Tangram II Manual

October 27, 2010

Research Team

Professors:

- Edmundo de Souza e Silva (UFRJ) (Tangram-II PI)
- Rosa Maria M. Leao (UFRJ) (Tangram-II co-PI)
- Richard R. Muntz (UCLA)
- William Cheng

Students:

Name	level	period	Institution
Adenilson Raniery S. Pontes	MSc	(1998-1999)	(PUC/BR)
Adriane de Quevedo Cardozo	MSc	(1999-2002)	(UFRJ/BR)
Ana Paula Couto Silva	PhD	(1999-2006)	(UFRJ/BR)
Antonio Augusto de Aragão Rocha	PhD	(2001-)	(UFRJ/BR)
Antonio Mauricio N. Tartarini	MSc	(1998-1999)	(PUC-RJ/BR)
Bernado Calil Machado Netto	MSc	(2001-2004)	(UFRJ/BR)
Bruno Cesar Barbosa Alves	PhD	(2009)	(UFRJ/BR)
Bruno Felisberto Martins Ribeiro	MSc	(2001-2004)	(UFRJ/BR)
Carlos F. de Brito	MSc	(1998-1999)	(UFRJ/BR)
Carolina C. Le Brum de Vielmond	MSc	(2001-2007)	(UFRJ/BR)
Daniel Ratton Figueiredo	MSc	(1998-1999)	(UFRJ/BR)
Daniel Sadoc Menasche	MSc	(2000-2005)	(UFRJ/BR)
David Silva Boechat	MSc	(2004-2007)	(UFRJ/BR)
Denise Jorge de Oliveira	undergrad.	(1998-1999)	(UFRJ/BR)
Edmundo Grune de Souza e Silva	MSc	(2002-2009)	(UFRJ/BR)
Flavio Pimentel Duarte	MSc	(1999-2003)	(UFRJ/BR)
Felipe Mendonça Alcure	undergrad.	(2000-2002)	(UFRJ/BR)
Fernando Jorge Silveira Filho	MSc	(2000-2006)	(UFRJ/BR)
Gaspare Giuliano Elias Bruno	PhD	(2009-)	(UFRJ/BR)
Guilherme de Melo B. Domingues	PhD	(2009-)	(UFRJ/BR)
Guilherme Dutra Gonzaga Jaime	PhD	(2001-)	(UFRJ/BR)
Hugo Hidequi Costa Sato	MSc	(2002-2007)	(UFRJ/BR)
Isabela Barreto Duncan	MSc	(2001-2006)	(UFRJ/BR)
Joao Carlos Guedes	PhD	(1994)	(UFRJ/BR)
Jorge Allyson Azevedo	MSc	(2000-)	(UFRJ/BR)
Kelvin de Freitas Reinhardt	MSc	(1999-2002)	(UFRJ/BR)
Luiz Rogerio G. de Carvalho	MSc	(1997-1998)	(UFRJ/BR)
Magnos Martinello	MSc	(1998-2001)	(UFRJ/BR)
Morganna Carmem Diniz	PhD	(1996-1997)	(UFRJ/BR)
Raphael S. de Moraes	undergrad.	(1998)	(UFRJ/BR)
Sidney Cunha de Lucena	PhD	(1997-2004)	(UFRJ/BR)
Yuguang Wu	PhD	(1994)	(UCLA/USA)

Copyright

[1] LAND: **L**aboratory for Modeling/**An**alysis and **D**evelopment of Computer and Communications Systems.
URL: http://www.land.ufrj.br

[2] UFRJ: Federal University of Rio de Janeiro, Brazil

Preface

TANGRAM-II is an environment for computer and communication system modeling and experimentation, developed for research and educational purposes. It provides a general user interface based on an object oriented paradigm and a variety of solvers to obtain the measures of interest. The environment also includes modules useful for computer network experimentation, and multimedia tools to aid in the modeling process and collaborative work.

The first version of TANGRAM was developed in Prolog [4] and a graphic interface called TGIF was also implemented and became later a full fledge independent general purpose sophisticated drawing tool [7]. From 1993 till 1994, several solvers were implemented, including those for transient analysis and for models with "deterministic events". The development of the TANGRAM-II environment started in 1997 [6].

The tool was completely re-designed and among the new features we mention [20]: a mathematical model generation implemented in C++; new constructs added to the language; new analytical solvers; a new user interface implemented in Java; TGIF constructs to facilitate the interaction with the new tool; an interactive simulator based on TANGRAM's paradigm. Recently, several modules have been implemented for computer network traffic modeling and analysis [34]. Finally, a Whiteboard tool and a voice transmission tool are being incorporated in the environment [9, 10, 27]. As soon as they are integrated with the environment, these tools will facilitate the development of models by groups at different locations.

The architecture of TANGRAM-II tool is shown in Figure 1.

If you want to know more information about our work visit our site at: **www.land.ufrj.br**

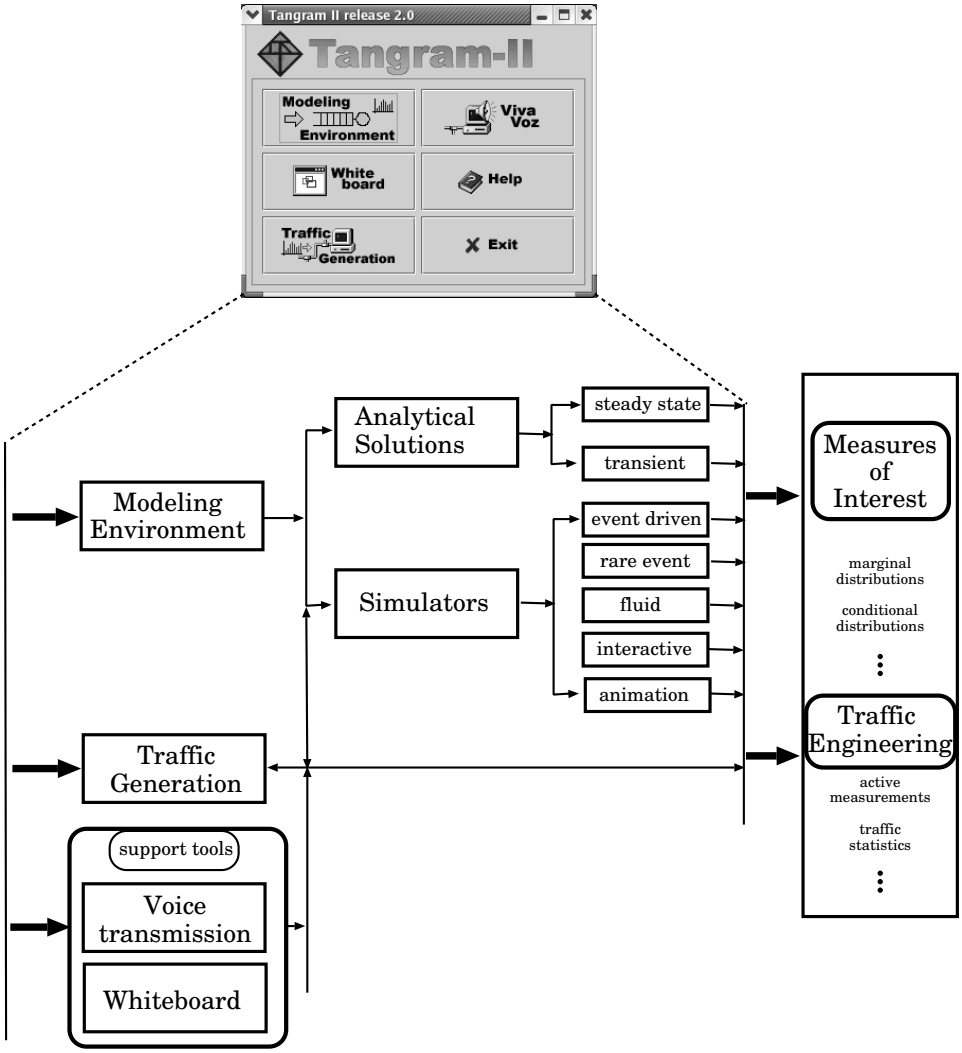


Figure 1: The Overview of the Architecture.

Acknowledgments

Many people contributed to TANGRAM-II, since this project began. Most people worked on several different parts of the tool. Some, though not directly involved with the tool, gave us invaluable advice. We want to say THANKS to everyone. In particular the project students were able to work together as a big family. It was a pleasure to see their enthusiasm. Below we acknowledge the people who contributed to the tool and indicate the main parts they were involved with.

The modeling paradigm was conceived by Dr. Steven Berson, Prof. Edmundo de Souza e Silva and Prof. Richard Muntz in 1989. This paradigm was adopted by the TANGRAM tool developed in 1989 and led by Richard Muntz.

The TANGRAM-II modeling environment was conceived by Prof. Edmundo de Souza e Silva and Prof. Rosa Maria Leão. Prof. Richard Muntz participated as a major consultant and in the discussions that led to the TANGRAM-II project. The current principal investigators are Edmundo de Souza e Silva and Rosa Maria Leão.

The drawing interface adopted by TANGRAM-II is the same as that conceived for TANGRAM (TGIF) and was developed by Dr. William Cheng. Several goodies were included in TGIF to support the needs of TANGRAM-II. Dr. Cheng implemented the modifications in TGIF and participated in the design discussions.

The first version of the mathematical model generator was implemented by Luiz Rogerio G. de Carvalho in 1997. This model was completely re-designed and re-implemented by Daniel Ratton Figueiredo and Carlos F. de Brito. The re-design included many new constructs that were added to the TANGRAM-II language, and reward-based measures. The hash function and AVL tree was implemented by Magnos Martinello.

The analytical solvers were designed/coded by: Yuguang Wu, João Carlos Guedes, Morganna Carmem Diniz, Edmundo de Souza e Silva, Carlos F. de Brito, João Abdalla and Ana Paula Couto da Silva. The simulator was designed by Daniel Ratton Figueiredo, and the FARIMA and FBM modules were implemented by Adenilson Raniery S. Pontes (under the guidance of Prof. Rosangela Coelho).

The module that calculates/plots measures of interest based on state probabilities was implemented by Antonio Mauricio N. Tartarini. The JAVA interfaces were designed and implemented by Kelvin de Freitas Reinhardt, Adriane de Quevedo Cardozo, Flavio Pimentel Duarte and Guilherme D. G. Jaime. They also have been debuggind and including several new features since 2000. The interfaces between the solvers and the JAVA screens was re-designed by Sidney Cunha de Lucena. The module used to visualize and permute matrices was designed and implemented by Fernando Jorge Silveira Filho and Daniel Sadoc Menasche.

The TANGRAM-II simulator (that includes support for rare event simulation) was designed and implemented by Daniel Ratton Figueiredo. Kelvin de Freitas Reinhardt implemented new features to support fluid simulation with the help of Bruno F. Martins Ribeiro. The TANGRAM-II parser has been improved and maintained by Jorge Allyson

Azevedo, Guilherme D. G. Jaime and Flavio Pimentel Duarte. Within the simulator, the module that generates random numbers was rebuilt by Guilherme D. G. Jaime and Fernando Fernando Jorge Silveira Filho. They also have been debuggind and including several new features to the simulator since 2000.

The traffic modeling environment was implemented by Sidney Cunha de Lucena. The traffic generator was implemented by Daniel Ratton Figueiredo, Carlos F. de Brito and Magnos Martinello. Fabiano de Azevedo Portella participated in the implementation of its user interfaces as well as the program for calculating traffic measures from the Tangram-II traffic trace format. Magnos Martinello implemented the traffic generator interface to ATM switches. Antonio A. de Aragão Rocha implemented new traffic generation features and options that were added to the 2.1 version of Tangram-II. Hugo Sato worked on the graphical interfaces for these new features.

The first version of the Whiteboard Tool was designed and implemented by Edmundo de Souza e Silva, William Cheng, Renato Santana, Carlos F. de Brito, Magnos Martinello, Raphael S. de Moraes, Denise Jorge de Oliveira and Flavio Pimentel Duarte. The develop team of the second version of the Whiteboard Tool was: Edmundo de Souza e Silva, William Cheng, Jorge Allyson Azevedo, Milena Scanferla and Daniel Sadoc.

Participated in the implementation of the voice tool, VivaVoz: Daniel Ratton Figueiredo, Flavio Pimentel Duarte. Carolina Vielmond and Edmundo Grune de Souza e Silva have implemented a graphical user interface for VivaVoz its voice mixer, a new trace colector, and are now the tool's current maintainers.

Ana Paula Couto da Silva was responsible for the examples and the TANGRAM-II tutorial, based on the contribution of many people, including those above. Daniel Figueiredo was the project leader till mid 1999, and he still acts as a consultant. Kelvin de Freitas Reinhardt, Adriane de Quevedo Cardozo, and Flavio Pimentel Duarte have been responsible for putting version 1.2-6 together and several subsequent versions. For version 2.1, Carolina Vielmond and Hugo Sato created the autotools scripts that enable the Tangram-II source code to be compiled on several Linux plattaforms.

We would like to thank our colleagues from the ALMADEM and COMIT projects: Prof. Berthier Ribeiro Neto, Dr. Jorge Moreira, Prof. Paulo Aguiar, Prof. Rosangela Coelho, Prof. Jose Augusto Suruagy Monteiro, Prof. Sergio Campos, Prof. Nelson Fonseca, Prof. Leana Golubchik and Prof. Don Towsley for the insightful discussions that led to many enhancements of TANGRAM-II.

We are in debt to Prof. Virgilio Almeida and Mr. Celso Deusdeti Costa from ProTem/CC (CNPq) who believed in the work of our group. CNPq (ProTem, Pronex, Free Software programs) and FAPERJ provided the main grants that supported our Laboratory.

Contents

List of Figures

Chapter 1

Modeling with TANGRAM-II

1.1 Introduction

System modeling and analysis is an important part of the design process of computer and communication systems in which one evaluates the efficacy of the system under consideration and compares different design alternatives. Over the last two decades, several tools have been designed to aid the user in developing performance and dependability models of those systems. Some tools are tailored to specific application domains, such as queuing network models, and availability modeling. Other tools allow specification of general modeling domains such are those based on Petri nets, those based on formal description language, and those that adopt a user interface description language specially developed for the tool. The tools also vary in terms of their user interface, the type of measures that can be obtained, and the analytic and/or simulation techniques that are available to solve the models.

During the development of a modeling tool, many issues must be addressed to facilitate the design process. On one hand, the user interface should be tailored to a particular application domain with which the user is concerned. For instance, if the user is developing an availability model, then the tool allows him/her to specify system components that can fail, interactions between components, repair policies, operational criteria, etc.

Most real system models have large space cardinalities, and so the main issue is how to deal efficiently with such large models. This problem influences both the generation phase of the state transition matrix, and the implementation of the solvers. The identification of special structures in the model is also a desirable feature. The type of model structure often influences the choice of the most effective solution technique.

Yet another issue is related to the interaction between the interface and the solvers. Several measures require special information to be provided by the user. In availability modeling, for instance, the user must specify the conditions in which the system is considered operational. In performability modeling, reward rates must be specified for the states.

If the model to be solved is non-Markovian, then depending on the solution technique used the interface has to provide more information than that required solving Markovian models.

TANGRAM-II was developed at Federal University of Rio de Janeiro (UFRJ), with participation of UCLA/USA and other Universities, for research and educational purposes and deals with several of the issues mentioned above. It combines a sophisticated user interface and new solution techniques for performance and availability analysis.

1.2 Architecture

In the paradigm used in the tool, the system modeled is represented by a collection of objects which interact by sending and receiving messages. The internal state of each object is represented by a set of integer-valued variables.

Events and messages and their associated conditions and actions define the behavior of an object. Events are generated spontaneously by an object, provided that the conditions specified when the object was defined are satisfied. These conditions are Boolean expressions evaluated using the current state of the object. Messages are just an abstraction used to represent the interaction among objects, and are delivered (and reacted to) in zero time. When either an event is executed or a message is received, a set of actions specified by the user is taken with a given probability distribution. As a result of an action taken, the object's state may change and messages may be sent to other objects in the model.A detailed explanation of how messages and events are executed in the Tangram-II simulator is given in section 4.

The tool provides a graphical interface to aid the user when creating a new object type or defining a model. It is based on the public domain TANGRAM Graphic Interface Facility TGIF, developed at UCLA [7].

The current version is implemented in C/C++ and has several solvers for transient and steady state analysis of performance and availability metrics. A robust simulator is also part of the solution methods and supports rare event simulation.

In order to create a new model, the user may employ different types of objects that can be retrieved from a library previously created, or he/she may build new object types from scratch. The objects are parameterized and instances can be declared with actual parameters specified for each instance.

The user starts building the model of a system by instantiating objects from the Object Library using the Modeling Environment module, provided that all types of objects needed are already in the library. Otherwise, new object types are first created and stored in the Object Library. Then, the objects are parsed and data structures are created for each object instance. New object types are defined and specified in terms of a graphical representation and their associated behavior. The interface of the tool displays a template to aid the user in the construction of a new object type.

After the model is built, mathematical description of the model can be generated (e.g a Markov Chain) and an appropriate solution technique can be used to obtain steady state and/or transient measures of interest. If the model is not Markovian or cannot be handled by the analytical solvers, then the simulator can be used to obtain different measures of interest.

Figure 2.2 shows the Modeling Environment Interface. The icons on the left hand side of the figure show the options available to the user:

1. Specification of a model (Model Specification Module);
2. Mathematical model generation (Mathematical Model Module);
3. Analytical solution of the model (Analytical Model Solution Module);
4. Measures-of-interest generation (Measures of Interest Module);
5. Traffic descriptors computation (Traffic Descriptors Module);
6. Simulation (Simulation Module);
7. Hidden Markov Models (HMM Module).

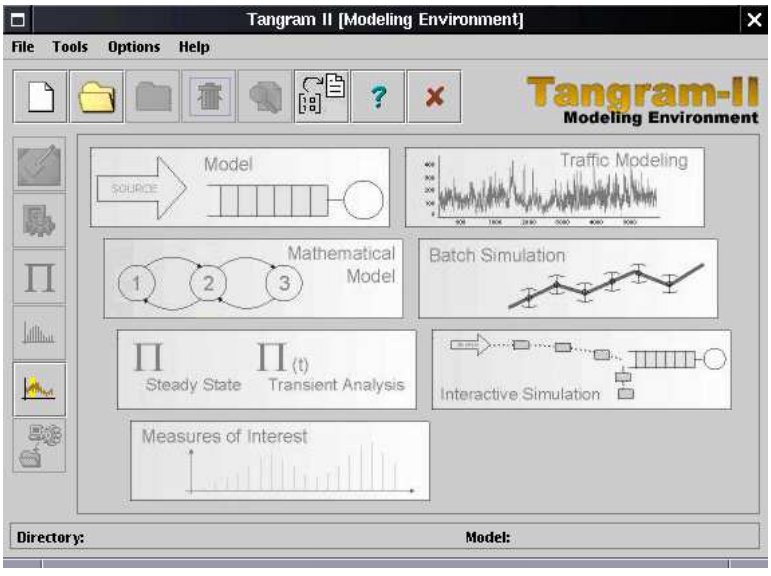


Figure 1.1: The Modeling Environment.

1.3 The Model Specification Module

In this module the objects are created and the model is specified using TGIF [7]; see Figure 1.2.

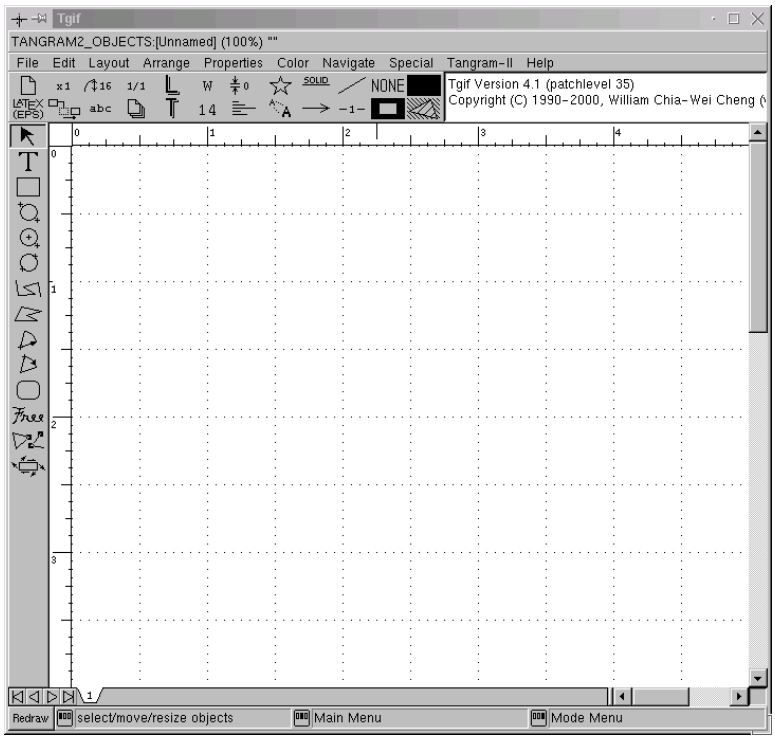


Figure 1.2: TGIF - TANGRAM Graphic Interface Facility

To create a new object type, the tool displays a template as shown in Figure 1.3. The template shows the state variable and all parameter types that can be defined by the user, and displays a set of fields to be filled in by the user indicating the events and messages of the object type and the associated conditions and actions. The object also may have an attribute called "rewards" that can be used to obtain reward measures using appropriate analytical or simulation solution methods. The box in the left-hand corner of the template can be modified to create an icon for the object type being defined.

After all objects are created, it is necessary to interconnect them using *ports*. Through these ports, *messages* are exchanged between objects. All variables are either state variables, constants or parameters. *State variables* are of type integer or integer-valued vector. *Constants* can be of type integer, float, object, or port. All state variables and constants must be initialized before the mathematical model is generated. *Parameters* need be initialized only before the analytical solution (they are not used in simulation).

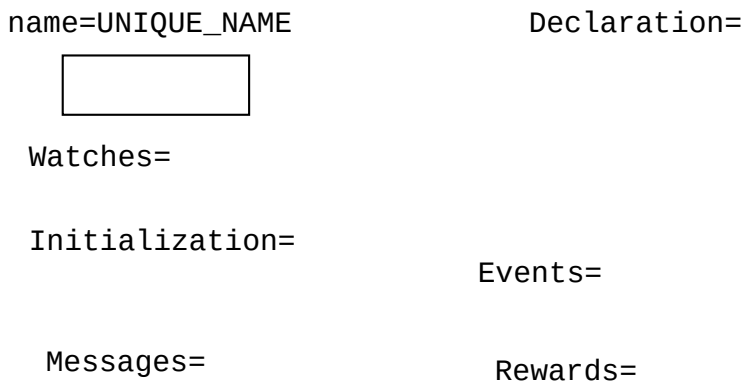


Figure 1.3: Template to define a new object type.

1.4 The Mathematical Model Module

The Mathematical Model Module explores reachable states in the model and calculates the transition rates between any two tangible states. When a state is generated, the generator must determine if it was previously explored, or if it is a new state. TANGRAM-II uses a hashing function to uniquely map states into identifiers, in order to save storage. The technique used identifies a lexicographic order for the state vectors and the state space is divided into sets according to this ordering. Then, a table that stores the number of states in each set is built. From this table, one can uniquely map the state vectors into identifiers with little effort.

The user specifies an initial state for each object. The state transition matrix generator finds a list of all events that can fire from that state. One event is then chosen to fire, and all messages that can be sent after an action is executed for that event, are found. All states with pending messages (yet to be delivered) are vanishing, and are not part of the final state space of the model. The algorithm recursively finds all states (vanishing or tangible) reachable from the current state, and the new tangible states found are inserted in the list of non-explored states (depth-first search).

The data structures generated for the class of non-Markovian models that can be solved by the tool are more complex than those needed for Markovian models, since many chains must be identified and generated as required by the solution method. Roughly, the generator first finds all tangible states assuming that all transitions in the model are exponential. Then, for non-exponential events defined by user, the generator finds one or more Markov chains associated with them. For details of the solution technique see [19].

1.5 Analytical Model Solution Module

Several solution techniques for obtaining steady state and transient measures are currently available in the TANGRAM-II environment. (See [12, 13, 18, 13] and also [45, 23] for references on some of the techniques implemented.) Since the tool was designed mainly to be used for research and education, a few traditional solution methods were implemented, as well as recently-developed algorithms. The tool provides several graphical interfaces to allow the specification of parameter values and to choose the appropriate solution technique. The main analytical solvers implemented in TANGRAM-II are shown in Figure 1.4.

1.6 Measures of Interest Module

In this module, several measures of interest can be evaluated and plotted. These include distributions of functions of state variables, and marginal and conditional probabilities.

1.7 Traffic Descriptors Module

This module allows the user to compute a few first- and second-order statistical measures (traffic descriptors) from a Markov reward model of a traffic source, or from a trace. The descriptors which can be calculated are: mean, variance, peak rate, burstiness, index of dispersion (IDC(t)), autocovariance(t), autocorrelation(t).

1.8 Simulation

A model can be solved by an analytical method or by simulation.

The tool supports traditional discrete-event simulation, which includes batch, interactive, and rare-event simulation.

1.8.1 Traditional Discrete-Event Simulation

1.8.1.1 Batch Simulation Module

In batch simulation the user specifies parameters such as number of runs, and stopping criteria such as simulation time and number of transitions.

1.8.1.2 The Interactive Simulation Module

If the user chooses to perform interactive simulation, the simulator communicates with the graphic interface using sockets. The simulator updates the state variables at each simulation step through the socket API. Then the user can visualize the evolution of the state variables after the execution of a specified number of steps.

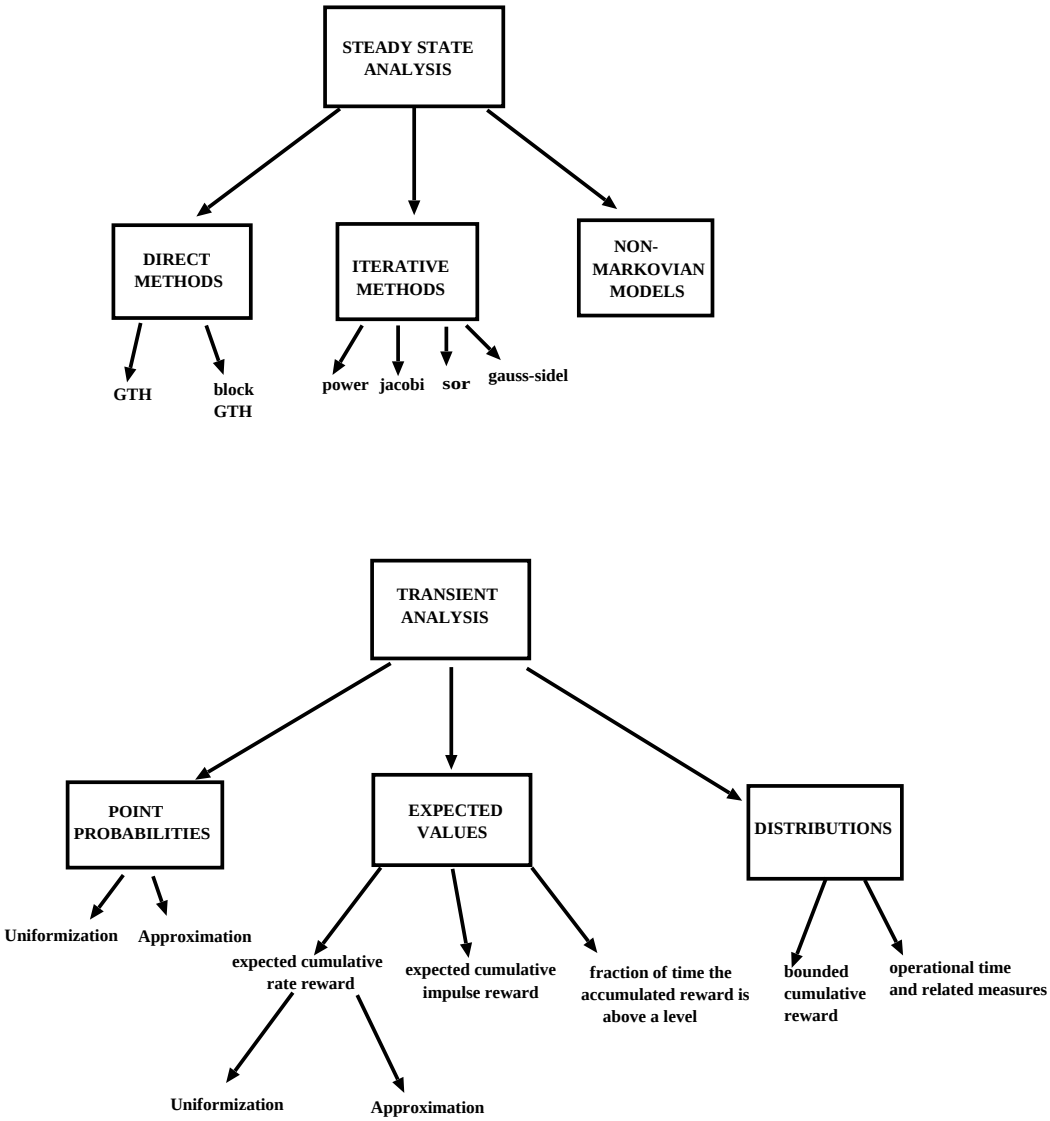


Figure 1.4: The Analytical Solvers.

1.8.2 Rare Event Simulation

Rare event simulation is implemented using the splitting/RESTART technique.

1.9 Hidden Markov Models Module

This module allows the user to work with hidden Markov models (regular or hierarchical), using the various methods and algorithms implemented therein, such as parameters estimation algorithms and forecasting techniques.

Tangram-II's Model Specification Module is used to design the model's chain structure. The HMM Module, then, loads this structure, and allows the user to specify any additional paramenter that the model may need. Once these specifications are done, the HMM model is created, and ready to be worked with.

1.10 Where we Go Next

In this section we presented a brief overview of the TANGRAM-II Modeling Environment. In the next chapter we will show how to use all features available in the Modeling Environment through a simple example.

Chapter 2

Getting Started

2.1 Introduction

The purpose of this chapter is to introduce the user to modeling process with TANGRAM-II using a very simple example. By following this tutorial, the user will be able to:

- Construct a model based on the TANGRAM-II object-oriented paradigm.
- Specify and solve the model, and obtain the measures of interest.

The first example we choose to model is the M/M/1/k queueing system. In this system there is a maximum number of customers that may be stored in a queue. This model has two types of objects: a source of packets, and the queue with its server. Objects of each type will be instantiated to create the system model.

The source object generates packets to the server object from a Poisson distribution with rate λ . The server object has a limited queue, and serves packets in FCFS order with exponentially-distributed service time. If a packet arrives at the server when the queue is full, the packet is dropped.

In the next sections we will present each one of the steps to create/evaluate a model:

- First we create the model with the Model Specification Module. In this step, the objects and the messages exchanged between them are defined.
- Next, we generate the state transition rate matrix. The Mathematical Model Module is used for that.
- Then we solve the model with the Analytical Model Solution Module and evaluate some measures of interest with the Measures of Interest Module.
- Finally, we simulate the model using the Simulation Module.

