
typedef struct state *vlab_state_t;
typedef struct datatype *vlab_datatype_t;
typedef struct datafield *vlab_datafield_t;

/*****************************************************************************/

extern int vlab_type_char (vlab_module_t, vlab_datatype_t *);
extern int vlab_type_unsigned_char (vlab_module_t, vlab_datatype_t *);
extern int vlab_type_byte (vlab_module_t, vlab_datatype_t *);
extern int vlab_type_short (vlab_module_t, vlab_datatype_t *);
extern int vlab_type_unsigned_short (vlab_module_t, vlab_datatype_t *);
extern int vlab_type_int (vlab_module_t, vlab_datatype_t *);
extern int vlab_type_unsigned (vlab_module_t, vlab_datatype_t *);
extern int vlab_type_long (vlab_module_t, vlab_datatype_t *);
extern int vlab_type_unsigned_long (vlab_module_t, vlab_datatype_t *);
extern int vlab_type_float (vlab_module_t, vlab_datatype_t *);
extern int vlab_type_double (vlab_module_t, vlab_datatype_t *);
extern int vlab_type_long_double (vlab_module_t, vlab_datatype_t *);
extern int vlab_type_long_long_int (vlab_module_t, vlab_datatype_t *);

extern int vlab_type_struct (vlab_module_t, vlab_datatype_t *);
extern int vlab_type_array (vlab_module_t, vlab_datatype_t *,
 int size);
extern int vlab_type_vararray (vlab_module_t, vlab_datatype_t *,
 vlab_datafield_t size);
extern int vlab_type_add (vlab_datafield_t *,
 vlab_datatype_t composed,
 vlab_datatype_t subtype);
extern int vlab_type_addnamed (vlab_datafield_t *,
 vlab_datatype_t composed,
 char *name,
 vlab_datatype_t subtype);

extern int vlab_type_commit (vlab_module_t, char *name,
 vlab_datatype_t type);
extern int vlab_type_named (vlab_module_t, vlab_datatype_t *t,
 char *name);
extern vlab_datatype_t vlab_gettype(vlab_module_t, char *type);
extern void vlab_setsize(vlab_datafield_t field, int value);

/*****************************************************************************/

extern char *vlab_strerror(int code);
```

```
extern int vlab_input(vlab_module_t, char*, vlab_datatype_t, vlab_port_t *);
extern int vlab_output(vlab_module_t, char*, vlab_datatype_t, vlab_port_t *);

extern void *vlab_getArgument(vlab_module_t, char *, vlab_datatype_t);
extern void *vlab_declareState(vlab_module_t, char *, vlab_datatype_t,
       vlab_state_t *);
extern void *vlab_declareParameter(vlab_module_t, char *, vlab_datatype_t,
   vlab_parameter_t *);
extern void vlab_updateState(vlab_state_t);
extern void vlab_updateParameter(vlab_parameter_t);

extern int vlab_updateparameter(vlab_parameter_t);
extern void *vlab_read(vlab_port_t, vlab_buffer_t *, long);
extern void *vlab_prepare(vlab_port_t, vlab_buffer_t *, long);
extern int vlab_write(vlab_buffer_t);
extern int vlab_release(vlab_buffer_t);

/*****************************************************************************/

struct vlab_specification {
  int version;
  char *name;
  void *(*ini)(vlab_module_t);
  void (*run)(vlab_module_t, void *);
};

/* internal use for runtime system */
extern int vlab_initialize(void);
extern int vlab_finalize(void);

/*****************************************************************************/
#endif
```

vlab.c

```
/*============================================================
 *
 * VLAB.MEX    Vlab running under MatLab, using the MEX-construction.
 *
 * Created by Joost Zoetebier. Last update: 07-03-2001.
 *============================================================*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <strings.h>
#include <thread.h>
#include <pthread.h>
#include <synch.h>
#include <dlfcn.h>
#include "control.h"
#include "vlabint.h"
#include "packing.h"

/* included on 22-11-2000 for MEX */
#include "mex.h"

#define _OK        0   /* Command completed normally */
#define _ERROR     1   /* The command couldn't be completed successfully */
#define _RETURN    2   /* The command requests that the current procedure returns */
#define _BREAK     3   /* The command requests that the innermost loop be exited */
#define _CONTINUE  4   /* Go on to the next iteration of the current loop */

#define maxargs 6   /* max. number of arguments accepted by the vlabprompt .
Max = 6 for " e connect m1 out n1 in ". */

extern struct moduledefs *defs; /* control.c */
int values[1024];

struct moduledesc;
struct experimentdesc;
struct streamdesc;
struct argumentdesc;
struct streamdesc {
    struct moduledesc *producer;
    char *outputport;
    struct moduledesc *consumer;
    char *inputport;
    struct streamdesc *next;
    struct queue *impl;
};
struct argumentdesc {
    char *name;
    int value;
    struct argumentdesc *next;
};
struct moduledesc {
    int ident;
    char *name;
    char *type;
    struct argumentdesc *arguments;
    struct moduledesc *next;
    vlab_module_t impl;
};
struct experimentdesc {
    int counter;
    char *name;
    struct moduledesc *modules;
    struct streamdesc *streams;
};

#if 0
static
#endif
void *
start(void *arg)
{
    vlab_module_t module = (vlab_module_t) arg;
    module->func(module, module->user);
    return NULL;
}

sigset_t new;

static void
run(struct experimentdesc *edesc)
{
    vlab_datatype_t inttype;
    struct moduledesc *mdesc;
    struct streamdesc *sdesc;
    struct argumentdesc *adesc;
    vlab_module_t module;
    vlab_port_t port;
    struct queue *queue;
    struct moduledefs *ptr;
    int i;

    sigemptyset(&new);
#if 0
```

```
    sigaddset(&new, SIGQUIT);
    pthread_sigsetmask(SIG_BLOCK, &new, NULL);
                                /* let the children catch the QUIT signal */
#endif
    for(sdesc=edesc->streams; sdesc; sdesc=sdesc->next) {
        sdesc->impl = queue = malloc(sizeof(struct queue));
        queue->type = NULL;
        queue->first = queue->last = NULL;
        queue->size = 0;
        mutex_init(&queue->mutex, USYNC_THREAD, 0);
        sema_init(&queue->queuesize, 0, USYNC_THREAD, 0);
        sema_init(&queue->queuefree, 1, USYNC_THREAD, 0);
    }
    for(mdesc=edesc->modules; mdesc; mdesc=mdesc->next) {
        mdesc->impl = module = malloc(sizeof(struct module));
        for(ptr=defs; ptr; ptr=ptr->next) {
            if(!strcmp(ptr->name, mdesc->type))
            break;
        }
        if(!ptr) {
            fprintf(stderr,"module %s does not exist\n",mdesc->type);
            abort();
        }
        module->parent     = edesc;
        module->arguments  = NULL;
        module->parameters = NULL;
        module->namedtypes = NULL;
        module->states     = NULL;
        module->ports      = NULL;
        vlab_type_int(module, &inttype);
        for(adesc=mdesc->arguments; adesc; adesc=adesc->next) {
            fprintf(stderr, "'%s': set argument '%s' to value %d\n",
                                    mdesc->name, adesc->name, adesc->value);
            vlab_setArgument(module, adesc->name, inttype, &adesc->value);
        }
        module->func = ptr->run;
        module->user = ptr->ini(module);

        if(pthread_create(&module->thread, NULL, &start, module)) {
            perror("pthread_create failed");
            abort();
        }
        else {
            fprintf(stderr, "Thread '%d' created.\n", module->thread);
        }
    }
    for(mdesc=edesc->modules; mdesc; mdesc=mdesc->next) {
        module = mdesc->impl;
        for(port=module->ports; port; port=port->next) {
            if(port->type == NULL)
                port->type = port->queue->type;
        }
    }
#if 0
    for(mdesc=edesc->modules; mdesc; mdesc=mdesc->next) {
        module = mdesc->impl;
        thr_continue(module->thread);
    }
#endif
}

static void
kill(struct experimentdesc *edesc)
{
    struct moduledesc *mdesc;
    vlab_module_t module;
    thread_t main_thr;

    for(mdesc=edesc->modules; mdesc; mdesc=mdesc->next) {
        fprintf(stderr, "Trying to kill thread '%s'...\t", mdesc->type);
        /*fprintf(stderr, "%d\n", module->thread);*/
        module = mdesc->impl;
        if( pthread_cancel(module->thread) )
            fprintf(stderr, "Cancelling thread failed.\n");
        else
            fprintf(stderr, "Thread successfully cancelled.\n");
    }
    fprintf(stderr, "End of killing threads.\n");
}


static void showState(struct experimentdesc *edesc)
{
    struct moduledesc *mdesc;
    struct streamdesc *sdesc;
    struct argumentdesc *adesc;
    int i;

    fprintf(stderr, "\n+ + + + +  CURRENT STATE  + + + + + + + +\n");
    fprintf(stderr, "+ Experiment: %s\t\t\t\t+\n", edesc->name);

    for(mdesc=edesc->modules; mdesc; mdesc=mdesc->next) {
        fprintf(stderr, "+ Module: '%s'\t\t\t\t+\n", mdesc->name);
        for(adesc=mdesc->arguments; adesc; adesc=adesc->next) {
            fprintf(stderr, "+\tArguments: '%s'\t= %d\t+\n",
                mdesc->arguments->name, mdesc->arguments->value);
        }
        sdesc = edesc->streams;
        if (sdesc != NULL)
```

