A PROBABILISTIC APPROACH TO ABSTRACT COMMUNICATION MODELS

by

Charu Jain
Bachelor of Computer Applications (Hons.),
Devi Ahilya University, 2002

PROJECT SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

In the School of Computing Science

© Charu Jain 2005

SIMON FRASER UNIVERSITY

Spring 2005

All rights reserved. This work may not be reproduced in whole or in part, by photocopy or other means, without permission of the author.
Approval

Name: Charu Jain

Degree: Master of Science, Computer Science

Title of Project: A Probabilistic Approach to Abstract Communication Models

Examining Committee: Dr. Ghassan Hamarneh, Chair

Dr. Uwe Glässer, Senior Supervisor

Dr. Ljiljana Trajkovic, SFU Examiner

Dr. Petra Berenbrink, Supervisor

Date Approved: March, 30, 2005
The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author’s written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

W. A. C. Bennett Library
Simon Fraser University
Burnaby, BC, Canada
Abstract

The "Abstract Communication Model for Distributed Systems" outlines a general method for describing executable specifications for all message-based distributed communication systems. This model is based on the ASM (Abstract State Machine) paradigm. The model is highly non-deterministic and is intended to be so in order to preserve the generality of its applicability. The ACM effectively separates the communication logic from the application logic during high-level modeling and simulation of distributed systems. This model could be specialized and used for modeling and simulation during very early stages of system analysis and design. Unlike other network simulators such as ns-2, ACM is a light-weight plug-in; high-level model for the communication component in a distributed application. ACM enables analysis at a semantic level, which is desirable during early stages of system design and analysis. Heavy-weight network simulators are unsuitable for such purposes; they could be used in later stages for accurate and elaborate simulations of the system implemented. Hence, the two approaches target complementary aspects of computer network modeling. The model as such is not implementable. In this project, we propose a natural extension of the ASM paradigm, which involves probability. We specialize the ACM partially by introducing probabilistic transitions where possible. Consequently, the model becomes amenable to statistical analysis, simulation and reasoning.
To my dear parents, and to Varun and Tarun,
for their unconditional love and support.
All professional men are handicapped by not being allowed to ignore things which are useless.

-Johann Wolfgang von Goethe
Acknowledgements

I would like to thank my senior supervisor Dr. Uwe Glässer for his constant support and guidance. This project would not have been possible without his help and timely advice despite his numerous other commitments. He has been an indefatigable source of inspiration throughout this work and I would like to take this opportunity to express my sincere gratitude. I would like to thank Dr. Ljiljana Trajkovic and Dr. Petra Berenbrink for their valuable comments on improving this project report. I would also like to thank my family for their patience throughout this work.
# Table of Contents

Approval ................................................................................................................. ii
Abstract ....................................................................................................................... iii
Dedication ..................................................................................................................... iv
Quotation ...................................................................................................................... v
Acknowledgements ................................................................................................... vi
Table of Contents ...................................................................................................... vii
List of Figures ............................................................................................................. ix
List of Specifications ................................................................................................. xi
List of Tables ............................................................................................................. xii
List of Abbreviations and Acronyms ......................................................................... xiii

## Chapter 1. Introduction ......................................................................................... 1
  1.1 Motivation and Objective .................................................................................. 4
  1.2 Project Report Organization ............................................................................ 6

## Chapter 2. Abstract State Machines ...................................................................... 8
  2.1 Basic Abstract State Machine .......................................................................... 8
    2.1.1 Parallelism .................................................................................................. 9
    2.1.2 Non-determinism ....................................................................................... 10
    2.1.3 Classification of functions ......................................................................... 10
  2.2 Distributed Abstract State Machine .................................................................. 11

## Chapter 3. Abstract Communication Model ........................................................... 13
  3.1 Abstract State Machines and AsmL .................................................................. 14
    3.1.1 States and Updates .................................................................................... 15
    3.1.2 Simulation of Agents in AsmL .................................................................. 17
  3.2 The Model ......................................................................................................... 19
    3.2.1 Communicator ........................................................................................... 20
    3.2.2 Application ............................................................................................... 21
    3.2.3 Message .................................................................................................... 23
  3.3 Probabilistic Modeling ....................................................................................... 24
    3.3.1 Probability and ASMs .............................................................................. 26
    3.3.2 Probabilistic Communicator Model ........................................................... 27
  3.4 Comparing ACM with Related Tools .................................................................. 29
    3.4.1 Architecture Description Languages ......................................................... 29
    3.4.2 Network Simulation Tools ......................................................................... 30
    3.4.3 Coordination Languages ........................................................................... 30

## Chapter 4. Interaction of ACM with the UPnP Model ............................................... 32
  4.1 Universal Plug and Play .................................................................................... 32
  4.2 Components in a UPnP Network ...................................................................... 33
    4.2.1 Devices ..................................................................................................... 35
    4.2.2 Services .................................................................................................... 35
    4.2.3 Control Points .......................................................................................... 36
  4.3 The Protocol ..................................................................................................... 36
List of Figures

Figure 1: The NAM window shows a multicast from Port 5 to Ports 2 and 10 (green packets). There is also a unicast from Port 12 to Port 8 (blue packets). ................................................................. 2

Figure 2: The NAM window shows delay and loss distribution for a communicator. ................................................................. 4

Figure 3: An instance of Abstract Communication Model [2]. ................................................................. 20

Figure 4: The window shows packets flowing from application #12 to application #8, and a form that shows the properties for application #8, its mailbox contains packets received until now. ........................................ 23

Figure 5: Statistical end-point of a communicator. ................................................................. 29

Figure 6: This picture depicts the relationship between devices, services, and control points in a UPnP network. A control point calls upon the services that are offered by UPnP enabled devices, which themselves can have control points using services offered by other devices. Derived from [20]. ...................................................... 34

Figure 7: Interaction of the UPnP model with the Abstract Communication Model [11]. ................................................................. 42

Figure 8: Class diagram showing inheritance from ACM classes [11]. ................................................................. 43

Figure 9: Role played by the Common Language Runtime environment. ................................................................. 48

Figure 10: The NAM package contains three components that work together in close collaboration. ................................................................. 49

Figure 11: The NAM window shows a multicast from Port 5 to Port 2 and Port 10 (green packets). There is also a unicast from Port 12 to Port 8 (blue packets). ................................................................. 50

Figure 12: This window shows statistics for the traffic between Port 12 and 8 in the previous simulation. ................................................................. 51

Figure 13: The window shows the connectivity and routes for Communicator 1 in the topology visible in the background. The Connections tab is made slightly transparent to enable the user to inspect the topology while modifying it, at the same time. ................................................................. 56

Figure 14: The window shows statistical details for a communicator. ................................................................. 57

Figure 15: The window shows a port form with tabs: mailbox, connections, and quantifiers that serve as an interface to modify various aspects of an application to achieve the desired behaviour. One may also control the speed of the message flow through the speed bar on bottom left corner. ................................................................. 58

Figure 16: The window shows the traffic form associated with the packet traffic between ports 12 and 8 and controls to modify the protocol for sending messages and the inter-message delays. ................................................................. 59
Figure 17: The window shows the mailbox tab of an application form, a message form describing details for the message in the mailbox, and a protocol form to customize the protocol for transfer. For each message the message form lists the intermediate receivers, the source, and the final destinations of the message.

Figure 18: UML Class diagram for the entire system. The triangular arrows depict inheritance, while the black diamonds depict composition. For instance, ComUI inherits from AgentUI while ComUI and Communicator contain a single object of the type of the other.

Figure 19: The main NAM window. There is a multicast from Port 5 to Ports 2 and 10 (green packets). There is also a unicast from Port 12 to Port 8 (blue packets).

Figure 20: Use cases.
List of Specifications

Spec 1:  In-place generic sorting .......................................................... 14
Spec 2:  Airport example, derived from [2] ........................................... 16
Spec 3:  Agents in NAM .................................................................... 17
Spec 4:  Simulation of agents in NAM .................................................. 18
Spec 5:  Communicator in NAM ............................................................ 21
Spec 6:  Application in NAM ................................................................. 22
Spec 7:  Messages in NAM ................................................................. 24
Spec 8:  Expression of non-determinism in AsmL ............................... 25
Spec 9:  Modified definition of Communicator that uses a probability distribution .......................................................... 27
Spec 10: Device and Control Point in AsmL. Adapted from [6] .............. 40
Spec 11: Description of Control phase for a device. Adapted from [6] ...... 41
Spec 12: Constructs to abstract from sequentiaity in AsmL ................. 45
Spec 13: Description of the global class model in NAM ....................... 52
Spec 14: Details of agent class in NAM .............................................. 53
Spec 15: Routing module initialization routine in NAM ....................... 54
Spec 16: Algorithm for creating the routing table in NAM .................. 55
List of Tables

Table 1: NAM Application Programming Interface........................................66
### List of Abbreviations and Acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACM</td>
<td>Abstract Communication Model</td>
</tr>
<tr>
<td>ADL</td>
<td>Architecture Description Language</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASM</td>
<td>Abstract State Machines</td>
</tr>
<tr>
<td>AsmL</td>
<td>Abstract State Machine Language</td>
</tr>
<tr>
<td>BCL</td>
<td>Base Class Library</td>
</tr>
<tr>
<td>DASM</td>
<td>Distributed Abstract State Machine</td>
</tr>
<tr>
<td>DHCP</td>
<td>Dynamic Host Configuration Protocol</td>
</tr>
<tr>
<td>GC</td>
<td>Garbage Collection</td>
</tr>
<tr>
<td>GENA</td>
<td>General Event Notification Architecture</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>IL</td>
<td>Intermediate Language</td>
</tr>
<tr>
<td>JIT</td>
<td>Just in Time Compiler</td>
</tr>
<tr>
<td>LRM</td>
<td>Language Reference Manual</td>
</tr>
<tr>
<td>NAM</td>
<td>Network Abstract Machine</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>SSDP</td>
<td>Simple Service Discovery Protocol</td>
</tr>
<tr>
<td>UPnP</td>
<td>Universal Plug and Play</td>
</tr>
<tr>
<td>VES</td>
<td>Virtual Execution System</td>
</tr>
</tbody>
</table>
Chapter 1. Introduction

The “Abstract Communication Model" describes the common part of all message-based distributed systems. It is general in the sense that it can be specialized to specify any distributed system at a high-level of abstraction. It is typically used to model the communication medium in high-level models of network-oriented distributed applications.