2.2 Starting TANGRAM-II

Start-TANGRAM-II by typing `tangram2` on the command line. The graphical interface appears as shown in Figure 2.1.

Figure 2.1: The Main Screen

To create and analyze a model, choose "Modeling Environment". The graphical interface appears as shown in Figure 2.2.

2.2.1 Step 1: Creating a Model

To create a model, choose the "Model Specification Module". In this step, we use the TGIF (TANGRAM Graphic Interface Facility), as shown in Figure 2.3.

2.2.1.1 The DOMAIN

First, we must choose the DOMAIN in which we want to work. A DOMAIN stores the types of objects that were created. For instance, we may create a DOMAIN that contains different service stations, each implementing a different queueing discipline.

To choose a DOMAIN, click on `Special` and, then on `DOMAIN` → `Change Domain`. A window with several options appears. Choose `TANGRAM2_OBJECTS`.

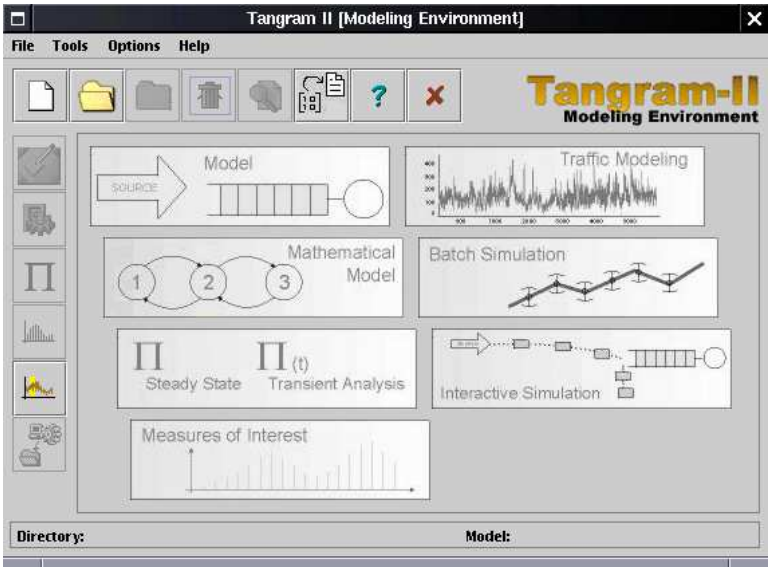


Figure 2.2: The Modeling Environment Module

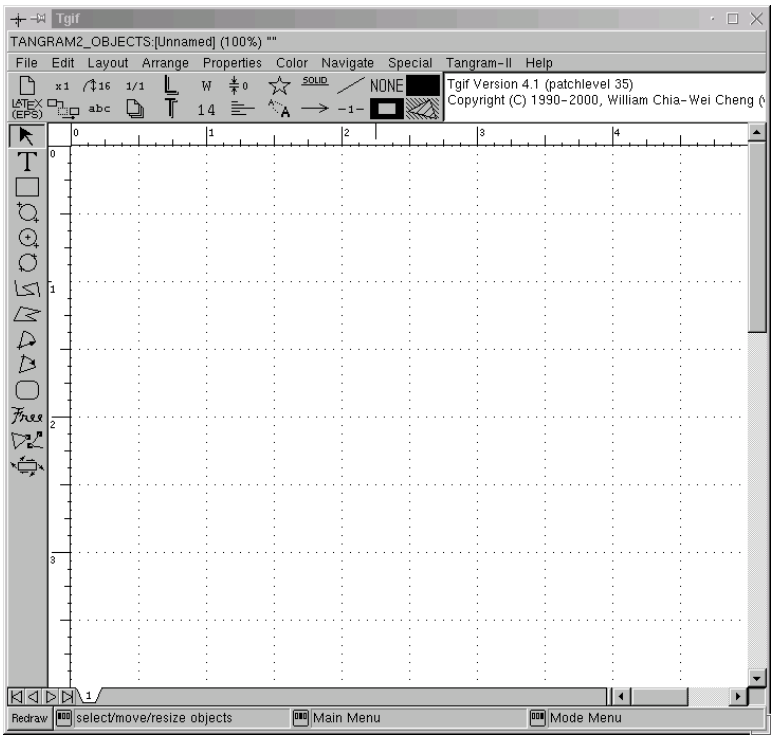


Figure 2.3: TGIF - TANGRAM Graphic Interface Facility

After this, we are able to use objects types stored in the library. If you can't see any Tangram-II objects in the list, and in the messages box of the main Tgif window there is an error like "path can't be created", it means that you need to set up correct Domain paths. One way to do that is by setting them into your `.Xdefaults` config file. In your home directory, open or create the `.Xdefaults` file (plain text) and put these lines at the end of the file:

```
Tgif*MaxDomains:  1
Tgif*DefaultDomain:0
Tgif*DomainPath0: TANGRAM2:/usr/local/Tangram2/Domain/TANGRAM2_OBJECTS:.
```

We can also create a new type of object from an object-template and store it in a new user DOMAIN. To do that, just increment the `MaxDomains` variable we just created inside the `.Xdefaults` file and create a new variable `DomainPath` with the new path and name you want to add as your newly-created user domain.

2.2.1.2 Specifying an object's attributes

To choose the object template click on `Special/Instantiate`. A window with the objects appears. Choose `Obj_Template.sym` and click on the interface. The object template is shown in Figure 2.4.

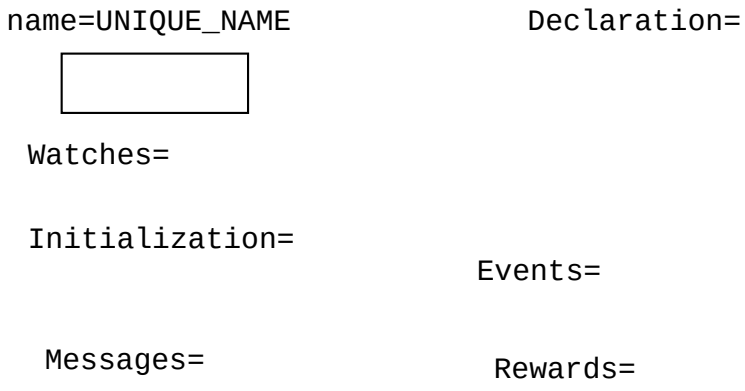


Figure 2.4: Template to define a new object type.

As shown in Figure 2.4, there are six attributes that describe the object: `Declaration`, `Watches`, `Initialization`, `Events`, `Messages`, and `Rewards`.

Declaration is used to specify the pre-defined-types: state variables, constants and parameters. Constants can be of type Integer, Float, Object or Port; parameters can be of type Integer or Float.

1. *State variables* can be of type Integer or Float. In each case, the following "flavors" are allowed:
   (a) *Scalar*. For example if we have a state variable named `Status`, it will be able to have the value 10 if its type is Integer, or 3.6 if its type is Float. Scalar state variables can only have non-negative values. (For the use of Float state variables see §2.3.2.5.)
   (b) *Vector*. The variable is represented by an array of Integer or Float values. NOTE: if we have a Integer vector, the maximum size allowed is 255. A Float vector has a maximum size of 127.
   (c) *Queue*. This type is similar to the Vector type, but the user can specify the desired size. There is no maximum size.
2. *Constants and parameters* can be of type Integer or Float. NOTE: We can represent the Integer and Float types in scientific-notation. E.g. Integer: `1e+10, 1E+05`; Float: `1.0e+10, 1.0e-10, 1.0E+10, 1.0E-10`.
3. *Object*: used to make references to others objects in the model;
4. *Port*: used to define connection-ports between objects. Objects can only communicate (via the message mechanism) if they have ports that are interconnected. NOTE: `port` is a reserved word, so an object's port cannot be simply named "port".

Initialization is used to initialize constants and state variables to numerical values.

Events is used to specify all the events in an object. An event is defined by a name, a distribution type, and its parameters. An event must also have a condition, and at least one action associated with it. Note that one or more messages can be sent during the execution of an action.

Watches lists all declared state variables. If we simulate the model interactively (see §3.1), the value of each watched variable is displayed at each step.

Messages is used to declare the messages that can be received. When a specific message is received, an action associated with that message is taken. NOTE: Messages arriving at a port are always received. There are no conditions associated with messages.

Rewards is used to specify rate or impulse-rewards. Rewards are used in the simulation and in the analytical transient solution.

We start by creating the packet_source object type. Recall that the behavior of an object is represented by events, and messages and their associated actions.

The syntax of an event is as follows:

```
event     = event_name(distribution, parameters of distribution);
condition =
action    = {
       ...
     };
```

IMPORTANT: *conditions* are boolean-expressions over state variables only. (The user can specify the condition to be always true. In this case, the reserved word `TRUE` is used).

The single event of the `packet_source` object is `Packet_Generation`. The distribution of the event is defined by the reserved word `EXP` which indicates an exponential distribution. The rate at which this event occurs is defined by the constant `pkt_rate`. In this case, there are no conditions for this event to occur, and this is specified by the keyword `TRUE`. Each time this event triggers, the action specified is executed. In this case, a message is sent. The message sent has the following syntax: `msg(name_of_port, object, data)`.

To edit an attribute, in this case the *Events* attribute, you must right-click exactly on the object, select `Edit Attribute in Editor` and then select the attribute you want to edit. In our case, we will edit the `Events` attribute. (You can also change the default editor, setting your preferred one in your `.Xdefaults` file, adding the command `Tgif*Editor: YOUR_EDITOR_HERE`.) The syntax of this event is as follows:

```
Events    =
event     = Packet_Generation (EXP, pkt_rate)
condition =  (TRUE)
action  = {
            msg(port_out,all,0);
          };
```

There is another way to edit object attributes in Tgif. It is possible to edit more than one attribute in the editor window. To access this feature you should use the right mouse button over the Tangram-II object and choose the `Edit Attribute Group In Editor` option. The available groups will be presented in the sub-menu.

Users can create their own attribute groups by editing the `.Xdefaults` file. For instance, to create an attribute group called `TANGRAM-II` to edit all Tangram-II attributes at once, do the following:

1. Add the following lines to the .Xdefaults file:

   ```
   Tgif.MaxAttributeGroups: 1
   Tgif.AttributeGroup0: TANGRAM-II:Declaration:Events:Messages
   :Rewards:Initialization:Watches (in the same line)
   ```

2. Reload your X configuration with the command `xrdb  /.Xdefaults`.

3. Restart Tgif.

The option `Tgif.MaxAttributeGroups` configures the number of attribute groups available. The option `Tgif.AttributeGroupX` is used to define the attribute group `X`. An attribute must be defined as:

```
Attribute_Name:<Attribute1:Attribute2:...:AttributeN>
```

IMPORTANT: `all` is a reserved word reserved-word!all that indicates the message is sent to all objects connected to the the port. The Messages-attribute is empty because no messages are received by a `packet_source` object.

The next step is to declarate the state variables and constants. This object type does not have state variables. It is a Poisson source that generates packets continuously. NOTE that at least one object must have a state variable in the model. Otherwise, the model can not be solved.

We must declare the variables used in the object. The Declaration-attribute for the `packet_source` is:

```
Declaration=
Const
Float: PKT_RATE;
Port: PORT_OUT;
```

Now, we must include all constants in the Initialization-attribute.

Our first object type is ready! Figure 2.5 shows the complete `packet_source` object type specification.

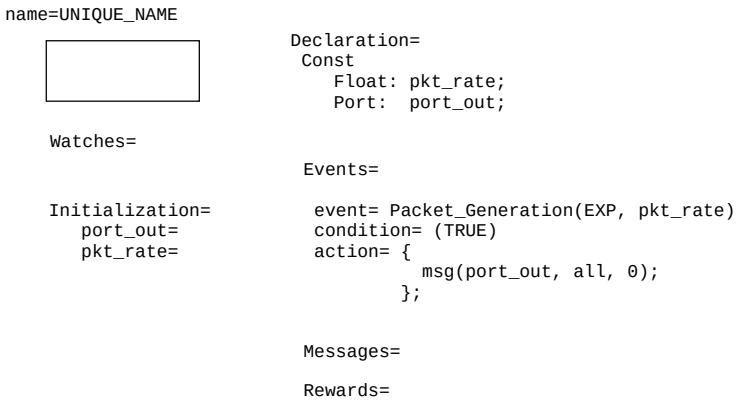


Figure 2.5: The `packet_source` object type.

After specifying the `packet_source` object type, now we will describe the server object type (queue plus server). We must follow the same steps. First, we will choose the object-template as before.

The next step is to define the state variable of the server (this should be the first step because the condition of the single event is based on the state variable). For this object type, the state is the queue. We can declare this state variable using the Declaration-attribute.

The single event of the server object is named `Packet_Service`. Each time this event triggers, the action specified is executed. In this case a packet is removed from the queue. The mean rate at which this event occurs is defined by the constant `service_rate`. The `Packet_Service` event can only occur if the queue is not empty ($queue > 0$) . The distribution of the event is exponential. When the event `Packet_Service` fires, a packet is removed from the queue. An action is specified using C-like constructs (see §D.2 in the appendix for the syntax).

```
Events =
event = Packet_Service (EXP, SERVICE_RATE)
condition =  (Queue > 0)
action = {
  int q;
  q = Queue -1;
  set_st ("Queue",q);
};
```

IMPORTANT: In an action , state variables can be used in the right-hand side of assignment expressions, but they *cannot* be modified except using the `set_ st()` (set state) function. This function assigns the value of a float or an integer constant to the specified state variable, and should be used only at the end of an action. This indicates that all state variables remain at the same value throughout the execution of an action. Their values are changed at once after the execution of the user code. The syntax is `set_st` ("state variable", integer variable). If the action is empty, it is necessary to put a ; before the `set_st` statement.

A server object type receives messages from other objects. The message represents the arrival of a packet to the server. In this model, the message is received from port `port_in` If the queue is not full, the packet is stored in the queue. Otherwise the packet is dropped (no action is taken).

```
Messages =
msg_rec = PORT_IN
action = {
  int q;
  q = Queue;
  if (Queue < queue_size)
```

```
    q = Queue +1;
    set_st("Queue", q);
};
```

In this object type, the constant PORT_IN is used to receive messages. The constant queue_size is used to represent the maximum queue size. The constant SERVICE_RATE is used to represent the mean service rate.

We must declare the constants using the Declaration-attribute and then, all constants and state variables should be included in the Initialization -attribute.

```
Declaration=
State Var
Integer: Queue;
Const
Float: SERVICE_RATE;
Integer: QUEUE_SIZE;
Port: PORT_IN;
```

Our second object type is ready! With the two object types defined above we are able to create an M/M/1/k model. There is one instance of each object type in this model. When a new object is instantiated, an unique name must be given together with initial values for all state variables and constants. In this example, the objects are named Poisson_Source and Server_Queue. NOTE: the Tgif tool has a maximum length of 255 for names of objects and variables. The other limit is the total number of bytes in each line in Tgif editor: 512 bytes.

For the constants and the Server_Queue state variable, we use the following values:

```
Poisson_Source:
* Initialization
       PKT_RATE= 80
       PORT_OUT= wire

Server_Queue:
* Initialization
      Queue = 0
      SERVICE_RATE = 100
      QUEUE_SIZE = 100
      PORT_IN = wire
* Watches
     Queue
```

Note that the port variable of each object receives the same name wire. This indicates that the two ports are connected.

The complete behavior of the model is as follows: with rate 80 packets per time unit, the source sends a message to the defined port (`wire`). All objects that are listening on this port receive this message. When received by the `Server_Queue`, this message increments the queue size, if and only if its queue is less that the maximum specified value (100 in this example). With rate 100 packets per time unit, the queue is served if and only if it is not empty.

2.2.1.3 Connecting objects

We can connect objects using either a single link or a broadcast link: see Figure 2.6. In each case the connected ports are assigned the same name, the name of the wire that connects the ports. Objects connected via a broadcast link can either send messages to all objects using the reserved word `all`, or to a specified object. Note that the broadcast-message sent is not received by the object that generated the message. In order to send a message to a specific object, the name of a variable must be used (instead of `all`). Of course, this variable must be initialized with the name of an object that will receive the message when the object is instantiated.

The connection between two or more objects can be done automatically. The `TANGRAM2_OBJECTS` domain has a port object named `port.sym` that is used in this connection. When we use this feature, the name of all ports connected is set automatically. This feature is particularly useful when the model is very large. The user can set automatically all ports in the model, or do it by hand, initializing the name of each declared port, using the `Initialization` attribute. See more details in §C.2.

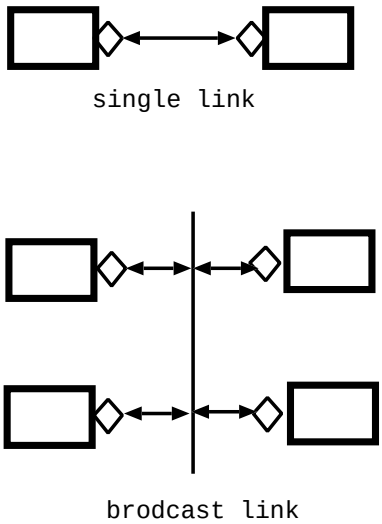


Figure 2.6: Single and Broadcast links.

Our first model is presented in Figure 2.7.

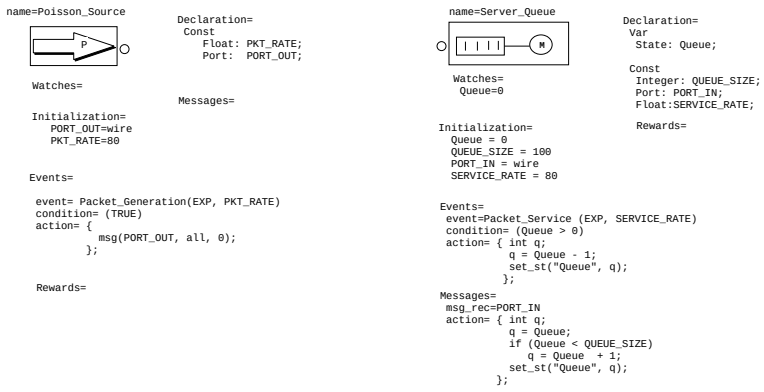


Figure 2.7: The M/M/1/k model.

In Figure 2.7 the objects have a different format. These objects (`Server_Queue` and `Poisson_Source`) are stored in the TANGRAM2_OBJECTS DOMAIN. §B.2 in the Appendix shows how to create a new object.

To save the new model, click on `File` → `Save new`. The model will be saved in the current directory.

2.2.2 Step 2: Generating the State Space

Once the model has been created , it can be solved either analytically or via simulation. Since all events are exponential in the M/M/1/k model, it is possible to generate the Markov chain and to solve it analytically. (The tool allows a class of non-Markovian models to be solved analytically as well. See chapter 9 for details.)

To generate the state space, click on "Mathematical Model Module". Figure 2.8 shows the corresponding graphical interface.

The parameter `Max number of states` limits the total number of states that will be generated by the program. If we set it to zero, no limit is used. This limit is useful during the debugging of the model, and it is advisable to explicitly set this parameter to avoid generating an unexpected very large state space due to specification errors.

In the next step, we must specifymaximum-values-for-all-state-variables. This information is useful for the hash function used by the search engine. Note that only a rough upper bound on the value of each state variable is needed, not a precise value. If we click on the `Extract` button, the name of the state variables are extracted from the model and displayed in the interface. The maximum values can then be set.

In our example model, the Server_Queue is limited to 100 and so the maximum value of the state variable queue (Server_Queue object) should be at least 100. Then

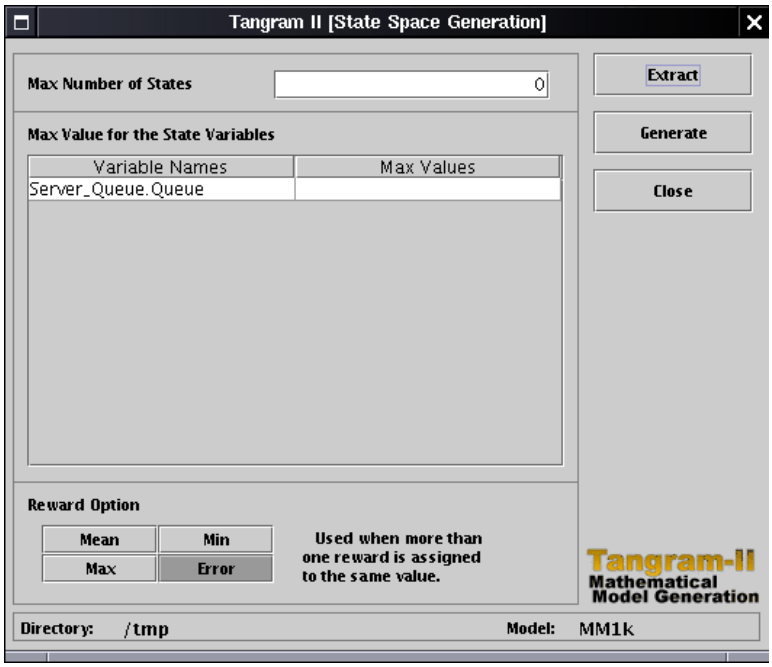


Figure 2.8: The Mathematical Model Module.

```
Variable Name                Max Value
Server_Queue.queue              100
```

Now we are able to generate the state space of the model. To run the generator program, click on the `Generate` button.

Several files are generated after this process is over. These files give the following information: the infinitesimal generator matrix of the model, the corresponding transition probability matrix of the model (if the matrix is uniformized), the state space of the model, and other information that will be used as input for other modules of the TANGRAM-II tool.

Now we are able to solve for the steady state of the model.

2.2.3 Step 3: Solving the Model (Analytically)

Once the state-transition matrix is generated, we can solve the model and obtain steady-state or transient measures. To solve the model, click on the "Analytical Model Solution button". The corresponding graphical interface for steady state solution of a Markovian model is shown in Figure 2.9.

Several solution methods are available. For detailed information on the use of different solvers see Chapter 4. In this example, we will use the GTH method. To select the GTH

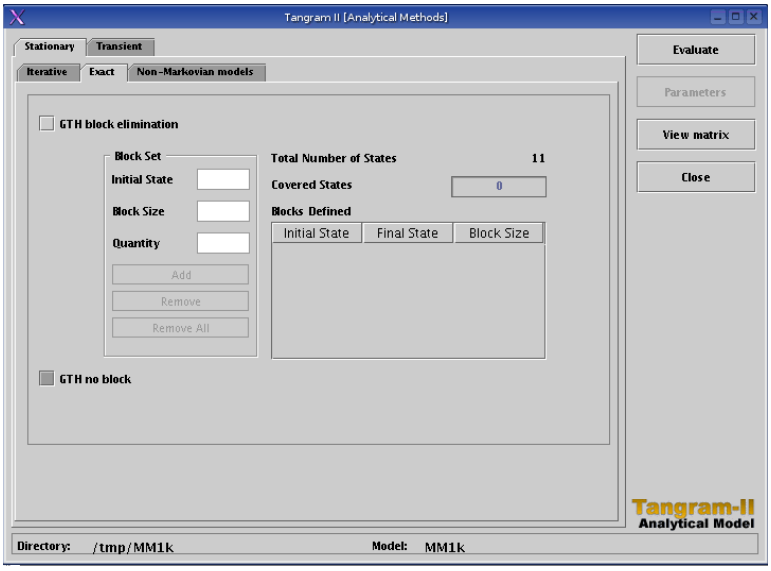


Figure 2.9: Steady State Analytical Methods.

method, choose `Solution Methods` → `Stationary` → `Exact`→ `GTH`. The file generated by the solver is <`name of the model`>`.SS.gth`. This file contains the stationary state probability vector and will be used as the input of the Measures of Interest Module.

2.2.4 Step 4: Obtaining the Measures of Interest

Various measures of interest can be calculated with TANGRAM-II, including measures obtained from functions of state variables, and conditional probabilities. For instance, in our M/M/1/k model, we can compute the expected number of packets in the system (queue + server) and the probability mass function of the number of the packets in the queue (pmf).

To obtain a measure of interest, click on the `Measures of Interest` button in the Modeling Environment interface. The graphical interface is shown in Figure 2.10.

As an example, choose "PMF of one or more state variables" to generate measures of interest. The user must specify the name of the measure of interest, e.g `pmf_queue`, that will be used to identify the file where the selected measure will be stored. Then the user must specify the file that contains the state probability vector. In our example that file has the form <`name of the model`>`.SS.gth` (in this example, it is the file `MM1k.SS.gth`).

All state variables of the M/M/1/k model appear in the "Choose Variables" box. Our example has only one state variable, so the only element in the box is `Server_Queue.queue`. Add the `Server_Queue.queue` variable into the right box. After clicking on the `Evaluate` button, the measure is evaluated provided that no mistakes were made. The message

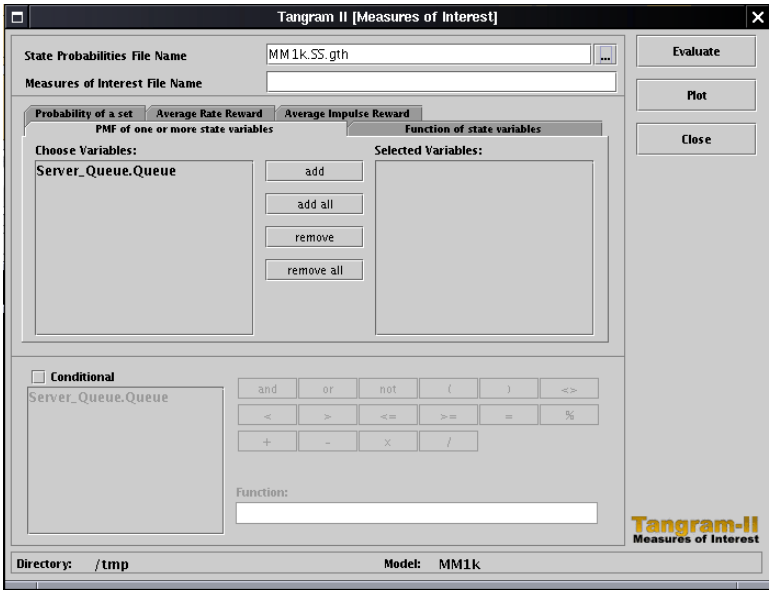


Figure 2.10: The Measures of Interest Module.

"Measures of interest generated" will pop up as soon as the calculations are concluded.

To plot the results, click on the `Plot` button and choose the file name that you specified to contain the Measures of Interest (in this case, `MM1k.IM.pmf_queue`). We can see that the expected value is shown, as well as the pmf of the number of packets in the queue (`GNUPlot` button). This plot is shown in Figure 2.11.

We can perform more experiments. For example, we can change the value of the service rate to 50. Then we generate the state transition probability matrix, the steady-state solution, and compute the same measure of interest for this service rate. Figure 2.12 shows the result.

We can also evaluate other measures. For example, the utilization of the queue, the expected number of customers in the queue (in this case $E[queue] = (n-1)\pi(n)$, where $\pi(n)$ is the probability that the system has n customers), the average time in the queue using Little's Law, etc.

Assume the measure of interest is the probability mass function of the queue size, conditioned on that the system has more than 50 customers. We encourage the user to check the `Conditional` box in Figure 2.10 and enter the proper condition. The result is shown in 2.13.

The user can also obtain a variety of measures of interest from reward models. Examples will be given in the following section.

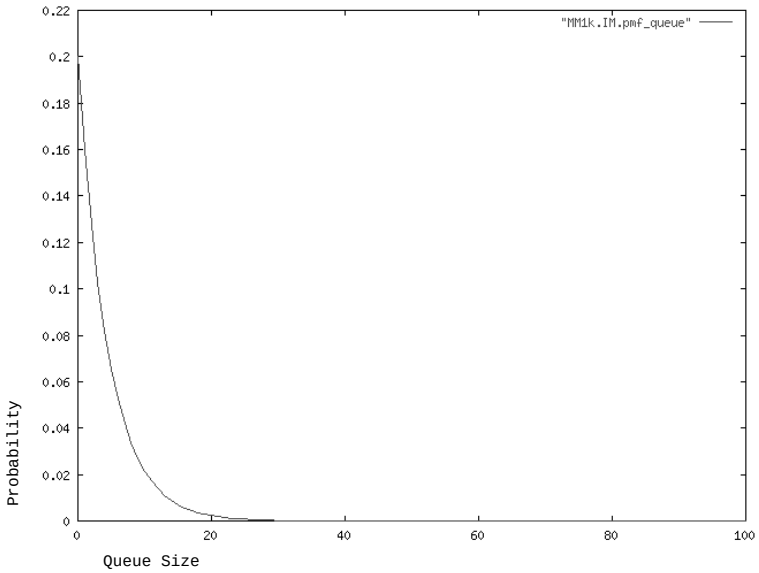


Figure 2.11: The plot generated by the PMF Module.

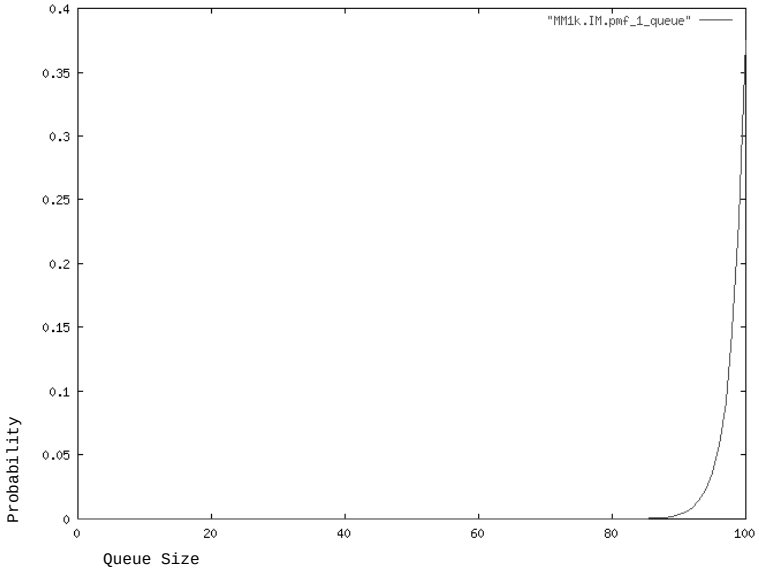


Figure 2.12: The plot generated by the PMF Module.

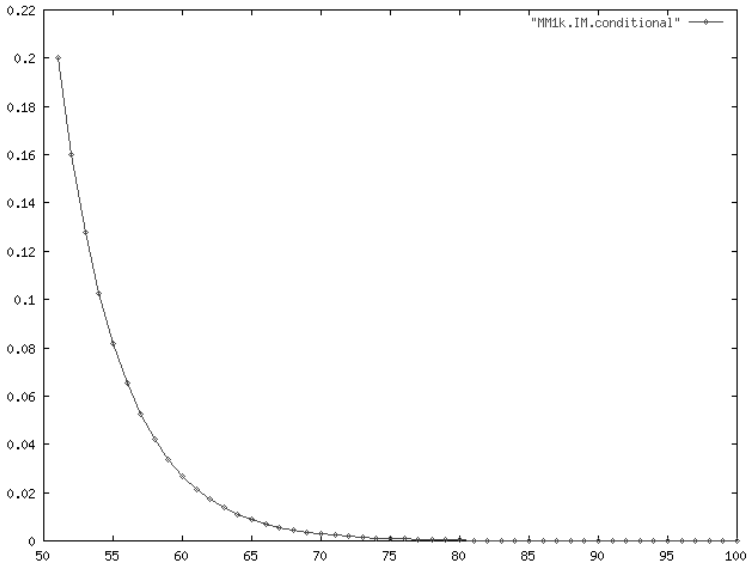


Figure 2.13: The plot generated by the PMF Module.