```
            fprintf(stderr, "+ Stream: from '%s' to '%s'\t\t+\n",
                    sdesc->inputport, sdesc->outputport);
    }
    fprintf(stderr, "+ + + + +   END OF CURRENT STATE  + + + +\n\n");

}


int vlab(char *argv[], int argc ) {
    struct vlab_specification *spec;
    struct experimentdesc *experiment;
    void *handle;
    int i, j, k, found = 0;
    char buffer[80];
    char *token, character;
    FILE *fp;
    struct moduledesc *module;
    struct streamdesc *stream;
    int rv; /* for reading the TXT-file */

    /* used for modules */
    struct argumentdesc *argument;
    int count, ds_index;
    char ds[256];
    char aux[30];

    struct moduledesc *tmp; /* 25-10-2000 - for determining modules in add */

    char temparg[80], temparg2[80], temparg3[80], lastarg[80],
            column1[80], column2[80], column3[80], column4[80], column5[80];
    int foundFirstDot = 0, foundSecondDot = 0, column;
    long temp;

    vlab_module_t mod;   /* 15-03-2001 - threads */


/************* fetching current state ******************/
    for (i=0; i < 80; i++) {
        column1[i] = column2[i] = column3[i] =
        column4[i] = column5[i] = temparg[i] = temparg2[i] = NULL;
    }

    /* finding the right experiment */
    fp = fopen("state", "r");
    if (fp == NULL) fprintf(stderr, "'state' not found!\n");
    else {
        while (fgets(buffer, sizeof(buffer), fp)) {
            for (i=0; i < 80; i++)
                column1[i] = column2[i] = column3[i] = column4[i] = column5[i] = NULL;

            j = 0;
            column = 1;
            for (i=0; i < strlen(buffer); i++) {
                while(buffer[i] != '\t' && buffer[i] != '\0' && buffer[i] != '\n') {
                    switch(column) {
                        case 1: column1[j++] = buffer[i++]; break;
                        case 2: column2[j++] = buffer[i++]; break;
                        case 3: column3[j++] = buffer[i++]; break;
                        case 4: column4[j++] = buffer[i++]; break;
                        case 5: column5[j++] = buffer[i++]; break;
                    }
                }
                column++;
                if (buffer[i] != '\0')   j = 0;
                /*if (buffer[i] == NULL || buffer[i] == '\n')   i = sizeof(buffer); */
            }

            if (!strcmp(column1, "experiment")) {
                /* a pointer is a long, so convert the string to a long using  atol */
                temp = atol(column3);
                experiment = (struct experimentdesc *) temp;
                rv = sscanf (column3, "%p",&experiment);
            }
            else if (!strcmp(column1, "module")) {
                temp = atol(column4);
                module = (struct moduledesc *) temp;
                rv = sscanf (column4, "%p",&module);
            }
            else if (!strcmp(column1, "argument")) {
                temp = atol(column5);
                argument = (struct argumentdesc *) temp;
                rv = sscanf (column5, "%p",&argument);
            }
            else if (!strcmp(column1, "stream")) {
                temp = atol(column2);
                stream = (struct streamdesc *) temp;
                rv = sscanf (column2, "%p",&stream);
            }
        }   /* end of 'while' */
        fclose(fp);
    }   /* end of 'else' */


/******** parsing command with dots in it ********
    * Only argv[0] can contain "." and has to be split if there are any dots present.
    * Examples of valid argv[0]'s:
    *     e.add
    *     e.m1.set
    */
```

```
    j = k = 0;
    for (i=0; i < 80; i++)
            temparg[i] = temparg2[i] = temparg3[i] = lastarg[i] = NULL;
    for (i=0; i < strlen(argv[0]); i++) {
        if (argv[0][i] == '.') {
            if (foundFirstDot)
                foundSecondDot = 1;
            else
                foundFirstDot = 1;
        }
        else {
            /* 30-11-2000 - copy each character to temparg[],
      * and copy argv[0] partly to argv[1], shift argv[1] to argv[2], etc.  */
            if (!foundFirstDot)
                temparg[i] = argv[0][i];      /* name of experiment */
            else if (!foundSecondDot)
                temparg2[j++] = argv[0][i];   /* 'add' or modulename */
            else
                temparg3[k++] = argv[0][i];   /* 'set' */
       }
    }

    if (foundFirstDot && !foundSecondDot) {
        for (i=0; i < strlen(argv[0]) ; i++) {
            if (i <= strlen(temparg2) )
                argv[0][i] = temparg2[i];     /* rewrite 'e.add' to 'add' */
            else
                argv[0][i] = NULL;
            }
        }
        if (foundSecondDot) {
            strcpy(lastarg, argv[2]);
            strcpy(argv[2], argv[1]);
            for (i=0; i < strlen(argv[0]) ; i++) {
                if (i <= strlen(temparg3) )
                    argv[0][i] = temparg3[i]; /* rewrite 'e.m1.set' to 'set' */
                else
                    argv[0][i] = NULL;
            }

            for (i=0; i < strlen(argv[1]) ; i++) {
                if (i <= strlen(temparg2) )
                    argv[1][i] = temparg2[i]; /* rewrite 'range' to 'm1' */
                else
                    argv[1][i] = NULL;
            }
        }
    }
    /******** end of parsing of dots ********/


    if(!strcmp(argv[0],"create")) {
        experiment = malloc(sizeof(struct experimentdesc));
        experiment->name    = strdup(argv[1]);
        experiment->counter = 0;
        experiment->modules = NULL;
        experiment->streams = NULL;

        fp = fopen("state", "w");
        if (fp != NULL) {
            fputs("experiment\t", fp);
            fputs(experiment->name, fp);
            fputs("\t", fp);
            fprintf(fp, "%p", experiment);
            fputs("\n", fp);
            fclose(fp);
        }
        else {
            fprintf(stderr, "Error writing file 'state'.\n");
            /* showState(experiment);*/    /* DEBUG: 15-02-2001  */
}

} else if(!strcmp(argv[0],"load")) {
        if((handle = dlopen(argv[1],RTLD_LAZY | RTLD_GLOBAL)) == NULL) {
            fprintf(stderr, dlerror() );
            return _ERROR;
        }
        fprintf(stdout, "%s succesfully loaded.\n", argv[1] );
        for(i=0; 2+i<argc; i++) {
            if((spec = dlsym(handle, argv[2+i])) == NULL) {
            dlclose(handle);
                fprintf(stderr, "module specification in object not found\n");
                return _ERROR;
            } else if(spec->version != 1) {
                dlclose(handle);
                fprintf(stderr, "module of incompatible specification\n");
                return _ERROR;
            }
            vlab__insmod(spec);
        }

} else if(!strcmp(argv[0],"exit")) {
        fprintf(stderr, "Quitting vlab...\n");
        fprintf(stderr, "Start killing threads.\n");
        kill(experiment);    /* kill all active threads */
        pthread_testcancel();
        fprintf(stderr, "Finished killing threads.\n");
        return _OK;
    }
```