In this project, we develop a specialization of the abstract communication model. This specialization modifies the behaviour of the communicating entities so that they more faithfully model subnets in TCP/IP networks. We modify the behaviour of the communicating entities by replacing non-determinism by probability wherever possible [2]. We call this specialized model “Network Abstract Machine" (NAM). The “Network Abstract Machine" is a Distributed ASM (DASM) constituted by a collection of agents that perform their computation steps concurrently. More specifically, the “Network Abstract Machine" is a specialization of the “Abstract Communication Model" [2].

The “Abstract Communication Model" introduces a special category of agents called communicators. A communicator abstractly represents a multitude of routers in a TCP/IP network. It transfers messages between applications running on hosts connected to the network.

In the TCP/IP world, the Internet at a very high level of abstraction can be seen as a collection of various nets and subnets that communicate to transfer messages between applications. These constituent subnets in the TCP/IP Internet have characteristics such as network delay, loss percentage, and throughput. A NAM communicator is specialized to model the behaviour of these subnets at a high level of abstraction. A glimpse of a network of communicators and hosts is shown in Fig.1.
A NAM communicator uses standard distributions modifiable by the user to generate random numbers for parameters such as network delay. For instance, a communicator using normal distribution with a mean $m$ and variance $\sigma$ would delay 97.7% of the messages by delays between $m-2\sigma$ and $m+2\sigma$. In this context, the usage of normally distributed random numbers for network delay may be seen as modeling the scenario where most packets are delayed by an average delay, only a few packets are delivered very late, and likewise a few packets are delivered very early.
The user can also set the loss threshold of a communicator to discard the packets delayed beyond a certain limit. One may see this as modifying the reliability levels of a subnet. A large loss threshold means a low loss percentage and hence a relatively reliable behaviour. In the TCP/IP world there are protocols that implement reliability, such as TCP, and protocols that provide an unreliable but relatively fast delivery such as UDP. In our model the parameters of a message must be set appropriately, to obtain such services.

In another instance, the behaviour of an overloaded sub-network with few resources and heavy traffic may be modelled by a communicator attached to a normally-distributed-random-number generator initialized with a high mean delay and a low loss threshold. The loss percentage as well as the average delay will be high. Alternatively, one may consider using distributions such as Bernoulli distribution, Beta distribution, or Binomial distribution to model non-deterministic parameters such as packet loss, packet corruption, and network connectivity. [7].

Figure 2 depicts the packet delay/loss details of a specific communicator. The packets delayed beyond 7 time units by this communicator are in-effect lost and are not sent further into the network.
Figure 2: The NAM window shows delay and loss distribution for a communicator.

1.1 Motivation and Objective

Unlike traditional engineering disciplines such as mechanical or electrical engineering, systems engineering heavily relies on informal documentation. Such informal documentation is necessary and may be informative. Nonetheless, being informal it involves ambiguity, incompleteness and sometimes inconsistency. Properly constructed mathematical models are consistent, avoid unintended ambiguity, and are complete at a specific level of abstraction. One may be able to prove certain properties of the design mathematically. Further, in contrast to informal documentation, the
mathematical models based on the *ASM paradigm* [4], are executable and may be used to explore and test the design. One may validate the models and generate test suites for conformance testing of implementations. Such mathematical models are based on a given set of informal requirements. While constructing such models, one is able to resolve ambiguity, inconsistency, and separate concerns. Gradually, the informal description gives rise to an executable mathematical model or to a hierarchy of such models [11].

In this project we study one such mathematical model: the Abstract Communication Model (ACM) for distributed systems [2]. We investigate the ways one might modify the behaviour of the communicating entities called *communicators* (Section 3.2.1) so that they closely model the behaviour of subnets in a TCP/IP network. The resulting specialization of the ACM is called “Network Abstract Machine” (NAM). We implement the NAM and construct a GUI to enable user interaction for control, inspection, and reasoning during simulations.

There is a need for a high-level model of the communication networks to enable communication between various entities in a high-level model of a message-based distributed system. Traditionally, high-level models of distributed message-based systems model communication by a miraculous transfer of messages from one entity to another. Such a situation may be ameliorated by constructing a high-level model of communication networks that is general and customizable to model the communication component of the system under consideration. Projects involving network-oriented applications require a configurable, executable, and yet high-level network model for the purposes of testing their internal logic as well as their interaction with the network. This need is fulfilled by a general high-level model of message-based distributed systems in [2] that we know as “Abstract Communication Model”. It models non-deterministic
characteristics of communication networks successfully. In particular, the Abstract Communication Model models concurrency and parallelism, non-deterministic failure of nodes, delay and uncertainty in message delivery.

One easy solution is to have various agents representing network applications communicate directly and have the network behaviour sort of integrated into the application, but this is not effective. The ACM provides a clear separation of the network from the application, and allows one to deal with two different aspects involved in a distributed message-based system, in a logical manner. One may configure the two components separately and test their interaction.

The Abstract Communication Model is not implementable as such due to its highly non-deterministic nature, which preserves the generality of its applicability. In this project we employ the probabilistic framework to substitute non-determinism partially [2]. We implement the resulting specialization that we call the “Network Abstract Machine” (NAM) and create a graphical user interface for it to enable user interaction, inspection, control and reasoning during simulations.

1.2 Project Report Organization

This project is organized as follows. Chapter 1 serves as an introduction to the work in this project and describes the motivation for this study. Chapter 2, Abstract State Machine, consists of a brief introduction to the ASM paradigm that is sufficient for the purposes of this project. For elaborate details on the ASM paradigm, the reader is referred to [4]. Chapter 3, Abstract Communication Model, consists of a detailed discussion of the ACM, which is followed by a discussion of Probabilistic Modeling. Chapter 4, Interaction of ACM with the UPnP (Universal Plug and Play) model, details the interoperability of a specialization of ACM with the high-level UPnP abstract state
machine model [6]. It also contains an introduction to the UPnP protocol and an introduction to the UPnP Abstract State Machine that is sufficient for our purposes. Chapter 5, Software Architecture, outlines issues concerning implementation and contains diagrams which describe the system architecture. We conclude with Chapter 6 which summarizes the ideas expressed in this project. Appendix A contains the end-user documentation. It also demonstrates the usage of the model through an example client.
Chapter 2. Abstract State Machines

The Abstract Communication Model (ACM) is based on Abstract State Machine (ASM) paradigm. In this chapter we give a brief introduction to ASM concepts. The definitions recalled here should be sufficient for the purposes of this project. For a rigorous treatment of the topic, the reader is referred to the original literature on the theory of Abstract State Machines [7], [4].

2.1 Basic Abstract State Machine

The ASM method is a systems engineering method that enables high-level modeling at a level of abstraction required by the application domain. It enables seamless transition during various phases of the systems development lifecycle, from requirements capture to their implementation. Abstract State Machines simulate arbitrary algorithms in a step-for-step manner and there is a substantial experimental confirmation [4], [8], as well as theoretical confirmation [13], [10] of this thesis.

A basic ASM is a single agent machine that consists of a program and an abstract state. The notion of an ASM state is similar to the notion of a mathematical structure [11]. An ASM signature \( \Sigma \) is a finite collection of function names and relation names. These functions are called basic functions. Relations are similar to functions, except that they always map to either \textit{true} or \textit{false}. Every ASM signature is assumed to contain static constants \textit{true}, \textit{false}, and \textit{undef}. A state \( U \) for the signature \( \Sigma \) consists of a non-empty set \( X \), which is the superuniverse of \( U \), together with the interpretations of the function names of \( \Sigma \). If \( f \) is an \( n \)-ary function name of \( \Sigma \) then its interpretation in state \( U \) is denoted by \( f^U \), which is a function from \( X^n \) into \( X \). If \( c \) is a nullary static function or constant in \( \Sigma \) then its interpretation \( c^U \) is an element of \( X \). The superuniverse \( X \) of the
state $U$ is denoted by $|U|$ and is also called the base set of a state. Similarly, an $n$-ary relation $r$ is a function from $X^n$ to $\{true, false\}$. The default value for dynamic functions is $undef$, and the default value for relations is $false$.