2.2.5 Reward Models

After describing the model, we can obtain different measures of interest using rate or impulse-rewards. *Rate* rewards are associated with the *states* in the model. These rewards are generic enough to permit the definition of a wide range of measures of interest. (For references on reward models and some of the techniques implemented, see [14, 16, 17].)

If a rate-reward r_i is associated with state i, them the system gains reward r_i per time unit spent in state i. *Impulse* rewards are associated with *state transitions*. If a reward ρ_{ij} is associated with the transition from state i to j, then the system gains a reward ρ_{ij} each time it makes a transition from i to j. The impulse-rewards may be used as a counting mechanism. An impulse reward can be associated with the triggering of an event, or with the reception of a message.

Below, we illustrate the use of rewards with the M/M/1/k model. Return to the "Model Specification Module", and open our model (using TGIF). We specify the rewards using the `Rewards` attribute:

```
Rewards=
  rate_reward = <name of reward>
  condition =
  value =     ;
```

Suppose that we are interested in calculating the utilization of the queue in the `Server_Queue` object. We can define a reward variable `utilization` that can express this measure of in-

terest. The user should specify the name of the reward, used as an identifier. The condition identifies the subset of states that will be associated with the reward. The reward value is specified in `value`. If the condition is not satisfied, that means we are not in our chosen subset of states and a reward of 0 is given.

```
Rewards=
  rate_reward = utilization
  condition = (Queue > 1)
  value = 1.0;
```

The expected value of the queue size can be specified as a rate-reward as follows:

```
Rewards=
  rate_reward = q_size
  condition= (TRUE)
  value= queue;
```

Note that the total accumulated reward in $(0, t)$ averaged over t is the expected queue size in the interval.

Suppose now we are interested in counting the number of customers served in $(0, t)$. An impulse-reward can be associated with the event `Packet_Service` and defined to have a value of 1 when this event triggers. We have:

```
Rewards=
  impulse_reward = served
  event= Packet_Service,1
  value= 1;
```

Clearly the accumulated impulse reward in $(0, t)$ averaged over t is the number of customers serviced per unit time.

Figure 2.14 shows the model and the objects attributes specified.

2.2.6 Step 5: Simulating the Model

To simulate the model, click on "Simulation Module". The graphical interface is shown in Figure 3.2.

A set of parameters used to perform the simulation must be given:

```
Runs: 5
Stop Condition=
Transitions: 1000
```

More details about simulation parameters are given in Chapter 4.

To run the simulation, click on the `Simulate` button. After simulation, all measures related to the rewards in the model are stored in the output file specified by the user. Such measures include, for instance, the total reward accumulated averaged over time.

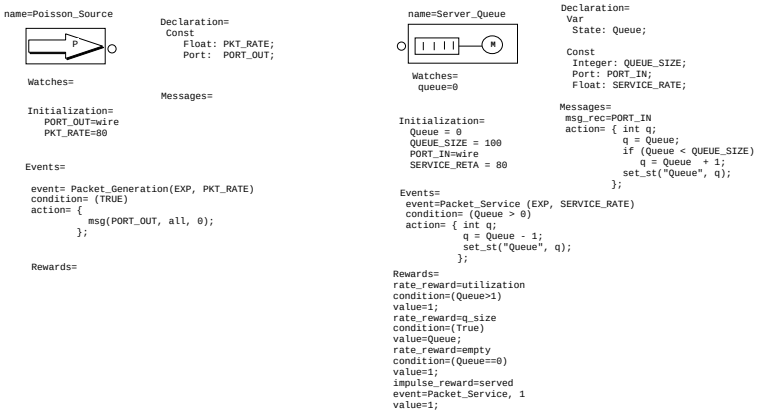


Figure 2.14: The model with Rewards

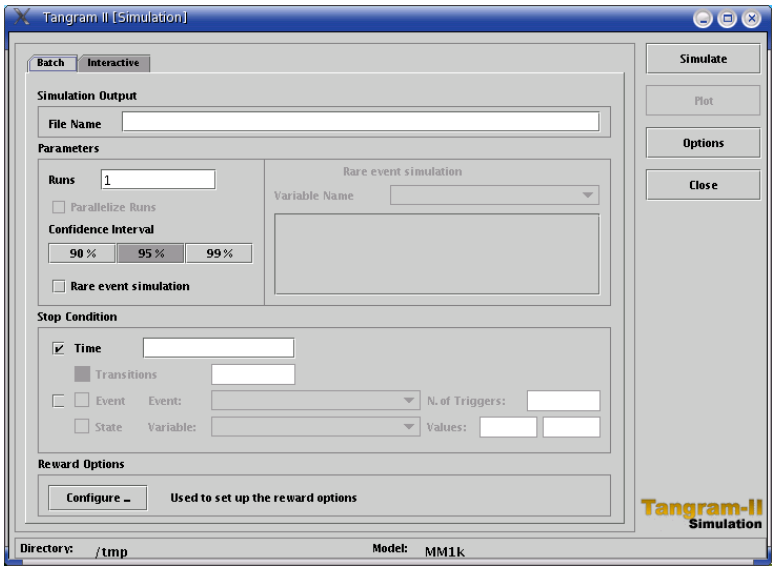


Figure 2.15: The Simulation Module

2.3 Simulation Programming with Tangram-II

Inside this section there are some Tangram-II programming features, particularities, and tips. These commands will not work with "Mathematical Model Generation".

2.3.1 Messages Between Objects

Messages allow communication between objects. A message has parameters `port`, `destination`, and `value`. Example:

- `msg(PORT_OUT, all, 4.56);`
- `msg(PORT_OUT, all, var1);`, where `var1` could be int or float.
- `msg(link, Server_Queue, vec);`, where `vec` could be `int vec[k];` or `int vec[];` for $k > 1$.

There are three ways to read the message body, depending on the message type:

1. `msg_data` is always available in the message action code, and it keeps the value of the message body (int or float). If the message sent was of a vector type, `msg_data` will have the value 0. For example,

   ```
   var = msg_data;
   ```

2. `msg_type` is always available too, and indicates the type of the message by an integer: 0 means an integer or float number, 1 means an integer vector (`INT_VEC`), and 2 indicates a float vector (`FLOAT_VEC`). E.g.

   ```
   if ( msg_type == 1 ) { ... }
   ```

3. integer vector type: the vector can be copied to a local integer vector, through the function `get_msg_data`:

   ```
   int vec[10];
   get_msg_data(vec);
   ```

   In this last case, if the sent vector is longer than the local vector, the excess part of the sent vector will be dropped. On the other hand, if the local vector is longer than the sent vector, the rest of the local vector will be 0-filled.

2.3.2 New commands

For more information see §3.3.

2.3.2.1 Obsolete commands

Commands `get_rew` and `set_rew` are now obsolete. They are replaced by the new pair `set_cr()` and `get_cr()`:

```
var = get_cr(reward_name);
```

where `get_cr` returns as a float the cumulative reward corresponding to `reward_name`.

```
var = set_cr(reward_name, value);
```

where the cumulative reward value of the reward `reward_name` will be set to `value`.

2.3.2.2 Commands `get_ir()` and `set_ir()`

Instantaneous rewards (i.r.) define how much the cumulative reward (c.r.) will increase by time unit. These i.r. values can be set in two different ways:

1. During the specification of the reward. Here the pairs condition-value defines all about our i.r.

   ```
   rate_reward = fluid1
   condition = (state == 1)
   value = -124.5;
   ```

   NOTE: in the above example, the value of the i.r. changes in accordance with the value of `state`, i.e. 0 when the condition is false and -124.5 otherwise.

2. Inside the `action` code, through the `set_ir` command, which overrides the previous value of the i.r. After using this command, this reward will increase by the value set. NOTE: Conditional values of i.r., as in the previous note, are only allowed in the reward declaration part. Once `set_ir` is used, the conditional is ignored. To restore the conditional declaration, use `unset_ir`.

   ```
   set_ir(fluid1, 100.5);
   set_ir(utilization, 0);
   ```

Commands `get_ir` and `unset_ir` are used in the code of an action to get an instantaneous reward's value and to enable automatic calculation, respectively. Their behavior will be defined by eq. (2.1) below.

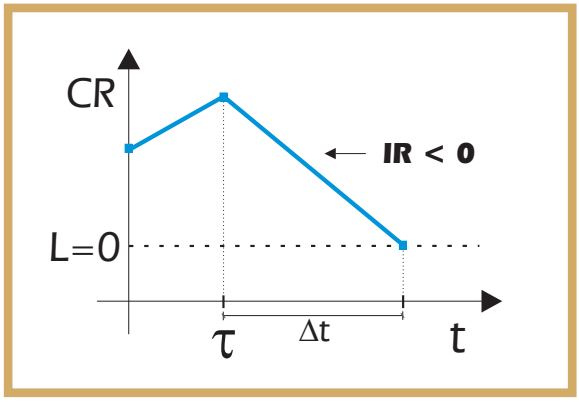


Figure 2.16: Triggering on a c.r.

```
x = get_ir(Queue_size);
unset_ir(utilization);
```

2.3.2.3 The special pseudo-event REWARD_REACHED

The special pseudo-event `REWARD_REACHED` was created in order to monitor values of cumulative rewards. Through the use of this event, actions can be taken depending on the level of a c.r.

The event triggers when a certain c.r. reaches a given value. The condition is

```
get_cr(reward) symbol limit
```

where `get_cr(reward)` represents the accumulated value of c.r. `reward`, `symbol` is one of "\/" or "/\", and `limit` represents the value to be reached. The symbol "\/" indicates that the trigger should occur when the c.r. crosses the limit from above, and "/\" that it should occur when the c.r. crosses the limit from below; see Fig. 2.16.

The occurrence time of the next trigger is calculated by the following expression:

$$\Delta t = \begin{cases} \dfrac{CR - L}{|IR|}, & \text{if } CR > L \text{ and } IR < 0, \\ \dfrac{L - CR}{|IR|}, & \text{if } CR < L \text{ and } IR > 0, \\ \infty, & \text{otherwise.} \end{cases} \tag{2.1}$$

where CR and IR represent, respectively, the accumulated reward and the instantaneous reward value of the reward specified in the condition, and L is the limit.

Multiple expressions like this can be used in the condition of the event, and the trigger time will be set to the minimum of all the values.

Beyond these special expressions, other expressions can be used to define the condition, like comparisons between state variables and constants. All expressions are evaluated, and if the ordinary ones return TRUE, the trigger times are calculated (the smaller will determine the event trigger), otherwise the event is disabled. NOTE: if the condition is TRUE, but no rewards will cross their limits, the event will be scheduled for an infinite time, that is, it will not be scheduled at all.

```
event = t0 (REWARD_REACHED)
condition =
  ((get_cr(fluid1) \/ 0) || (get_cr(fluid2) \/ 0))
action = { ... }

event = FullBuffer (REWARD_REACHED)
condition = ( (Status==1) && (get_cr(buffer) /\ B) )
action = { ... }
```

WARNING: In order to use the `REWARD_REACHED` event for some reward, this reward must accumulate only using the `set_ir` command. For rewards that accumulate as

```
rate_reward=buffer
bounds=0,B
condition=(SourceStage==1)
value=lambda-C;
condition=(SourceStage==0)
value=-C;
```

the `REWARD_REACHED` event cannot be used. The declaration of a reward to use with a `REWARD_REACHED` event *must* be as follows:

```
rate_reward = buffer
bounds = 0,B
condition = (FALSE)
value = 0;
```

And in the event declaration

```
event = Activate_Source(EXP,alpha)
condition = (SourceStage==0)
action = {
  float accum;
  ...
  accum = lambda-C;
  set_ir(buffer,accum);
  ...
```

```
};

event = BufferFull(REWARD_REACHED)
condition = (get_cr(buffer) /\ B)
action = {...};
```

Another limitation is when there are two simultaneous events at least one of which is a `REWARD_REACHED` event, and the same reward is updated on both events. The `REWARD_REACHED` event occurs after all other events, so the value that will be assigned to the reward will be the one given by the `REWARD_REACHED` event. The possibility of selecting which reward value will be assigned (mean, maximum, minimum) is not implemented yet.

Also, a reward should be used as a condition only in one `REWARD_REACHED` event.

2.3.2.4 The special reward `rate_reward_sum`

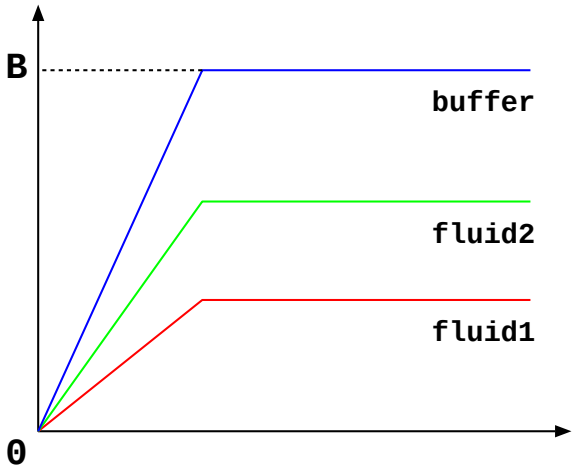
To monitor and control a set of rewards, there is a special reward that keeps the sum of the accumulated reward of a set of specified rewards.

An example of the great importance of this feature is the use of the `rate_reward_sum` to represent a shared buffer that has more that one reward to indicate each class of traffic. Fluid classes can be specified and the total shared buffer can be defined as a bound in the `rate_reward_sum` that monitors all classes. As in one single reward, the c.r. value of the `rate_reward_sum` can be bounded by a range, and the simulator maintains the correct ratio of each c.r. value of the individual rewards.

To illustrate this feature, we are going to analyze the following example:

```
rate_reward_sum = buffer
bounds = 0, B
rewards = fluid1 + fluid2;

rate_reward = fluid1
cr_bounds = 0, B
condition = (FALSE)
value = 0;

rate_reward = fluid2
cr_bounds = 0, B
condition = (FALSE)
value = 0;
```

Suppose that *fluid1* and *fluid2* represent two classes of traffic and they share the same buffer B. Each fluid should be bounded according to the buffer size: $0, B$. The amount of buffer will be given by a `rate_reward_sum` called `buffer`. The behavior of the fluids will



depend upon the `rate_reward_sum`. If *fluid1* and *fluid2* are growing and the buffer reaches the value B, the c.r. values of each fluid will be frozen because the buffer size can't exceed B.

This is a very simple example, but many other cases are treated. This complete treatment makes the feature robust enough to represent all situations where the global bound can affect the rewards.

One short example of this complexity is 3 rewards a, b, c, where a and b are growing and c is decreasing. If the `rate_reward_sum` reaches the bound B, the amount of c.r. that will be released by the decreasing c is distributed proportionally to the increasing a and b. At this point, none of the rewards are frozen. c continues decreasing at the same rate, and a, b will suffer a slope change that will limit their growth according to the distributed rate as shown in Fig. 2.17.

NOTE: All rewards whose sum will be gathered in a `rate_reward_sum` must not appear in any other `rate_reward_sum` reward. Tangram-II will report the error "Duplicated reward reference xxx. The rewards reference must be mutually exclusive." in this case.

2.3.2.5 State Variables of type Float

The Tangram-II simulator can deal with float state variables. This feature allows construction of continuous-state models, which cannot be solved analytically. There are two types of float state variables: scalars and vectors. The new syntax of the state variable declaration is

```
Declaration =
  Var:
    Integer: Status;
    Float: Fvar;
```

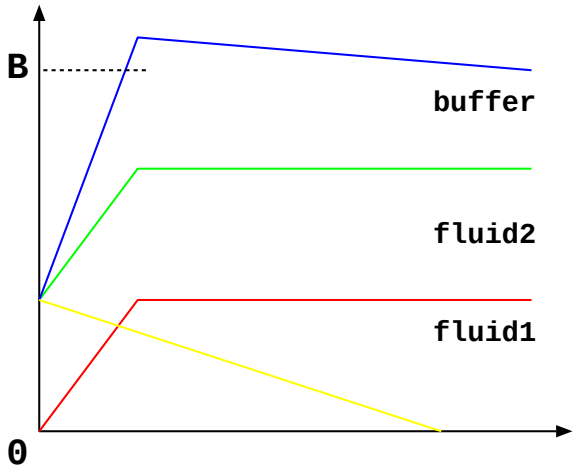


Figure 2.17: Illustration of bound affecting rewards.

```
Float: Fvec[10];
```

The use of these new variables is similar to that of the integer ones. The difference is in the use of local variables of float type to avoid type cast errors:

```
float fvar;
fvar = 1.0/3*Fvar;
set_st( Fvar, fvar );
```

or for vectors,

```
float fvec[10], aux;
get_st( fvec, "Fvec[]" );
...
fvec[2] = aux;
set_st( "Fvec[]" , fvec);
```

2.3.2.6 Type cast

Now is possible to use of the type cast operator of the C language in assignments:

```
int i, var_i;
float f;
  ...
i = (int) 20/State;
f = (float) var_i;
```

2.3.2.7 Float/Integer Queue

This kind of state variable allows using a dynamic structure to keep the state values in the simulator. All other state variables have a static size, defined during the modeling.

`FloatQueue` and `IntegerQueue` variables can be viewed as double-ended queues. The commands `save_at_head()`, `save_at_tail()`, `restore_from_head()`, and `restore_from_tail` manipulate these queues. `restore_from_head` and `restore_from_tail` remove the element from the queue. One limitation is that Tangram-II dqueue operations only support vectors, and Tangram's vectors must have dimensions ≥ 2.

```
Declaration =
  Var:
    FloatQueue: Statusqueue(2);
```

Inside an event or message:

```
float b[2];
FloatQueue statusqueue(2);
...
get_st(statusqueue, "Statusqueue");
b[0] = 1.0;
b[1] = 2.5;
save_at_tail(statusqueue,b);
set_st("Statusqueue", statusqueue);
```

Inside another event or Message:

```
float a[2];
FloatQueue statusqueue(2);
...
get_st(statusqueue, "Statusqueue");
restore_from_head(statusqueue, a);
set_st("Statusqueue", statusqueue);
```

For more information, see the fluid server FIFO example.

2.4 Where to Go Next

In this chapter we illustrated how to build and solve a simple model. Tangram-II provides much more modeling power and flexibility than is possible to demonstrate in a short tutorial and we encourage the user to use the tool an explore the possibilities using a simple model such as the M/M/1/k or any other small model the user is comfortable with. We also encourage the user to compare analytical and simulation results. In the next chapters, we describe many options available in the tool.

Chapter 3

Simulating with TANGRAM-II

3.1 Introduction

The main purpose of this chapter is to describe the simulation solver modules that is part of TANGRAM-II tool. As in the analytical model, the model has to be fully specified using the Model specification Module. To select the module use the simulation button.

3.2 The TANGRAM-II Simulator

When analytical methods can not be used to solve the model, simulation is the method of choice. Simulation can be used for solve any kind of model specified in the Model Specification Module. In this case, the events can have a general distribution, selected from a set of pre-defined distributions. Alternatively, the times of occurrence of an event can be read from a file.

3.2.1 Models with Rewards

After describing a model, a user can specify different measures of interest using rewards. Rewards are values that can be associated with system *states* (rate rewards) or system *transitions* (impulse rewards). Rewards are generic enough to allow the definition of a wide range of measures.

A reward (rate or impulse) can be defined for a particular object using the Rewards object attribute. This way, rewards are specified from the state variables of the associated object. In order to use variables from different objects, we should use the global reward instruct (see the object Global_Rewards.sym) global reward object. The syntax used to define a global reward is the same as that used for a reward. The variables used to evaluate the condition can be state variables or rewards (impulse or rate) defined for the objects of model.

3.2.2 Discrete Event Simulation

3.2.2.1 Messages and Events

TANGRAM-II allows the description of general models (that is it is not tailored for a particular application). As such it is important to be aware of how TANGRAM-II handles messages and events to proper construct a model.

Messages are executed in zero time, so one issue is the order they are executed. TANGRAM-II executes the messages in the order they are generated by the actions. When an event triggers the event may send messages to other objects. The action associated to each message received may in turn generate more messages and the process continues until there is no more messages to be executed. When a message is processed during an action is included in a list of messages to be executed. After all messages in an action are processed the list of pending messages is searched, the action corresponding to the first message is executed and possibly new messages are added at the tail of the list.

When the simulation starts, samples for all the enabled events are generated and included in the event list in increasing order of time to trigger. (We recall that the value of the state variables changes only after the action is completely processed, including the execution of all messages associated with the action.) Whenever an event is enabled, a sample is generated. If more than one event is enabled after the execution of an action, then samples for them are generated and included in the event list. Assume that one of the enabled events triggers and the new state reached (after the execution of the action associated with the event) is such that all the other enabled events remain enabled. The TANGRAM-II simulator does *not* regenerate new samples for the events that remain enabled. The samples remain in the event list until either they trigger, or the associated event is disabled. In the last case the disabled events are removed from the event list.

To illustrate the process, consider an M/M/1/k queue. Assume that there are $n > 1$ packets in queue when at time t a packet departs from the server. A new customer enters service and a new sample for the departure event is generated, say to trigger at time $t_d > t$. Now assume a new customer arrives at $t < t_a < t_d$. The new state after the arrival is $n > 1$ and so the departure event scheduled to trigger at t_d remains in the event queue, and and no new sample for the departure event is generated.

Although the behavior above handles most situations, there are cases when the user needs to re-sample an event after an action is executed even if the event remains enabled in the new state. Although the current version of the simulator does not allow the user to automatically ask for re-sampling an event this can be easily accomplished as follows. Assume that two events E_1 and E_2 of object O need to be re-sampled after any action is executed. Add a state variable Flush with gets only 2 values:0 and 1 (true or false). Then: (a) duplicate events E_1 and E_2 (name the "new" events as E_1' and E_2'; (b) add a condition $Flush = 0$ for events E_1 and E_2 and $Flush = 1$ for events E_1' and E_2'; (c) whenever an event triggers and the user wants to re-sample events E_1 and E_2, change the value of the $Flush$ state variable. Assume that $Flush = 0$ and E_1 and E_2 are enabled. If an event

E triggers and $Flush$ changes value, E_1 and E_2 are disabled even if the other conditions are enabled. Therefore, the samples for E_1 and E_2 are removed from the event list, and new samples are generated for E_1' and E_2', since they became enabled. Since all actions in E_1' and E_2' are identical to E_1 and E_2, what was accomplished with $Flush$ was simply to re-sample the events. An example will be given in the example section 9 in example 9.5.

The user is encouraged to use the debug option in the `config` (ModelSpecific) menu and follow the execution of messages and events.

TANGRAM-II also offers the user the capability of generate several samples of an event after an action is executed. This is called *event cloning*, see §3.2.2.3.

3.2.2.2 Event Distributions

The interval between the occurrence of two events can have one of the following distributions:

1. Exponential - (EXP, rate)
2. Deterministic - (DET, rate)
3. Uniform - (UNI, lower_value, upper_value)
4. Gaussian - (GAUSS, mean, variance)
5. Erlang - (ERLANG, mean, number_of_stages)
6. Log Normal - (LOGNORM, mean,variance)
7. Pareto - (PAR, scale, shape)
8. Truncated Pareto - (TRUNCPAR,scale,shape, max_value). In this distribution, if the value of sample is more than the parameter `max_value`, the sample is truncated at `max_value`.
9. Weibull - (WEIB, scale, shape)
10. FBM - (FBM, mean, variance, maximum level, time scale, Hurst)
11. FARIMA - (FARIMA, mean, variance, number of samples, time scale, Hurst)
12. File - (FILE, file name).

FBM and FARIMA The FBM (Fraction Brownian Motion) and FARIMA (Fractional ARIMA - Auto Regressive Integrated Moving Average) process can be used to model traffic that exhibits long-range dependence. These distributions may be used in multimedia source models to represent video and voice traffic. In the FBM and FARIMA distributions some parameters have to be specified:

- *maximum level* the total number of samples generated is $2^{(\text{max level})}$.
- *time scale* time interval between two consecutive samples.
- *number of samples* total number of samples generated.

File Distribution When the File distribution is associated to an event, the time instants where the event occurs are read from a file previously declared by the user. When more than one simulation run is done, the time instants of each run must be declared in separated files named `filename.r_`i, where i denotes the index of the run starting from 0. `filename`, without extension, can be used in place of `filename.r0`. Within these files, the time instants are described by the following line format:

```
I_TIME   N_EV
```

This indicates that `N_EV` equally-spaced events occurred during the interval `I_TIME`, starting from previous interval, and the last event coincides with `I_TIME` (see Figure 3.1).

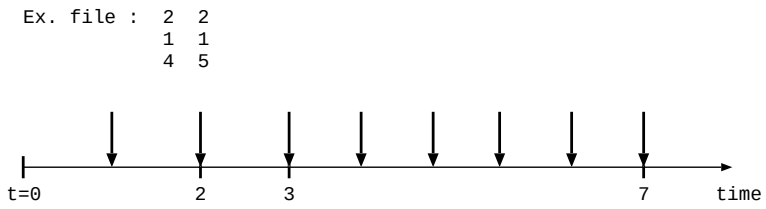


Figure 3.1: File distribution trace format

NOTE: When the stop condition of the simulation is not satisfied and (1) the number of samples of the FBM or FARIMA distribution is exhausted, or (2) the end of the file containing the distribution is reached, a box displaying a warning message appears and the simulation stops.

3.2.2.3 Event Cloning

Sometimes it is necessary to generate more than one sample of an event when the event triggers. *Event cloning* is the mechanism used to implement this feature. (An example will be presneted in chapter 9.) To clone an event, it is necessary to use the following function: `clone_ev<`*`name of event`*`>`.

3.2.2.4 Batch Simulation

After modeling the system, we can solve it using batch-simulation. The Batch Simulation Module is shown in Figure 3.2.

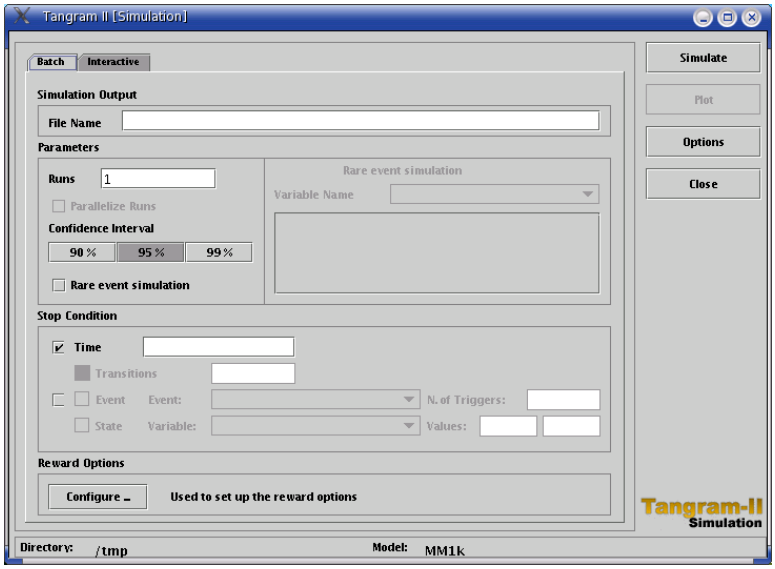


Figure 3.2: The Batch Simulation Module

The input parameters are :

- *Simulation File Name*: this file will store the result of all rewards specified in the model.
- *Runs*: total number of simulation runs.
- *Parallelize Runs*: check it to distribute the simulation runs through a network of workstations.
- *Confidence Interval*: the default value is 95%.
- *Stop Condition*: the simulation of the model will be interrupted in the following cases :
  - *Time* - Total time of each run, when only this parameter is specified.
  - *Number of Transitions* - If only this parameter is specified, each run stops when the specified number of transitions is reached.
  - *Only Events* - When only this parameter is specified, each runs stops when the specified event triggers the given number of times.

  - *(Time and Transition) or (Time and Event) or (Time and State)* - When the user specifies one of these pairs, the simulation will be interrupted when one of the conditions is satisfied.
  - *State and Time* - This stop condition uses a state variable of an object. The user should specify a lower and a upper bound for the state variable and the simulation stops when one of the bounds is reached. NOTE: if the "state" option is used, the user must also specify the "time" stop condition. This is a safety feature, to force the simulation to halt in case neither the lower nor the upper state variable bounds are reached.

- *Reward Option* - Press button `Options` to open the Reward-Options-Window

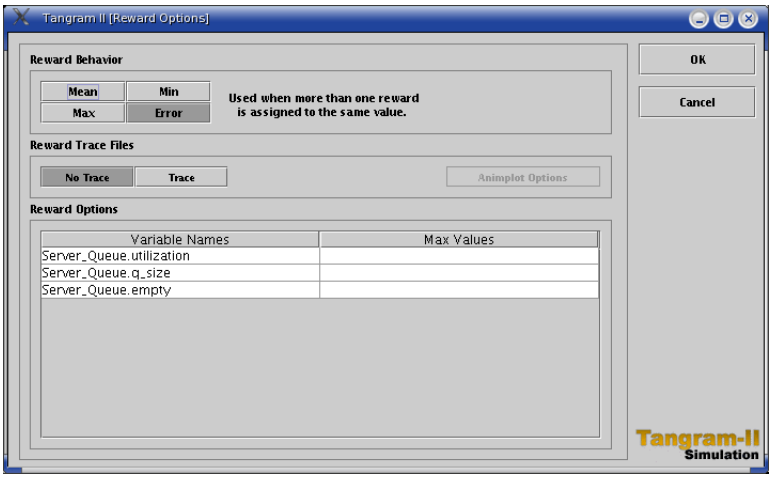


Figure 3.3: Reward Options Window

  - *Reward Behavior.* This option is used whenever there are two or more conditions/values associated with the same reward in the model. If two or more different values are associated with the same reward at a state, only one reward value can be associated to that state. The user can define the following values to resolve the multiple assignment:
    * *Mean* - The reward value to be assigned to the state is the arithmetic mean of all values associated with true conditions.
    * *Maximum* - The reward value is the maximum of all possible values.
    * *Minimum* - The reward value is the minimum of all possible values.
    * *Error* - In this case, an error message is given when more than one condition is true for a specific state. This option is useful to catch model specification errors if the user has no intention to allow multiple reward values to be assigned to a state.

– *Reward Trace Files.* Click `Trace` to generate traces from your rewards. The first column of the file is the simulation time and the second is the reward value. Reward values are only written into the trace file when the reward changes its value. NOTE: Use it with caution because you may generate huge files, refer to the FAQ for knowing how to compress these files.
– *Reward Levels Specification.* Set max values if you want to know how much time the reward was above that level.