```
    /*
     * experiment-block
     */
    else if(!strcmp(argv[0],"add")) {
        fprintf(stderr, "adding module...\n");

        if(argc != 3) {
            fprintf(stderr, "arguments mismatch: add moduleName moduleType\n");
            return _ERROR;
        }
        module = malloc(sizeof(struct moduledesc));
        module->name        = strdup(argv[1]);
        module->type        = strdup(argv[2]);
        module->ident       = experiment->counter++;
        module->next        = experiment->modules;
        module->arguments   = NULL;
        experiment->modules = module;
        tmp = experiment->modules;  /* 26-10-2000 added to get 'set'-routine to work properly */
        fprintf(stderr, "module->name = %s\n", module->name);

        fp = fopen("state", "a+");
        if (fp != NULL) {
            fputs("module\t", fp);
            fputs(module->name, fp);
            fputs("\t", fp);
            fputs(module->type, fp);
            fputs("\t", fp);
            fprintf(fp, "%p", module);
            fputs("\n", fp);
            fclose(fp);
        }
        else {
            fprintf(stderr, "Error writing file 'state'.\n");
            /* showState(experiment);*/    /* DEBUG: 15-02-2001  */
}

    } else if(!strcmp(argv[0],"connect")) {
        fprintf(stderr, "connecting modules...\n");
        if(argc != 5) {
            fprintf(stderr, "arguments mismatch: connect producerModule "
                "producerPort consumerModule consumerPort\n");
            return _ERROR;
        }
        stream = malloc(sizeof(struct streamdesc));
        found = 0;
        for(module=experiment->modules; module; module=module->next) {
            if(!strcmp(argv[1],module->name)) {
                found = 1;
                break;
            }
}
        if (found) {
            stream->producer   = module;
            stream->outputport = strdup(argv[2]);
            fprintf(stderr, "outputport connected...\n");
            found = 0;
            for(module=experiment->modules; module; module=module->next) {
                if(!strcmp(argv[3],module->name)) {
                    found = 1;
                    break;
                }
}
            if (found) {
                stream->consumer  = module;
                stream->inputport = strdup(argv[4]);
                fprintf(stderr, "inputport connected...\n");
                stream->next = experiment->streams;
                experiment->streams = stream;
                /*fprintf(stderr, "writing stream-state...\n");*/
                fp = fopen("state", "a+");
                if (fp != NULL) {
                    fputs("stream\t", fp);
                    fprintf(fp, "%p", stream);
                    fputs("\n", fp);
                    fclose( fp);
                }
                else {
                    fprintf(stderr, "Error writing file 'state'.\n");
}
            }
             else {
                fprintf(stderr, "Couldn't find the second module '%s' ", argv[3] );
                fprintf(stderr, "so I can't connect the modules.\n");
 }
        }
        else {
            fprintf(stderr, "Couldn't find the first module '%s' ", argv[1] );
            fprintf(stderr, "so I can't connect the modules.\n");
}
        /* showState(experiment);*/    /* DEBUG: 15-02-2001  */

    } else if(!strcmp(argv[0],"start")) {
        fprintf(stderr, "Starting experiment...\n");
        showState(experiment);
        run(experiment);
    }
    /*
     * module-block
     */
```

```
    else if(!strcmp(argv[0],"set")) {
        found = 0;
        for(module=experiment->modules; module; module=module->next) {
            if(!strcmp(argv[1],module->name)) {
                found = 1;
                break;
            }
        }
        if (found) {
            fprintf(stderr, "Setting argument...%s %s\n", argv[2], lastarg );
            argument = malloc(sizeof(struct argumentdesc));
            argument->name    = strdup(argv[2]);
            argument->value   = atoi(lastarg);   /* lastarg == argv[3] */
            argument->next    = experiment->modules->arguments;
            module->arguments = argument;
            /*fprintf(stderr, "writing argument-state...\n");*/
            fp = fopen("state", "a+");
            if (fp != NULL) {
                fputs("argument\t", fp);
                fprintf(fp, "%s\t%s\t%s\t", argv[2], lastarg, argv[1]);
                fprintf(fp, "%p\n", argument);
                fclose(fp);
            }
            else {
                fprintf(stderr, "Error writing file 'state'.\n");
                /* showState(experiment);*/    /* DEBUG: 15-02-2001  */
}
        }
        else {
            fprintf(stderr, "Couldn't find module '%s' ", argv[1] );
            fprintf(stderr, "so I can't set the requested argument '%s'.\n", argv[2] );
}

    } else if(!strcmp(argv[0],"get")) {

for(module=experiment->modules; module; module=module->next) {
            if(!strcmp(argv[1],module->name)) {
                vlab_getState(module->impl, argv[2], values);   /* 'get m3 values 10' */
                /*
                fprintf(stderr, "Thread '%d' active.\n", pthread_self() );
                mod = module->impl;
                if (mod->thread != pthread_self() ) {
                    fprintf(stderr, "mod->thread = %d,\t", mod->thread );
                    fprintf(stderr, "pthread_self() = %d\n", pthread_self() );
                }
                */
                count = atoi(argv[3]);
                fprintf(stderr, "count = '%d'\n", count );
                ds_index = 0;
                fprintf(stderr, "\nResult get: ");
                for(i=0; i<count; i++) {
                    sprintf(aux,"%d",values[i]);
                    fprintf(stderr, "%d ", values[i] );
                }
                fprintf(stderr, "\n");
            }
        }

    } else if(!strcmp(argv[0],"put")) {

        for(module=experiment->modules; module; module=module->next) {
            if(!strcmp(argv[1],module->name)) {
                values[0] = atoi(argv[3]);
                vlab_setParameter(module->impl, argv[2], values);
                fprintf(stderr, "Putting '%s' at %d\n", argv[2], values[0]);
            }
        }

}
    else if (!
( !(strcmp(argv[0],"create")) || !(strcmp(argv[0],"load"))
  || !(strcmp(argv[0],"add")) || !(strcmp(argv[0],"connect"))
  || !(strcmp(argv[0],"start")) || !(strcmp(argv[0],"set"))
  || !(strcmp(argv[0],"get")) || !(strcmp(argv[0],"put")) ) ) {
        fprintf(stderr, "'%s' is not a valid argument.\n", argv[0]);
    }

    return _OK;

}


void mexFunction( int nlhs, mxArray *plhs[], int nrhs, const mxArray*prhs[] )
{
    int status, buflen, i, mrows, ncols;
    char *input_buf[maxargs], *args[maxargs];
    char *result;
    double *y;

    /* Check for proper number of arguments */
    if (nrhs != 3) {
        mexErrMsgTxt("Three input arguments required.");
    } else if (nlhs > 1) {
        mexErrMsgTxt("Too many output arguments.");
    }

    /* Input must be strings */
    if (mxIsChar(prhs[0]) != 1)
        mexErrMsgTxt("Argument 1 must be a string");
```

```
    if (mxIsChar(prhs[1]) != 1)
        mexErrMsgTxt("Argument 2 must be a string");
    if (mxIsChar(prhs[2]) != 1)
        mexErrMsgTxt("Argument 3 must be a string");

    for (i=0; i < nrhs; i++) {
        if (prhs[i] != NULL) {
            buflen = (mxGetM(prhs[i]) * mxGetN(prhs[i])) + 1;
            args[i] = mxCalloc(buflen, sizeof(char));
            status = mxGetString(prhs[i], args[i], buflen);
        }
    }
#if 0
    if (status != 0)
        mexWarnMsgTxt("Not enough space creating input string. String is truncated.");
#endif

    mexWarnMsgTxt("Starting vlab...");
    status = vlab(args, nrhs);
/*
    switch(status) {
        case 0: result = "_OK"; break;
        case 1: result = "_ERROR"; break;
        case 2: result = "_RETURN"; break;
        case 3: result = "_BREAK"; break;
        case 4: result = "_CONTINUE"; break;
        default: result = "_OK";
    }
*/

    /* for returning the "get"-value to MatLab */
    if(!strcmp(args[0],"get")) {
        mrows = 1;
        ncols = atoi( args[3] );       /* number of requested values */
        plhs[0] = mxCreateDoubleMatrix(mrows, ncols, mxREAL);
        y = mxGetPr( plhs[0] );
        for (i = 0; i < ncols; i++) {
            y[i] = values[i];
            /*fprintf(stderr, "y[%d] = '%f'\n", i, y[i]);*/
        }
        fprintf(stderr, "\nFinished copying values.\n");
    }
/*
    else {
        plhs[0] = mxCreateString(result);
    }
*/

    return;

}
```

vlab_gui.m

```
function vlab_gui(option)

if nargin == 0 % set up
  hndl=figure(1) ;
  if ~get(hndl,'userdata')
      close(hndl)
      return
  end

  set(hndl,'menu', 'none', 'pos', [500 500 300 300]);