An ASM program consists of transition rules, which are mainly of three types. These transition rules transform abstract states. We have conditional rules of the form:

$$\text{if } Condition \text{ then } Updates.$$ 

$Condition$ is an arbitrary predicate logic formula whose interpretation evaluates to $true$ or $false$. $Updates$ is a finite set of assignments of the form 

$$f(t_1, \ldots, t_n) := t,$$

where $f$ is an $n$-ary function, and $t_1, \ldots, t_n, t$ are terms [13]. First $t_1, \ldots, t_n, t$ are evaluated to their values $v_1, \ldots, v_n, v$ and then the value of $f(v_1, \ldots, v_n)$ is updated to $v$.

The pair $(f, o)$ where $f$ is a function and $o$ an object of the right kind, defines a location in a state. The location-value pairs $(loc, v)$ are called updates and represent the basic units of state change. To fire this update in a state, replace the current content of $loc$ with $v$.

2.1.1 Parallelism

At a high-level of abstraction, it might be desirable to abstract from sequentiality. Until now, a single-agent ASM did a bounded amount of work in every step. The agent changed the ASM state from $S_1$, to $S_2$, to $S_3$,... In this section we will introduce agents that perform a substantial amount of work in a single step. A step may involve numerous parallelism. Such work, in principle may be executed by several auxiliary agents executing in parallel. Nevertheless on a natural level of abstraction of an algorithm, such work is accomplished by a single agent, and those auxiliary agents are invisible. In this regard, the $forall$ construct enables one to abstract from the order of execution where it is irrelevant.
forall \( x \) with \( \varphi(x) \)
\[ R(x) \]
The rule \( R \) is executed for every \( x \) that satisfies the condition \( \varphi \), typically \( x \) will have some free occurrences in \( R \) which are bound by the quantifier.

### 2.1.2 Non-determinism

Similarly, non-determinism is often required to model algorithms at a high-level of abstraction. The basic model is extended with the `choose` construct to allow a convenient representation of non-determinism when details are irrelevant.

```
choose x with \( \varphi(x) \)
\[ R(x) \]
```
The meaning of such a rule is to execute the rule \( R \) for any \( x \) that satisfies the property \( \varphi \). It may be used in particular to represent environmental forces that are not necessarily algorithmic.

We often express the range of the quantifiers by usual set-notation or by a mixture of set and property notation. For instance, let \( X \) stand for a set:

```
choose x \in X with \( \varphi(x) \)
\[ R(x) \]
```

### 2.1.3 Classification of functions

In support of principles of modularization, information hiding, data abstraction and separation of concerns, the ASM method exploits the following distinction among the types of functions and locations. An ASM \( M \) may have static functions that never change during a run of \( M \). Static functions are defined by the initial state of the ASM, and handling of such functions is clearly separated from the description of system dynamics. Then there are dynamic functions. Static 0-ary functions and dynamic 0-ary functions can be thought of as constants and variables of programming respectively. Dynamic
functions, for instance, can be thought of as generalization of array variables or hash tables.

In general, ASM runs may be affected by the environment. The environment manifests itself through basic functions. These functions which are updated by the environment are called *external functions*. A typical external function is the input provided by the user. Basic non-external functions are called *internal functions*.

External functions that are updatable by the environment only and not by the rules of ASM M, are called *monitored functions*. They appear in the update sets produced by M but not on the left-hand side, rather as values on the right-hand side. To describe a combination of internal and external control of functions, we have *interaction* or *shared functions*. Such functions are updatable by the rules of ASM M as well by the environment or, in general, by other agents in a multi-agent system. Typically a protocol is required to guarantee the consistency of updates. Internal functions that are updatable by the rules of ASM M only, are called *controlled functions*. Such functions appear on the left-hand side of at least one update produced by the rules of M. These functions are not updatable by the environment and constitute the internally controlled part of the dynamic state of M.

### 2.2 Distributed Abstract State Machine

In this section, we consider multi-agent computations. We do not suppose that the agents are deterministic or do a bounded amount of work in each step. We introduce a special function *Self*, which is interpreted differently by different agents. An agent a interprets *Self* as a, that is the agent itself. From a global point of view, agents are elements of a dynamic universe AGENTS. As for single-agent ASMs, a Distributed Abstract State Machine (DASM), D has an initial state and a program that defines its
behaviour. The initial state of $D$ is the union of the initial states of all the ASMs that constitute $D$. Similarly, the program of $D$ is an indexed set PROGRAM of the modules that define the behaviour of constituent ASM agents.

The agents in the dynamic universe AGENTS perform their computation steps concurrently. Further, we classify the agents that make a move in a computation step as active agents. This is essentially a generalization; the original definition can be seen as a special case where all the agents are active. The new definition may be convenient when the initial state specifies all agents and their programs, and these agents are activated and deactivated during DASM evolution [13]. For the “Abstract Communication Model”, the activation and deactivation of communicators (Section 3.2.1) is a convenient mechanism to model network connectivity. Two or more agents may do contradictory updates in a step; such conflicts are resolved through the definition of partially ordered runs [13]. For a detailed mathematical foundation for Distributed Abstract State Machines, and treatment of issues related to conflict resolution, the reader is referred to [13].
Chapter 3. Abstract Communication Model

After a brief introduction to "Abstract State Machines" (ASMs) in the Chapter 2, we are ready for a detailed treatment of the "Abstract Communication Model" (ACM) [2]. Distributed and concurrent systems consist of two logically separate parts: the application logic and communication medium for the interaction of distributed components. While modeling such systems, it is desirable to separate these concerns for accuracy, modularization, correctness and clarity of the resulting model. Such separation of concerns is natural and enables one to customize each component as required and study their interaction. The ACM models the communication medium for the interaction of distributed components and may be specialized to suit the needs of the domain under consideration. In general, the ACM models the common part of all message-based distributed communication systems. The Abstract Communication Model was initially developed to specify particular network architecture namely the Universal Plug and Play architecture, but it is general, in the sense that it can be specialized to model any distributed system at the required level of abstraction, and verify the correctness of the specifications. In Chapter 4 we present an example of a model of a distributed system, which uses a specialization of the ACM for modeling a communication network.

In this chapter, we introduce the reader to AsmL which is used to define the executable model of the ACM. It is based on the ASM paradigm. Then we study detailed executable specifications for the ACM.
3.1 Abstract State Machines and AsmL

In order to deploy ASMs in the industrial environment, we require an industrial-strength language. AsmL (ASM Language) is one such language that is developed at Microsoft Research. It is fully integrated with the Microsoft .NET platform (Section 5.1.3). The Abstract Communication Model is defined using AsmL. In this section, we focus only on those aspects of AsmL that are actually used in this report, for a full description of AsmL features the reader are referred to [13]. We assume that the reader is familiar with basic ASM theory described in Chapter 2. We explain how fundamental ASM concepts of states and updates are reflected in AsmL. We then define the semantics for the core constructs of AsmL, through an example. Then, we introduce additional functionality into the modeling framework, which enables us to simulate distributed ASMs. Introduction of this additional functionality is required, since the current version of AsmL lacks support for true concurrency.

Let us first consider a small example that introduces the reader to some salient features of AsmL and how it stands apart from other programming languages. Spec 1 describes an AsmL program to sort a list of numbers in non-decreasing order:

\[
\begin{align*}
\text{var } & \text{ } A = [3, 10, 5, 7, 1] \\
\text{indices } & = \{0, 1, 2, 3, 4\} \\
\text{Main()} & \\
\text{step until } & \text{ fixpoint} \\
\text{choose } & \text{ } i \text{ in indices, } j \text{ in indices} \\
\text{where } & \text{ } i < j \text{ and } A(i) > A(j) \\
\text{A(i)} & := A(j) \\
\text{A(j)} & := A(i) \\
\text{step} & \\
\text{WriteLine(A)} & \text{ } \text{ } \text{// prints [1, 3, 5, 7, 10]} \\
\end{align*}
\]

Spec 1: In-place generic sorting.
These executable specifications use an abstract state machine for in-place sorting. The machine performs sequential steps that swap the values of A whose elements are denoted by indices i and j such that i is less than j and the values A(i) and A(j) are out of order. It runs until no further updates are possible, that is, until the sequence is in order. As a final step, it prints the sorted sequence. The state of the machine at each step is entirely characterized by the value of the sequence A in that step.

The specification is minimal in the following ways.

- The choose expression does not say how the two indices are selected, only that whatever indices are chosen must be distinct indices of out-of-order elements. Hence, many sorting algorithms, including quicksort and bubble sort would be consistent with what we have specified.

- Our example does not specify how the swap operation happens. The values of the variables change as an atomic transaction. This leaves each implementation to decide how to perform the sequential swap, for instance, with an intervening copy to a temporary location.

### 3.1.1 States and Updates

The state of an AsmL model is given by the values of the state vocabulary symbols that occur in the model program. Each vocabulary symbol is a static function (constant) or a dynamic function (variable) (Section 2.1.3). All vocabulary symbols are strongly typed. Variables are marked with the keyword var, their values may change from state to state, whereas constants maintain their initial values. AsmL has a rich type system containing type constructs for sequences, maps, sets, etc.
Spec 2 describes an Asml program derived from [2]; there is a fixed enumerated type Airport whose elements are airports ARN, CPH, and SEA. In ASM terms, Airport is a static universe where each element is a static nullary function. Airline is a class; semantically it is a dynamic universe that consists, in a given state of the Airline objects created so far, name is a unary function from Airline to String. Flights is a dynamic unary function from Airline to Flight tables. DomesticAirline is a subset of Airline. The global variable myAirline is either of type Airline or has a special null value (indicated by question mark) that is also its initial value.

```plaintext
enum Airport
    ARN
    CPH
    SEA

type Flights = Set of (Airport, Airport)

class Airline
    name as String
    var flights as Flights

class DomesticAirline extends Airline
    var myAirline as Airline? = null
```

Spec 2: Airport example, derived from [2].