3.2.2.5 Parallelize Runs

Tangram II uses PVM (Parallel Virtual Machine) to parallelize simulation runs. This requires that rsh and PVM be installed in your system. Binaries and source code of the PVM can be found at `http://www.csm.ornl.gov/pvm/pvm_home.html`. It is also interesting to install XPVM, PVM's graphical interface.

3.2.2.6 Configuring your Network of Workstations

Either the directory that you are running your model is replicated on each machine, or it is a distributed directory (NFS or some other distributed file system). The second option is better.

Create in your home directory(ies) a `.rhosts` file with the workstations you will use. Start xpvm or pvm and add these workstations to the pvm network (PVM command `add <workstation>`; XPVM: `Hosts`→`Other Host` or read `Help`→`Hosts` to see how to create and use the `.xpvm_hosts` configuration file). Now click on the Tangram's `Parallel` check box and then the `Simulate` button. If you are using xpvm, you will be able to see the interaction between the workstations.

NOTE: if you cannot add a machine to your PVM network, try running **rsh <machine>** and see if it requires the password to login. PVM is not able to deal with passwords. The next step is to check, on each machine, if the `$PVM_ROOT` environment variable has your PVM's root path (TANGRAM II needs it also). Tips:

- If you keep xpvm open, you should reset your Trace and your View before running your simulation again.
- If xpvm hangs, which is quite common, just kill it and restart again, your simulation runs with pvmd and will not be stopped.

If the trace generation is enabled, each workstation will write a trace file on it is own.

3.2.2.7 Interactive Simulation

TANGRAM-II allows interactive simulation to be performed. Then all state variables described in the model are updated during the evolution of the simulation. This kind of simulation is useful for debugging the model, and for educational purposes.

Animation is another feature available to the user. In this case, the object has pre-defined animations that are executed according to the states the object reaches.

The animation is specified by the user, by introducing a new attribute to an object: the `Animation` attribute. Using this attribute, the user should specify the animation of the object and others related objects. Note that the TGIF object being used in the model specification should not be used for simulation with animation. A separate TGIF object should perform the animation. The procedures, of calling an animated object and of specifying an animation, has to be done using TGIFs internal commands, that are similar to a programming language. TGIF supports a few animation primitives that can be used to perform the animations. A good reference of all supported commands can be found in the TGIF manual (`http://bourbon.cs.umd.edu:8001/tgif`). After a simulation step, the animation starts and a pre-defined number of animation steps are executed before resuming the simulation. "Simultaneous" animation of different objects are allowed and each animation is performed only at specified steps. The Interactive Simulation Module is shown in Figure 3.4.

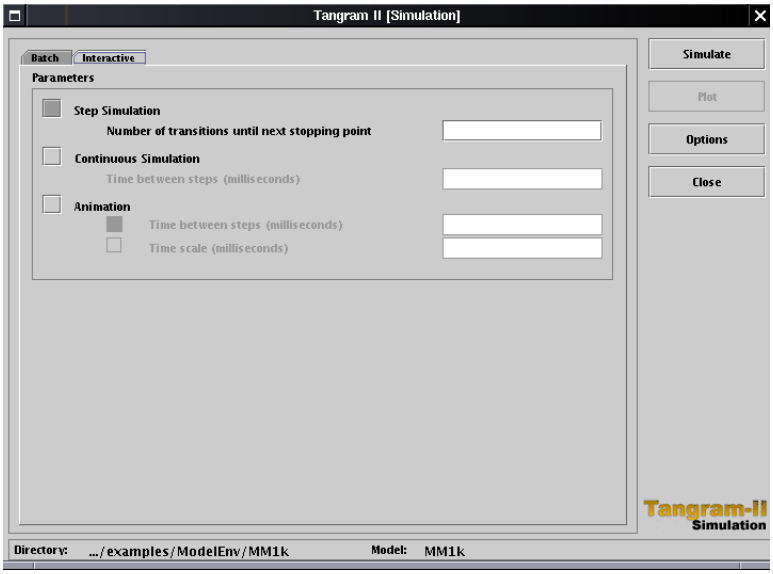


Figure 3.4: The Interactive Simulation Module.

We have three types of interactive simulations. The first, called *step simulation*, allows the user to specify the number of steps to be executed before the simulation is interrupted. In the second type, *continuous simulation*, the simulation is stopped at each transition of the model. Finally, the third type is simulation with animation. Each time the simulation stops, the user can observe on the TGIF canvas the current values of the model's state variables.

The following parameters must be given to an interactive simulation :

1. **Step Simulation** - The parameter to be given is the number of events executed between two consecutive stopping points.

2. **Continuous Simulation** - In this case, the parameter is the time, in milliseconds, that the TGIF interface, will be frozen before the next transition takes place.

3. **Animation** - The user can choose one of the following parameters: *time between steps* or *time scale*. The first parameter is the same as the parameter "time" defined for the continuous simulation. The second is defined to allow the user to observe the transitions of the model at a time scale proportional to the rate that they occur. For example, if the time till the next transition is 10 units and the time scale is 100ms, when this transition fires, the simulation stops for 1000ms.

After entering the parameters, the user can click on the `Simulate` button. Then, a box appears with the options `start`, `step`, `end` and `close` as shown in Figure 3.5. To start, the user should click on the `start` button, and a TGIF screen with the model is opened (in this screen a new box at the upper left-hand corner appears with the number of transitions of the model and the simulation time - see figure 3.6).

The simulation behavior depends on the type previously chosen. If the user choose step-simulation, the values of the state variables are displayed at each step. To proceed to the next step, the user has to click on the `step` button (see Figure 3.5). To end the simulation, the user has to click on the `end` button which closes the TGIF interface, and then on the `close` button.

If the option chosen by the user, was continuous simulation, he/she should click on the `step` button. Then, after each transition of the model, the values of the states variables are displayed at intervals equal to the given time between steps. To stop the simulation, the user must click on the progress indicator of the TGIF interface (see Figure 3.6). After stopping, she can resume the simulation by clicking on the `step` button or she can finish the simulation by clicking on the `end` button.

If the option chosen by the user, was continuous-simulation, he/she should click on the `step` button. Then, after each transition of the model, the values of the states variables are displayed at intervals equal to the given time between steps. To stop the simulation, the user must click on the progress indicator of the TGIF interface (see Figure 3.6). After stopping, she can resume the simulation by clicking on the `step` button, or she can finish the simulation by clicking on the `end` button.

In the animation option, the procedure to be employed by the user is the same as for continuous simulation.

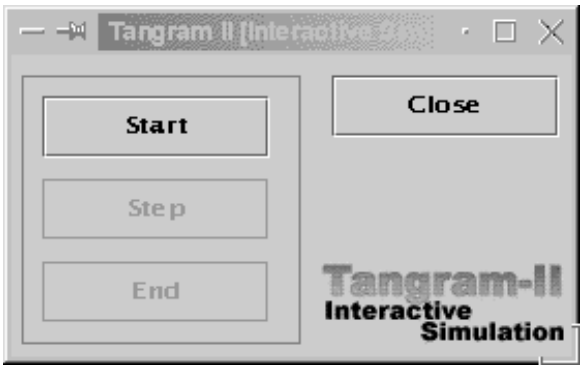


Figure 3.5: The box used to control interactive simulation

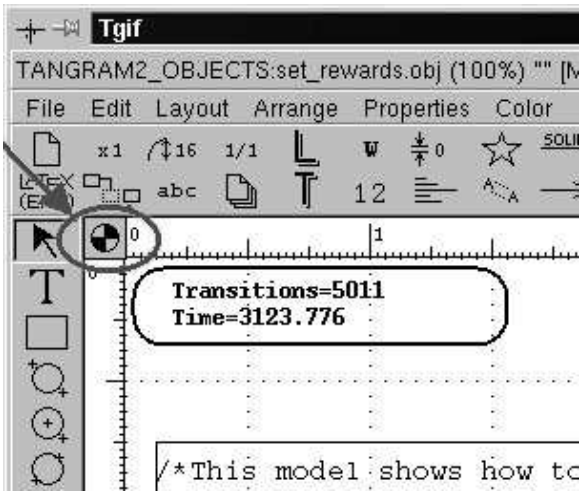


Figure 3.6: TGIF interface - Progress Indicator - Interactive simulation

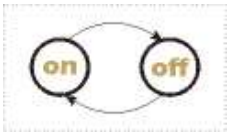


Figure 3.7: On-Off source.

3.3 Fluid Simulation

In last section, traditional simulation was the main issue. In that case, the events to be executed in the simulation were listed in time order. The simulation goes on with the execution of the events, and the scheduling of the new events in the list. A time distribution for the new event must be indicated in the model.

In *fluid simulation*, beyond the traditional events, there are new events related to the fluid behavior. Unlike traditional simulation, where, e.g., a simple server may have a simple event, a fluid simulation server has several events, like emptying a buffer, filling a buffer, or changing a rate. For execution of these events, a special behavior is required, as to monitor fluid levels, or foreseen event time execution.

3.3.1 On-off source

Five parameters must be specified in the `Initialization` attribute.

- `FLUID_CLASS` - Indicates the class number that will receive the fluid traffic generated by the source in the subsequent queue server object. Typical values range between 1 and 3 in the presented examples.
- `ONOFF_RATE` - Indicates the transition rate from the "on" to the "off" state.
- `OFFON_RATE` - Indicates the transition rate from the "off" to the "on" state.
- `FLUID_RATE` - Indicates the rate of the volume of fluid that will be generated in the on state.
- `PORT_OUT` - Responsible for the connection between the source and the subsequent object. A string should be specified, and it must be the same as the `port_in` in the next object. Two identical names indicate the direct association of the objects.

IMPORTANT: Suppose that we have a model where an on-off source feeds a buffer. The initial on-off source state is "on". Although the buffer size has to be increased until the first change to the "off" state, this does not happen. The buffer size only increases after the first change to the "on" state. In this case, we can initialize the source with "off" and declare an event with high rate that changes the source state to "on".

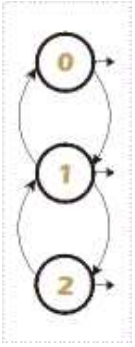


Figure 3.8: 3-state MMFS source

3.3.2 3-state MMFS source

Five parameters must be specified in the `Initialization` attribute.

- `FLUID_CLASS` - The class number that will receive the fluid traffic generated by the source in the subsequent queue server object. Typical values range between 1 and 3 in the presented examples.
- `RATE_0` - The volume rate of the fluid that will be generated in state 0.
- `RATE_1` - The volume rate of the fluid that will be generated in state 1.
- `RATE_2` - The volume rate of the fluid that will be generated in state 2.
- `TRANS_0_1` - The transition rate from state 0 to state 1.
- `TRANS_1_0` - The transition rate from state 1 to state 0.
- `TRANS_1_2` - The transition rate from state 1 to state 2.
- `TRANS_2_1` - The transition rate from state 2 to state 1.
- `PORT_OUT` - Responsible for the connection between the source and the subsequent object.

3.3.3 Channel

Three parameters must be specified in the `Initialization` attribute.

- `DELAY` - Indicates the the propagation delay in the channel. Values should be greater than 0.
- `PORT_IN` - Name of port where the fluid came from.
- `PORT_OUT` - Port responsible for the connection with the subsequent object.

Figure 3.9: Channel

Figure 3.10: Sink

3.3.4 Sink

One parameter must be specified in the `Initialization` attribute. This object can support up to 3 classes of fluid.

- `PORT_IN` - Port from where the fluid came.

3.3.5 server_queue_FIFO - CS

Five parameters must be specified in the `Initialization` attribute. This object can support up to 3 classes of fluid.

- `B` - A float number representing the shared buffer size.
- `C` - A float number representing the service rate capacity.
- `PORT_IN` - Where the fluid came from.
- `PORT_OUT` - Responsible for connection with the subsequent object.
- `MYSELF` - This string must contain the object name.

IMPORTANT: The routing is done by this object, so you must set in the `Messages` attribute the output port and the fluid class for each departure rate. This part of the code is labeled by

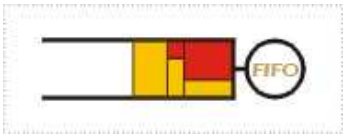


Figure 3.11: server_queue - FIFO

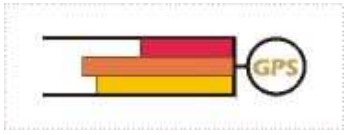


Figure 3.12: server_queue - GPS-CS

```
"/*-- send newdeparture rates -- (ROUTING ) --- */"
```

By default all messages are sent to `port_out` and the fluid class remains the same (fluid 1 is send to fluid class 1 of the subsequent queue, etc).

3.3.6 server_queue_GPS - CS

Eight parameters must be specified in Initialization attribute. Note: This object can support up to 3 classes of fluid.

- `B` - A float representing the shared buffer size.
- `C` - A float representing the service rate capacity.
- `FI1` - A float that represents the weight of the class 1 fluid.
- `FI2` - A float that represents the weight of the class 2 fluid.
- `FI3` - A float that represents the weight of the class 3 fluid.
- `PORT_IN` - Indicates where the fluid came from.
- `PORT_OUT` - Connection with the subsequent object.
- `MYSELF` - The object name.

NOTE: The sum $fi1 + fi2 + fi3$ must be equal to 1. Important: The routing is done by this object, so you must set in the `Messages` attribute the output port and the fluid class for each departure rate. The same as server_queue_FIFO - CS object.

3.3.7 server_queue_GPS - CP

Eight parameters must be specified in the `Initialization` attribute. This object can support up to 3 classes of fluid. The parameters are the same as for server_queue_GPS - CS.

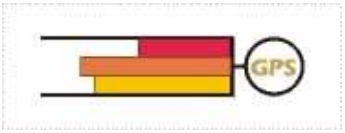


Figure 3.13: server_queue - GPS-CP

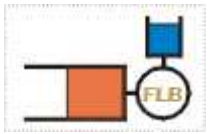


Figure 3.14: server_queue - GPS-CP

3.3.8 fluid_leaky_bucket

Seven parameters must be specified in the `Initialization` attribute. This object can support just one fluid class.

- `FLUID_CLASS` - The class number that will receive the fluid traffic released by the flb in the subsequent queue server object. Typical values range between 1 and 3 in the presented examples.
- `CREDIT_RATE` - A float that indicates the rate of credit generation for the bucket.
- `BUCKET_SIZE` - The size of the bucket, a float-type constant.
- `BUFFER_SIZE` - The size of the data buffer, a float-type constant.
- `PORT_IN`, `PORT_OUT`, `MYSELF` - As above.

3.4 Where to Go Next

In this chapter we have described the simulation solver available in TANGRAM-II. The next chapter will present the Matrix Visualization and States Permutation tool.

Chapter 4

Solvers

4.1 Introduction

This chapter describes the analytical solvers that are available in the TANGRAM-II tool. The solvers are selected using either the Analytical Model Solution or Simulation buttons. The model has to be fully specified using the Model Specification Module. Before solving the model analytically, the user has to generate the proper mathematical model using the Mathematical Model Generation button.

All analytical solvers require the generation of the mathematical model, i.e. the state space and the state transition rate matrix for Markovian models. Therefore the Mathematical Model Module must be executed before we are able to use any of the analytical solvers.

4.2 Steady-state analytical solvers

In a large number of applications, it is necessary to find the steady-state solution of the system being analyzed. Although the analyst is often interested in the system behavior during a finite time interval $(0, t)$, the steady-state solution may be a good approximation for values of t "sufficiently large". From this solution, several measures of interest can be calculated, for instance the system availability, which is the fraction of time the system remains operational during $(0, \infty)$. Other examples of measures include the average number of tasks processed per unit of time (throughput), the expected time to process a task, etc.

We can divide the methods used to obtain the steady state solution into *direct* and *iterative* methods. A method is called direct when the exact solution is obtained after a finite number of steps. On the other hand, a method is called iterative when it produces a sequence of approximate solutions that converge to the exact value.

In general, direct methods are appropriate when the state space of the model is not very large and when the corresponding state transition matrix is not sparse. Iterative methods,

on the other hand, are appropriate when the state transition matrix is large and sparse, since they preserve the sparseness of the matrix.

For Markovian models the direct methods implemented are GTH and block GTH, and the iterative methods are SOR, Jacobi, Gauss-Seidel, and Power.

4.2.1 Direct Methods - GTH and Block GTH

The GTH algorithm is a steady-state direct method that has nice properties. Basically, the algorithm works by eliminating states, one at a time from the set of states of the model. In other words, from the initial transition probability matrix, the stochastic complement of the matrix for a state is obtained and, at each subsequent step, a new stochastic complement is calculated from that found in the preceding step. Finally, an upper-triangular matrix U is obtained and the system $\pi = \pi U$ is solved.

The block version of the algorithm is also implemented. In this case, *blocks* of states are eliminated at a time, instead of a single state. As a consequence, the procedure requires calculation of the inverse of the diagonal blocks of the uniformized stochastic matrix.

One advantage of the GTH method is that is does not require any subtractions. Another interesting advantage is that the approach has a probabilistic interpretation. One problem is the "fill-in" that may occur in the matrix, which can destroy the sparseness of the original. However the method does preserve an existing banded structure (which is common in many system models).

To use this method, choose the button Analytical Model Solution. Choose `Stationary` → `Exact`. The interface is shown in Figure 4.1.

NOTE: If the method to be used is the GTH block, we must specify the final state and the block size (the number of states in the block). In chapter 5 we show how the user can specify the blocks after visualizing the transition probability matrix.

4.2.2 Iterative Methods - Jacobi, Gauss-Seidel, Power, and SOR

In many computer and communication system models, the probability transition matrix is very large but also sparse. In this case, the *iterative* methods are particularly attractive, since they preserve the sparseness of the matrix. In these methods, a sequence of approximate values $\pi^{(k)}$ for the stationary probability vector is generated, which should converge to the solution π. Each iteration of the method has a cost approximately equal to the cost of multiplying a vector by a (sparse) matrix. Therefore, the number of iterations is crucial in determining the total cost of the algorithm.

The Jacobi, Gauss-Seidel, and Successive Over-Relaxation (SOR) methods are classical solution methods for a linear system. These and the Power method are used to solve the system $\pi = \pi P$, where P is a stochastic matrix. The interface for iterative methods is shown in Figure 4.2.

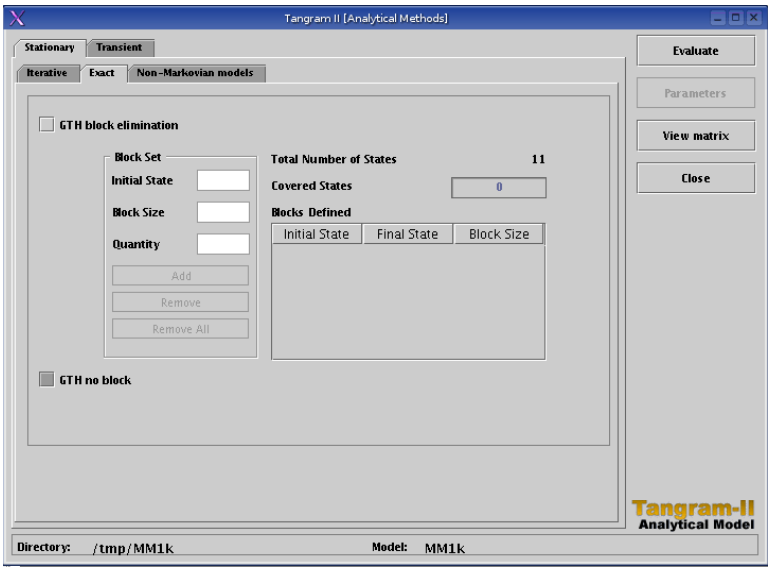


Figure 4.1: The Stationary Exact Methods.

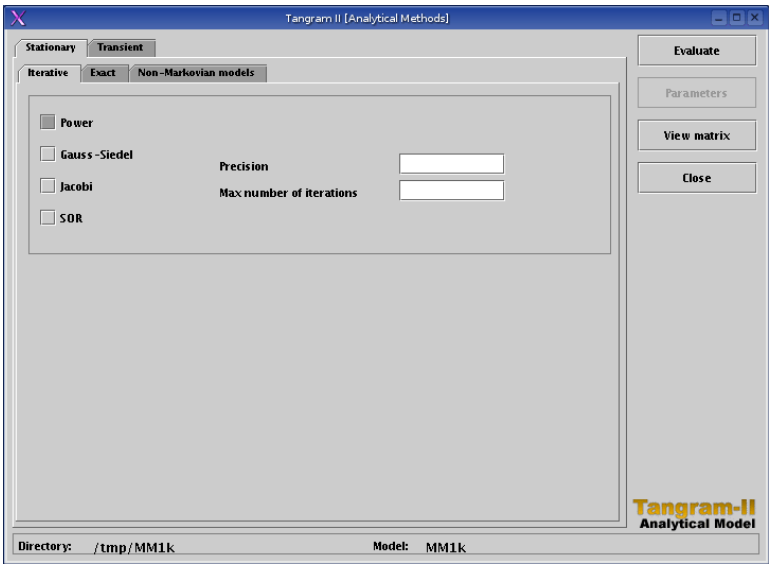


Figure 4.2: The Stationary Iterative Methods.

Two parameters have to be defined for these methods: the precision and the maximum number of iterations. The maximum number of iterations is used as a stopping condition, if the convergence is slow. If the method does not converge in "max number of iterations", it stops running and the user is notified.

4.2.3 Non-Markovian Models

The tool allows the solution of a class of non-Markovian models. Presently, the models that can be solved are restricted to those in which at most one deterministic event is enabled at any one time. The deterministic events may be enabled by any event and disabled by exponential events.

In this method an embedded Markov chain is constructed at special time points (*embedded* points). In any interval between embedded points, there may be either zero or a single deterministic event enabled. For instance, assume that the model consists of a single-server queue with deterministic service times. The embedded points are the service completion instants and the beginning of a busy period. Once the embedded points are determined, the transition probabilities between them are found, and then the measures of interest (see [19] for details on the solution technique).

To use this method, choose the button Analytical Model Solution and then click on Non-Markovian models. The interface is shown in 4.3.

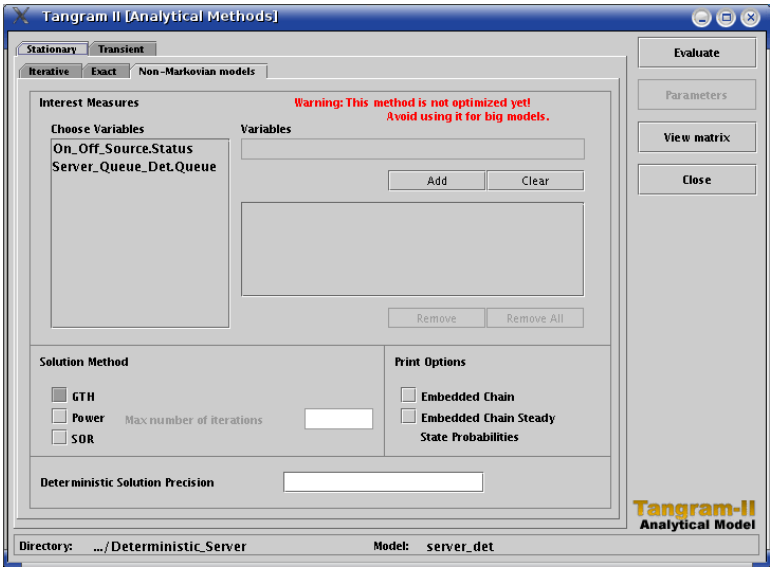


Figure 4.3: Non-Markovian Models.

The following parameters must be specified :

Measures of Interest The measures of interest are the marginal probabilities that the system spent in the specified states (see chapter 9 for details).

Solution Methods Steady-state solution for solving the embedded chain (Methods :, Power, or SOR).

Precision

Max number of iterations as defined for Iterative Methods.

During the intervals in which a deterministic event is enabled, a group of objects may evolve independently of other groups in the model. The solution technique we employ can take advantage of this behavior to reduce the computational costs of calculating the transition probabilities between embedded points.

NOTE: It is not allowed to run PMF when the file selected is from a non-Markovian solution.

NOTE: Every non-Markovian model must have an `independent_chains` object. This object specifies the deterministic event, a set of "chains", and associated objects. A set of objects associated with a chain must evolve independently of other objects associated with another chain. This special object can be found in the library (Domain `TANGRAM2_OBJECTS`). In the "Model with Deterministic Server" example (chapter 9), we show an example of the use of the `independent_chains` object.

WARNING: The order of the objects in any chain must be exactly the same as the order of the list of objects in the Mathematical Model Module window.

WARNING: The specification of the "independent chains" is for the sophisticated user who is familiar with the solution technique. If the user is not familiar with that technique, he should associate all objects with a single chain for each deterministic event.

4.3 Transient analytical solvers

In many cases, the modeler is interested in calculating measures for a relatively "short" interval, and so the results obtained from the steady state solution ($t \to \infty$) are not good approximations for the desired measures.

There are many examples which show the importance of determining the transient behavior of the system being modeled. For instance:

1. In the analysis of systems that have to remain operational in a given interval of time (usually called the system *mission time*), such as on-board aircraft computers, satellite systems, etc. In these cases, possible questions to be answered are related to the probability that the system will fail during the mission time. A failure in the system can be catastrophic or can cause considerably loss of revenues.

2. Consider a model of a data communication channel with limited buffer. There are many important questions to be answered such as: "what is the probability that a packet is lost due to buffer overflow, during a given interval of time ?" Or "how long does it take until a packet is lost ?"

3. Transient analysis is also useful to determine equilibrium results in many models. For instance, the steady-state behavior of regenerative processes may be characterized by the behavior of the process over a cycle (finite time between two regeneration points).

See [18] for a survey on transient-analysis.

The interface for transient analysis is shown in Figure 4.4. It can be seen from the figure that we can obtain three types of transient measures: Point-Probabilities, Distributions, and Expected Values (see also Figure 1.4.

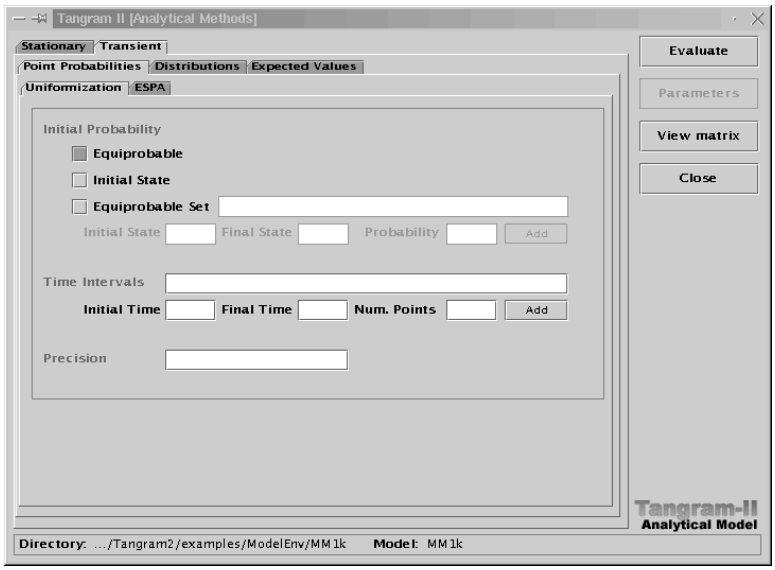


Figure 4.4: The Transient Methods.

Most of our transient solution methods are founded on the Uniformization technique (see references [12, 13, 18, 13, 45, 23].)

4.3.1 Point Probabilities

4.3.1.1 Uniformization Technique

The interface to compute Point Probabilities is shown in Figure 4.5.

The parameters to be given are :

1. **Initial Probability**. This parameter specifies the probability vector at time zero.

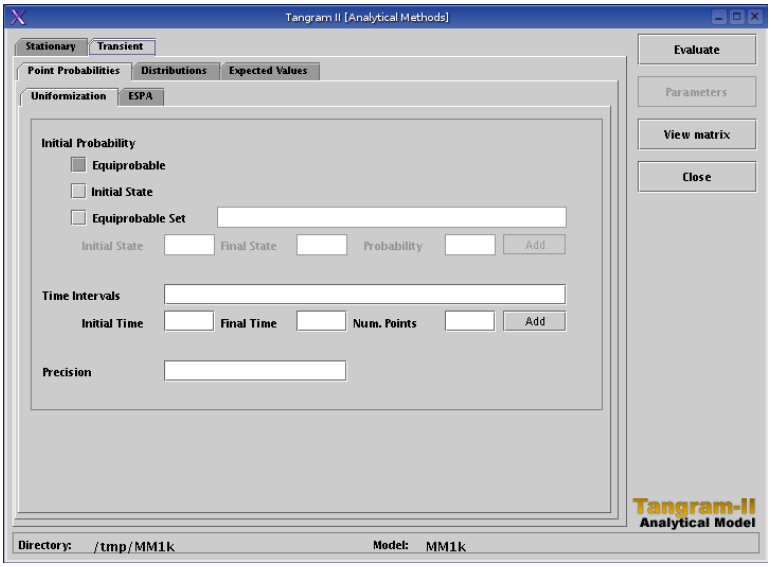


Figure 4.5: Point Probabilities Interface - Uniformization Technique.

- *Equiprobable*: all states in the model have the same Initial-Probability.
- *Initial State*: the initial state specified in the model has Initial Probability equal to one and all other states have Initial Probability equal to zero.
- *Equiprobable Set*: each state in the set specified has the same Initial Probability.

2. **Time Intervals**. This specifies all the intervals at which the point probability will be calculated.

   - *Initial Time*: specifies the first observation point. This time must be greater than zero.
   - *Final Time*: this is the last observation point.
   - *Number of points*: this specifies the total number of observation points in a time interval, including the Initial and the Final Time.

3. **Precision**. Error bound.

4.3.1.2 Approximation Technique

One of the most widely-used techniques to obtain transient measures is the Uniformization method. However, although uniformization has many advantages, the computational cost required to calculate transient probabilities may be very large for stiff-models (transition rates that differ by several orders of magnitude). This occurs because the computational

cost of uniformization is proportional to Λt, where t is the length of the observation period and Λ is a parameter which is greater than or equal to the largest absolute value of the diagonal elements of the infinitesimal generator Q. Stiff-models may give rise to large Λt values.

In [25], an efficient method to calculate transient state probabilities of Markov models and cumulative expected reward measures over a finite interval, based in the approach of [43] is proposed. In that work, the measures can be computed from iterative and direct solution techniques. For more details, see [25].

4.3.1.2.1 Direct Method The interface to compute an approximation for the transient state probabilities based on [25], using a direct method, is shown in Figure 4.6. This method has computational advantages when the matrix has a special structure.