  % Set up uicontrols; 'Tag' them as belonging to a group
  uicontrol('Style', 'slider', 'Position', ...
     [20 40 200 20], 'Tag', '#1', 'min', 0, ...
     'max', 100, ...
     'Callback', 'vlab_gui set');

  % Set up text uicontrols; 'Tag' them as belonging
  % to a group

  uicontrol('Style', 'text', 'String', num2str(0), ...
     'Tag', '#1', 'Position', [230 40 40 15]);

  % Set up push button control
  uicontrol('Position', [20 10 60 20], 'String', ...
     'Create', 'Callback', 'vlab_gui create');
  uicontrol('Position', [120 10 60 20], 'String', ...
     'Start', 'Callback', 'vlab_gui start');
  uicontrol('Position', [220 10 60 20], 'String', ...
     'Quit', 'Callback', 'vlab_gui quit');
    set(gca,'unit','pixel','position',[20 65 250 200])
  drawnow

elseif strcmp(option, 'set')
  % Set the text above the slider to reflect the string
  % value.  Get handle to accompanying text uicontrol --
  % i.e., find matching 'Tag'

  % TODO: Eerst controleren of de modules wel aangemaakt zijn, door boolean?
  % --> checken of create == true en

  %if (created == 1 & started == 1)
     txthndl = findobj(gcf, 'Style', 'text', ...
     'Tag', get(gco, 'Tag'));

     % Set value to current value of slider
     %set(txthndl, 'String', num2str(get(gco, 'value')));
     threshold = num2str(get(gco, 'value'));
     set(txthndl, 'String', threshold );
     vlab('put','m2','threshold', threshold );
  %else
  %  fprintf('First click on [create], then on [start]...\n');
  %end

elseif strcmp(option, 'create')
    % Prepare vlab
    vlab('load','/home/jgztbier/vlab/matlab/modrand.so','mod_rand');
    vlab('load','/home/jgztbier/vlab/matlab/modhist.so','mod_hist');
    vlab('load','/home/jgztbier/vlab/matlab/moddisp.so','mod_disp');
    %fprintf('Creating experiment...\n');
    vlab('create','e');
    %fprintf('Adding modules...\n');
    vlab('e.add','m1','rand');
    vlab('e.add','m2','hist');
    vlab('e.add','m3','disp');
    %fprintf('Setting modules...\n');
    vlab('e.m1.set','range','10');
    vlab('e.m2.set','range','20');
    vlab('e.m3.set','range','30');
    %fprintf('Connecting modules...\n');
    vlab('e.connect','m1','out','m2','in');
    vlab('e.connect','m2','out','m3','in');
    %created = 1;

elseif strcmp(option, 'start')
    vlab('start');
    fprintf('Starting bar-graph...\n');
    counter = 0
    figure(1)
    set( 1, 'userdata',1 )
    while findobj ('String','Start')
     %fprintf('Counter = %d\n', counter);
     y = vlab('get','m3','values','10')
        bar(1:10,y)
        drawnow
    end

elseif strcmp(option, 'quit')
    vlab('exit');
    fprintf('Closing window...\n');
    set(1 , 'userdata', 0 )
    close
    return
end
```

D.1.2 Simulink interface

sfun_input.m

```
function [sys,x0,str,ts] = sfun_input(t,x,u,flag,P1)
%SFUNTMPL General M-file S-function template
%   With M-file S-functions, you can define you own ordinary differential
%   equations (ODEs), discrete system equations, and/or just about
%   any type of algorithm to be used within a Simulink block diagram.
%
%   The general form of an M-File S-function syntax is:
%       [SYS,X0,STR,TS] = SFUNC(T,X,U,FLAG,P1,...,Pn)
%
%   What is returned by SFUNC at a given point in time, T, depends on the
%   value of the FLAG, the current state vector, X, and the current
%   input vector, U.
%
%   FLAG   RESULT             DESCRIPTION
%   -----  ------             --------------------------------------------
%   0      [SIZES,X0,STR,TS]  Initialization, return system sizes in SYS,
%                             initial state in X0, state ordering strings
%                             in STR, and sample times in TS.
%   1      DX                 Return continuous state derivatives in SYS.
%   2      DS                 Update discrete states SYS = X(n+1)
%   3      Y                  Return outputs in SYS.
%   4      TNEXT              Return next time hit for variable step sample
%                             time in SYS.
%   5                         Reserved for future (root finding).
%   9      []                 Termination, perform any cleanup SYS=[].
%
%
%   The state vectors, X and X0 consists of continuous states followed
%   by discrete states.
%
%   Optional parameters, P1,...,Pn can be provided to the S-function and
%   used during any FLAG operation.
%
%   When SFUNC is called with FLAG = 0, the following information
%   should be returned:
%
%      SYS(1) = Number of continuous states.
%      SYS(2) = Number of discrete states.
%      SYS(3) = Number of outputs.
%      SYS(4) = Number of inputs.
%               Any of the first four elements in SYS can be specified
%               as -1 indicating that they are dynamically sized. The
%               actual length for all other flags will be equal to the
%               length of the input, U.
%      SYS(5) = Reserved for root finding. Must be zero.
%      SYS(6) = Direct feedthrough flag (1=yes, 0=no). The s-function
%               has direct feedthrough if U is used during the FLAG=3
%               call. Setting this to 0 is akin to making a promise that
%               U will not be used during FLAG=3. If you break the promise
%               then unpredictable results will occur.
%      SYS(7) = Number of sample times. This is the number of rows in TS.
%
%
%      X0     = Initial state conditions or [] if no states.
%
%      STR    = State ordering strings which is generally specified as [].
%
%      TS     = An m-by-2 matrix containing the sample time
%               (period, offset) information. Where m = number of sample
%               times. The ordering of the sample times must be:
%
%               TS = [0      0,      : Continuous sample time.
%                     0      1,      : Continuous, but fixed in minor step
%                                      sample time.
%                     PERIOD OFFSET, : Discrete sample time where
%                                      PERIOD > 0 & OFFSET < PERIOD.
%                     -2     0];     : Variable step discrete sample time
%                                      where FLAG=4 is used to get time of
%                                      next hit.
%
%               There can be more than one sample time providing
%               they are ordered such that they are monotonically
%               increasing. Only the needed sample times should be
%               specified in TS. When specifying than one
%               sample time, you must check for sample hits explicitly by
%               seeing if
%                  abs(round((T-OFFSET)/PERIOD) - (T-OFFSET)/PERIOD)
%               is within a specified tolerance, generally 1e-8. This
%               tolerance is dependent upon your model's sampling times
%               and simulation time.
%
%               You can also specify that the sample time of the S-function
%               is inherited from the driving block. For functions which
%               change during minor steps, this is done by
%               specifying SYS(7) = 1 and TS = [-1 0]. For functions which
%               are held during minor steps, this is done by specifying
%               SYS(7) = 1 and TS = [-1 -1].

%   Copyright (c) 1990-1998 by The MathWorks, Inc. All Rights Reserved.
%   $Revision: 1.12 $

%
```

```
% The following outlines the general structure of an S-function.
%
switch flag,

  %%%%%%%%%%%%%%%%%%%
  % Initialization %
  %%%%%%%%%%%%%%%%%%%
  case 0,
    [sys,x0,str,ts]=mdlInitializeSizes(t,P1);

  %%%%%%%%%%%%%%%%
  % Derivatives %
  %%%%%%%%%%%%%%%%
  case 1,
    sys=mdlDerivatives(t,x,u,P1);

  %%%%%%%%%%
  % Update %
  %%%%%%%%%%
  case 2,
    sys=mdlUpdate(t,x,u, P1);

  %%%%%%%%%%
  % Outputs %
  %%%%%%%%%%
  case 3,
    sys=mdlOutputs(t,x,u);

  %%%%%%%%%%%%%%%%%%%%%%%%
  % GetTimeOfNextVarHit %
  %%%%%%%%%%%%%%%%%%%%%%%%
  case 4,
    sys=mdlGetTimeOfNextVarHit(t,x,u);

  %%%%%%%%%%%%%
  % Terminate %
  %%%%%%%%%%%%%
  case 9,
    sys=mdlTerminate(t,x,u);

  %%%%%%%%%%%%%%%%%%%%
  % Unexpected flags %
  %%%%%%%%%%%%%%%%%%%%
  otherwise
    error(['Unhandled flag = ',num2str(flag)]);

end

% end sfuntmpl

prev_t = 0;