We view a state as a kind of memory. A location of a state is either a nullary variable $f$ or a pair $(f, o)$ consisting of a unary dynamic function $f$ and an object $o$ of the right type. We say that a location $f$ or $(f, o)$ is variable if $f$ is dynamic. The value or content of a location $f$ or $(f, o)$ in a given state $A$ is the value of $f$ or $f(o)$ in $A$ denoted by $f^A$ or $f^A(o)$ respectively. An update is a pair $l \rightarrow a$ where $l$ is a location and $a$ is a value of type appropriate for $l$. An update is trivial in a state $A$ if $a$ is the content of $l$ in state $A$. An ASM program consists of rules that spawn such updates.
The updates set produced in a step must be consistent. An update set $U$ is consistent if, for each location $i$ in $U$, if there are two updates of the form $i \to a$, $i \to b$ then $a = b$.

### 3.1.2 Simulation of Agents in AsmL

Abstract Communication Model is essentially a Distributed ASM (Section 2.2). Ideally, a distributed runtime environment is suitable for the execution of distributed abstract state machines. The current distribution of AsmL (Section 3.1) does not have runtime support for true concurrency, this is a work in progress. We simulate concurrent behaviour by the means of interleaving as explained below.

Our DASM consists of agents which are viewed as objects of the class `Agent`. An agent $a$ has a function `program`, that defines its behaviour in every step. Also, an agent $a$ has a `mailbox`, and a method `InsertMessage` that is used by other agents to send messages to $a$. The agent $a$ also has a method `isActive`, that determines whether $a$ will make a move in a step of the DASM.

```java
public class Agent
    
  public typedef as String = "Agent"
  public primitive var mailbox as Mailbox = new Mailbox()
  public primitive var adjacentAgents as Set of Agent = ()
  public primitive var nHop as Map of Agent to Agent = ()
    // Hoping table of this agent. Initially empty. It is built using
    // Bellman-Ford shortest path algorithm, and updated everytime the
    // topology changes

  public abstract program()
    // A function describing the behaviour of this agent at every
    // computational step.

  public isActive() as Boolean
    return true
    // By default isActive returns true. It may be modified to model
    // non-deterministically. Subnets may appear or disappear

  Spec 3: Agents in NAM.

Spec 3: Agents in NAM.
```

17
The set \texttt{adjacentAgents} is a set of agents that are directly connected to this agent. An agent can directly insert messages into the mailboxes of other agents, only if they are adjacent to this agent. The keyword \texttt{primitive} directs the compiler to make the corresponding field accessible outside the AsmL domain, for instance, one may like to have access to the information in \texttt{adjacentAgents} to render the topology graphically with Visual C#. The field \texttt{nextHop} is the routing table maintained by an agent. It is dynamic and is updated as the topology changes. \texttt{nextHop} contains a map, which ordains the best choice for forwarding a message, given the destination address. It is constructed and updated using Bellman-Ford shortest path algorithm [14]. We have a class \texttt{Message}, objects instantiated from which, are the messages that flow in the system. A message may be lost, duplicated, become corrupt, or delivered out-of-order. In TCP/IP networks an additional protocol layering at the transport level provides reliable and in-order delivery of messages. In our system, messages that are delivered using the TCP protocol may be delayed but are never lost or corrupted. Then we have the class \texttt{Mailbox}, which has the functionality to collect messages for its agent and transform them as required, so that they are ready to be forwarded to the next receiver in a step. The objects of this class and their fields need not be accessed outside the AsmL assembly, hence the keyword \texttt{internal}.

\begin{verbatim}
RunAgents()
    forall a in ChooseSubset({a|a in agents where a IsActive()})
        a.program()
\end{verbatim}

\textbf{Spec 4: Simulation of agents in NAM.}

For simulation purpose, the model has a global function \texttt{RunAgents()}. By firing \texttt{RunAgents()} we perform a single step of the top-level system. Thus, at each global step of the system, we make a non-deterministic choice; some, none or all the active agents may perform a step. Alternatively, one may use probability distributions to choose
among agents (see Section 3.3 for more details) to make a move. A case may arise when two or more agents do contradictory things, for example, if there is a shared memory then one agent may write 7, while the other writes 10. In such cases an exception may be raised, or choose_subset may be rewritten to avoid all such possible clashes.

### 3.2 The Model

After a brief but sufficient introduction to AsmL, agents and their simulation in AsmL, we are ready for a detailed treatment of the abstract communication model for distributed systems. High-level modeling of distributed message-based systems requires a high-level model of a communication network, which is flexible and adaptable to the problem domain. At best, there must be a clear separation between the application logic and the communication logic, to enable modular design and a separation of concerns. Such a separation is natural and makes the model amenable to modular design and analysis, it enables the user to concentrate on either component separately, tune the parameters, modify behavior and study the interaction between the components. The Abstract Communication Model describes a general way to create a high-level model of the communication network. It may be specialized in various ways to obtain the required behaviour of the communication network component.

Initially, a model was developed to provide a high-level communication media for the UPnP project [11]. This model was found to be a specialization of the “Abstract Communication Model” that is so general that every distributed system that involves a communication network gives rise to a specialization of ACM [2]. In this section, we discuss various components of the abstract communication model; we will discuss the interaction of a specialization of the ACM with the UPnP high-level model in Chapter 4. For several other examples, the reader is referred to [2]. We also discuss the possible
techniques for refining the existing ACM towards a more realistic executable model.

Figure 3 shows an instance of the Abstract Communication Model.

![Diagram of Abstract Communication Model](image)

**Figure 3:** An instance of Abstract Communication Model [2].

### 3.2.1 Communicator

The Abstract Communication Model introduces a special category of agents called **communicators**. Each communicator abstractly represents a multitude of routers in a TCP/IP network [2]. The abstraction of TCP/IP Internet into such subnets may be found in [15]. Intuitively, different communicators represent disjoint subnets as shown in Figure 3, but no such restriction is imposed and one may freely choose the scope and the size of the objects that are to be modelled using a communicator. The communicators transfer messages between applications connected to them. Although this model was obtained by an abstraction from the TCP/IP networks, it is general and is used to deal with very different networks.

The Abstract Communication Model is highly non-deterministic and is intended to be so in order to preserve the generality of its applicability. Consider Spec 5 which defines a **communicator**: 

20
class Communicator extends Agent

override typeOf as String = "Communicator"

override myView as ComUI = new ComUI(me)
  //myView contains the user interface for this Communicator,
  //a pointer to the assl back-end is passed to the user interface
  //through the keyword me

abstract readyToDeliver(message as Set of Message) as Set of Message

override Program()
  forall msg in me.readyToDeliver(me.mailbox)
    remove msg from me.mailbox
    //delete the message from my mailbox, because it is being
    //forwarded.
  forall msg in me.resolveMessage(msg)
    //Consider all resolved messages. A multicast message with two
    //co-opted receivers may be resolved into two messages each with:
    //one recipient among these.
    let a = me.Recipient(msg)
    if a = null then
      //If no immediate recipient is found, which will be one among the
      //adjacent agents
      a.insertMessage(msg)
      //Forward the message

Spec 5: Communicator in NAM

One has to provide some implementation of the construct readyToDeliver in order to make the model executable. In communication networks, message delivery depends on several factors such as network connectivity, network load that effect network delays, and network throughput. Clearly, these factors depend on the system state. For instance increased network load increases the average network delay. One may be able to model such parameters using probability distributions that depend on the system state (see Section 3.3 for further details).

3.2.2 Application

public class Application extends Agent
override typeOf as String = "Application"
override myView as AppUI = new AppUI(me)
  //myView contains the user interface for this application,
var myCommunicator as Communicator? = null

//override the method that this application connects to

override program()
    for all msg to me:ReadyToDeliver(me.mailbox)
        remove msg from me.mailbox
        //delete the message
    for all m in me.ResolveMessage(msg)
        //consider all resolved messages
        if myCommunicator = null then
            //if this application is connected to a network
            myCommunicator.insertMessage(m)
        //forward the message

Spec 6: Application in NAM.

This is the simplest of all the classes because it encapsulates the general functionality of an application connected to a network. The specification is minimal and is intended to be extended by the application logic of a client using the abstract communication model for modeling the network component (See Chapter 4 for an example, and Figure 8 that depicts this concept). At the very least, a basic application attached to the network is expected to pass on the messages in it’s outbox to the communicator it is connected to.
3.2.3 Message

```java
public class Message
    public typeof as String = "Message"

    primitive var myView as MsgUI = new MsgUI(me)
        //myView contains the user interface for this Message.
        //A pointer to the back-end is passed to the user interface
        //through the keyword me.
```
A message, in general, is multicast to all its receivers. If the message is unicast then the set receivers is a singleton set. If the message is broadcast then the set receivers contains addresses of all the applications on the network. Both are special forms of multicast [15].

### 3.3 Probabilistic Modeling

In this section we discuss possible probabilistic techniques to extend the generic ACM to a more realistic high-level network model. Non-deterministic transitions have traditionally been used to model distributed and concurrent systems. Non-determinism provides an abstraction over scheduling, network delays, failures and randomization. However, a probabilistic model can capture these sources of non-determinism more precisely and enable statistical analysis, simulations and reasoning.