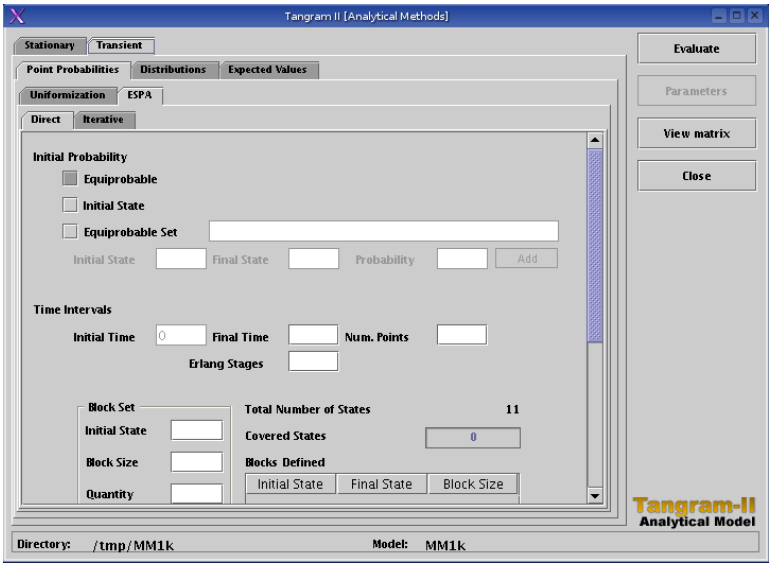


Figure 4.6: Point Probabilities Interface - Approximation Technique (Direct).

The input parameters are:

1. **Initial Probability**. The initial probabilities at time zero.
2. **Time Intervals**. The time points at which point probabilities will be calculated.
   - *Final Time*: this time is the last observation point.
   - *Number of points*: the total number of observation points in the given time interval.
   - *erlang Stages*: the total number of Erlang Stages to be used in the approximation (See [43, 22] for more details about this parameter).

3. **Block Set**. The direct method assumes that the matrix is partitioned into K blocks. The user has to define the initial state, the block size, and the number of blocks.

4. **Measures of Interest**

   - *State Probability*: with this option the point probability at time t is obtained.
   - *Probability of a set*: with this option, the probability that the system is in a set of states at time t is calculated. In this case the user has to specify the set of states using the global reward object `Global_Rewards.sym` . The states included in the set are those that satisfy the condition defined for the global reward.
   - *Expected Value*: with this option the expected value at time t of one state variable is calculated. In this case the user has to specify in the model a rate-reward with the value of the state variable.

5. **State_var**. This parameter is specified only if the "Expected Value" option is chosen.

4.3.1.2.2 Iterative Method The interface to compute the state probability approximation by an iterative method, is shown in Figure 4.7. This method is indicated for sparse matrices.

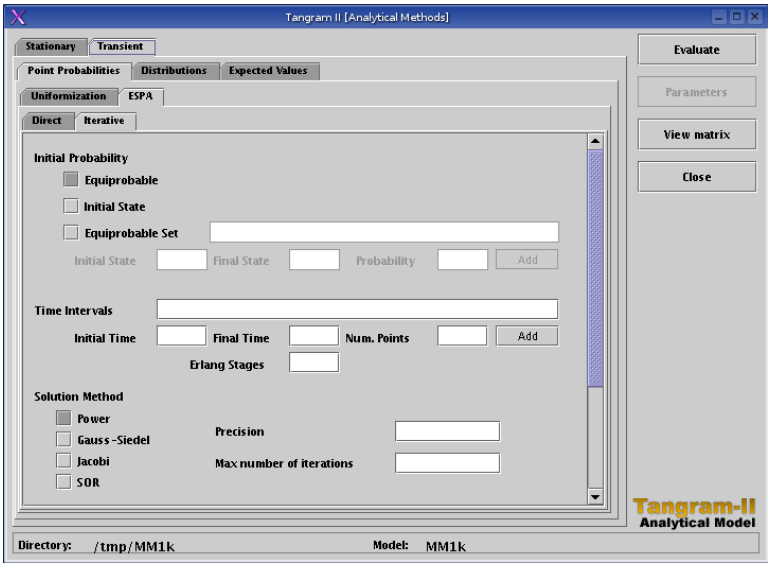


Figure 4.7: Point Probabilities Interface - Approximation Technique (Iterative).

The input parameters are :

1. **Initial Probability**. The initial probabilities at time zero.

2. **Time Intervals**. This option specifies all intervals where the point probability will be calculated. In this case we can specify different intervals, each with a different Erlang stages parameter. The intervals are non-overlapping and exhaustive, and the first begins at $t = 0$.

3. **Solution Method**. We must choose the iterative method that will be used in the calculations. These methods were described in the previous section.

4. **Measure of Interest**. As above.

5. **State_var** As above.

4.3.2 Distributions

Two algorithms are implemented to compute the following measures: *cumulative reward distribution* and *operational time distribution* (and related measures). (See [12, 14, 17, 11].)

4.3.2.1 Cumulative Reward Distribution

The tool calculates the distribution of the cumulative reward $CR(t)$ in the following cases: (1) when the random variable $CR(t)$ is not bounded, (2) when the random variable is bounded by L_{bound} and U_{bound}.

One measure that can be obtained with this algorithm is the transient queue length distribution (and from that, the packet loss ratio as a function of time). Let $b(t)$ be the number of packets stored in a limited buffer and $M = \{M(t), t \geq 0\}$ the process that models the traffic source (Markov reward model). It is not difficult to see that if C is the channel capacity and if we associate to state s a rate reward $r_{c(s)} = \lambda_s - C$, then the random variable $CR(t)$ is equal to the buffer size at t provided that $CR(t)$ is limited between 0 and the maximum buffer size B.

The interface for this method is presented in Figure 4.8.

The input parameters: ,*Initial-Probability*, *Time-Intervals*, and *Precision* have to be specified as in the Uniformization technique. To choose the *Reward Name*, the user have to click on the little button on the right hand side of the box with the *Reward Name*. Then, another window with the name of all rewards, specified by the user will appear and the user will be able to select one of them. The probabilities will be computed for the *Reward Levels* given. For example, if we give as reward levels 1000 and 1500, the tool outputs will be $P[CR(t) > 1000]$ and $P[CR(t) > 1500]$ plus the probabilities calculated for the lower and upper bound, provided that bounds are given for the reward $CR(t)$.

4.3.2.2 Cumulative Operational Time Distribution

The tool computes the distribution of the cumulative operational time and other measures of interest, such as: (1) expected availability, (2) reliability, (3) expected lifetime. The

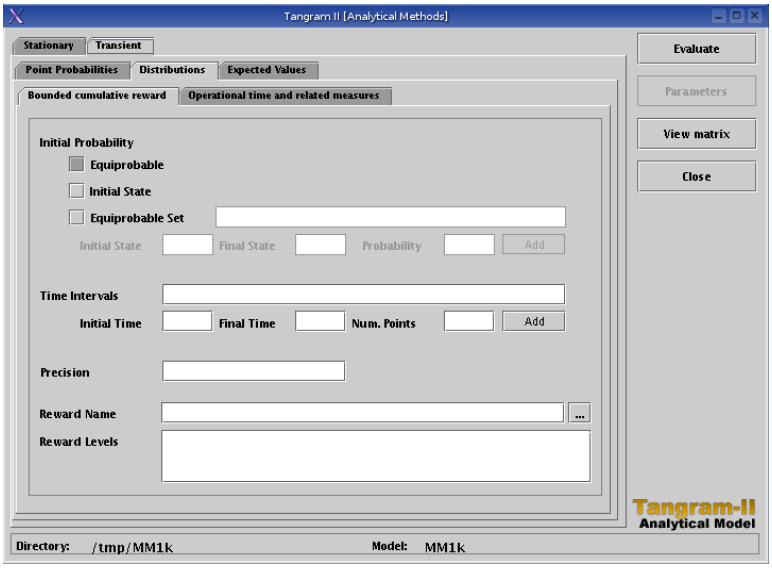


Figure 4.8: Cumulative Reward Distribution Interface

interface for this method is presented in Figure 4.9.

The parameters to be specified are the same as for the Cumulative Reward Distribution . The *Reward Levels* are given as a fraction of time, i.e. a number between 0 and 1.

4.3.3 Plotting 3D or 2D graphics for time-varying measures

In order to plot the transient measure of interest with time the following steps should be followed:

1. Solve the model using a transient analytical method. Select, for instance, the Transient/Point Probabilities/Uniformization tab, under the Analytical Methods window, and fill in the parameters. Then click on the Evaluate button.

2. On the Measures of Interest window, select more than one file for the State Probabilities File Name (for instance all files with `TS.pp` extension). The selection is done by holding the shift (or control) keyboard button and clicking the mouse left button. Note that several names will appear in the State Probabilities File Name window. If you want to generate a file to be plotted in 3D, you have to select one of the following options: "PMF of one or more state variables", "Function of state variables", or "Probability of a set" and click on the Evaluate button. Then a file with a `.3d` extension will be generated.

3. On the Measures of Interest window, click on the Plot button. There are two options:

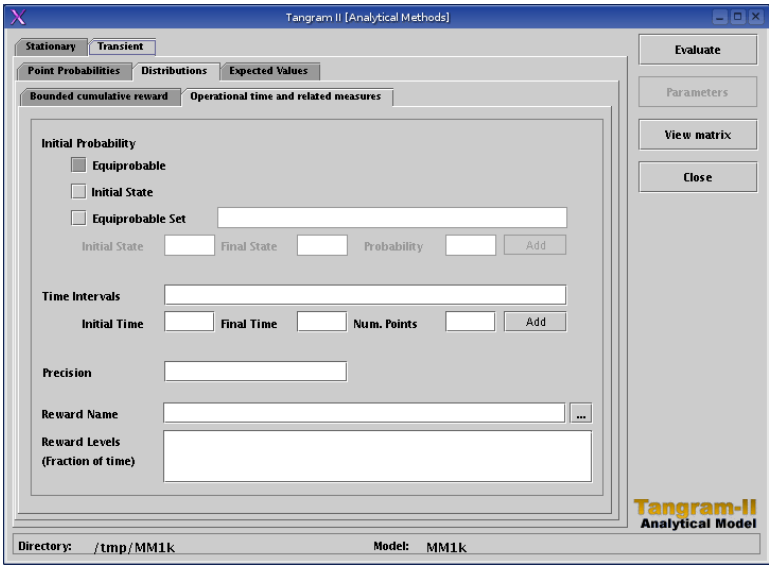


Figure 4.9: Cumulative Operational Time Distribution Interface

(a) Select a file with the `.3d` extension and click on the GNUPlot button. Then a 3D graph will be shown, representing, for instance, the evolution of the probability mass function with time of the previously selected state variable.

(b) If, instead, you select multiple input files by holding the shift or control keys, and then clicking on the GNUPlot button, a 2D graphic will be plotted with all selected files. Note that GNUplot has a pre-defined sequence of colors for graphics with multiple functions. The order the files are selected determines the sequence of colors used in the plot.

The user is encouraged to solve the simple MM1k example to familiarize him/herself with all the features and the solution files generated by Tangram-II.

4.3.4 Expected Values

4.3.4.1 Expected Cumulative Rate Reward

4.3.4.1.1 Uniformization Technique The interface for this method is shown in Figure 4.10.

The parameters are the same as for the Cumulative Reward Distribution Method . WARNING: the method gives as output the expected cumulative reward $E[CR(t)]$, not $E[CR(t)/t]$.

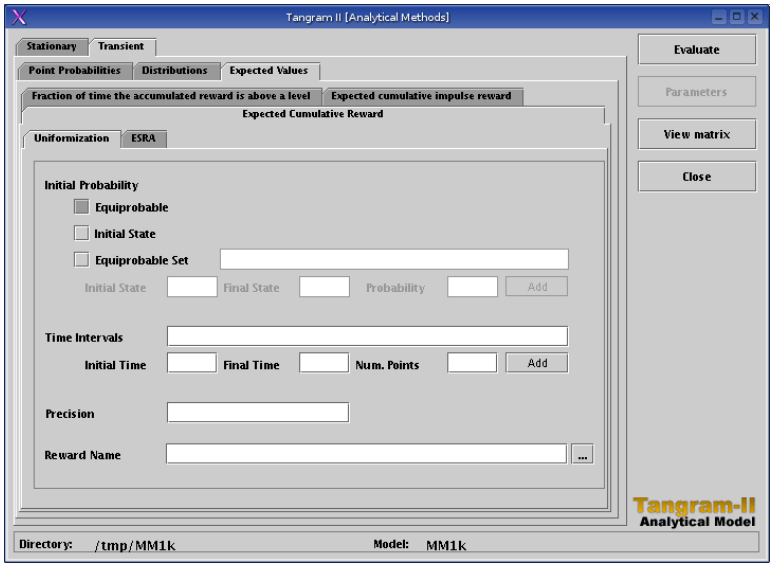


Figure 4.10: Expected Cumulative Rate Reward Interface - Uniformization Technique.

4.3.4.1.2 Approximation Technique An approximation to the expected cumulative rate reward $E[CR(t)]$ can also be obtained similarly to the method used to calculate the point probabilities (See [25] for more details.)

1. *Expected Cumulative Reward Approximation - Direct Method.* The interface to compute the expected cumulative reward approximation, using a direct method is shown in Figure 4.11.

   The input parameters are the same as for the Point Probability Approximation Method. However, when a Measure of Interest is chosen, the Expected Cumulative Reward Approximation method does not allow the Expected Value option .

2. *Expected Cumulative Reward Approximation - Iterative Method.* The interface to compute the expected cumulative reward approximation, using an iterative method, is shown in Figure 4.12.

   The input parameters are the same as for the Point Probability Approximation Method. However, when a Measure of Interest is chosen, the Expected Cumulative Reward Approximation method does not allow the Expected Value option .

4.3.4.2 Fraction of Time the Accumulated Reward is above a Level

This measure can be used, for example, to compute the expected time a finite buffer is above a given level during $(0, t)$. The interface for this method is presented in Figure 4.8 (it is the same as for the Expected Cumulative Reward Distribution).

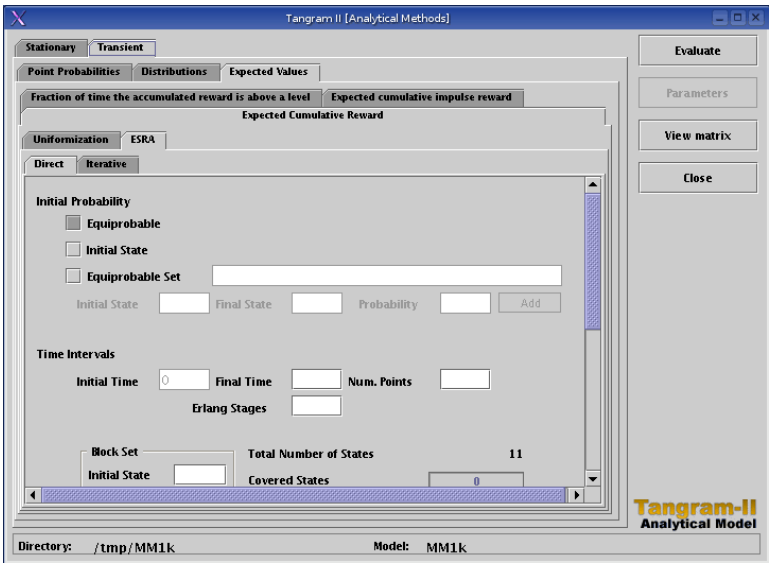


Figure 4.11: Expected Cumulative Rate Reward Interface - Approximation Technique.

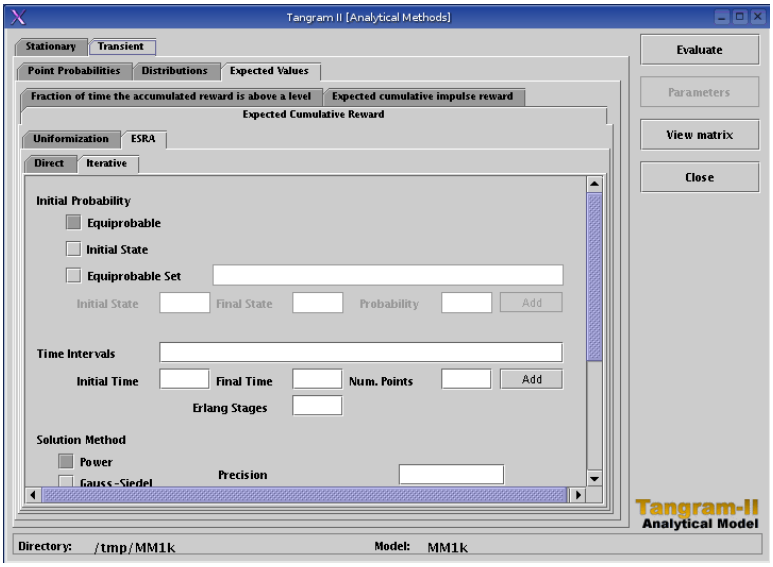


Figure 4.12: Expected Cumulative Rate Reward Interface - Approximation Technique.

4.3.4.3 Expected Cumulative Impulse Reward

The number of occurrences of some events during an interval is another important random variable to be considered in the analysis of computer systems. For instance, we may be interested in the number of failures of a given component during $(0, t)$; or in the number of times a given component caused the system to fail. In this case, the rewards are associated with the transitions of the Markov chain model.

We can calculate the expected cumulative impulse-reward of the given events using the Expected Cumulative Impulse Reward (See [23]).

The interface for this method is shown in Figure 4.13.

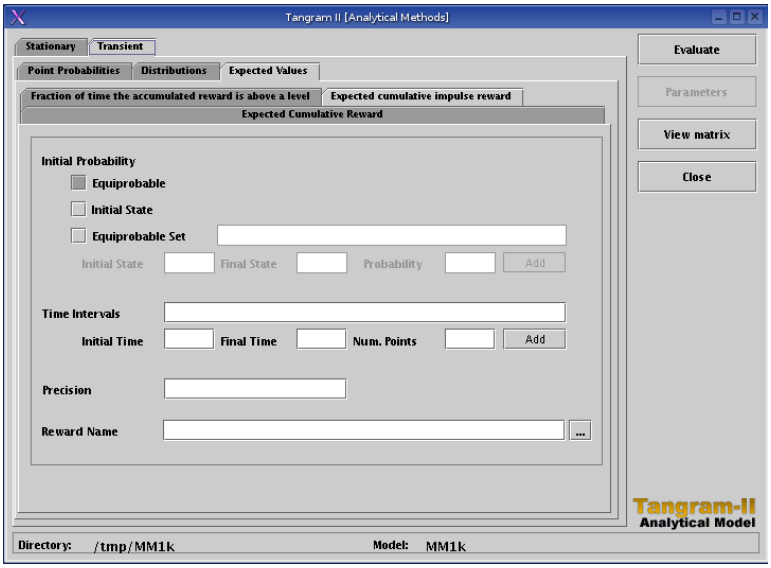


Figure 4.13: Expected Cumulative Impulse Reward Interface - Uniformization Technique.

The parameters are the same as for the Expected Cumulative Rate Reward - Uniformization Technique.

4.4 Where to Go Next

In this chapter we have described the analytical solver available in TANGRAM-II. The next chapter will present the simulation solver.

4.5 References

1. **Iterative Methods - SOR, Gauss-Siedel, Jacobi and Power**: [45, 23]

2. **Direct Methods - GTH and GTH block**: [45, 23, 30, 29]
3. **Non-Markovian Models**: [19, 15] and references therein.
4. **Uniformization Technique and Expected Cumulative Reward**: [32, 13, 18].
5. **Efficient Solution - Probability Approximation and Reward Approximation:** [43, 25]
6. **Bounded Cumulative Reward Distribution and Fraction of Time the Accumulated Reward is above a Level**: [11, 21]
7. **Operational Time and Related Measures**: [12, 14]
8. **Performability Measures, Reward Models**: [16, 14, 17]

Chapter 5

Matrix Visualization - State Ordering

5.1 Introduction

Direct analytical solution techniques can take advantage of special structure in the generator matrix. It is interesting to be able to change the order of the states variables of the model, to obtain different structures which hopefully may lower the computational costs of the solution.

This chapter describes the Matrix Visualization - State Ordering tool. It allows the user to change the order of the state variables of the model and to visualize the state transition matrix obtained after each permutation.

5.2 How to use the Matrix Visualization - State Ordering

To use the Matrix Visualization - State Ordering, choose the Analytical Model Solution button, and then, the View Matrix button. The interface is shown in Figure 5.1.

The main options of the Matrix Visualization - State Ordering interface are :

1. **Ordering of States**

   - `Use permutation`. This option is used when we want to see the structure of the matrix with the permutation of the state variables that is defined in the `List of State Variables` box. When we want to see the structure of the matrix after the chain is generated (the matrix visualization program does not guarantee any pre-defined state ordering), we do not select this option.
   - `List of State Variables`. Used to obtain a state-ordering. The number on the left side of each variable represents its current position in the variable vec-

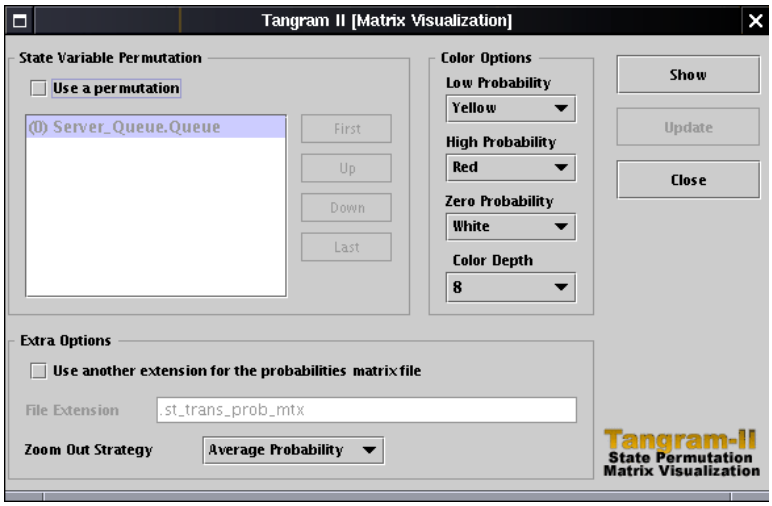


Figure 5.1: The Matrix Visualization - States Permutation Interface.

tor. It is possible to change the order of the variables with the following buttons: `First`, `Up`, `Down`, `Last`. The state variables are used to obtain a specific state-ordering. For example, if the state variables are ordered as shown in Figure 5.1, then the states are ordered in such a way that the last state variable varies faster than the previous state variable, in increasing order. The tool generates the uniformized state transition probability matrix from the generator matrix.

2. **Color Options**. This option is used to associate colors to probabilities of states (Low, High and Zero probabilities). All the colors can be chosen from the following set black, white, red, green, blue, yellow, cyan, purple. The color depth defines the number of colors used to represent the probabilities. For example, if color depth is 4 then the number of colors used is $2^4 = 16$.

3. **Extra Options**

   - *Use another extension to define the state transition probability matrix file.* A new extension is used for the probability matrix file instead of the default *i*`<model name>.st_trans_prob_mtx`.
   - *Zoom out strategy.* This option is useful when the number of elements in the matrix is greater than the number of pixels in the window canvas.
     - Maximum Probability: the probability shown will be the maximum probability (in the transitions that occupy the pixel).
     - Minimum Probability: the probability shown will be the minimum probability (in the transitions that occupy the pixel).

  - Average Probability: the probability shown will be the average probability (in thetransitions that occupy the pixel).

- **Buttons**
  - `Show` Display the matrix with the configurations specified in the interface.
  - `Update` Generates new files (generator matrix, etc) according to the new state ordering.
  - `Close` Closes the window.

The interface to visualize the matrix is shown in Figure 5.2.

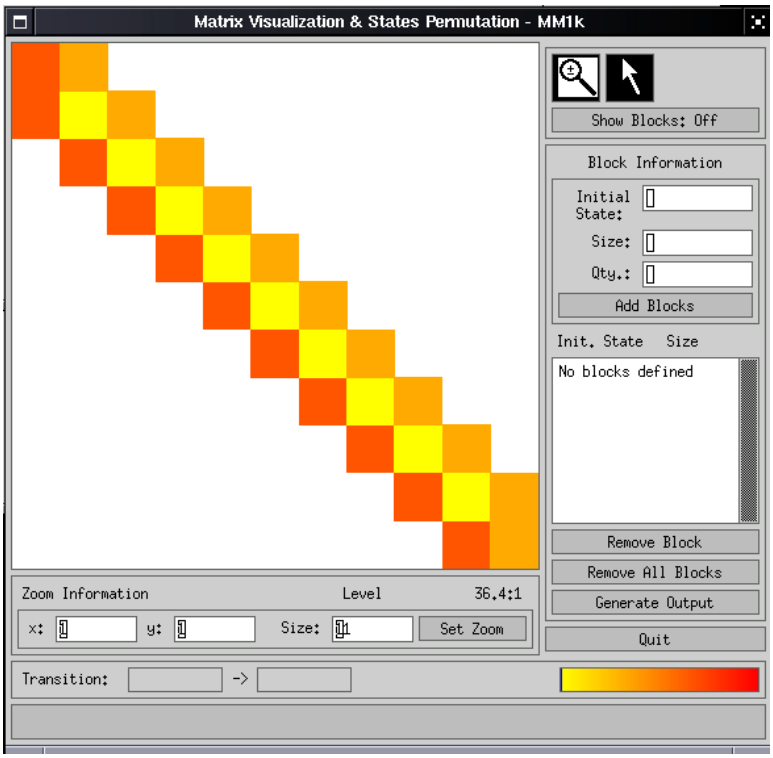


Figure 5.2: The Matrix Visualization Interface.

The main options of the Matrix Visualization interface are:

- On the left, the matrix is shown.
- The `Zoom Information` displays information concerning the current zoom. There are options to zoom to a specific area of the matrix. The given parameters are the coordinates and the size of the area (in number of states).

- The tool bars are on the upper right corner. It is possible to choose among the following options: . The option `Show Blocks: On/Off` is used to show the blocks defined for the matrix and that will be used in block solution methods (e.g. GTH Block).
- The `Block Information` option allows the definition of matrix blocks. The user has to specify the initial state of a block, the size of a block and the total number of blocks with this size. The button `Generate Output` generates a file with (the extension of this file is `.blocks`). This file will be used as input for a solution method based on the block elimination (e.g. block GTH).

5.3 Where to Go Next

The next chapter will present some examples that will be useful to explore the power of the Model Environment Module of the TANGRAM-II tool.

Chapter 6

Traffic Modeling

6.1 Introduction

The modeling and analysis of computer network traffic has been an area of extensive research over the last ten years, as new multimedia applications over the Internet become common.

In order to conduct a performance study, several steps are needed [34]. First, one must understand the characteristics of the traffic competing for the resources under investigation. We can: (a) analyze measurements taken from the actual *aggregated* traffic of an existing network; (b) and/or focus on traces of traffic, for instance a trace from a given application such as voice and video transmission; (c) or concentrate on different "classes" from a given application. (For instance, for video transmission application, the classes of action movies, or the class of lectures, etc.)

A large number of models have been proposed in the literature. They include Markovian models, and models that possess long-range dependence. The user should be able to use these traffic models as input to a model that includes the system resources in order to conduct performance studies.

The TANGRAM-II environment provides a set of tools to measure traffic, obtain descriptors, and experiment with different models. The modeler is able to: (a) use statistics from real traces; (b) choose from different traffic models (which includes Markovian, FBM and FARIMA models); (c) calculate descriptors from the models to be able to match parameters and/or verify statistical differences from the model to the measured data; (d) create a "complete" performance model which includes the traffic model and the resources under study; (e) solve it via simulation or analysis and; (f) conduct experiments with traffic generators over a laboratory environment.

The TANGRAM-II modeling environment includes traffic modeling tools and a traffic generator, that are integrated with the simulation and the analytical modeling tools.

6.2 Traffic Modeling

The tool can calculate statistics from a pre-recorded stream (trace). The module accepts as input, for instance, the number of bytes per a given interval and produces first and second order statistics as output. The first-order statistics are: average traffic rate (bits/time unit), variance, and burstiness. The second order statistics are: autocovariance, autocorrelation, and index of dispersion per count. The IDC(t) for instance, is obtained, for a finite data set, by dividing the set into non-overlapping intervals and taking these intervals as different sample paths for the random process $\mathcal{N}$. In the interface we call "window" the non-overlapping intervals. We can also obtain other measures, such as the fraction of time above a given rate.

Let $X(n)$ be the nth sample of your trace file.

- The autocorrelation is given by

$$\text{autocor}(\tau) = \frac{E[X(n)X(n+\tau)] - E[X(n)]^2}{E[X(n)^2] - E[X(n)]^2}. \tag{6.1}$$

- The autocovariance is given by

$$\text{autocov}(\tau) = E[X(n)X(n+\tau)] - E[X(n)]^2. \tag{6.2}$$

- The IDC is given by

$$\text{IDC}(\tau) = \frac{E[N(k\tau)^2] - E[N(k\tau)]^2}{E[N(k\tau)]}, \tag{6.3}$$

  where $N = \{N(k\tau),\ k\tau \geq 0\}$ is a stationary process that counts the amount of data in the interval $[0, k\tau]$, with $k \in [0, \texttt{Number of points}]$. As your trace is a sample path, not a stationary process, it will be split into several windows (sample paths).

Figure 6.1 shows the interface to compute some statistics from a pre-recorded stream.

The parameters to be given are:

Trace Name The name of the file that contains the pre-recorded stream.

Number of samples Number of samples in the trace file.

Time Scale Time between two consecutive samples in the trace (Δt).

Maximum Time Lag The measure will be computed from 0 to this value.

Number of points The total number of observation points, from zero to maximum time lag. Note that τ = Maximum time lag/Number of points.

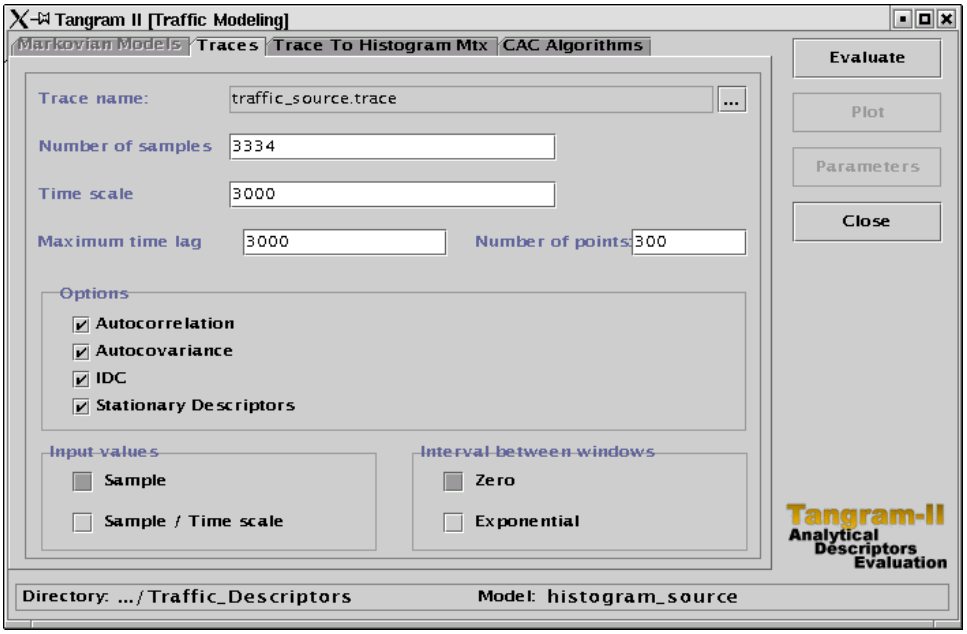


Figure 6.1: Interface to obtain traffic statistics from a trace

Input Values
- `Sample`: $X(n) =$ `sample`, where `sample` are the sample values given in your trace.
- `Sample/Time scale`: $X(n) =$ `sample`$/\Delta t$.