%
%=============================================================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the S-function.
%=============================================================================
%
function [sys,x0,str,ts]=mdlInitializeSizes(t,P1)

%
% call simsizes for a sizes structure, fill it in and convert it to a
% sizes array.
%
% Note that in this example, the values are hard coded.  This is not a
% recommended practice as the characteristics of the block are typically
% defined by the S-function parameters.
%
sizes = simsizes;

sizes.NumContStates  = 0;
sizes.NumDiscStates  = 1;
sizes.NumOutputs     = 0;
sizes.NumInputs      = 1;
sizes.DirFeedthrough = 1;
sizes.NumSampleTimes = 1;   % at least one sample time is needed

sys = simsizes(sizes);

%
% initialize the initial conditions
%
x0  = zeros(1,1);

%
% str is always an empty matrix
%
str = [];

%
% initialize the array of sample times
%
ts  = [0 0];

prev_t = t;

%
% initialize vlab-modules
%
```

```
    vlab('load','/home/jgztbier/vlab/matlab/simulink/modinput.so', ...
                                                        'mod_input');
    vlab('e.add','m1','input');
    vlab('e.m1.set','range','10');
% end mdlInitializeSizes

%
%=============================================================================
% mdlDerivatives
% Return the derivatives for the continuous states.
%=============================================================================
%
function sys=mdlDerivatives(t,x,u,P1)

%dx = zeros(size(x));
%
%l = floor (u)
%
%if l > 0 & l < size(x,1)
%   dx(l+1)=1./dt;
%end
%
% sys = dx;
sys = [];

% end mdlDerivatives

%
%=============================================================================
% mdlUpdate
% Handle discrete state updates, sample time hits, and major time step
% requirements.
%=============================================================================
%
function sys=mdlUpdate(t,x,u, P1)

%pause(0.01);
vlab('put','m1','value',num2str(u));
% sprintf('input@%f: %f -> "%s"\n', t, u, num2str(u))

sys = u;

% end mdlUpdate

%
%=============================================================================
% mdlOutputs
% Return the block outputs.
%=============================================================================
%
function sys=mdlOutputs(t,x,u)

sys = [];

% end mdlOutputs

%
%=============================================================================
% mdlGetTimeOfNextVarHit
% Return the time of the next hit for this block.  Note that the result is
% absolute time.  Note that this function is only used when you specify a
% variable discrete-time sample time [-2 0] in the sample time array in
% mdlInitializeSizes.
%=============================================================================
%
function sys=mdlGetTimeOfNextVarHit(t,x,u)

sampleTime = 1;    %  Example, set the next hit to be one second later.
sys = t + sampleTime;

% end mdlGetTimeOfNextVarHit

%
%=============================================================================
% mdlTerminate
% Perform any end of simulation tasks.
%=============================================================================
%
function sys=mdlTerminate(t,x,u)

sys = [];

% end mdlTerminate
```

sfun_hist.m

```
function [sys,x0,str,ts] = sfun_hist(t,x,u,flag,P1)
%
% The following outlines the general structure of an S-function.
%
switch flag,

  %%%%%%%%%%%%%%%%%%
  % Initialization %
  %%%%%%%%%%%%%%%%%%
  case 0,
    [sys,x0,str,ts]=mdlInitializeSizes(t,P1);

  %%%%%%%%%%%%%%%%
  % Derivatives %
  %%%%%%%%%%%%%%%%
  case 1,
    sys=mdlDerivatives(t,x,u,P1);

  %%%%%%%%%%
  % Update %
  %%%%%%%%%%
  case 2,
    sys=mdlUpdate(t,x,u, P1);

  %%%%%%%%%%
  % Outputs %
  %%%%%%%%%%
  case 3,
    sys=mdlOutputs(t,x,u,P1);

  %%%%%%%%%%%%%%%%%%%%%%
  % GetTimeOfNextVarHit %
  %%%%%%%%%%%%%%%%%%%%%%
  case 4,
    sys=mdlGetTimeOfNextVarHit(t,x,u);

  %%%%%%%%%%%%%%
  % Terminate %
  %%%%%%%%%%%%%%
  case 9,
    sys=mdlTerminate(t,x,u);

  %%%%%%%%%%%%%%%%%%%%
  % Unexpected flags %
  %%%%%%%%%%%%%%%%%%%%
  otherwise
    error(['Unhandled flag = ',num2str(flag)]);

end

% end sfuntmpl

prev_t = 0;

%
%=============================================================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the S-function.
%=============================================================================
%
function [sys,x0,str,ts]=mdlInitializeSizes(t,P1)

%
% call simsizes for a sizes structure, fill it in and convert it to a
% sizes array.
%
% Note that in this example, the values are hard coded.  This is not a
% recommended practice as the characteristics of the block are typically
% defined by the S-function parameters.
%
sizes = simsizes;

sizes.NumContStates  = 0;
sizes.NumDiscStates  = 1;
sizes.NumOutputs     = 1;
sizes.NumInputs      = 0;
sizes.DirFeedthrough = 1;
sizes.NumSampleTimes = 1;   % at least one sample time is needed

sys = simsizes(sizes);

%
% initialize the initial conditions
%
x0  = 1;

%
% str is always an empty matrix
%
str = [];

%
% initialize the array of sample times
%
ts  = [0 0];
```

```
prev_t = t;

%
% initialize vlab-modules
%
    vlab('load','/home/jgztbier/vlab/matlab/simulink/modhist.so', ...
                                                            'mod_hist');
    vlab('e.add','m2','hist');
    vlab('e.m2.set','range','10');
    vlab('e.connect','m1','out','m2','in');
% end mdlInitializeSizes

%
%=============================================================================
% mdlDerivatives
% Return the derivatives for the continuous states.
%=============================================================================
%
function sys=mdlDerivatives(t,x,u,P1)

%dx = zeros(size(x));
%
%l = floor (u)
%
%if l > 0 & l < size(x,1)
%   dx(l+1)=1./dt;
%end
%
% sys = dx;
sys = [];

% end mdlDerivatives

%
%=============================================================================
% mdlUpdate
% Handle discrete state updates, sample time hits, and major time step
% requirements.
%=============================================================================
%
function sys=mdlUpdate(t,x,u, P1)

if ( t < 0.2)
    vlab('put','m2','threshold', num2str(P1));
end

x = vlab('get','m2','count','1');

sys = x;

% end mdlUpdate

%
%=============================================================================
% mdlOutputs
% Return the block outputs.
%=============================================================================
%
function sys=mdlOutputs(t,x,u,P1)

if x == 1
sys = 1;
else
sys = 0;
end


% end mdlOutputs

%
%=============================================================================
% mdlGetTimeOfNextVarHit
% Return the time of the next hit for this block.  Note that the result is
% absolute time.  Note that this function is only used when you specify a
% variable discrete-time sample time [-2 0] in the sample time array in
% mdlInitializeSizes.
%=============================================================================
%
function sys=mdlGetTimeOfNextVarHit(t,x,u)

sampleTime = 1;    %  Example, set the next hit to be one second later.
sys = t + sampleTime;

% end mdlGetTimeOfNextVarHit

%
%=============================================================================
% mdlTerminate
% Perform any end of simulation tasks.
%=============================================================================
%
function sys=mdlTerminate(t,x,u)

sys = [];

% end mdlTerminate
```

sfun_disp.m

```
function [sys,x0,str,ts] = sfun_disp(t,x,u,flag,P1)

%
% The following outlines the general structure of an S-function.
%
switch flag,

  %%%%%%%%%%%%%%%%%%%
  % Initialization %
  %%%%%%%%%%%%%%%%%%%
  case 0,
    [sys,x0,str,ts]=mdlInitializeSizes(t,P1);

  %%%%%%%%%%%%%%%%
  % Derivatives %
  %%%%%%%%%%%%%%%%
  case 1,
    sys=mdlDerivatives(t,x,u,P1);

  %%%%%%%%%%
  % Update %
  %%%%%%%%%%
  case 2,
    sys=mdlUpdate(t,x,u, P1);

  %%%%%%%%%%
  % Outputs %
  %%%%%%%%%%
  case 3,
    sys=mdlOutputs(t,x,u);