A number of factors, such as processor scheduling and network delays, failures, and explicit randomization, generally result in nondeterministic execution in concurrent and distributed systems. A well known consequence of such non-determinism is an exponential number of possible interactions which in turn makes it difficult to reason rigorously about concurrent systems. For example, it is infeasible to use techniques such as model checking to verify any large-scale distributed systems. In fact, some distributed systems may not even have a finite state model.
A non-deterministic rule may potentially produce several distinct updates. Consider the Spec 8.

\[
\text{ReadyToDeliver(messages as Set of Message) as Set of Message}
\]
\[
\text{return ChooseSubset(\{m | m in messages\})}
\]

Spec 8: Expression of non-determinism in AsmL

There are \(2^{\text{messages}}\) possible subsets of which a subset is chosen randomly by the chooseSubset construct. Such non-determinism abstracts over message scheduling, message delays, message loss and network throughput due to any particular implementation. This also abstracts over the amount of work done in a single logical time-step (because we cannot predict the number of messages that are chosen for delivery). However, it is possible to abstract over such details more accurately, by introducing probabilities. We now choose a subset with some probability. A message may have a higher probability of being chosen for delivery than some other message due to factors such as longer wait time in a network. Such a probability distribution models first-in-first-out (FIFO) scheduling. One might consider using other probability distributions to model various queuing strategies.

A large class of concurrent systems may become amenable to a rigorous analysis if we are able to quantify some of the probabilities of transitions. For example, network delays can be represented by variables from probabilistic distributions that depend on some function of the system state. Similarly, network connectivity, failure rates, message corruption, and loss, may also have a probabilistic behavior. A probabilistic model can capture the statistical regularities in such systems and enable us to make probabilistic guarantees about its behavior.
3.3.1 Probability and ASMs

As described (Section 2.1, 3.1.1), an ASM has a program that produces a consistent set of updates in every step. Modeling systems at a particular level of abstraction necessitates ignoring details irrelevant to that level of abstraction. Hence, it may be desirable to forgo complex implementation of program rules that produce precise updates. One may simply not have enough information to implement such rules, or such an attempt might introduce complexity that is undesirable at the specific level of abstraction. One way to model the system at the desired level of abstraction is by making non-deterministic updates in a state, which is precisely modelled by making a randomized choice through \texttt{choose} and \texttt{chooseSubset} constructs of AsmL. A better way to deal with such non-determinism would be to associate probabilities with all the possible updates which are amenable to probabilistic modeling.

Further, in a DASM run, a step may involve executing a subset of agents that are chosen on the basis of some property of the distributed system. However, it may be the case that implementing the particular policy that accurately models this property of the system, is not desirable or is impossible at the concerned level of abstraction. Hence, one may resort to making a non-deterministic choice of a subset of agents that run in a step. We say that these agents are \textit{active} in a particular step. Again, in some cases it might be possible to associate probability distributions with the activeness pattern of agents. In the Abstract Communication Model, the activeness pattern of communicators effectively models the network connectivity.

One may consider using probability distributions such as Bernoulli distribution, Beta distribution, Binomial distribution for modeling parameters such as network delay, network connectivity, message loss, and message corruption [7].
3.3.2 Probabilistic Communicator Model

Consider the modified specs for a communicator shown in Spec 9. The choice of the messages to deliver in a step is not completely non-deterministic; rather it is determined by the statistical end-point of the communicator.

```java
public class Communicator extends Agent
override typeOf as String = "Communicator"
override myView as COMU1 = new COMU1(self)
    //myview contains the Communicator user interface

ReadyToDeliver(messages as Set of Message) as Set of Message
    //returns a subset of messages for forwarding in the current step.
step
    forall msg in messages where msg.hopDelay = 0 and not msg.lost
        msg.hopDelay := myStats.nextDelay()
        //hopDelay for a message is simply the time it will be
        //enqueued before being forwarded to the next receiver
step
    forall msg in messages where not msg.lost
        msg.hopDelay = msg.hopDelay - 1
step
    return (msg | msg in messages and m.hopDelay <= 0 and not msg.lost)

override program()
step
    foreach m in ReadyToDeliver(me.mailbox)
    step
        if(m.hopDelay > myStats.lossThreshold) then
            m.lost := true
            //THIS BIT INDICATES that the message is lost forever.
            //THIS message will not be further considered for delivery
        else
            m.deliver()
        step
        m.delivered := true
        m.myview.updateView()
step
    mybase.program()
```

Spec 9: Modified definition of Communicator that uses a probability distribution.

A Communicator delays all the undelivered messages in its mailbox by the next semi-random delay returned by its statistical-end (that is customizable by the user, through a well-defined interface, see Figure 5), checks whether it is delayed beyond a
given threshold, and then accordingly decides whether to drop the message or schedule it for delivery at some point of time in the future. Note that we do not make a completely non-deterministic choice on which messages to deliver in some step. Rather, we use a probability distribution to model parameters such as packet delay in the network. Instead of setting a loss threshold, one may consider using distributions such as Beta distribution, Binomial distribution and Bernoulli distribution for modeling parameters such as packet loss and network connectivity [7].

The class ComStats provides random number generators for normally distributed random numbers and other relevant distributions. As in real networks, most of the packets are delayed by a reasonable delay — only a few packets are delayed very long, and, similarly, only a few of them are delivered very quickly. Such a delaying behaviour is modelled by normally distributed random numbers with adjustable parameters. For instance, the behaviour of an overloaded sub-network with few resources and heavy traffic may be modelled by a communicator hooked to a normally distributed random number generator initialized with a high mean delay and a low loss threshold. So, the loss percentage as well as the average delay will be high.

Then there are methods that abstract the behaviour of IP networks into analytical models. Solving the model numerically yields performance metrics that are close to those of real networks. Yong et al., [16] describe the behaviour of networks carrying responsive and unresponsive (with respect to congestion) traffic through (coupled) ordinary differential equations. Such numerical models might replace the random number generators of our statistical end-point ComStats, if one wants a communicator to represent a large IP network with a certain topology of routers and a certain number of TCP and UDP data flows. Figure 5 depicts the user interface for customizing the statistical end-point for any communicator in a network.
3.4 Comparing ACM with Related Tools

In this section we describe the complementary relation (derived from [2]) of executable specifications using the ASM paradigm, with other paradigms and tools including Architecture Description Languages (ADLs), Network Simulation tools, and Coordination Languages.

3.4.1 Architecture Description Languages

ADLs are used to specify a system’s conceptual architecture rather than its actual implementation. ADLs are necessary to bridge the gap between informal “boxes and lines” diagrams and programming languages which are deemed too low-level for application design activities [2]. From that perspective AsmL is an ADL, and abstract
communication model can be seen as an ADL artifact. However, there is an important difference. Specifications described with AsmL are executable unlike those described with ADLs. They may be used to explore the model state space, generate test cases, do conformance checking [17] to provide behavioural interfaces for components, and so on.

The issue of ADLs versus ASMs is addressed in [18].

3.4.2 Network Simulation Tools

Among the simulation tools for network research, ns-2 developed by the VINT project [19] is the most widely used. The network simulator ns-2 is a heavy weight, discrete event simulator targeted for accurate and detailed simulations. It offers elaborate support for simulation of TCP, routing, and multicast protocols over wired and wireless networks. When compared with ns-2, ACM is a light-weight, plug-in, high-level model for the communication component in a distributed application. Unlike ns-2, ACM enables analysis at a semantic level, which is desirable during early stages of system design and analysis. ns-2 is unsuitable for such purposes; it could be used in later stages for accurate and elaborate simulations of the system implemented. Hence, the two approaches target complementary aspects of computer network modeling [2].

3.4.3 Coordination Languages

Coordination Languages, for example Linda provide a means to program distributed systems. A coordination language provides a communication model for that purpose. For instance, in Linda, the coordination is achieved by the means of tuples that live in a (conceptually) shared space [2]. However, the abstraction level of this communication component is fixed, unlike the ACM. Furthermore, a coordination language operates on the top of a conventional programming language such as C,
whereas AsmL is a single language. As a result, the task of analysis becomes much easier in our case [2].
Chapter 4. Interaction of ACM with the UPnP Model

Until now we have focused on the behaviour of the ACM, now we describe its interaction with other high-level models that require a network in between. In this section we summarize the basic characteristics of the Universal Plug and Play (UPnP) model. We focus on those aspects of UPnP model that serve as a prelude for discussing its interoperability with a specialization of ACM (rather than on the internal behaviour of the UPnP components). The integration of Device Plug and Play (PnP) with operating systems enables an easy setup, configuration, and addition of peripherals to a personal computer. Universal Plug and Play (UPnP) extends this simplicity to include the entire network, enabling discovery and control of devices, including networked devices and services, such as network-attached printers, Internet gateways, and consumer electronics equipment. In addition, UPnP is designed to support zero-configuration, seamless proximity networking, and automatic discovery for a breadth of device categories from a wide range of vendors. For a detailed discussion on the UPnP model, the reader is referred to [6], [20]. We discuss the interaction of the UPnP model with a specialization of the ACM.

4.1 Universal Plug and Play

Proximity, zero-configuration networks imply that a device can dynamically join a network, obtain an IP address, convey its capabilities, and learn about the presence and capabilities of other devices. These devices can subsequently communicate with each other directly through peer to peer networking. Such devices include intelligent appliances, wireless devices, and PCs of all form factors.
The scope of UPnP is large enough to encompass many existing as well as new
scenarios including home automation, printing and imaging, audio/video entertainment,
kitchen appliances, automobile networks, and proximity networks in public venues [20].