Interval between windows The interval between two consecutive windows: it can be an exponential random variable or zero. It is used for the computation of IDC.

Notice that Δt and τ are related. It does not make sense, for example, to set $\Delta t = 2$ and $\tau = 1$, as there is no sample at odd times. For these mistakes, Tangram II prints an error on your terminal window.
NOTES:

1. Your trace file should contain just the sample values. The tool assumes that they are evenly-spaced by Δt.

2. When a time scale does not make sense, as in calculating the autocorrelation of a delay trace, set Δt to 1.

If the traffic model is Markovian, first and second order statistics can be obtained using the recursions given in [29]. The overall model containing the traffic model plus the network resources model can be built using TANGRAM-II. TANGRAM-II also calculates several measures of interest, such as loss probabilities.

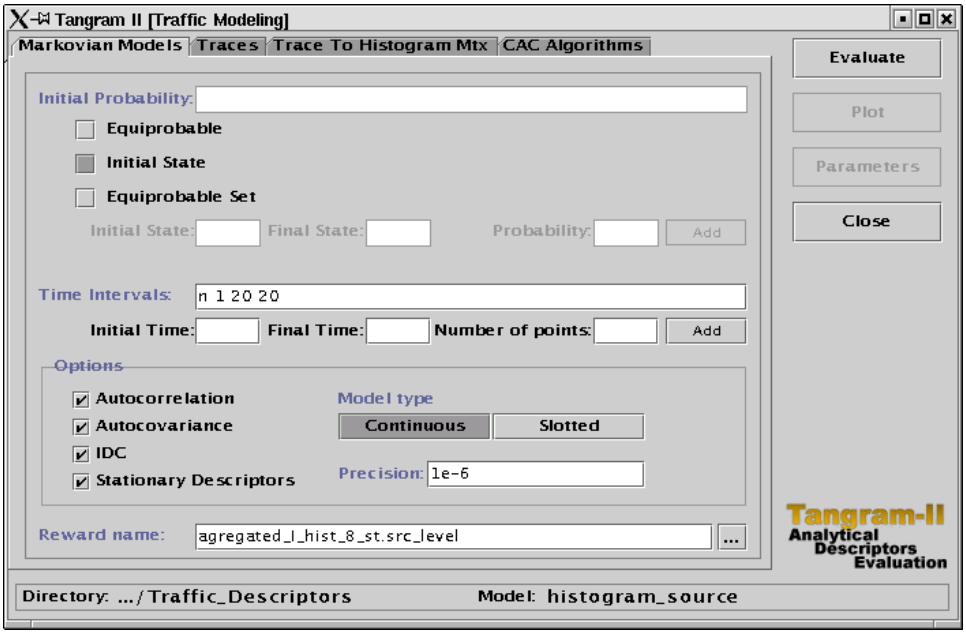


Figure 6.2: Interface to obtain traffic statistics from a markovian model

Figure 6.2 shows the interface to compute some statistics from a Markovian model. The parameters `Initial Probability`, `Time Intervals`, and `Precision` have to be provided for second-order statistics only. The parameter `Reward Name` is the name of the reward in the model that will be used to calculate the traffic model statistics (in general, the reward represents the traffic source rate). The parameters to be given are:

Initial Probability This specifies the probability vector at time zero:

- `Equiprobable` All states in the model have the same initial probability.
- `Initial State` The initial state specified in the model has initial probability 1, and all other states have initial probability 0.
- `Equiprobable Set` Each state in the set specified has the same initial probability.

Time Intervals The intervals at which to compute the traffic model statistics:

- `Initial Time` The first observation point; this time must be greater than zero.
- `Final Time` The last observation point.
- `Number of points` The total number of observation points in a time interval, including the initial and the final time.

Precision Error bound.

The user is not limited to Markovian traffic models. The TANGRAM-II simulator allows the specification of inter-event times obtained from samples from FARIMA or FBM processes. In this case the user must specify the mean rate, variance, time scale, and Hurst parameter.

From either the Markovian models or the FARIMA Distributions, FARIMA and FBM Distributions, FBM, second order statistics can be obtained, by direct recursions (if the model is Markovian) or from a trace generated by the simulator.

6.3 Connection Admission Control (CAC) Algorithms

CAC algorithms should predict the fraction of the network resources that will be consumed by the traffic generated by each application. One of the most important resources is channel bandwidth. The problem of bandwidth allocation, in particular in ATM environments, has been addressed in a number of works.

Traffic descriptors, in particular those standardized by the ATM forum (UPC), play an important role in conveying the minimum amount of traffic information to the algorithms. Briefly, in many of these works, the loss probability is estimated based on the traffic descriptors and amount of available bandwidth.

The CAC module implements two CAC algorithms [31, 38], which are useful to provide a basis for comparison against laboratory measurements and other possible CAC algorithms. The CAC module calculates the *effective capacity* and the *number of admitted sources* based on the traffic descriptors specified by the user, the buffer size, link capacity, and the desired QoS. As shown in Figure (6.3) the user can specify the following parameters:

Transmission Capacity This parameter specifies the transmission capacity of network node in (bytes/s).

Buffer Size The size of the buffer in bytes, used in the effective capacity calculations.

QoS - Loss Probability The desired loss probability.

The traffic descriptors used for each CAC algorithm are as follows.

6.3.1 Regulated Traffic Algorithm

Source Peak Rate The source peak rate in bytes/s.

Source Average Rate The source average rate in bytes/s.

Maximum Burst Size The maximum burst size or the leaky bucket size in bytes.

6.3.2 Non-Regulated Traffic Algorithm

Source Peak Rate The source peak rate in bytes/s.

On-Off Rate The transition rate from on to off.

Off-On Rate The transition rate from off to on.

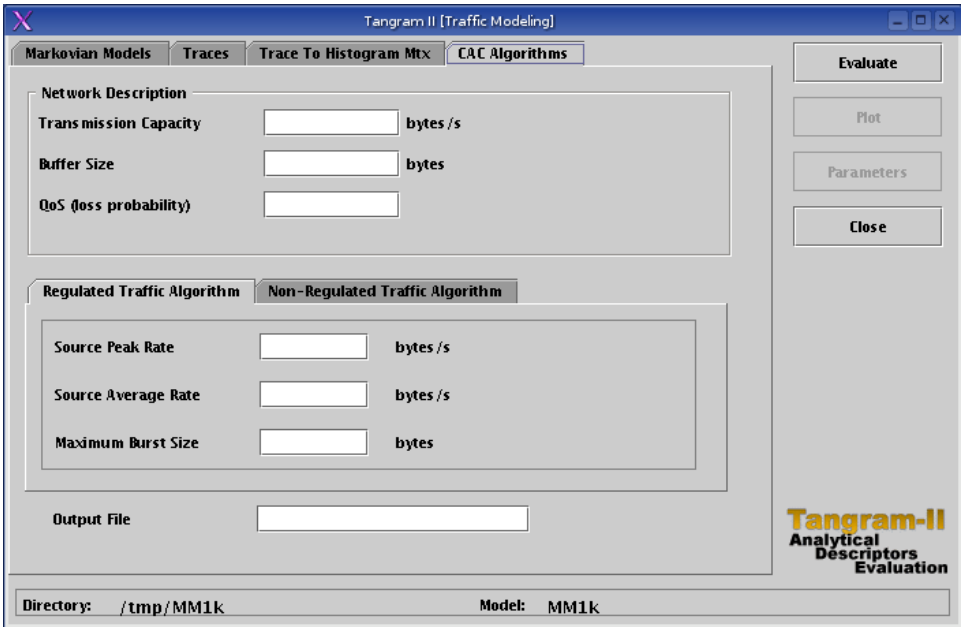


Figure 6.3: Interface of Tangram-II to CAC algorithms

6.4 Where to Go Next

In this chapter we introduced the Traffic Modeling module of the Tangram-II tool. This module offers several possibilities to the user, such as: (a) use statistics from real traces; (b) choose from different traffic models (which includes Markovian, FBM and FARIMA models); (c) calculate descriptors from the models to be able to match parameters and/or verify statistical differences from the model to the measured data; (d) create a "complete" performance model which includes the traffic model and the resources under study; (e) solve it via simulation or analysis, and (f) conduct experiments with traffic generators over a laboratory environment. We also encourage the user to go through the Traffic Generation tool, which will be described in the next chapter.

Chapter 7

Traffic Generator Tool

7.1 Introduction

One of the main goals of teletraffic engineering is to be able to predict, with sufficient accuracy, the impact of the traffic generated by the applications on the network resources, and evaluate if the required QoS is being achieved. With the increasing demand for multimedia applications, to know the internal network characteristics became essential for improving the quality of these applications. Loss, consecutive loss, round-trip-time (RTT), one-way delay (OWD), jitter, bottleneck bandwidth, bottleneck buffer size, drop rate, among others, are some of those network characteristics commonly measured. To discover these parameters, methods have been proposed.

The *Tangram-II Traffic Generator* is a powerful tool for discovering network characteristics. In the first version, the tool was able to estimate only some basic parameters (jitter, loss, and consecutive loss). However, the current version is able to estimate many others. Beside the parameters mentioned before, the *Tangram-II Traffic Generator* has implemented methods to estimate bottleneck bandwidth, one-way delay, bottleneck buffer size, drop rate, and other network measures. To estimate some of these metrics, state-of-the-art algorithms were implemented inside the *Tangram-II Traffic Generator* source, as for example the methods to estimate the one-way delay without any synchronization devices.

The *Traffic Generator* developed is capable of injecting packets in the network at intervals in accordance to the user specifications. It supports UDP/IP and native ATM through the ATM adaptation layer (AAL5). When UDP/IP is chosen the tool employs the BSD socket standard. Traffic for native ATM is supported using the API developed by Werner Almesberger (available in [1, 2]) for Linux. However, to make available the ATM generation interface, the tool employs a check to verify some dependences. The traffic generator interface is shown in Figure 7.1.

Warning: Special care must be taken for a very high UDP traffic generation rate.

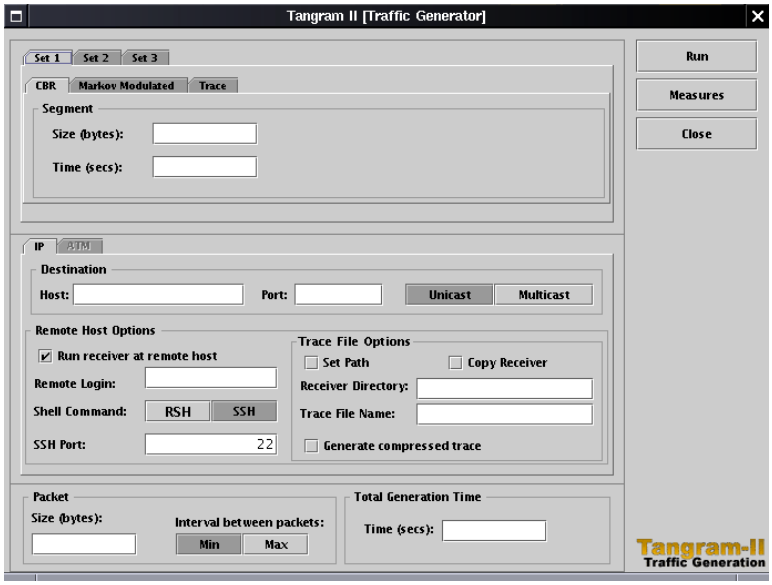


Figure 7.1: Interface Tangram-II to Traffic Generator.

Losses may occur even at local output buffer.

As mentioned above, with *Tangram-II Traffic Generator* many measures of interest may be obtained. To estimate so many metrics, different directions and models for the probe generations are required. These requirements make the tool structure to be directly related to them. The actual tool structure is illustrated by the Figure 7.2.

7.1.1 Using *Tangram-II Traffic Generator*

Before using the *Tangram-II Traffic Generator*, the user must decide what measures of interest he wishes to estimete. The metrics to be estimated depend directly on a correct probe generation. For each set of measures, a certain model and direction of probe generations will be used by the tool. Figure 7.2 shows the possible metrics collected from each type of measurement, and the graffical user interface of the tool is shown in Figure 7.1.

7.1.1.1 Probe Generation Direction

- *One-way Measurement(Set1):* In this measurement format, probes are sent from a source up to a destination. This type of measurement supplies simple metrics as results. As mentioned before, only the parameters not relative to different clocks may be estimated by this way.

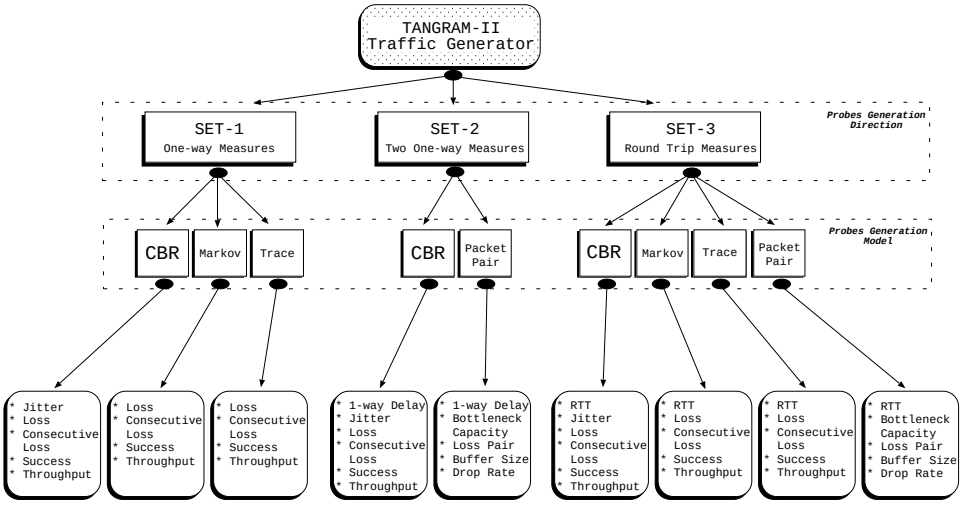


Figure 7.2: Tangram-II Traffic Generator Structure.

- *Two One-way Measurement(Set2):* To obtain measures considering different clocks, it is necessary to send probes in two ways. In this measurement module, two hosts send probes to each other. Two one-way probe generations are started at the same time and in opposite direction. For this reason the method is called *Two one-way*. With the variation in the size of packets, this measurement format allows the application of algorithms to remove Skew and Offset from the traces.

- *Round-trip Measurement(Set3):* In this option, the *Tangram-II Traffic Generator* works like a *Ping* tool, but in an application layer level. The Probes are sent from a network point to another host; when received by this remote host, the packets are echoed (by the application) to the sender again. It is important to note that in this format there is no problem with clocks. The sending and receiving clocks are the same, given that the timestamps are from the same machine.

7.1.1.2 Probe Generation Model

- *Constant Bit Rate (CBR):* In this case, the traffic is generated as a deterministic model and the interval between packet generation is constant. Depending on the probe generation format and the measures of interest, the size of packets may be varied in this model.

- **Markov Modulated:** The traffic is generated according to a continuous-time Markov reward model with finite states space. In this case a reward rate is associated to each state in the model. Let r_j be associated to state j. Then the transmission rate when the model is at state j is r_j bps. The model can be specified using the Tangram-II

modeling environment [6, 20], that generates the state transition rate matrix for the model and associated reward rates. (Any other tool can be used to generate the transition matrix, provided that the file format is that required by the generator.)

The traffic generator tool first obtains the state transition probability between any two states as follows. Let Δ_j be the sum of the output rates out of state j, that is, $\Delta_j = \sum_k \lambda_{j,k}$, where $\lambda_{j,k}$ is the transition rate from state j to k. Then the transition probability from state j to k is $Prob_{j,k} = \frac{\lambda_{j,k}}{\Delta_j}$.

Assume that the system is at the state j. Since the amount of time in j is a random variable exponentially distributed with rate Δ_j, a sample of this random variable is first generated. Clearly, the number of bytes that must be sent till be next transition is $r_j\Delta_j$. The traffic generator then generates a random sample to determine the next state from the state transition probabilities (the $P_{j,k}$) and the process continues till the total generation time is reached.

- *Trace:* The tool can generate the traffic based on a trace file. This trace must be in a specific format, where each line contains in the first collum the amount of time since the last packet sent and in the second collum the packet size. It is important to note that the trace file can be obtained from a simulator or from any real multimedia data such as from a video coded in MPEG. Especific tools can generate these trace files from a MPEG video file. The Tangram-II simulator is completely integrated with the traffic generator and, if this is the simulator of choice, the modeler can generate traffic with different characteristics including long range dependent traffic models such as FARIMA and FBM. Therefore, a wide range of options is available to the user.

- *Packet Pair:* Similar to the CBR generation, this module can generate traffic in a constant interval. But, in spite of only one packet, a pair of them is sent. Also as in the CBR model, depending on the probe generation format, the size of packets may be varied in this model.

7.1.1.3 Generating Traffic Features

Generating CBR traffic from *Set 1* and *Set 2*, the user can specify the size L of the packets to be transmitted, the total generation time T, and the number of bytes for frame D. The amount of traffic is specified in terms of frames. Clearly, the number of packets N generated per frame depends of frame size and packet size, that is $N = \frac{D}{L}$.

The generator supports ATM traffic generation only using *Set 1*. However, IP traffic is supported by all other generation set. In IP option the host name and destination port must be given. In ATM option, VPi, VCi and the *Traffic Classes* UBR or CBR must be specified, For CBR traffic only the peak rate must be input. For Markov modulated traffic, the model name must be provided.

In *Set 1*, the packets generated from CBR and Markov Modulated models a frame can be transmitted in two forms: they are either sent one after another at the beginning of the frame generation interval or each packet is sent uniformly spread over the interval. The choice is made using the option *Interval between packets*. (*min* indicates that packets are transmitted in sequence at the link rate, as shown in Figure 7.3, and *max* indicates that packets are uniformly spread during the interval between two frames, as indicated in Figure 7.4.)

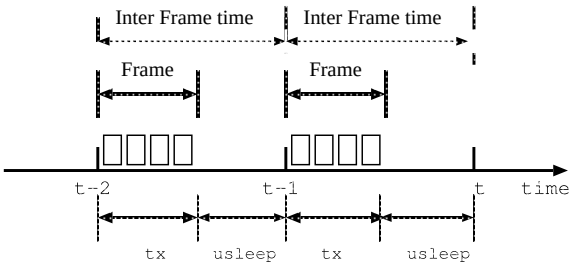


Figure 7.3: Generation mode - *min*

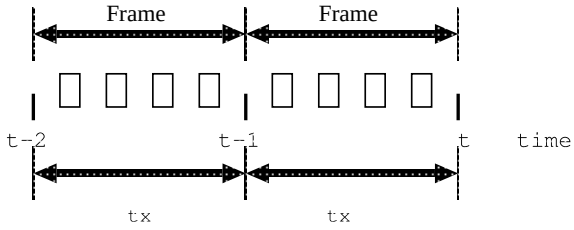


Figure 7.4: Generation mode - *max*

It should be observed in Figure 7.3 that the generation process *sleeps* between two consecutive **frames**. In this case, after transmitting the last packet of a frame, the generator calculates the residual time till the beginning of the transmission of the next frame. If the residual time is positive, then the *usleep()* system call is executed, otherwise the next frame is transmitted immediately. Note that only one *usleep()* system call is executed for each frame, and no *busy wait* is used. The Figure 7.4 shows the packets being transmitted uniformly over the interval between frames. In this case, the generation process sleeps after transmitting each **Packet**.

The user can launch a remote traffic receiver, using the receiver options as shown in Figure 7.1. The IP-Traffic-Receiver captures the packets sent by the traffic generator and creates a trace file, used to calculate statistics. This module requires the destination host and UDP port.

WARNING: Make sure the firewalls of the sender and receiver machines are open for the

chosen UDP port before generating traffic.

The parameters to be given are:

Remote Login - This parameter specifies the user remote login. The user needs a remote account, as well permition to execute the receiver binary `traffgen_recv` on remote host.

Local Trace File Name - The name of trace file that will be generated by the *IP Traffic Receiver*. The trace collected is sent back to the machine that launched the receiver (and generated traffic), and stored in the file `Local Trace File Name`, specified in the parameter.

Shell Command - This specifies the type of remote connection. The user can choose security connection (`ssh`) or just remote connection (`rsh`). Please see the respective man pages for details. The remote machine needs to provide a connection for user's authentication. `ssh` allows configurations to avoid asking for passwords, see tips below.

Receiver Directory - The receiver binary files must be copied to directory at the remote account. If it has already been done, the path must be indicated. Otherwise, the `TANGRAM2_HOME` environment variable will be considered.

Generate compressed trace - The trace file can be sent compressed from the *IP Traffic Receiver* to the local machine if this option is selected.

TIPS: The `ssh` client can perform a login without asking for a password. In the remote user's home create the file `.ssh/environment`, with the line `PATH=/bin:/usr/bin:/usr/local/bin:$TANGRAM2_HOME`. The file `.ssh/known_hosts` should contain the remote host name.

7.2 Traffic Measures

The IP Traffic Measures generate measures of interest based on the trace file created by the Traffic Receiver.

The interface for the IP traffic measures is shown in Figure 7.5. It can be seen from the figure that we can obtain several types of measures: loss, success, throughput, jitter, band capacity, buffer size, drop rate and delay. The measures that can be estimated from a trace depend on the traffic generation model adopted.

7.2.1 Measure Parameters

Primary Trace File - This parameter (the TraffGen output) can be passed by text filed or file chooser (search only TraffGen trace files in the current directory, but it is possible to change it).

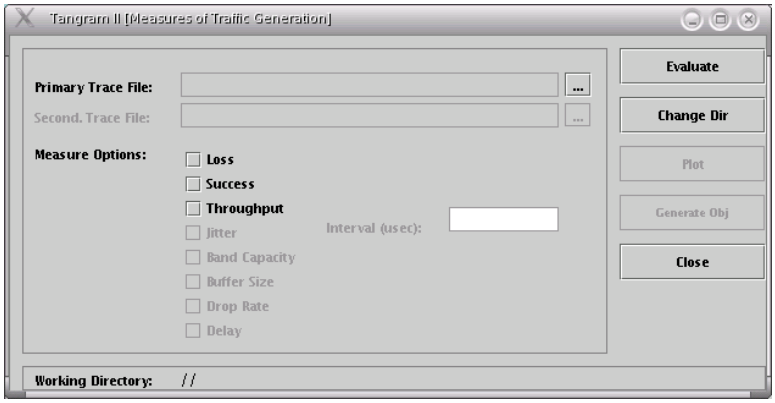


Figure 7.5: Interface of Tangram-II to IP traffic measures

Second Trace File - Depending on the **Primary Trace File** chosen, this parameter becames available. Some metrics, such as One Way Delay, need both traces to be estimated. As above, the file name can be passed by text field or file chooser.

Output File Name - Optional parameter used to substitute `out` in the output file name.

Measure Options - Allows user to choose measures.

Evaluate - Allows user to evaluate the chosen measures.

Plot - Allows to plot evaluated traces.

Close - Return to the Measures window.

NOTE 1: If the packets arrive out of order, they are considered lost. Duplicate packets are discarded.
NOTE 2: The jitter measure is enabled only if the trace was generated with maximum interval between packets (`Max` button at bottom left of the Traffic Generator interface) and using the CBR model.

7.2.2 Plotting the output of measures

The Plot window contains some important characteristics, as shown in Figure 7.6. The top left list displays all traffgen measures in the current directory. Some information about the selected file is shown in the text area. The right side contain some buttons:

Properties - Allows user to configure the graphics.

Delete - Remove selected file

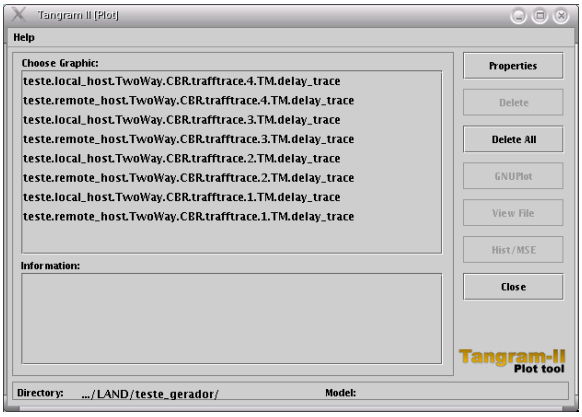


Figure 7.6: Interface of Tangram-II to Plot statistics measures

Delete all - Remove all files in the currently directory.

GNUPlot - Plot the selected trace. Several traces may be selected.

View File - Uses the default text editor to open the selected file.

Hist/MSE - Opens Histogram generation and MSE estimation window.

Close - Return to the Measures window.

7.2.3 Histogram generation and MSE estimation

The interface for the Histogram generation and MSE estimation is shown in Figure 7.7. From this window is possible to plot a histogram from the selected trace. It is also possible to compare the trace distribution with the selected distributions. The user can select if the MSE method for those distributions must be used to select the best distribution to represent the results. The plot selection defines the mode of visualization.

The result of MSE, after estimation, is shown in the textbox. At the bottom left of the window there are some buttons with functions already explained.

7.3 Measuring with *Tangram-II Traffic Generator*

In this section we present a few results obtained by the *Tangram-II Traffic Generator*. Results are presented from each probe generation model. First we demonstrate measures taken from a *one-way* traffic generation. Later, measurements from the *two one-way* set are presented, including offset and skew removal. Round trip measures and comparative results, between *round trip* and the *two one-way* measures, are shown to give an insight about the validity of the offset and skew removal.

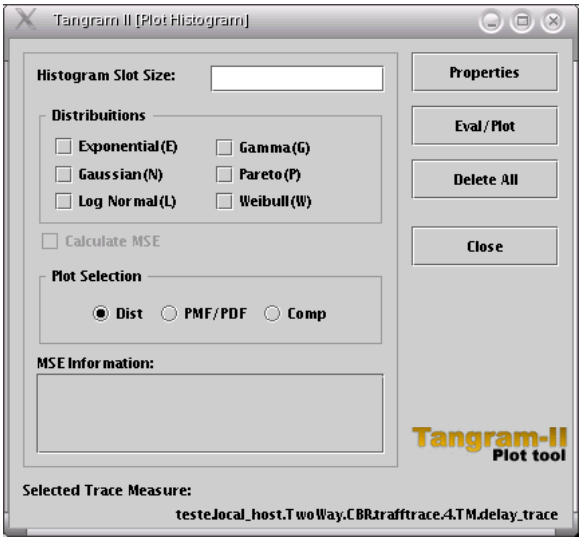


Figure 7.7: Interface of Tangram-II to Histogram genaration and MSE estimation

The only purpose of the results presented here is to show the tool's functionality and its possible parameter estimations. We do not intend to present any conclusive result about the Internet or IP network characteristics.

7.3.1 Measuring in One-way

In this case, the traffic is generated in a single direction. From these traces only a few basic parameters can be estimated. We show that some other measures of interest can be estimated if these traces are collected in parallel with other techniques.

Using Set1, from *Tangram-II Traffic Generator*, traffic was generated from a MPEG trace file. With specific tools, videos in a MPEG format became traces of the frame size of a video file. One of these MPEG traces was used as input in a traffic generation between UFRJ and University of Massachusetts at Amherst. From the collection, the likelihood of consecutive loss and consecutive success was estimated by the tool. These estimations are shown in Figure 7.8, (A) and (B) respectively.

7.3.2 Measuring in Two One-way

Certain measures of interest need simultaneous and opposite directions in traffic generation. In Set2 of the *Tangram-II Traffic Generator* the user is allowed to apply this kind of measure.

From the collections, important measures of interest are estimated by the tool. After collecting the traffic, the estimated delay is usually not accurate, because of the problems

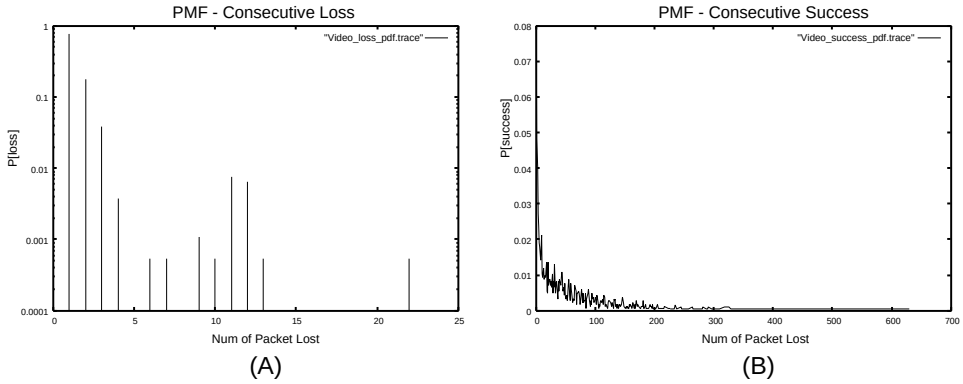


Figure 7.8: PMF of loss (A) and success (B) of videos packets.

with different clocks. These problems are shown in Figure 7.9A. However, algorithms [36, 47] are implemented in the tool to solve the Offset and Skew problems. The one-way delay of the probes, after applied to the algorithms in the initial trace, is shown in Figure 7.9B. A PMF estimation of the delay tried by the probes in both directions is shown in Figure 7.9C.

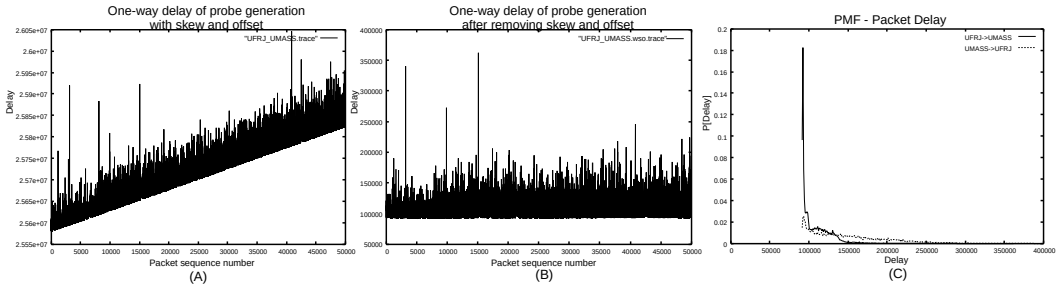


Figure 7.9: Delay calculation of probes generation: (A) with Skew and Offset, from one of the traces (B) After removing Skew and Offset, from one of the traces (C) Histogram of delay probability with both traces.

7.3.3 Measuring in Round Trip

Also implemented in the *Tangram-II Traffic Generator*, the measurement in *Round trip way* allows the user to estimate parameters in this probe generation model. As shown in Figure 4, using this technique it is possible to estimate many measures of interest, but in this case they refer to the coming and going of the probes. Figure 7.10 shows a PMF estimation of the round trip time from several probes sent from UFRJ and echoed by UMASS to the sender again.