  %%%%%%%%%%%%%%%%%%%%%%%
  % GetTimeOfNextVarHit %
  %%%%%%%%%%%%%%%%%%%%%%%
  case 4,
    sys=mdlGetTimeOfNextVarHit(t,x,u);

  %%%%%%%%%%%%%
  % Terminate %
  %%%%%%%%%%%%%
  case 9,
    sys=mdlTerminate(t,x,u);

  %%%%%%%%%%%%%%%%%%%%
  % Unexpected flags %
  %%%%%%%%%%%%%%%%%%%%
  otherwise
    error(['Unhandled flag = ',num2str(flag)]);

end

% end sfuntmpl

prev_t = 0;

%
%=============================================================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the S-function.
%=============================================================================
%
function [sys,x0,str,ts]=mdlInitializeSizes(t,P1)

%
% call simsizes for a sizes structure, fill it in and convert it to a
% sizes array.
%
% Note that in this example, the values are hard coded.  This is not a
% recommended practice as the characteristics of the block are typically
% defined by the S-function parameters.
%
sizes = simsizes;

sizes.NumContStates  = 0;
sizes.NumDiscStates  = 10;
sizes.NumOutputs     = 10;
sizes.NumInputs      = 0;
sizes.DirFeedthrough = 1;
sizes.NumSampleTimes = 1;   % at least one sample time is needed

sys = simsizes(sizes);

%
% initialize the initial conditions
%
x0  = zeros (1,10);

%
% str is always an empty matrix
%
str = [];

%
% initialize the array of sample times
%
```

```
ts  = [-1 0];

prev_t = t;

%
% initialize vlab-modules
%
    vlab('load','/home/jgztbier/vlab/matlab/moddisp.so','mod_disp');
    vlab('e.add','m3','disp');
    vlab('e.m3.set','range','10');
    vlab('e.connect','m2','out','m3','in');
% end mdlInitializeSizes

%
%=============================================================================
% mdlDerivatives
% Return the derivatives for the continuous states.
%=============================================================================
%
function sys=mdlDerivatives(t,x,u,P1)

%dx = zeros(size(x));
%
%l = floor (u)
%
%if l > 0 & l < size(x,1)
%   dx(l+1)=1./dt;
%end
%
% sys = dx;
sys = [];

% end mdlDerivatives

%
%=============================================================================
% mdlUpdate
% Handle discrete state updates, sample time hits, and major time step
% requirements.
%=============================================================================
%
function sys=mdlUpdate(t,x,u, P1)

y = vlab('get','m3','values','10');

if (y == x')
    pause(0.00001);
end

sys = y;

% end mdlUpdate

%
%=============================================================================
% mdlOutputs
% Return the block outputs.
%=============================================================================
%
function sys=mdlOutputs(t,x,u)

sys = x;

% end mdlOutputs

%
%=============================================================================
% mdlGetTimeOfNextVarHit
% Return the time of the next hit for this block.  Note that the result is
% absolute time.  Note that this function is only used when you specify a
% variable discrete-time sample time [-2 0] in the sample time array in
% mdlInitializeSizes.
%=============================================================================
%
function sys=mdlGetTimeOfNextVarHit(t,x,u)

sampleTime = 1;    %  Example, set the next hit to be one second later.
sys = t + sampleTime;

% end mdlGetTimeOfNextVarHit

%
%=============================================================================
% mdlTerminate
% Perform any end of simulation tasks.
%=============================================================================
%
function sys=mdlTerminate(t,x,u)

sys = [];

% end mdlTerminate
```

simulink2vlab.mdl

```
Model {
  Name   "simulink2vlab"
  Version   3.00
  SimParamPage   "Solver"
  SampleTimeColors   off
  InvariantConstants   off
  WideVectorLines   off
  ShowLineWidths   off
  ShowPortDataTypes   off
  StartTime   "0.0"
  StopTime   "300.0"
  SolverMode   "Auto"
  Solver   "FixedStepDiscrete"
  RelTol   "1e-3"
  AbsTol   "auto"
  Refine   "1"
  MaxStep   "auto"
  InitialStep   "auto"
  FixedStep   "0.2"
  MaxOrder   5
  OutputOption   "RefineOutputTimes"
  OutputTimes   "[]"
  LoadExternalInput   off
  ExternalInput   "[t, u]"
  SaveTime   on
  TimeSaveName   "tout"
  SaveState   off
  StateSaveName   "xout"
  SaveOutput   on
  OutputSaveName   "yout"
  LoadInitialState   off
  InitialState   "xInitial"
  SaveFinalState   off
  FinalStateName   "xFinal"
  SaveFormat   "Matrix"
  LimitMaxRows   off
  MaxRows   "1000"
  Decimation   "1"
  AlgebraicLoopMsg   "warning"
  MinStepSizeMsg   "warning"
  UnconnectedInputMsg   "warning"
  UnconnectedOutputMsg   "warning"
  UnconnectedLineMsg   "warning"
  InheritedTsInSrcMsg   "warning"
  IntegerOverflowMsg   "warning"
  UnnecessaryDatatypeConvMsg "none"
  Int32ToFloatConvMsg   "warning"
  SignalLabelMismatchMsg  "none"
  ConsistencyChecking   "off"
  ZeroCross   on
  SimulationMode   "normal"
  BlockDataTips   on
  BlockParametersDataTip  on
  BlockAttributesDataTip  off
  BlockPortWidthsDataTip  off
  BlockDescriptionStringDataTip off
  BlockMaskParametersDataTip off
  ToolBar   on
  StatusBar   on
  BrowserShowLibraryLinks off
  BrowserLookUnderMasks   off
  OptimizeBlockIOStorage  on
  BufferReuse   on
  BooleanDataType   off
  RTWSystemTargetFile   "grt.tlc"
  RTWInlineParameters   off
  RTWRetainRTWFile   off
  RTWTemplateMakefile   "grt_default_tmf"
  RTWMakeCommand   "make_rtw"
  RTWGenerateCodeOnly   off
  ExtModeMexFile   "ext_comm"
  ExtModeBatchMode   off
  ExtModeTrigType   "manual"
  ExtModeTrigMode   "oneshot"
  ExtModeTrigPort   "1"
  ExtModeTrigElement   "any"
  ExtModeTrigDuration   1000
  ExtModeTrigHoldOff   0
  ExtModeTrigDelay   0
  ExtModeTrigDirection   "rising"
  ExtModeTrigLevel   0
  ExtModeArchiveMode   "off"
  ExtModeAutoIncOneShot   off
  ExtModeIncDirWhenArm   off
  ExtModeAddSuffixToVar   off
  ExtModeWriteAllDataToWs off
  ExtModeArmWhenConnect   off
  StartFcn   "vlab('start');"
  StopFcn   "vlab('exit');"
  Created   "Tue Mar 20 16:27:40 2001"
  Creator   "jgztbier"
  UpdateHistory   "UpdateHistoryNever"
  ModifiedByFormat   "%<Auto>"
  LastModifiedBy   "jgztbier"
  ModifiedDateFormat   "%<Auto>"
```

```
  LastModifiedDate   "Thu Mar 29 15:16:45 2001"
  ModelVersionFormat   "1.%<AutoIncrement:50>"
  ConfigurationManager   "none"
  BlockDefaults {
    Orientation     "right"
    ForegroundColor     "black"
    BackgroundColor     "white"
    DropShadow     off
    NamePlacement     "normal"
    FontName     "Helvetica"
    FontSize     10
    FontWeight     "normal"
    FontAngle     "normal"
    ShowName     on
  }
  AnnotationDefaults {
    HorizontalAlignment     "center"
    VerticalAlignment     "middle"
    ForegroundColor     "black"
    BackgroundColor     "white"
    DropShadow     off
    FontName     "Helvetica"
    FontSize     10
    FontWeight     "normal"
    FontAngle     "normal"
  }
  LineDefaults {
    FontName     "Helvetica"
    FontSize     9
    FontWeight     "normal"
    FontAngle     "normal"
  }
  System {
    Name     "simulink2vlab"
    Location     [338, 573, 838, 833]
    Open     on
    ModelBrowserVisibility  off
    ModelBrowserWidth     200
    ScreenColor     "white"
    PaperOrientation     "landscape"
    PaperPositionMode     "auto"
    PaperType     "usletter"
    PaperUnits     "inches"
    ZoomFactor     "100"
    AutoZoom     off
    ReportName     "simulink-default.rpt"
    Block {
      BlockType       SubSystem
      Name       "Experiment"
      Description       "vlab"
      Ports       [1, 2, 0, 0, 0]
      Position       [170, 96, 230, 169]
      InitFcn       "vlab('create','e');"
      ShowPortLabels       on
      System {
Name "Experiment"
Location [453, 337, 914, 479]
Open on
ModelBrowserVisibility off
ModelBrowserWidth 200
ScreenColor "white"
PaperOrientation "landscape"
PaperPositionMode "auto"
PaperType "usletter"
PaperUnits "inches"
ZoomFactor "100"
AutoZoom on
Block {
  BlockType   Inport
  Name   "In"
  Position   [25, 33, 55, 47]
  Port   "1"
  PortWidth   "-1"
  SampleTime   "-1"
  DataType   "auto"
  SignalType   "auto"
  Interpolate   on
}
Block {
  BlockType   "S-Function"
  Name   "m1"
  Ports   [1, 0, 0, 0, 0]
  Position   [80, 25, 140, 55]
  FunctionName   "sfun_input"
  Parameters   "10"
  PortCounts   "[]"
  SFunctionModules   "''"
}
Block {
  BlockType   "S-Function"
  Name   "m2"
  Ports   [0, 1, 0, 0, 0]
  Position   [195, 25, 255, 55]
  FunctionName   "sfun_hist"
  Parameters   "20"
  PortCounts   "[]"
  SFunctionModules   "''"
}
Block {
```