UPnP uses standard TCP/IP and web technologies, to achieve a seamless
integration with the existing networks. Using standardized protocols allows UPnP to
benefit from a wealth of experience and knowledge and makes interoperability an
inherent feature. UPnP is distributed, open network architecture, defined by the
protocols used. It is independent of any particular operating system, programming
language, or physical medium (just like the Internet).

In this chapter, we deal with the interoperability aspects rather than details of
individual components. Components operate concurrently and interact with each other
by exchanging messages over the communication network. They use actuators and
sensors to interact with the external world, which is the environment into which the whole
system is embedded. The ASM paradigm allows us to combine synchronous as well as
asynchronous computations. The component models themselves are parallel
compositions of synchronously operating ASMs. The system as a whole is a composition
of asynchronously operating components, called agents.

### 4.2 Components in a UPnP Network

A UPnP network consists of devices, services, and control points. They are
described briefly in this section (see Figure 6).
Figure 6: This picture depicts the relationship between devices, services, and control points in a UPnP network. A control point calls upon the services that are offered by UPnP enabled devices, which themselves can have control points using services offered by other devices. Derived from [20].
4.2.1 Devices

A UPnP device consists of services and nested devices. For instance, a VCR device may consist of a tape transport service, a tuner service, and a clock service. A TV/VCR combo device would consist of a nested device as well [20].

UPnP devices are associated with specific sets of services and embedded devices. For instance, services within a VCR will be different from those within a printer. In a UPnP network, different working groups standardize on the different set of services that a particular device type will provide. All of this information is captured in an XML device description document that the device must host. In addition to the set of services, the device description also lists the properties (such as device name and icons) associated with the device.

4.2.2 Services

The smallest unit of control in a UPnP network is a service. A service exposes actions and models its state with state variables. For instance, a clock service could be modelled as having a state variable current_time which defines the state of the clock and two actions set_time and get_time which allow you to control the service. Similar to the device description, this information is part of an XML service description standardized by the UPnP forum [20]. A pointer (URL) to these service descriptions is contained within the device description document. Devices may contain multiple services.

A service in a UPnP device consists of a state table, a control server, and an event server. The state table models the state of the service through state variables and updates them when the state changes. The control server receives action requests (such as set_time), executes them, updates the state table and returns responses. The event server publishes events to interested subscribers anytime the state of the service
changes. For instance, the fire alarm service would send an event to interested
subscribers when its state changes to “ringing”.

4.2.3 Control Points

A control point in a UPnP network is a controller capable of discovering and
controlling other devices. After discovery, a control point could:

- Retrieve the device description and get a list of associated services.
- Retrieve service descriptions for interesting services.
- Invoke actions to control the service.
- Subscribe to the service’s event source. Anytime the state of the service
  changes, the event server will send an event to the control point.

Ideally, devices incorporate control point functionality (and vice-versa) to enable
ture peer-to-peer networking [20].

4.3 The Protocol

UPnP is a layered protocol built on the top of TCP/IP by combining various
protocols such as DHCP, SSDP, SOAP, and GENA. It supports dynamic configuration of
any number of devices offering various kinds of services requested by the control points.
To perform control tasks, a control point needs to know about the devices that are
online, the services that are advertised and the duration for which they are available.
The UPnP protocol describes six basic phases to achieve this (Derived from [20]).

- Addressing: Each device must have a Dynamic Host Configuration
  Protocol (DHCP) client and search for a DHCP server when the device is
  first connected to the network. If a DHCP server is available, the device
  must use the IP address assigned to it. If no DHCP server is available,
the device must use Auto IP to get an address. In brief, Auto IP defines how a device intelligently chooses an IP address from a set of reserved private addresses, and is able to move easily between managed and unmanaged networks. A device may implement higher layer protocols outside of UPnP that use friendly names for devices. In these cases, it becomes necessary to resolve friendly host (device) names to IP addresses. Domain Name Services (DNS) resolve names to IP addresses. A device that requires or uses this functionality may include a DNS client and may support dynamic DNS registration for its own name to address mapping.

- **Discovery:** Once devices are attached to the network and addressed appropriately, discovery can take place. Discovery is handled by SSDP. When a device is added to the network, SSDP allows that device to advertise its services to control points on the network. The fundamental exchange is a discovery message containing a few, essential specifics about the device or one of its services, for example its type, identifier, and a pointer to its XML device description document.

- **Description:** The next step in UPnP networking is description. After a control point has discovered a device, the control point still knows very little about the device. For the control point to learn more about the device and its capabilities, or to interact with the device, the control point must retrieve the device's description from the URL provided by the device in the discovery message. Devices may contain other logical devices and services. The UPnP description for a device is expressed in XML and includes vendor-specific, manufacturer information including the model.
name and number, serial number, manufacturer name, URLs to vendor-specific web sites, and so forth. The description also includes a list of any embedded devices or services, as well as URLs for control, eventing, and presentation.

- **Control:** After a control point has retrieved a description of the device, the control point has the essentials for device control. To learn more about the service, a control point must retrieve a detailed UPnP description for each service. The description for a service is also expressed in XML and includes a list of the commands, or actions, the service responds to, and parameters or arguments, for each action. The description for a service also includes a list of variables; these variables model the state of the service at run time, and are described in terms of their data type, range, and event characteristics. To control a device, a control point sends an action request to a device’s service. To do this, a control point sends a suitable control message to the control URL for the service (provided in the device description). Control messages are also expressed in XML using SOAP. In response to the control message, the service returns action specific values or fault codes.

- **Eventing:** A UPnP description for a service includes a list of actions the service responds to and a list of variables that model the state of the service at run time. The service publishes updates when these variables change, and a control point may subscribe to receive this information. The service publishes updates by sending event messages. Event messages contain the names of one of more state variables and the current value of those variables. These messages are also expressed in
XML and formatted using GENA. A special initial event message is sent when a control point first subscribes; this event message contains the names and values for all evented variables and allows the subscriber to initialize its model of the state of the service. To support multiple control points, all subscribers are sent all event messages, subscribers receive event messages for all evented variables, and event messages are sent irrespective of the reason for state variable change (in response to an action request or due to a state change).

- **Presentation:** If a device has a URL for presentation, then the control point can retrieve a page from this URL, load the page into a browser, and depending on the capabilities of the page allow a user to control the device and/or view device status. The degree to which each of these can be accomplished depends on the specific capabilities of the presentation page and device.

### 4.4 UPnP Abstract State Machine

A high-level executable DASM model of the UPnP architecture, based on the informal requirement specification [20], could be found in [6] and [21]. The UPnP model consists of an ensemble of devices and control points with the communication network in-between them. The ASM paradigm allows us to combine *synchronous* as well as *asynchronous* execution models. The component models themselves are parallel compositions of *synchronously* operating ASMs, while the system as a whole is a composition of *asynchronously* operating ASMs which we know as *agents*. 

39
4.4.1 Device Model

This model describes, how a device behaves in a UPnP compliant way. The network connectivity of a device depends on external factors, and may therefore change anytime without prior notice. The device model specification is a synchronous parallel composition of a number of rules operating in parallel; where different rules describe different protocol phases.

```java
class ControlPoint extends Application
class Device extends Application
    abstract isConnected() as Boolean
    Program()
        if me.isConnected() then
            me.RunAddressing()
            me.RunDiscovery()
            me.RunDescription()
            me.RunControl()
            me.RunEventing()
            me.RunPresentation()
```

Spec 10: Device and Control Point in AsmL. Adapted from [6]

Here, we focus only on the control phase given by RunControl, which involves direct interaction with the communicators. Every device offers a set of services, which may be called by any of the control points through messages. These devices reply to control points, and the reply may include a response message which informs the caller whether the call succeeded or not and may include some return value.

4.4.2 Interaction with ACM

Control point and devices interact through exchange of messages over a TCP/IP network where network characteristics, such as bandwidth, dimension, and reliability are left unspecified. In general, the communication is considered to be neither predictable nor reliable, that is, messages arrive with varying delays, they may be out-of-order, and some messages may never arrive. Devices may appear or disappear without notice, and
there is no guarantee that a requested service will be available in a given state, or become so in the future. In particular, an available service may not remain available until a certain control task using this service has been completed.