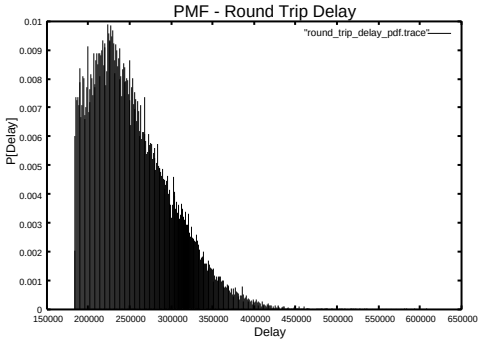


Figure 7.10: Round Trip Delay from packets generated at the same instant of the One-way.

The initial purpose of implementing the round trip module was to compare its results with the one generated by the *Two one-way* set. However, several metrics may also be estimated using this measurement.

We suppose that generating both measurements together may give us an insight about the accuracy of the one-way estimation. In this way, we believe that measuring the round trip delay and the OWD at the same time makes it possible to avaliate the correctness of the Offset and Skew removal. To consider a correct estimation, not only the mean delay obtained from the round trip measurement and the sum of both one-way measurement in oposite directions should be close, but also the convolutions of both PMF curves estimated by one-way delay must be close to the curve generated by round trip delay. Figure 7.11 shows in (A) the closeness of both PMF estimations (RTT and convolution function).

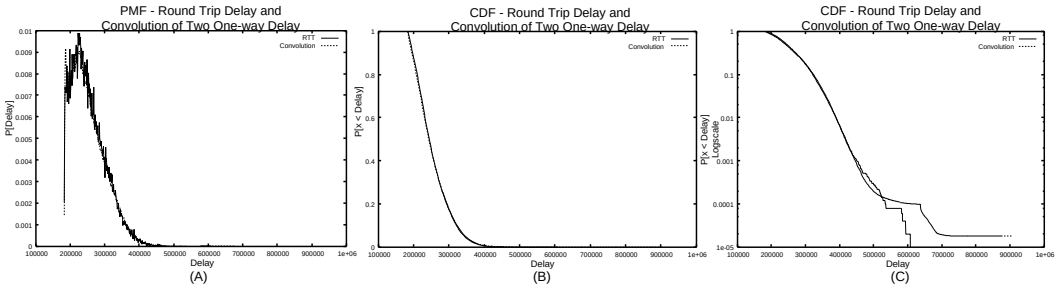


Figure 7.11: Comparison between Delay calculations: (A) PMF, (B) CDF.

Figure 7.11 also shows in (B) and (C) the closeness of both curves, but in this case in the form of the cumulative distribution function (CDF). In (c) a log scale is applied to the y-axis.

7.3.4 Estimating delay distribution

The one-way delay results obtained by the tool may be used to estimate a best distribution for characterizing the behavior of this metric during the experiment. Using the simple method of moments, parameters of the distributions are estimated. The distributions estimated are plotted beside the trace collected in the attempt to identify the best distribution for characterizing this collection. A simple example of this estimation is shown in Figure 7.12, where only three distributions were used. Methods like mean squared error have been implemented to aid in identyifing the distribution with best fit.

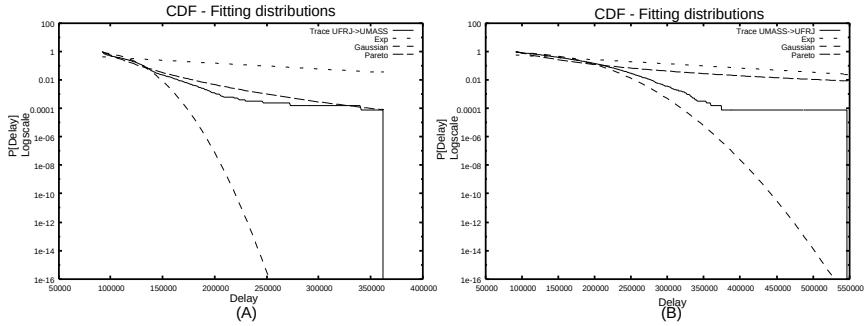


Figure 7.12: Estimating distribution of delay tried by probes: (A) UFRJ-UMASS (B) UMASS-UFRJ.

7.4 References

1. Some important references about traffic generation: [6, 20, 3, 37, 40, 41, 42]

Chapter 8

Hidden Markov Models Module

8.1 Introduction

A hidden Markov model (HMM)[39] is a statistical model commonly used to represent systems whose observable outcomes depend on the system's states, which are, themselves, not directly observable. HMMs have become very popular in several different fields, such as speech recognition, bioinformatics and computer networks.

Tangram-II's `HMM Module` allows users to create and work with two different classes of hidden Markov models: regular hidden Markov models[39] and hierarquical hidden Markov models[44]. The regular HMM has, associated with each state, a probability distribution which determines the symbol's emission in that state. The hierarchical HMM, on the other hand, has associated with each state a Markov chain, which is responsible for the symbol emissions. TANGRAM-II supports four different types of HMMS (the regular HMM and three different hierarquical HMMs), and each is described, in details, in sections 11.5.3, 11.5.4, 11.5.5 and 11.5.6 of this manual.

In this chapter, we explain how an HMM model can be created with TANGRAM-II, and what interesting metrics can be generated with it.

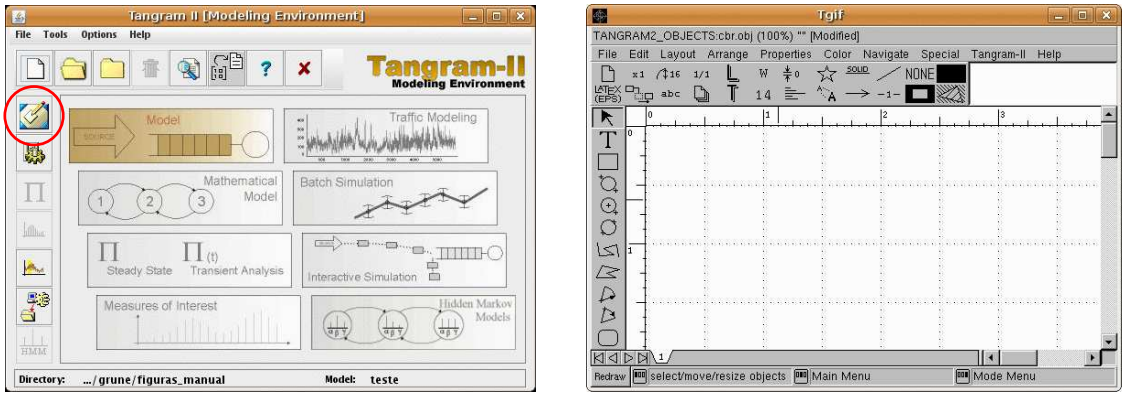
8.2 Creating Hidden Markov Models with TANGRAM-II

When creating an HMM, the most cumbersome job is designing its chain structure (or structures, if the HMM is hierarchical). This is especially true when the chain has a large number of states. Imagine having to specify, manually, every transition of a 30 state hidden Markov model. This means you would have to write $30 \times 30 = 900$ values! For this reason, TANGRAM-II allows the user to create a hidden Markov model chain structure with Tangram-II's `Model Specification Module`, using the objects therein, and load it directly into the `HMM Module`.

The basic idea behind this method is to have the `HMM Module` interpret the state tran-

sition matrix created by TANGRAM-II for the model designed by the user, and, from it, extract the necessary information to assemble the hidden Markov chain. So all the user has to do is build his hidden Markov model in TANGRAM-II, using its objects, events, and messages, have TANGRAM-II generate the state transition matrix that represents it and, finally, have the `HMM Module` load the created HMM model. Next, we will explain how this can be done, using a simple example.

The first step is to open Tangram-II's `Model Specification Module`, shown in Figure 8.1(b). This can be done by clicking the **Model Specification button**, in Tangram-II's Modeling Environment, shown in Figure 8.1(a). Now, we can create our Markov model



(a) (b)

Figure 8.1: Opening Tangram-II's Model Specification Module

just like we would create a normal Tangram-II model. Let's have a look.

Suppose we want to create a hierarchical Gilbert hidden Markov model (a description of this type of HMM can be found in section 11.5.4 of this manual and in the work of [44]) with 30 hidden states, and the following characteristics:

i) Every transition (S_i, S_{i+1}), in the hidden (upper-level) chain, occurs with a 0.5 probability. If S_i is the chain's last state, there is a transition to itself, also with a 0.5 probability.

ii) Every transition (S_i, S_{i-1}), in the hidden (upper-level) chain, occurs with a 0.4 probability. If S_i is the chain's first state, there is a transition to itself, also with a 0.4 probability.

iii) Every transition (S_i, S_i), in the hidden (upper-level) chain, occurs with a 0.1 probability.
Note: The transitions defined here do not expunge the transitions defined in (i) and (ii), for the first and last states. Thus, transition $(S_0, S_0) = 0.1 + 0.4$ *and transition*

$(S_N, S_N) = 0.1 + 0.5$, *where* S_0 *represents the first state and* S_N *represents the last state.*

iv) There are no other transitions in the hidden chain, i.e, all other transitions have a 0.0 probability of occurring.

v) In every hidden state S_i, the transition probability from lower-level state I_0 to lower-level state I_1 is 0.9, and the transition from lower-level state I_1 to lower-level state I_0 is 0.5.

vi) In every hidden state S_i, the probability of starting in lower-level state I_1 is 0.7.

Let's start by building the upper-level chain. To this purpose, we will use the `Markov_Chain` Tangram-II object, illustrated in Figure 8.2. This object defines, by default, the state variable N, which describes the states of the Markov chain, and the constans FWD_RATE, BCK_RATE and MAX_CHAIN_SIZE which describe, respectively, the transition rate from state S_i to state S_{i+1}; the transition rate from state S_{i-1} to state S_i; and the total number of states of the chain. Since the upper-level chain we are looking to build has 30 states, our first step will be to set $MAX\_CHAIN\_SIZE = 29$[1]. Next, we will specify the transition probabilities between the chain's states.

Every event in TANGRAM-II has a rate associated with it, which defines the rate at which the event occurs. **When reading the state transition matrix created by TANGRAM-II, which specifies the rate at which transitions occur, the `HMM Module` will assume that every rate is, actually, a probability.** For this reason, it is easy, for the user, to specify the transition probabilities for the Markov chain he is creating. All he has to do is set the rate of the event which describes the transition to the same value as the probability he wishes that transition to have. Thus, in our case, the event that describes transitions (S_i, S_{i+1}) (the Forward_Transition event) will have its rate set to 0.5, which can be done by assigning this value to the constant FWD_RATE. Similarly, the event which describes transitions (S_i, S_{i-1}) (the Backward_Transition event) will have its rate set to 0.4, which can be done by assigning this value to the constant BCK_RATE. The two remaining events in the `Markov_Chain` object, ZerotoZero_Transition event and NtoN_Transition event, will have to have their rates changed to BCK_RATE and FWD_RATE, respectively, in order to satisfy the chain border conditions described in (i) and (ii).

This takes care of characteristics (i) and (ii). Figure 8.3(a) shows our Tangram-II model so far, and Figure 8.3(b) shows the upper-level Markov chain we have created up to now, by using the `Markov_Chain` object.

To finalize our upper-level chain, there is, still, one thing left to do: address characteristic (iii). To this purpose, we will create a new event, in the `Markov_Chain` object, which we will call `Same_ST`, that does not change the value of the state variable `N`. Its rate will

[1] Remeber that Tangram-II's variables start from 0.

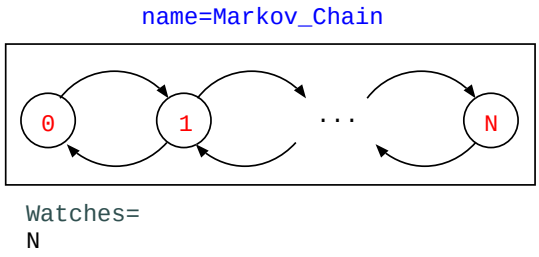
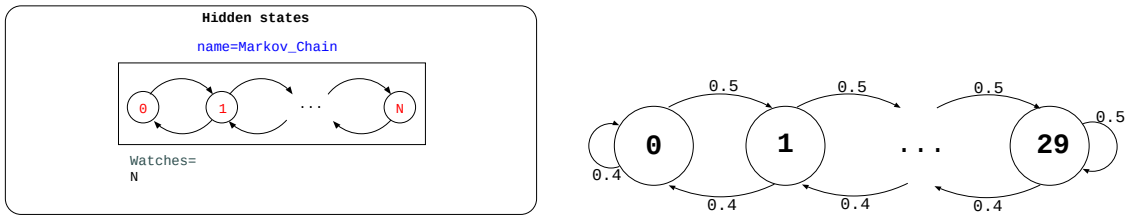


Figure 8.2: Tangram-II's Markov Chain object.



(a) (b)

Figure 8.3: (a)Tangram-II model with the `Markov_Chain` object, and (b) the partial upper-level Markov chain created with it.

have, of course, the same value as the transition probability we specified in characteristic (*iii*), which is 0.1.

```
event = Same_ST( EXP, 0.1 )
condition = ( TRUE )
action =
{
    int n;

    n  = N;

    set_st( "N", n  );
};
```

An so, we have finished building our upper-level Markov chain. Our next step will be to assemble the hierarchical chain, i.e., the Gilbert model that we want to put inside each hidden state. To build it, we will use the `On_Off_Source` object from TANGRAM-II. It will have one state variable, called `Status`, which will indicate the state that the model is currently in (if it is in state I_0 or I_1), and four events. Each event will describe one possible transition of the chain, i.e., transitions I_0 to I_1, I_0 to I_0, etc., and will have its rate set accordingly to (*v*), as shown below:

```
Events=
event = I0_to_I1( EXP, 0.9 )
condition = ( Status == 0 )
action =
{
    int status;

    status = 1;

    set_st( "Status", status );
};
```

This takes care of characteristic (v), and now there is only characteristic (vi) left. In a hierarchical model, every lower-level state has an initial probability, i.e, a probability of being chosen as the starting state, once a transition to its upper-level state has happened. For this reason, the `On_Off_Source` object needs to know when the upper-level chain has made a transition, so it can choose its initial state. In order to do this, we will have to introduce a slight modification in the `Markov_Chain` object events. Every time a hidden transition event happens, it will have to send a message to the `On_Off_Source` object, informing it that a upper-level transition has happened. So, as an example, the `Forward_Transition` event of the `Markov_Chain` object will have a message routine introduced to it as follows:

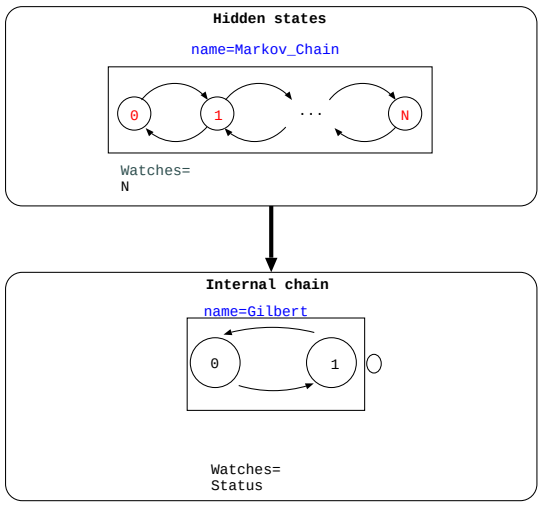
```
Events=
event= Forward_Transition ( EXP, FWD_RATE )
condition= ( N < MAX_CHAIN_SIZE )
action =
{
    int n;

    n = N + 1;

    /* Send message to lower-level chain */
    msg( LOWER_LEVEL_PORT,  all, n  );

    set_st( "N", n  );
};
```

and the `On_Off_Source` object will deal with it in the following way:

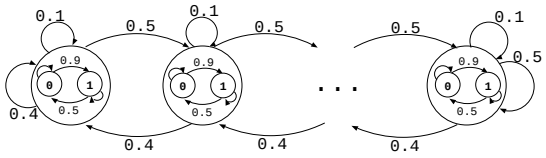
```
Messages=
msg_rec = LOWER_LEVEL_PORT
action =
```

```
{
    set_st( "Status", 0 );
} : prob = 0.3;
{
    set_st( "Status", 1 );
} : prob = 0.7;
```

And we are done! Figure 8.4(a) shows the model we have just created in TANGRAM-II, and Figure 8.4(b) its corresponding hierarchical Gilbert hidden Markov chain structure.



(a)



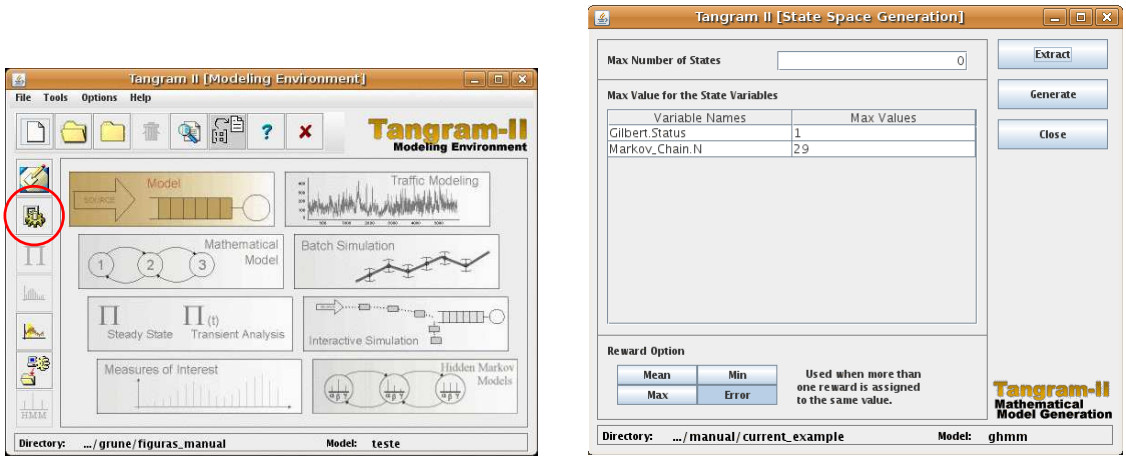
(b)

Figure 8.4: (a)Tangram-II model created. (b)Hierarchical Gilbert hidden Markov model build with the model of (a).

This concludes the structure design part. But before loading this model into the `HMM Module`, TANGRAM-II has to, first, generate the state transition matrix of the model. This can be done with Tangram-II's `Mathematical Model Generation` module, shown in Figure 8.5.

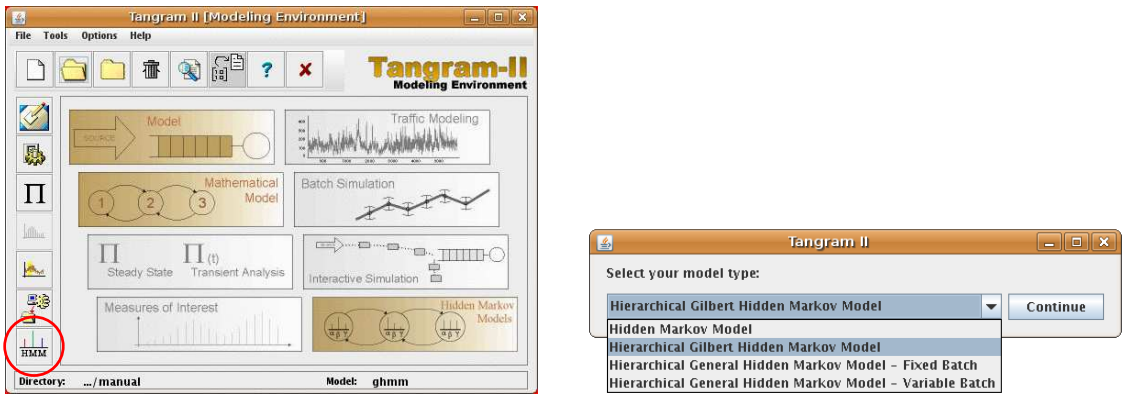
8.3 Loading a Hidden Markov Model into the HMM Module

Once the model is generated by TANGRAM-II, the user may, finally, load it into the `HMM Module`. To open the `HMM Module`, just click in the `HMM Module` button, shown in Figure 8.6(a). This will open the first window of the `HMM Module` interface, which is shown in Figure 8.6(b).



(a) (b)

Figure 8.5: Mathematical Model Generation Module (a) selection button; (b) state space generation interface.



(a) (b)

Figure 8.6: HMM Module (a) selection button; (b) model selection interface.

In the first window of the `HMM Module` interface, the user may select which type of hidden Markov model he wants to work with. As stated previously, the `HMM Module` supports four types of HMMs, described below:

1. **Hidden Markov Model:** a regular HMM which has, associated to each state, a symbol emission probability distribution.

2. **Hierarchical Gilbert Hidden Markov Model**: a hierarchical HMM which has, associated to each state, a Gilbert Markov model, which is responsible for the symbol emissions. This model has a fixed batch size (number of symbols emitted between two transitions in the hidden chain) whose value is determined by the user. An example of this model is illustrated on Figure 11.3.

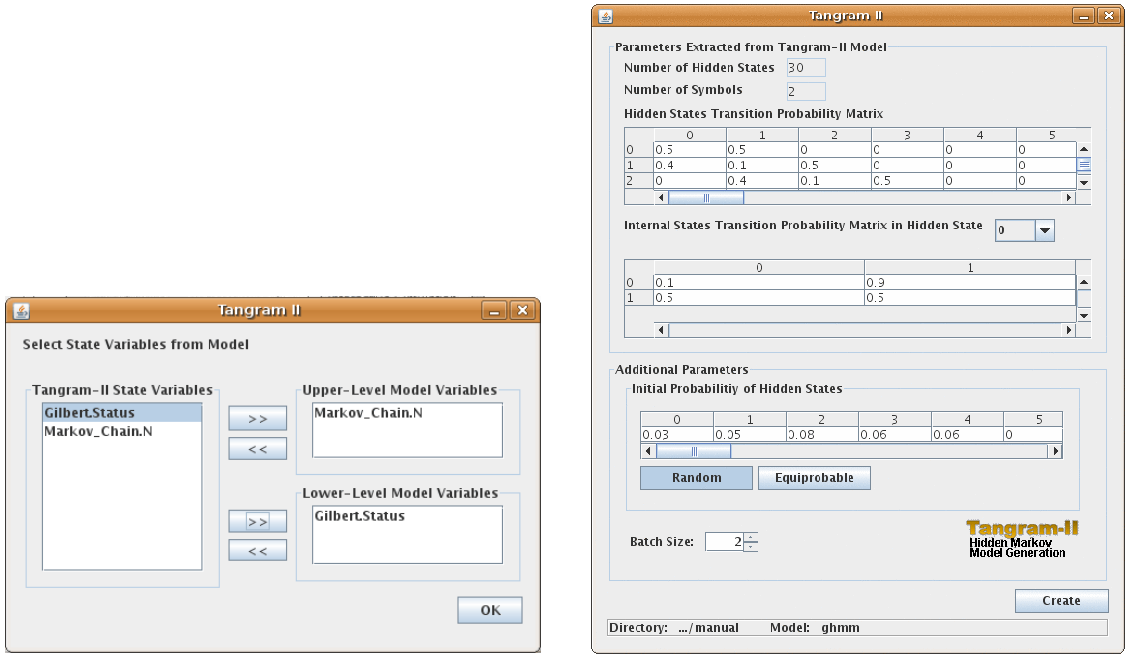
3. **Hierarchical General Hidden Markov Model - Fixed Batch**: a hierarchical HMM which has, associated to each state, a custom Markov chain, which is responsible for the symbol emissions. This model has a fixed batch size (number of symbols emitted between two transitions in the hidden chain) whose value is determined by the user. An example of this model is illustrated on Figure 11.4.

4. **Hierarchical General Hidden Markov Model - Variable Batch**: a hierarchical HMM which has, associated to each state, a custom Markov chain with an absorbing state, which is responsible for the symbol emissions. This model has a variable batch size (number of symbols emitted between two transitions in the hidden chain) whose value is determined by the number of transitions (in the lower-level chain) it takes to reach the absorbing state. An example of this model is illustrated in Figure 11.5.

After choosing one of these models, the `HMM Module` will load the chain structure designed by the user in the `Model Specification Module`. To see how this can be done, let's continue with the example we have been working on.

Recall that, in section 8.2, we designed a hierarchical Gilbert hidden Markov model, whose structure is illustrated in Figure 8.4(b). Thus, when opening the `HMM Module` interface, we will choose the **Hierarchical Gilbert Hidden Markov Model** option, as shown in Figure 8.6(b), and click **OK**. This will open the second `HMM Module` interface, shown in Figure 8.7(a). In it, the user will have to specify which of his Tangram-II model's state variables correspond to the chain's upper-level states, and which correspond to the chain's lower-level states. This is necessary in order to allow the `HMM Module` to correctly interpret the Markov chain created by TANGRAM-II, and, consequently, accurately extract the HMM model's structure designed in the `Model Specification Module`. In our example, recall that we used the state variable `Markov_Chain.N` to represent the upper-level chain, and the state variable `On_Off_Source.Status` to represent the lower-level chains. Hence, we will choose them accordingly, as illustrated in Figure 8.7(a), and click **OK**.

Our HMM's structure is now created. The only thing left for us to do is specify any additional parameters that might be required by the model we chose to work with. This



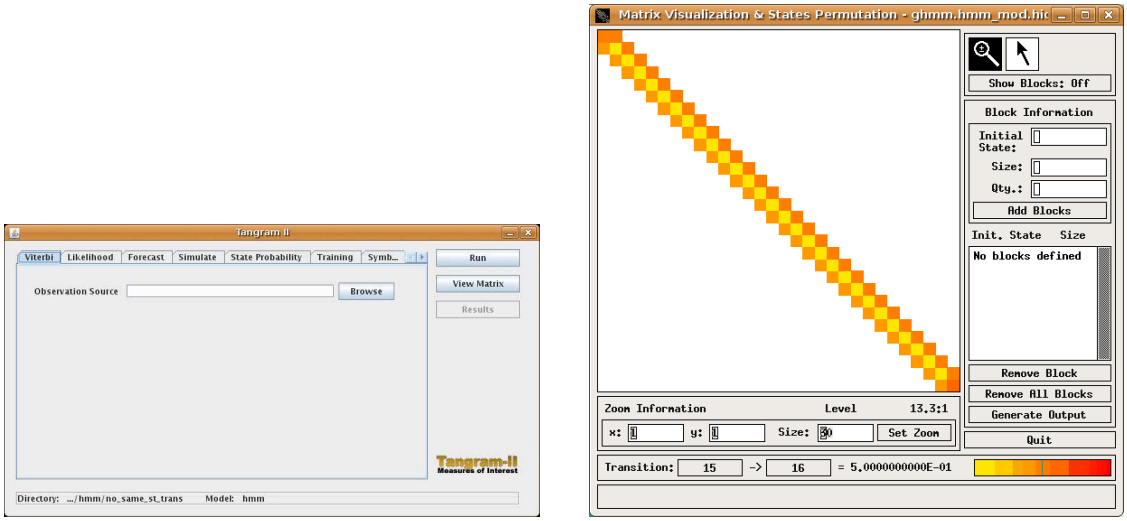
(a) (b)

Figure 8.7: HMM Module (a) state variable selection interface; (b) additional parameter specification interface.

will be done in the third `HMM Module` interface, which is illustrated in Figure 8.7(b). In the case of the hierarchical Gilbert HMM, these parameters are: *initial probability of hidden states* and the *batch size*. Once they are specified, all you have to do is click on the **Create** button, and the model you designed in Tangram-II's `Model Specification Module` will, finally, be loaded into the `HMM Module`.

8.4 Working with the HMM Module

Once the model is loaded, a new `HMM Module` interface, similar to the one illustrated in Figure 8.8(a), will appear. It shows the user all the methods and algorithms that can be used with the specific type of HMM model he chose to work with. These methods and algorithms are the same ones implemented for the corresponding `MTK` plugins, and, for this reason, we will not describe them here. The user who wishes to learn more about each one, should refer to section 11.5 of this manual.



(a) (b)

Figure 8.8: HMM Module (a) method's and algorithm's interface; (b) chain structure visualization.

Let's continue with our example. As stated above, the interface shown in Figure 8.8(a) shows us the available methods and algorithms for the hierarchical Gilbert HMM model. To execute any of them, just click on it's corresponding tab, specify the necessary parameters, and click on the **Run** button. Each method's output will be stored in a file (which is described in appendix **??**), and it's result can be viewed by clicking the **Results** button. The `HMM Module` also allows the user to view every chain's structure of his HMM (both upper-level chain and lower-level chains) using Tangram-II's `Matrix Visualization`

`Module`. All he has to do is click on the **View Matrix** button and select which chain he wishes to see. Figure 8.8(b) shows the hidden chain of the model we have created.

Chapter 9

Examples

9.1 Introduction

This chapter presents simple models to exemplify the modeling paradigm, how to obtain analytical and/or simulation solutions, and how to calculate measures of interest. All the examples described in this chapter are in a directory named EXAMPLES.

9.2 The MMPP/Leaky Bucket Model

9.2.1 Model Description

In this example, we will model a system with three objects:

MMPP source this kind of source can generate packets with different Poisson rates. The Poisson rate depends on the state of the source.

Leaky Bucket this is an access controller to the network. In this model, an event generates credits that will be consumed when a packet generated by the MMPP source is transmitted. These credits are generated periodically (this event has an exponential distribution), and are stored until a maximum of M. When a packet arrives and there are no credits available, the packet is stored in a finite buffer.

Server Queue this object represents a queue with a server that has an exponential service time distribution.

In this model, the MMPP source generates a packet that is sent to the Leaky Bucket and eventually delivered to the Server Queue. If there is a credit upon arrival, the Leaky Bucket routes the packet to the queue. Otherwise, the arriving packet is stored if there is space available in the buffer. Figure 9.1 shows the model.

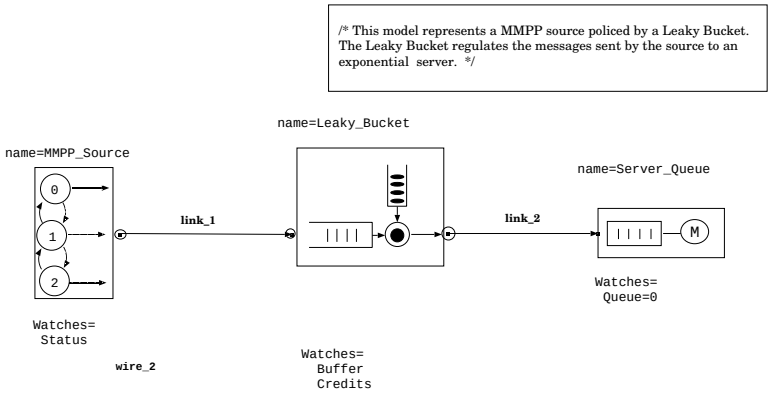


Figure 9.1: The MMPP Model

9.2.2 Solving the Model

In this section, we are interested in the transient behavior of the model. We will solve the MMPP/Leaky Bucket Model using transient-analysis. Our interest is the state probabilities at different time points. This method produces an output file for each interval specified. To solve the model, click on the Analytical Model Solution button. Choose the Transient section and then Point Probabilities. The interface is shown in Figure 9.2.