```
  BlockType   "S-Function"
  Name   "m3"
  Ports   [0, 1, 0, 0, 0]
  Position   [285, 70, 345, 100]
  FunctionName   "sfun_disp"
  Parameters   "30"
  PortCounts   "[]"
  SFunctionModules   "''"
  MaskIconFrame   on
  MaskIconOpaque   on
  MaskIconRotate   "none"
  MaskIconUnits   "autoscale"
}
Block {
  BlockType   Outport
  Name   "full"
  Position   [400, 33, 430, 47]
  Port   "1"
  OutputWhenDisabled   "held"
  InitialOutput   "[]"
}
Block {
  BlockType   Outport
  Name   "Values"
  Position   [400, 78, 430, 92]
  Port   "2"
  OutputWhenDisabled   "held"
  InitialOutput   "[]"
}
Line {
  SrcBlock   "In"
  SrcPort   1
  DstBlock   "m1"
  DstPort   1
}
Line {
  SrcBlock   "m2"
  SrcPort   1
  DstBlock   "full"
  DstPort   1
}
Line {
  SrcBlock   "m3"
  SrcPort   1
  DstBlock   "Values"
  DstPort   1
}
      }
    }
    Block {
      BlockType       UniformRandomNumber
      Name       "Uniform Random\nNumber"
      Position       [40, 119, 70, 151]
      Minimum       "0"
      Maximum       "9.99999999"
      Seed       "0"
      SampleTime       "0"
    }
    Block {
      BlockType       SubSystem
      Name       "bar"
      Ports       [1, 0, 0, 1, 0]
      Position       [320, 134, 360, 166]
      ShowPortLabels       on
      System {
Name "bar"
Location [97, 410, 595, 710]
Open off
ModelBrowserVisibility off
ModelBrowserWidth 200
ScreenColor "white"
PaperOrientation "landscape"
PaperPositionMode "auto"
PaperType "usletter"
PaperUnits "inches"
ZoomFactor "100"
AutoZoom on
Block {
  BlockType   Inport
  Name   "y"
  Position   [65, 118, 95, 132]
  Port   "1"
  PortWidth   "-1"
  SampleTime   "-1"
  DataType   "auto"
  SignalType   "auto"
  Interpolate   on
}
Block {
  BlockType   TriggerPort
  Name   "Trigger"
  Ports   [0, 0, 0, 0, 0]
  Position   [205, 75, 225, 95]
  TriggerType   "either"
  ShowOutputPort   off
  OutputDataType   "auto"
}
Block {
  BlockType   MATLABFcn
```

```
  Name   "MATLAB Fcn"
  Position   [185, 160, 245, 190]
  MATLABFcn   "bar(u)"
  OutputWidth   "0"
  OutputSignalType   "auto"
}
Line {
  SrcBlock   "y"
  SrcPort   1
  Points   [70, 0]
  DstBlock   "MATLAB Fcn"
  DstPort   1
}
      }
    }
    Line {
      SrcBlock       "Uniform Random\nNumber"
      SrcPort       1
      DstBlock       "Experiment"
      DstPort       1
    }
    Line {
      SrcBlock       "Experiment"
      SrcPort       2
      DstBlock       "bar"
      DstPort       1
    }
    Line {
      SrcBlock       "Experiment"
      SrcPort       1
      Points       [105, 0]
      DstBlock       "bar"
      DstPort       trigger
    }
  }
}
```

D.2 XML experiment description VLAM-G

XML header

```
<?xml version="1.0" standalone="yes"?>
<!-- This file describes an experiment topology -->
<!-- This file is generated by the Front-End -->
<!-- DO NOT edit this file manually! -->
<!-- Version VlReadTopologyFromDatabase: \$Revision: 1.1 \$ -->
```

XML body

```
<topology>
  <module definition 1>
  <module definition 2>
  <module definition 3>
  ...
  <module definition n>

  <instance definition 1>
  <instance definition 2>
  <instance definition 3>
  ...
  <instance definition n>

  <connection definitions 1>
  <connection definitions 2>
  ...
  <connection definitions (n-1)>
</topology>
```

XML module definition

```
<module name="moduleName" moduleId="moduleId">
  <cpuTimeRequest>value (double)</cpuTimeRequest>
  <memoryRequest>value (double)</memoryRequest>
  <storageDemand>value (double)</storageDemand>
  <executable>
    <platform>platformName</platform>
    <location>pathname/executable</location>
  </executable>
  <port name="portName">
    <dataType>integer/double/...</dataType>
    <direction>input/output</direction>
    <replication>true/false</replication>
  </port>
</module>
```

XML instance definition

```
<instance instanceId="Id" moduleId="moduleId">
  <host>hostname</host>
  <dn>GRID certificate</dn>
  <position x="value (int)" y="value (int)"/>
</instance>
```

XML connection definition

```
<connection name="connectionName">
  <output port="portName" instanceId="value (int)" />
  <input port="portName" instanceId="value (int)" />
  <!-- Data to draw this connection -->
  <point x="value (int)" y="value (int)" />
  <point x="value (int)" y="value (int)" />
</connection>
```

D.3 MDL2XML

```
function mdl2xml(mdlfile, xmlfile)
% mdl2xml - converts a Simulink .MDL-file to a VLAM-G .xml-file

if nargin < 2
  error('Not enough input arguments. Usage:  mdl2xml(mdlfile, xmlfile) ');
end

blks = find_system(gcs, 'Type', 'block');
moduleNames = get_param(blks, 'moduleName');
moduleIds = get_param(blks, 'moduleId');
platforms = get_param(blks, 'platform');
locations = get_param(blks, 'proglocation');
datatypes = get_param(blks, 'datatype');
directions = get_param(blks, 'direction');
replications = get_param(blks, 'replication');
hosts = get_param(blks, 'host');
dns = get_param(blks, 'dn');

% determine portnames and put them in a table with the corresponding moduleName and instanceID
j = 1; k = 1;
blocktypes = get_param(blks, 'BlockType');
for i=1:length(blocktypes)
    if findstr( cell2mat(blocktypes(i)), 'SubSystem')
        outports(j,:) = [ moduleNames(i) ( get_param(blks(i), 'outport1') ) {i-1} ];
        j=j+1;
        outports(j,:) = [ moduleNames(i) ( get_param(blks(i), 'outport2') ) {i-1} ];
        j=j+1;
    else
        portname = get_param(blks(i), 'inport1');
        if (~isempty(cell2mat(portname)))
            inports(k,:) = [ moduleNames(i) ( portname ) {i-1} ];
            k=k+1;
        end
        portname = get_param(blks(i), 'outport1');
        if (~isempty(cell2mat(portname)))
            outports(j,:) = [ moduleNames(i) ( portname ) {i-1} ];
            j=j+1;
        end
    end
end
%inports                        % DEBUG
%[rows, cols] = size(inports);  % DEBUG
%outports                       % DEBUG
%[rows, cols] = size(outports); % DEBUG

lines = get_param(gcs, 'Lines');
j = 1;
for i = 1:length(lines)
    sources{j} = get_param(lines(i).SrcBlock, 'Name');
    destinations{j} = get_param(lines(i).DstBlock, 'Name');
    j=j+1;
    sources{j} = lines(i).SrcPort;
    destinations{j} = lines(i).DstPort;
    j=j+1;
end
%sources                        % DEBUG
%destinations                   % DEBUG