Communicators enable communication between a device and a control point. Every device that is connected to the network is associated with a communicator. A device sends messages by inserting them into the mailbox of the communicator to which it connects. Consider Spec 11 for a device's Control phase.

```plaintext
---
type Service
class Device
  services as Set of Service
  var communicator as Communicator
  abstract Call(s as Service, msg as Message) as Message
  abstract IsServiceRequest(m as Message, s as Service) as Boolean
  abstract RunServices()
  var address as Address? = null
  RunControl()
    if address ≠ null then
      me.RunServices()
      choose msg in me.mailbox
      remove msg from me.mailbox
      choose s in me.services where
        me.IsServiceRequest(msg, s)
        let reply = me.Call(s, msg)
        communicator.InsertMessage(reply)
---

Spec 11: Description of Control phase for a device. Adapted from [6]

The control phase of the protocol is executed only when the device obtains a valid address after a successful execution of the addressing phase. When active the control phase handles service requests one at a time and runs the services. Note that the RunServices() is executed in parallel with the handling of the service requests. The number of services that are allowed to run simultaneously depends on the definition of the device described by a particular implementation of the method RunServices().

Actions and events in the external world as represented by the environment into which the system is embedded affect the system under consideration in various ways.
For instance, the transport of messages over the communication network is subject to varying delays and some messages may never arrive. Also, the system configuration may itself change unpredictably with the devices going offline without notice. Therefore, an additional GUI to allow for user-controlled interaction is introduced (see the Figure 7). However, this does not implies that the environment behaves in a completely unpredictable fashion; rather one may formulate reasonable integrity constraints on external actions and events where possible.

Figure 7: Interaction of the UPnP model with the Abstract Communication Model [11].
Figure 8: Class diagram showing inheritance from ACM classes [11].
Chapter 5. Software Architecture

The implementation utilizes the language interoperability features of Microsoft Visual Studio .NET platform. The mathematical model is implemented using AsmL, a Graphical User Interface (GUI), statistical and visual analysis/reporting components are created using Visual C# .NET. The interoperability between these components is made possible through the common language runtime that is integrated with the .NET platform. In this section we briefly discuss the salient features of AsmL .NET and Visual C# .NET and how they are suited for our purposes. Then we describe the detailed component architecture in both the domains. The Application Programming Interface (API) and an example client using a specialization of ACM can be found in Appendix A.

5.1 Choice of Language and Platform:

In this section we briefly describe features of programming languages used for this project, in particular we discuss their suitability for our purposes. We have used the integrated development environment of Visual Studio .NET 2003 for this project, and we discuss it briefly in this section.

5.1.1 AsmL .NET

AsmL is a software specification language based on abstract state machines [13]. It is used for creating human-readable, machine-executable models of a system in a way that is minimal and complete with respect to any desired level of abstraction. Specifications written in AsmL are called executable specifications.

AsmL has constructs to support parallelism such as forall. Such constructs are required for modeling at a high-level, where it may be desirable to
abstract from sequentiality or the order of execution of statements. Consider Spec 12 as an example.

```
RunAgents()
    forall a in ChooseSubset({a | a in agents where a.IsActive()})
    a.program()
```

Spec 12: Constructs to abstract from sequentiality in AsmL.

Clearly, the actual order in which the chosen set of agents make their moves is irrelevant and this is well-expressed through the `forall` construct. Here, concurrency is simulated through interleaving during the actual execution. This example also illustrates the expression of intended ambiguity through the `ChooseSubset` construct. At a given level of abstraction, one might not be interested in the policy used to select a subset of active agents that execute their programs and this is well-expressed through `ChooseSubset` construct that selects on a random basis. One might notice that there is a possibility of conflicts in such a scenario, when two or more agents do contradictory updates. This might be avoided by using some policy to select a subset of agents, in an implementation of the `ChooseSubset` construct, at a level of abstraction lower than this level in the hierarchy. AsmL constructs such as `ChooseSubset` and `Choose` may also be used for describing non-deterministic environmental factors that do not follow any algorithmic pattern.

The current version, AsmL 2 (AsmL for Microsoft .NET), is embedded into Microsoft Word and Microsoft Visual Studio.NET. It uses XML and Word for literate specifications. It is fully interoperable with other .NET languages. AsmL generates .NET assemblies, which can be executed from the command line, linked with other .NET assemblies, or packaged as COM components.
Due to the above mentioned features AsmL .NET is an excellent choice for
describing the semantics of the “Abstract Communication Model” and to specialize it as required.

AsmL .NET is not a visual language and, hence, does not have a GUI builder like other visual .NET languages. Fortunately, it can be used in close collaboration with Visual C# .NET that has an excellent GUI development support.

5.1.2 Visual C# .NET

Visual C# .NET is the newest among the .NET compliant languages. Like AsmL and many other .NET compliant languages, it is based on object-oriented paradigm. It has a visual interface and intuitive new language constructs that greatly simplify the development process. It is fully integrated with AsmL .NET. One may use reflection to
discover properties of AsmL components to manipulate and project them as desired.

Hence, the dynamic link library that is constructed consists of two assemblies,

- The high-level primary functionality of the Abstract Communication Model is defined using AsmL .NET
- The graphical user interface, statistical and visual analysis/reporting components are created in Visual C# .NET

These components work in close collaboration to achieve the desired functionality.

5.1.3 Development Platform: Visual Studio .NET

Developer tools, such as Microsoft Visual Studio® .NET 2003 provide an integrated development environment (IDE) for maximizing developer productivity with
the .NET Framework. The .NET Framework is composed of the common language runtime and a unified set of class libraries. Let us briefly discuss these two components.

5.1.3.1 Common Language Runtime

The common language runtime (CLR) is responsible for run-time services such as language integration, security enforcement, memory, process, and thread management. In addition, the CLR has a role at development time when features such as life-cycle management, strong type naming, cross-language exception handling, and dynamic binding reduce the amount of code that a developer must write to turn business logic into a reusable component.

CLR enables cross-language development and deployment, which means cross-language inheritance, cross-language debugging and exception handling. The code in any .NET compliant language is first converted into Intermediate Language (IL) as shown in Figure 10, which is then processed by the Intermediate Assembler (ILASM). The CLR supports a Common Type System (CTS), which is intended to support a wide range of languages. Common Language Specification (CLS) defines a subset of Common Type System, which all language compilers targeting the CLR must adhere to.

All compilers under .NET will generate Intermediate Language no matter what language is used to develop an application. In fact, CLR will not be aware of the language used to develop an application. All language compilers will generate a uniform, common language called Intermediate Language as seen in Figure 9. The original code is not compiled to machine native code but to an intermediate form which does not contain any hardware/software specific information and hence is potentially platform independent. This code is not executable as such; CLR converts IL into executable machine specific code. This is the role played by CLR along with many other functions
Such as garbage collection. For more details, the reader is referred to [22].

5.1.3.2 Class Libraries

Base classes provide standard functionality such as input/output, string manipulation, security management, network communication, thread management, text management, and user interface design features.

The Base Class Libraries (BCL) serve as fundamental building blocks for any application developed in the .NET framework, be it an ASP.NET application, a Windows Forms application, or a Web Service. The BCL generally serves as the main point of interaction with the runtime; they consist of libraries such as ADO.NET, XML libraries, ASP.NET, and Windows Forms libraries. The ADO.NET classes enable developers to interact with data accessed in the form of XML through the OLE DB, ODBC, Oracle, and SQL Server interfaces. XML libraries enable XML manipulation, searching, and translations. The ASP.NET classes support the development of Web-based applications and Web services. The Windows Forms classes support the development of desktop-based smart client applications.
Together, the class libraries provide a common, consistent development interface across all languages supported by the .NET Framework. For more details, the reader is referred to [22].

5.2 The System Model

Figure 10 depicts the constituent components of the *Network Abstract Machine (NAM)* package. The NAM package consists of the AsmL model component, the GUI component, and the statistics and reporting component. Figures 11 and 12 show a glimpse of the GUI and the statistics and reporting components, respectively.

![Diagram of NAM package](image)

*Figure 10: The NAM package contains three components that work together in close collaboration.*
Figure 11: The NAM window shows a multicast from Port 5 to Port 2 and Port 10 (green packets). There is also a unicast from Port 12 to Port 8 (blue packets).
5.2.1 Class Model

Model is a global class that is not instantiated; instead it enforces static variables that describe current system state and record all instantiated objects in the application that is using the network model.

```java
public class Model {
    primitive shared var agents as Set of Agent = {}
    primitive shared var edges as Set of (Agent, Agent) = {}
    primitive shared var coms as Set of Communicator = {}
    primitive shared var ports as Set of Application = {}
    primitive shared var stepnumber = 1
    primitive shared var trffs as Set of Traffic = {}
    //trffs can be thought of as a set of pairs: (sender, set of receivers)
    //each such traffic flow that is set up by the user is attained through
    //parameters such as inter-message delay and number of messages to deliver
}
```
shared playStep()
step
stepNumber := stepNumber + 1
currTime := currTime + timeincr
step
forall f in trffs
f.playStep()
//create message and forward to the network
step
forall a in agents
a.program()

Spec A: Description of the global class model in NAM.

The playStep() method when executed furthers the system to the next state. The granularity of the timescale is controlled by the value of timeincr. For instance, if timeincr is 0.2 ms, then two messages scheduled to be transmitted at 15.7 ms and 15.8 ms on the timescale, will be sent in the same step when currtime has advanced to 15.8 ms. On the other hand, a timeincr of 0.1 will result in a finer time granularity, this will result in the transmission of these messages in two different but sequential steps when the currtime reaches 15.7 ms and 15.8 ms respectively. The purpose of defining the various global variables will become clear after perusing the Routing Module section (Section 5.2.3).

5.2.2 Agent

public class Agent
public primitive shared typeOf as string = "Agent"
primitive var adjacent_agents as Set of Agent = {}
primitive var non_adj_agents as Set of Agent = {}
primitive var myView as Agent := new Agent(me)
primitive guid as GUID = Model.getIcn()(
primitive var mailbox as Mailbox := new Mailbox(me)
primitive var next_hop as Map of Agent to Agent = {}
primitive var dist as Map of Agent to Integer = {} //dist records the distance of this agent from all the other agents
//In the network, it is subsequently used by the Bellman Ford algorithm
//to create a routing table for shortest path distances
private primitive var throughput as Double = 0.0

52
primitive var delay as Integer = 0
primitive var desiredInput as Double = 0.50
private primitive var received as Integer = 0
    //number of packets that are forwarded until now, initially 0
private primitive var delivered as Integer = 0
    //number of packets that are received until now, initially 0

public getIDC() as GUID
    return me.guid
    //every agent that is instantiated receives a globally unique identifier
    //that is returned by this function.

Agent()
    //This code is executed when an agent is instantiated
    me non_adj_agents := Model.agents
    forall a in Model.agents
        add me to a non_adj_agents

Spec 14: Details of agent class in NAM.

Variables adjacent_agents and non_adj_agents represent sets of Agent that are adjacent, respectively non-adjacent to this instance of Agent. An object of class Agent is disconnected when it is created and hence it is added to non_adj_agents set of all existing agents. Its own set, non_adj_agents is initialized to all existing agents, whereas the set, adjacent_agents is kept empty (see the constructor Agent()). There are methods connect() and disconnect() that internally manage the connections of an instance of Agent.

The map variable nextHop contains the routing table for this instance of Agent. It is constructed dynamically whenever the topology of the network changes. The map variable dist records the distance from this agent to every other agent in the system. Section 5.2.3 describes the implementation of the Bellman-Ford shortest path algorithm, which computes the routing table for all agents in the system.

5.2.3 The Routing Module

public class Model
private shared InitBellmanFord()
  step
  nodes := {}
edges := {}
  forall v in agents
    step foreach a in agents
      v.nextHop := (u->v.defaultRoute())
      //For any destination, initialize the next hop to the
      //default route from this agent, which could be any agent
      //among its adjacent agents.
v.dist := ->
  step
  forall c in agents where c.isActive()
    add c to nodes
    step foreach cl in c.adjacentAgents where cl.isActive()
      add (c, cl) to edges

Spec 15: Routing module initialization routine in NAM.

As is evident, following the notation of graph theory, the network is a set of nodes
that represent active agents in the system and the connections between the agents are
modelled through the map variable edges.

public class NAM
  private shared weight(n1 as Agent, n2 as Agent) as Integer
    if ((c1, c2) in edges) then
      return 1
    else
      return 100
  private shared InitDist(n as Agent)
    forall v in nodes - {n}
      v.dist := {n->100}
      //Distance of a node from itself is 0 and infinite(100)
      //from the rest.

shared convergeBellmanFord()
  step
    InitBellmanFord()
    step
    foreach n in nodes
      step InitDist(n)
    step until fixpoint
      step foreach (u, v) in edges
        if (u.dist(n) + v.dist(n) + Weight(u, v)) then
\[
\begin{align*}
    u.\text{dist}(n) &:= v.\text{dist}(n) + \text{weight}(u, v) \\
    u.\text{nextHop}(n) &:= v \\
\end{align*}
\]

step
\[
    t.\text{nextHop}(t) := t
\]

\textit{Spec 16: Algorithm for creating the routing table in NAM.}

\subsection*{5.2.4 Communicator}

The fundamental details for a communicator are described in Sections 3.2.1 and 3.3.2. In this section, we describe the user interface and how it is bound with the mathematical model of the communicator defined in AsmL. Figure 13 shows a communicator form that can be used by the user to customize the behaviour of a communicator.
Figure 13. The window shows the connectivity and routes for Communicator 1 in the topology visible in the background. The Connections tab is made slightly transparent to enable the user to inspect the topology while modifying it, at the same time.
Figure 14: The window shows statistical details for a communicator.

5.2.5 Application

The fundamental details for an application are in section 3.2.2. In this section we describe the user interface for an application and how it interacts with the application back-end defined with AsmL. Figures 15, 16, and 17 describe the user interface for an application, which can be used to customize the behaviour of an application.
Figure 15: The window shows a port form with tabs: mailbox, connections, and quantifiers that serve as an interface to modify various aspects of an application to achieve the desired behaviour. One may also control the speed of the message flow through the speed bar on bottom left corner.
5.2.6 Traffic/Message User Interface

Figure 16: The window shows the traffic form associated with the packet traffic between ports 12 and 8 and controls to modify the protocol for sending messages and the inter-message delays.
Figure 17: The window shows the mailbox tab of an application form, a message form describing details for the message in the mailbox, and a protocol form to customize the protocol for transfer. For each message the message form lists the intermediate receivers, the source, and the final destinations of the message.

5.2.7 Class Diagram

Figure 18 describes inheritance and aggregation relationship between prominent classes that constitute the system.
Agent, Communicator, and Port (alias for Application) are AsmL classes that are discussed in the previous sections (Section 3.1.2, 3.2.1, and 3.2.2). In order to facilitate user interaction with objects of these classes, user interfaces are developed in Visual C#. The class GenericUI utilizes the facility of reflection provided by the .NET framework class library. It is defined as follows:

```csharp
namespace GUI
{
    /// <summary>
    /// GenericUI uses reflection to discover the properties of
    /// an assembly; It is the parent class of all UI classes.
    /// </summary>
    public class GenericUI
    {
        private static System.Windows.Forms.Form PropertiesForm;
        private object myAgent;
        //The back-end assembly for this UI class
```
The GenericUI uses reflection to discover the properties of myAgent and then initializes propertiesForm accordingly. For instance, ComUI that is a subclass of GenericUI, discovers the attributes of Communicator and generates a form to manipulate the properties of Communicator. Figure 20 shows the main NAM window with an imported topology defined previously. It provides a facility to resume simulations when desired.
Figure 19: The main NAM window. There is a multicast from Port 5 to Ports 2 and 10 (green packets). There is also a unicast from Port 12 to Port 8 (blue packets).

5.2.8 Use Cases

Figure 20 shows use cases for interaction between a client using the ACM and the client applications that communicate via the ACM.
Figure 20: Use cases.
Chapter 6. Conclusion

In this project, we have studied the Abstract State Machine paradigm in detail and considered the possibility of associating probabilities with state transitions. The Abstract Communication Model [2] is a generic model and is not implementable as such. The specialization of the Abstract Communication Model that we have implemented is geared towards modeling communication networks more faithfully. The main idea is to employ probabilistic techniques to model the reality that is amenable to such techniques. We have implemented the Abstract Communication Model after introducing probabilistic transitions and constructed an interactive graphical user interface to facilitate user interaction for inspection, reasoning, and control during simulations. We have explored the interaction of the ACM with the high-level UPnP model. We have developed an external probability distribution generator and we have used it to associate probabilities with ACM transitions.

The specialization of the Abstract Communication Model that we have implemented is more deterministic in the sense that it uses an external probability distribution generator to schedule messages to be forwarded in various steps and to select among the messages to be forwarded in a step. This specialization modifies the behaviour of the original non-deterministic communicator to use probabilities, to model the behaviour of a subnet more faithfully and is customizable by the user to achieve the desired behaviour.

One may investigate into the possibility of introducing the probability into the ASM framework and integrating it with AsmL.
Appendix A

End User Documentation:

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoadTopology(xml file)</td>
<td>Activates a pre-defined topology</td>
</tr>
<tr>
<td>Communicator()</td>
<td>Instantiates a new Communicator and returns its unique ID which is an Integer</td>
</tr>
<tr>
<td>Port()</td>
<td>Instantiates a new Port and returns its unique ID which is an Integer</td>
</tr>
<tr>
<td>Message()</td>
<td>Instantiates a new Message and returns its unique ID which is an Integer</td>
</tr>
<tr>
<td>InsertMessage(port_id, msg_id)</td>
<td>Insert the message with ID equal to msg_id into the mailbox of the port with ID equals port_id, the message is passed to the network for delivery in the next computational step.</td>
</tr>
<tr>
<td>Traffic(sender, receivers, interval)</td>
<td>Creates a steady data flow from sender to a number of receivers, where the interval between sending is also a parameter to the function.</td>
</tr>
<tr>
<td>NamForm()</td>
<td>Creates a new instance of NAM window with a well-defined graphical user interface for interaction with the network.</td>
</tr>
<tr>
<td>PortIds()</td>
<td>Returns a set of integer IDs of all the ports present in the topology.</td>
</tr>
<tr>
<td>GetMsgs()</td>
<td>Usually called by an application attached to a port to obtain all the messages in the port's mailbox and empty the mailbox.</td>
</tr>
</tbody>
</table>

Table 1: NAM Application Programming Interface.

An Example Client in C#:

```csharp
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using Microsoft.VisualBasic;
using NAM;

namespace NAMTest
{
    // This namespace describes a client using ACR
    {
        private System.ComponentModel.IContainer components;

        public NAMTest()
        {
            // Required for Windows Form Designer support
            InitializeComponent();
        }
    }
}
```
protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        if (components != null)
        {
            components.Dispose();
        }
    }
    base.Dispose(disposing);
}

#region Windows Form Designer generated code

/// <summary>
/// The main entry point for the application.
/// </summary>
///<STAThread>
static void Main()
{
    NAM.NamForm f = new NAM.NamForm();
    //Create topology
    f.createTpl();
    //Get port IDs
    ArrayList portIDs = f.getPortIDs();
    //Create a message with `sender` as port 12 and
    //`receiver` as port 8
    NAM.Message m = new NAM.Message(12, "data string");
    m.addReceiver(8);
    //Create traffic
    Flow flow = new Flow(12);
    flow.addReceiver(8);
    flow.enabled = true;
    //Get all messages in a port's mailbox and
    //empty the mailbox
    Microsoft.AAML.Set s = Model.getMessage(12);
    Application.Run(f);
}
#endregion

67
References