% start writing the xmlfile by using the writeXML-functions
fprintf('Writing %s...\n', xmlfile);
writeXMLheader(xmlfile);
elements = size(moduleNames, 1);

for i = 1:elements
    writeXMLmodule(xmlfile, cell2mat( moduleNames(i) ), cell2mat( moduleIds(i) ), cell2mat( platforms(i) ),
        cell2mat( locations(i) ), sources, destinations, inports, outports, cell2mat( datatypes(i) ),
        cell2mat( directions(i) ), cell2mat( replications(i) ) );
end

for i = 1:elements
    writeXMLinstance(xmlfile, (i-1), cell2mat( moduleIds(i) ), cell2mat( hosts(i) ), cell2mat( dns(i) ) );
end

% writing connections
connectionName = 'Exp_Top_';
for i = 1:elements
    connectionName = strcat(connectionName, int2str(i));
end
connectionName = strcat(connectionName, '_');
for i = 1 : length(lines)

    % find modulename for the beginning of this connection
    for j=1:elements
        if ( findstr( cell2mat(moduleNames(j)), cell2mat(sources((2*i)-1)) ) )
            outputportID = (j-1);
        break;
    end
    end
    % find modulename for the ending of this connection
    for j=1:elements
        if ( findstr( cell2mat(moduleNames(j)), cell2mat(destinations((2*i)-1)) ) )
            inputportID = (j-1);
        break;
```

```
    end
    end
    connectionNr = strcat(connectionName, int2str(i-1) );
    writeXMLconnection(xmlfile, connectionNr, cell2mat( sources(2*i) ), outputportID, cell2mat( destinations(2*i) ),
        inputportID, cell2mat(sources((2*i)-1)), cell2mat(destinations((2*i)-1)), inports, outports );

end


fprintf('Done.\n');




function writeXMLheader(filename)     % Subfunction
% write XML-header to file
fid = fopen(filename, 'w');
fprintf(fid, '<?xml version=\"1.0\" standalone=\"yes\"?>\n');
fprintf(fid, '\n');
fprintf(fid, '<!-- This file describes an experiment topology -->\n');
fprintf(fid, '<!-- This file is generated by the Front-End -->\n');
fprintf(fid, '<!-- DO NOT edit this file manually! -->\n');
fprintf(fid, '<!-- Version VlReadTopologyFromDatabase: $Revision: 1.1 $ -->\n');
fprintf(fid, '\n');
fprintf(fid, '<topology>\n');
fclose(fid);


function writeXMLmodule(filename, moduleName, moduleId, platform, location, source, dest, inputports, outputports,
                        dataType, direction, replication )    % Subfunction
% write XML-specification of a module to file
fid = fopen(filename, 'a');
fprintf(fid, '  <module name=\"%s\" moduleId=\"%s\">\n', moduleName, moduleId);
fprintf(fid, '    <cpuTimeRequest>10.0</cpuTimeRequest>\n');
fprintf(fid, '    <memoryRequest>0.5</memoryRequest>\n');
fprintf(fid, '    <storageDemand>0.5</storageDemand>\n');
fprintf(fid, '    <executable>\n');
fprintf(fid, '      <platform>%s</platform>\n', platform);
fprintf(fid, '      <location>%s</location>\n', location);
fprintf(fid, '    </executable>\n');
fclose(fid);
for i=1 : 2 : length(source)
    if (findstr( cell2mat(source(i)), moduleName))
        writeXMLport(filename, moduleName, cell2mat(source(i+1) ), inputports, outputports, dataType, 'output', replication);
    end
end
for i=1 : 2 : length(dest)
    if (findstr( cell2mat(dest(i)), moduleName))
        writeXMLport(filename, moduleName, cell2mat( dest(i+1) ), inputports, outputports, dataType, 'input', replication);
    end
end
fid = fopen(filename, 'a');
fprintf(fid, '  </module>\n\n');           % added extra white line for clearity
fclose(fid);


function writeXMLport(filename, modName, portNr, inputports, outputports, dataType, direction, replication)     % Subfunction
% write XML-specification of a port to file - subfunction of writeXMLmodule
fid = fopen(filename, 'a');
name = portnr2portname(portNr, direction, modName, inputports, outputports);
fprintf(fid, '    <port name=\"%s\">\n', name);
fprintf(fid, '      <dataType>%s</dataType>\n', dataType);
fprintf(fid, '      <direction>%s</direction>\n', direction);
fprintf(fid, '      <replication>%s</replication>\n', replication);
fprintf(fid, '    </port>\n');
fclose(fid);


function writeXMLinstance(filename, instanceID, moduleId, host, dn )        % Subfunction
% write XML-specification of an instance of a module to file
fid = fopen(filename, 'a');
fprintf(fid, '  <instance instanceId=\"%d\" moduleId=\"%s\">\n', instanceID, moduleId);
fprintf(fid, '    <host>%s</host>\n', host);
fprintf(fid, '    <dn>%s</dn>\n', dn);
% x and y-coordinates are not parsed and used in the RTS
% fprintf(fid, '    <position x=\"%d\" y=\"%d\"/>\n' , positionX, positionY);
fprintf(fid, '  </instance>\n\n');          % added extra white line for clearity
fclose(fid);


function writeXMLconnection(filename, connectionName, oppNr, oppID, ippNr, ippID, srcModName, dstModName, inputports, outputports)        % Subfunction
% write XML-specification of a connection between two modules to file
% oppNr = number of outputport
% oppID = ID of outputport
% ippNr = number of inputport
% ippID = ID of inputport
oppName = portnr2portname(oppNr, 'output', srcModName, inputports, outputports);
ippName = portnr2portname(ippNr, 'input', dstModName, inputports, outputports);
fid = fopen(filename, 'a');
fprintf(fid, '  <connection name=\"%s\">\n', connectionName);
fprintf(fid, '    <output port=\"%s\" instanceId=\"%d\" />\n', oppName, oppID);
fprintf(fid, '    <input port=\"%s\" instanceId=\"%d\" />\n', ippName, ippID);
fprintf(fid, '  </connection>\n');
fclose(fid);



function portname = portnr2portname(portnumber, direction, modName, inputports, outputports)        % Subfunction
```

```
% convert a given portnumber (from a Simulinkmodule) to the associated masked portname
% direction must be 'input' or 'output', denoting the direction of the port.
if (~isnumeric(portnumber))
    portNr = str2num(portnumber);
else
    portNr = portnumber;
end
if (findstr('output', direction ))
    counter = 0;
    [rows, cols] = size(outputports);
    for i = 1:rows
        if (findstr( cell2mat( outputports(i,1) ), modName))
            counter = counter + 1;
            if (counter == portNr)
                portname =  cell2mat( outputports(i,2) );
            end
        end
    end
else
    counter = 0;
    [rows, cols] = size(inputports);
    for i = 1:rows
        if (findstr( cell2mat( inputports(i,1) ), modName))
            counter = counter + 1;
            if (counter == portNr)
                portname = cell2mat( inputports(i,2) );
            end
        end
    end
end




% ---- The function below is NOT used in the current implementation of MDL2XML,
% ---- which is based on requesting parameters from the Simulink environment.
% ---- The getMDLparameter-function directly accesses the .MDL-file to retrieve
% ---- the requested parameter. It is kept in this code for educational
% ---- purposes to show another approach to solve this problem.
% ----
function [varargout] = getMDLparameter(filename, parameterName)        % Subfunction
% retrieve the requested parameter from the .mdl-file
fid = fopen(filename, 'r');
%frewind(fid);    % to be sure that we start at the beginning of the file
j = 1;
while ( ~feof(fid))

    % read line
    line = fgetl(fid);  % discard newline character
    %line = fgets(fid);  % keep newline character
    if findstr(line, parameterName)      %check for existence of specified tag

        k = 1;
        for i=1:(length(line))
            if (line(i) == '"')
                %fprintf('DEBUG: found quote mark at position %d\n', i);
                quote(k,:) = i;
                k=k+1;
            end
            i=i+1;
        end
        varargout{j} = line(quote(1)+1:quote(2)-1)
        j=j+1;

    end

end
fclose(fid);
```