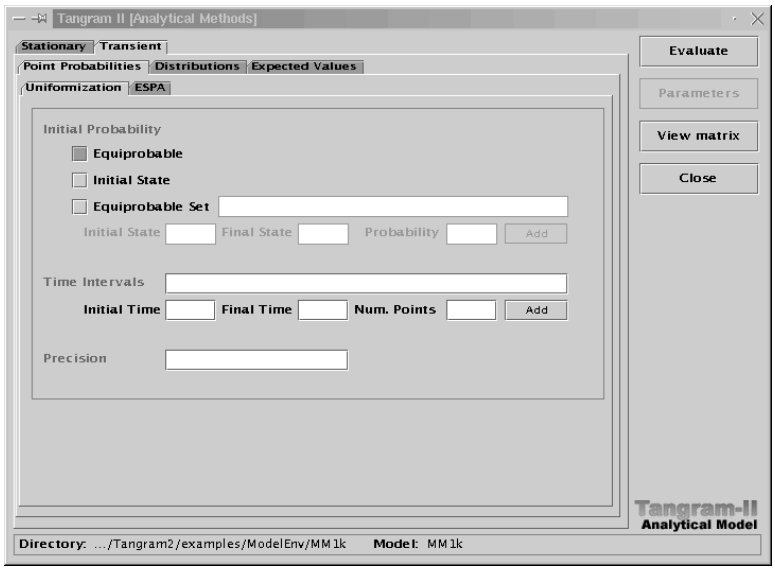


Figure 9.2: The Point Probabilities Method.

In the next step we input the following parameters: Initial Probability, Time Intervals

and Precision. These parameters are important to solve the model.

```
Initial Probability - Equiprobable
Time Intervals: n 0.1 10 10
Initial Time = 0.1
Final Time = 10
Number of points = 10
Precision = 1.0e-05
```

The Point Probabilities method generates files that will be used to calculate the measures of interest (e.g, probability mass function, expected value and so on). The names of the files generated are `<name of model>.<TS>.<pp>.<final time interval>`.

For example, suppose that we want to know the distribution of the number of the packets in the Leaky_Bucket buffer at time $t = 10$. Then it is necessary to use the Measures of Interest Module (PMF of one or more state variables) and choose the appropriate file with the probabilities. This file is generated by the Point Probabilities Method. In this case, choose the file `MMPP.TS.pp.1.0000000e+01` and choose the state variable `Leaky_Bucket.buffer` in the Choose Variables box. Click on the `Evaluate` button, and on the `Plot` button. We can obtain the PMF of the number of the packets in the buffer (Fig 9.3).

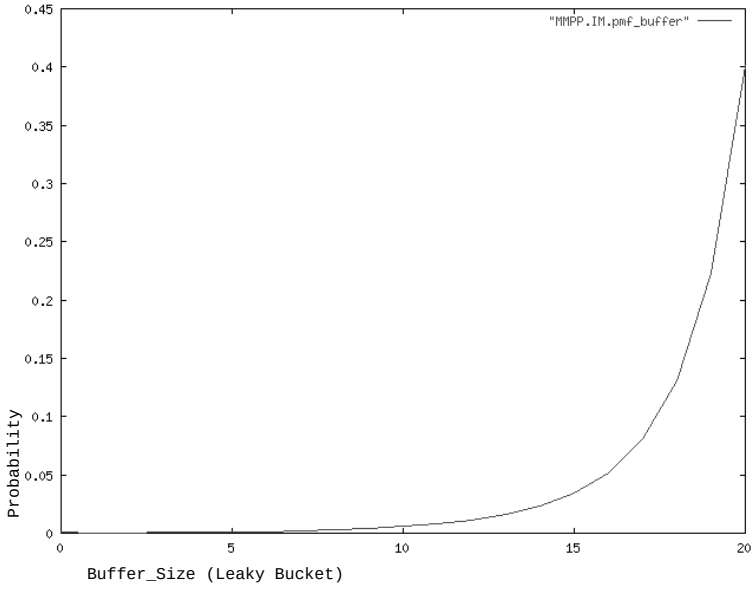


Figure 9.3: The Buffer size PMF

9.3 Model with a Deterministic Server

9.3.1 Model Description

The second model that we present is composed by two objects: an On_Off Source and a finite queue with deterministic service time. This model is shown in Figure 9.4.

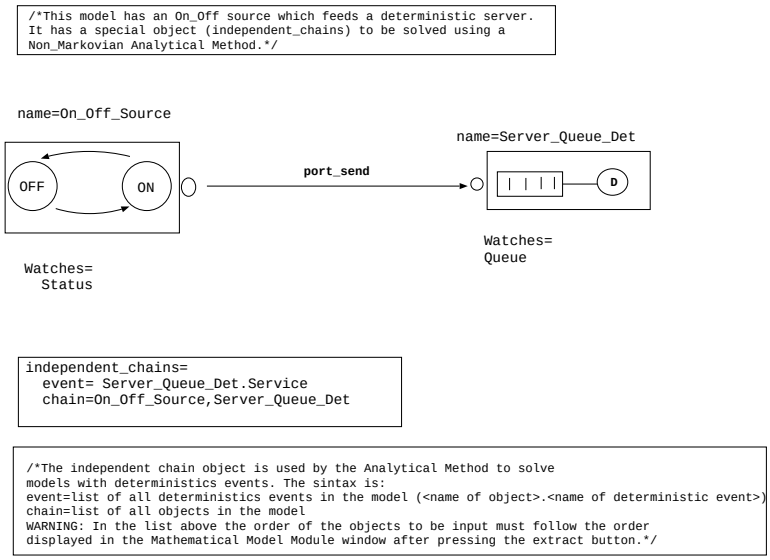


Figure 9.4: The Deterministic Server Model

9.3.2 Solving the Model

To solve the model, click on the Analytical Model Solution button. Choose the Stationary Methods → Non-Markovian Models.

Suppose, in this model, that we want to obtain the fraction of time the queue has n customers. We then need to choose the state variable `queue` to calculate the measure of interest.

Note that the "independent_chains" object must be specified. In this case, there are no objects that behave independently of each other while the deterministic event is enabled, therefore all objects in the model are included in "chains" for this event.

The model results are stored in file `<model_name>.SS.NM.marginal_probs`. This file includes the marginal probabilities of each state variable you select. In the file `<model_name>.SS.NM.expected_cycle_length` the expected length of the intervals between embedded points are stored.

The tool also creates other files that may be useful to the user. For instance, the user can choose (using the interface) to generate the embedded Markov chain used in the solution,

and the steady state solution of that chain. For a complete list of the files generated by the method please refer to appendix A.

9.4 Output queueing Model

The main purpose of this section is to show how to calculate measures of interest with the Measures of Interest module.

9.4.1 Model Description

This model represents an 2×2 output queue ATM switch architecture. In this model, when a packet arrives at the switch it is immediately routed to the appropriate output channel and no blocking occurs in the switch fabrics. We have two On_Off sources that transmit packets to the Switch_2x2 object. When the event Packet_Generation triggers, a source sends a packet to the switch that with probability p routes the packet to queue 1 (first output) and with probability $1-p$ routes it to queue 2 (second output). The On_Off_Source_1 object sends packets using the port named `connection_1` and the On_Off_Source_2 object sends packets using the port named `connection_2`. The model is shown in Figure 9.5.

9.4.2 Solving the Model

After the generation of the Markov chain, we solve for the steady state of the model using the Analytical Model Solution module. To solve the model, click on the Analytical Model Solution button. Choose either the Exact or Iterative Methods.

9.4.3 Measures of Interest

For this model we will pay special attention to how to use the Measures of Interest module. Click on the Measures of Interest module. The interface is shown in Figure 9.6.

For any measure of interest, we must select a file that contains the steady-state or transient probabilities of the model. This file is generated by the solution method chosen in the Analytical Model Solution module. In this example, we use the file generated by the GTH no block Method. We also have to specify the file to store the results that we want to obtain (calculated measures of interest). The Measures of Interest module has three different sections:

PMF of one or more state variables - In this section we are able to calculate the probability mass function of one or more state variables. For this, it is necessary to choose the variable of interest in the Choose Variables box. If we want to obtain conditional probabilities, we must select the Conditional box. Then the conditional pmf of the selected state variables will be calculated. Suppose that we want to obtain

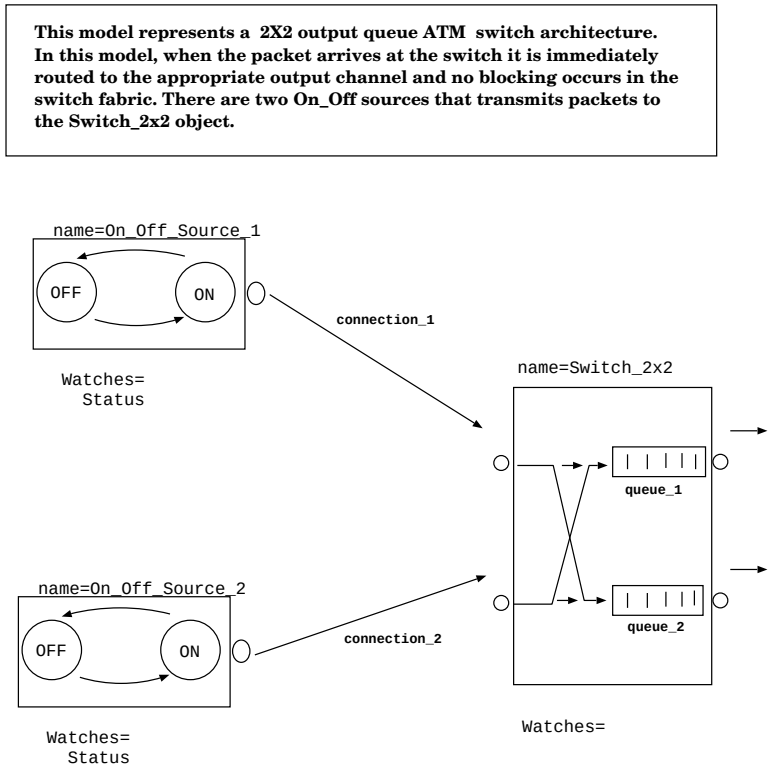


Figure 9.5: The Outputqueueing Model

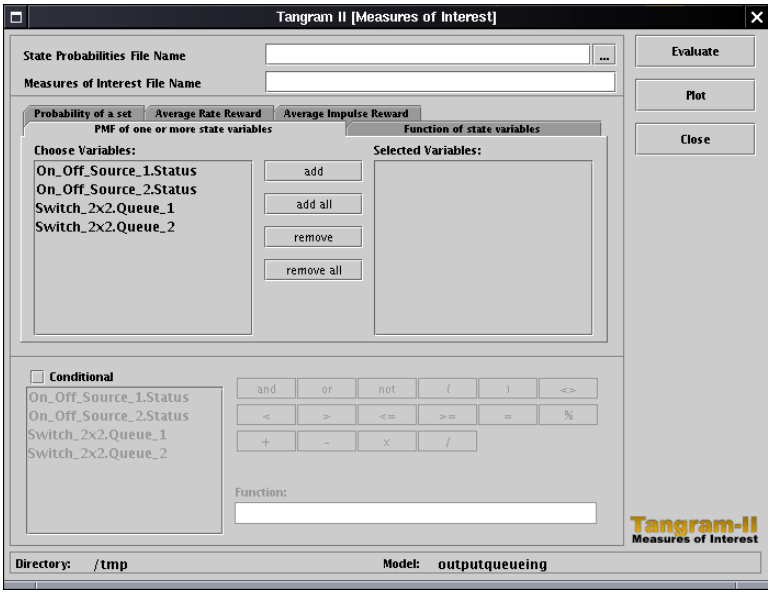


Figure 9.6: The Measures of Interest module.

1. *PMF of the Switch_2x2.queue_1* - Choose the `Switch_2x2.queue_1` variable, select a name for the measure of interest file and click on the `Evaluate` button. To see the result, click on the `Plot` button, select the file generated, and click on the `GNUPlot` button. Figure 9.7 shows the result.

2. *PMF of the Switch_2x2.queue_1 conditioned on the state of the On_Off_Source_1 and of the On_Off_Source_2* - To calculate the PMF of `Switch_2x2.queue_1` conditioned on the On_Off_Source_1 and the On_Off_Source_2 being in the ON state, specify the condition
`(On_Off_Source_1 = 1) & (On_Off_Source_2 = 1)`
IMPORTANT: We must use parentheses to specify functions. If parentheses are not employed carefully, wrong results may be generated.

Function of state variables - Suppose we want to calculate the probability of a function of two or more state variables. For example, we may be interested in the probability that a state variable is equal to three times the value of another state variable. To specify the appropriate function we select the corresponding tabbed pane in Figure 9.6. As another example, suppose that we have two objects in the model with their respective state variables and we want to obtain the probability that the sum of these two state variables is less than a specific value. To do this, we must specify the function `(state_variable_1_name) + (state_variable_2_name) < value`.
If we select the Conditional option, the conditional probability of this function of the

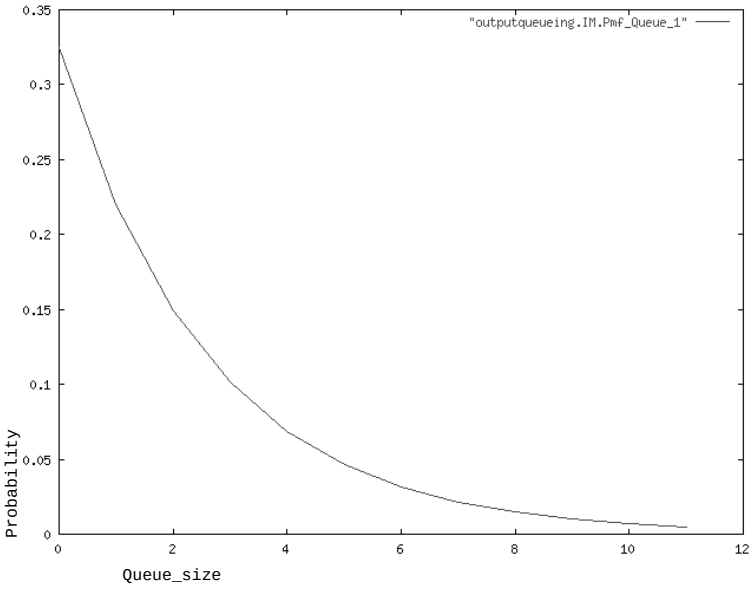


Figure 9.7: The PMF of the Switch_2x2.queue_1 object.

state variables is computed.
Other examples:

- If we want to obtain the probability that the sum of `Switch_2x2.queue_1` and `Switch_2x2.queue_2` is equal to 2, we have to specify the following function: `(Switch_2x2.queue_1 + Switch_2x2.queue_2) = 2`. We can also calculate the probability that `Switch_2x2.queue_1` equals twice `Switch_2x2.queue_2`. This measure is specified by `(Server_Queue_1.queue) = 2*(Server_Queue_2.queue)`.
- Conditional Behavior of Queues 1 and 2 - Suppose that we want to obtain the probabilities above, conditioned on the On_Off_Source_1 being ON. In this case it is necessary to click on the Conditional option and construct the function `(On_Off_Source_1.status = 1)`.

Probability of a set - This section is used when we want to obtain the probability of a set of states. We can also use the Conditional option as in the previous sections. For example, suppose that we want to obtain the probability that the size of Queue 1 is 2 and the size of Queue 2 is 3, given that Source 2 is ON. To do this we specify the function `(Switch_2x2.queue_1 = 2) & (Switch_2x2.queue_2 = 3)`. The condition is `(On_Off_Source_1 = 1)`.

Average Rate Reward - This section is used when we want to obtain the average at

time t or in steady-state. In fact, it is computed as an inner product of the reward vector and the probability vector.

Average Impulse Reward - This section is used when we want to obtain the average impulse reward at time t or in steady-state. We consider the probability that the model is in a state, say s_i, multiply by the probability that occurs an transition to a other state, say s_j, and multiply by the impulse reward. Then we sum over all possible combinations.
NOTE: In "PMF of one or more state variables", "Function of state variables", and "Probability of a set", we can select more than one state probabilities file. In this case, the PMF will generate a specific file for each time interval and a file with the interest measure for all time intervals.

9.5 Traffic Model

9.5.1 Model Description

In this example we model a video source using a birth-death process which feeds a limited queue. In this model, the source generates packets at a rate that depends on the state of the birth-death model. This source model results from the superposition of a number of ON-OFF sources. The video source has two exponential events: Enable_Source, that increases the number of active ON-OFF sources and Disable_Source that reduces the number of active ON_OFF sources. The other event (Packet Generation) occurs if the number of active sources is at least one.

The behavior of the Server_Queue object is very simple: the packet is stored in the queue, if there is space available in the buffer. The service time of the server has exponential distribution. Figure 9.8 shows the model.

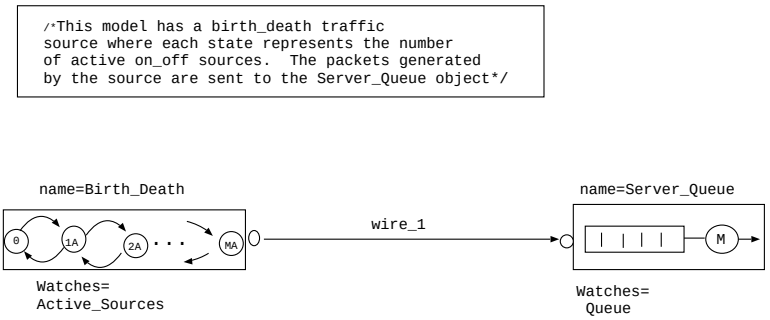


Figure 9.8: The Traffic Model.

9.5.2 Solving the Model

In this example we are interested in the steady state behavior of the system. To solve the model, click on the Analytical Model Solution button. Choose the Exact or Iterative Methods. In this case, you can choose any method.

After this step, we are able to calculate the measures of interest. For example the user can obtain the PMF of the number of active sources, and the probability that the number of on-off sources is greater than a given value conditioned on the queue being at its maximum value.

Choose the Measures of Interest module. Then choose the PMF, select the state variable `active_sources`, and click on `Plot` to obtain the first measure above.

Now assume that you want to solve this model by simulation. Please refer to chapter 3 and the section on messages and events to learn about the event execution process. The model above can be immediately simulated. Recall that an enabled event E is not re-sampled if another event triggers and E remains enabled in the new state. The following situation occurs in the example above. Assume that only one source is active. Then 2 events of the object source can trigger in this state and 2 samples are in the event list. Assume that the event corresponding to the disabling of a source triggers. The event corresponding to enabling a source remains in the event list. However, this event was generated with rate $(\texttt{active\_sources} - 1) \times \alpha$. When it triggers, the new state will be one active source and therefore no event with rate $\texttt{active\_sources} \times \alpha$ will ever be generated. The user should follow the steps in chapter 3 to build a model that generates new samples of events after an action (even if the events remain enabled), and compare the resulting models.

9.6 Set Cumulative Rewards Values

Reward models can be easily defined in TANGRAM-II. As described in chapter 2 an extremely rich set of measures can be defined using rate-rewards (which are associated with states in the model) and/or impulse-rewards (which are associated with state transitions). We recall that we can attribute values to a reward by defining conditions over the state variables associated to an object. Furthermore, global rewards can be defined, and their value set when boolean conditions over state variables from different objects are satisfied. The reward values accumulated can be modified when an action is executed, as we will see in the example below.

9.6.1 Model Description

The main purpose of this example is to show how the cumulative value of a reward defined for an object can be modified by the execution of an action. The main function to accomplish this is called `set_cr` . We can also obtain the current value of the amount of

reward accumulated using the `get_cr` function. `Note`: these functions can be used only in simulation.

The model is shown in Figure 9.9. In that figure, a source sends packets to a finite exponential queue. The behavior of the source is described in Figure 9.10. While in state where the variable state is equal to zero, the source generates packets at an exponential rate. From Figure 9.10, the state variable status can only change its value when the elapsed time since the last change is at least 10. This elapsed time is stored in the reward `elapsed_time`.

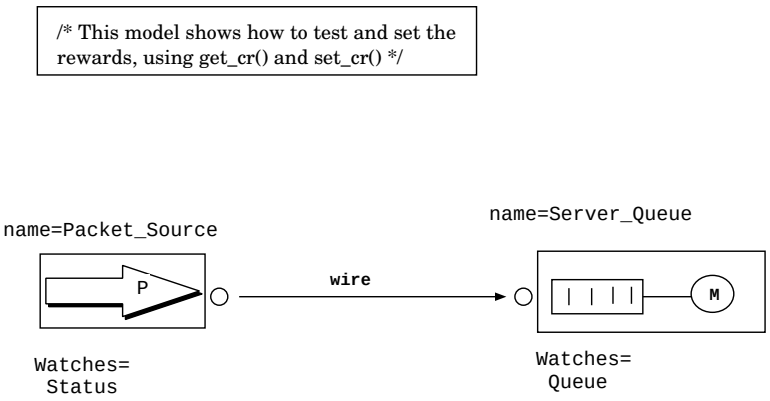


Figure 9.9: Set Cumulative Rewards Values.

The description of the Server_Queue object is shown in Figure 9.11.

NOTE: `get_cr` gets the total value of the corresponding rate reward accumulated so far, and `set_cr` sets the cumulative rate reward value. We can use `get_ir` to get the total value of the corresponding impulse reward accumulated so far and `set_ir` to set the impulse reward cumulative value. Again, these functions can be used only in simulation.

DEBUGGING TIP: If you need some information about the value of any variable during simulation you can use `set_cr` to generate a trace containing the values.

Imagine you want to know the values assumed by the queue during simulation. Suppose this state var is called QUEUE. How to do it step by step:

1. Specify a reward like this:

```
rate_reward = NAME_RELATED
condition=(FALSE)
value=0;
```

   This reward will not accumulate by itself. Instead, you need to specify its value with the `set_cr` command.

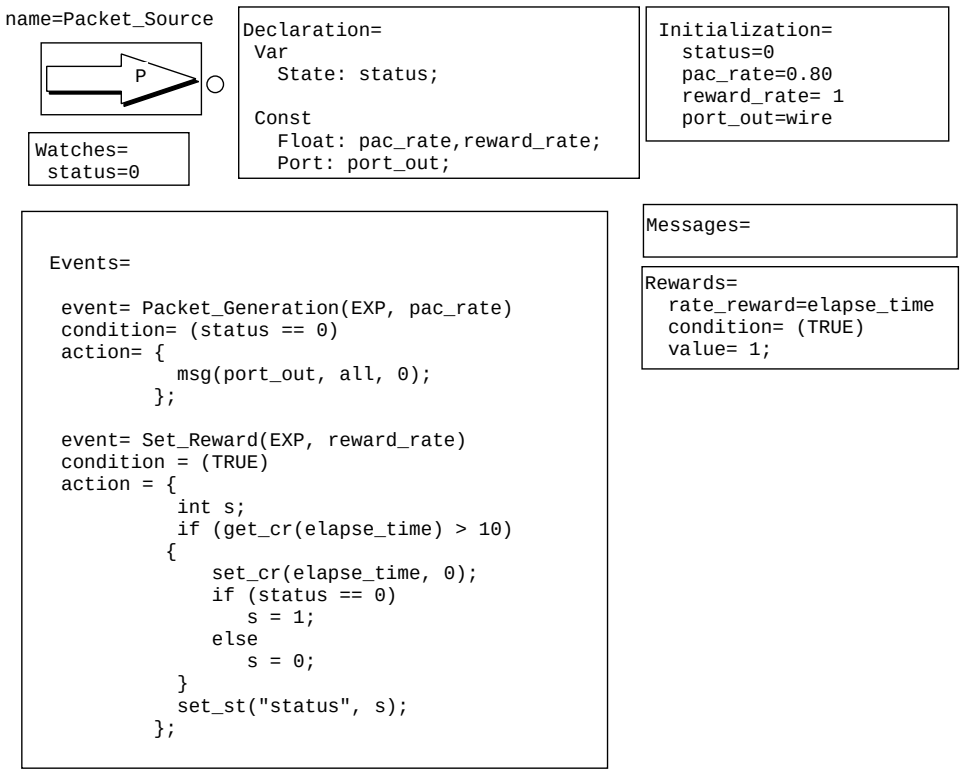


Figure 9.10: The Packet Source object (Set Cumulative Rewards Values).

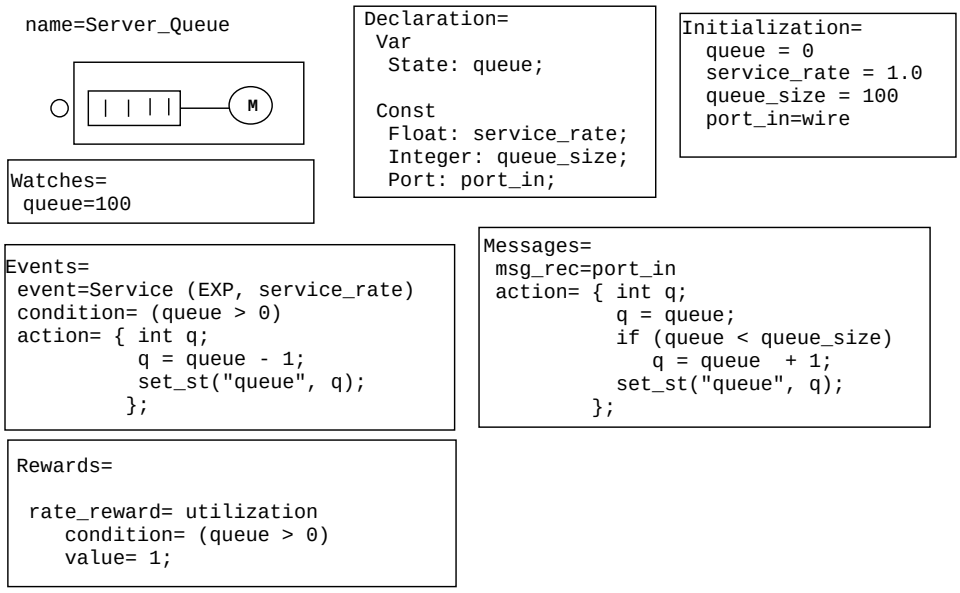


Figure 9.11: The Server Queue object(Set Cumulative Rewards Values).

2. Inside the code (message or event of the same object) you put the following command: `set_cr( NAME_RELATED, QUEUE)`. This command will modify the cr value of the reward called `NAME_RELATED` to `QUEUE` value and will plot the new value in a trace file under condition 3.

3. To generate the trace file you need to press the trace button at the Reward Option (button) in the Simulation window. You can visualize the values as text in any editor or plot it with Gnuplot.

In order to make the debug action and the analysis of the simulation time easier the following commands can be used:

1. `get_simul_time()`. This command is used to get the current simulation time (float). Examples:

```
(1) var_float = get_simul_time();
(2) if (get_simul_time () > var_time)
    {
      do_something;
    }
```

2. `printf`. This command is similar to the print command found in the C language. Example:

```
printf ("Debug message1: %d %f \n", int_var, float_var);
```

3. `fprintf`. This the same as the `fprintf` found in the C language. Example:

```
fprintf (stderr, "Debug message2: %d %f \n", int_var, float_var);
fprintf (stdout, "Debug message3: %d %f \n", int_var, float_var);
```

9.7 Event Cloning

Event-cloning is useful when more than one sample of an event has to be generated when an action is executed. Models that contain queues with multiple servers benefit from this construct.

9.7.1 Model Description

In this example, we have two objects: a Poisson Source and an Infinite Server Queue. The service event has a uniform distribution . Each message sent by the source and received by the Infinite Server (indicating the arrival of a new packet), generates a new sample of the event `Service`. Figure 9.12 presents the model and Figures 9.13 and 9.14 show all the attributes of each object.

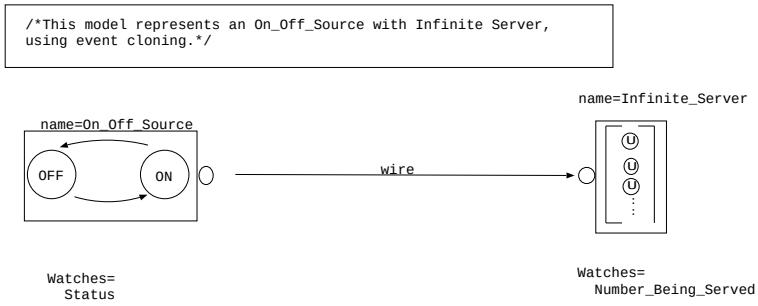


Figure 9.12: Event Cloning Model.

IMPORTANT It is important to understand how samples of an event are generated when we clone an event. Let us recall how samples are automatically generated by the simulator, when *no* cloning is used in the model. A new sample is generated whenever: (a) an event is enabled (from a disabled condition) and (b) whenever an event executes, provided that the conditions for the event to occur remain satisfied. Note that only one sample is active at a time. When the `clone_ev()` executes in an action, the following rule is used to generate a sample :

> A new sample is generated after `clone_ev()` is issued when the current object state is such that the event that will be cloned is enabled *and* it remains enabled after the action is executed.

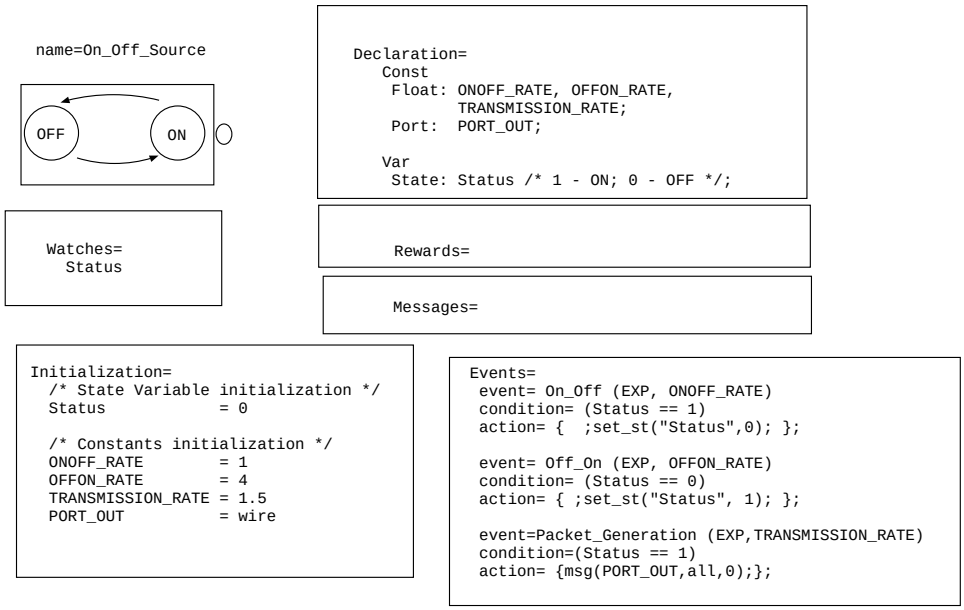


Figure 9.13: The ON_OFF Source object (Event Cloning Model).

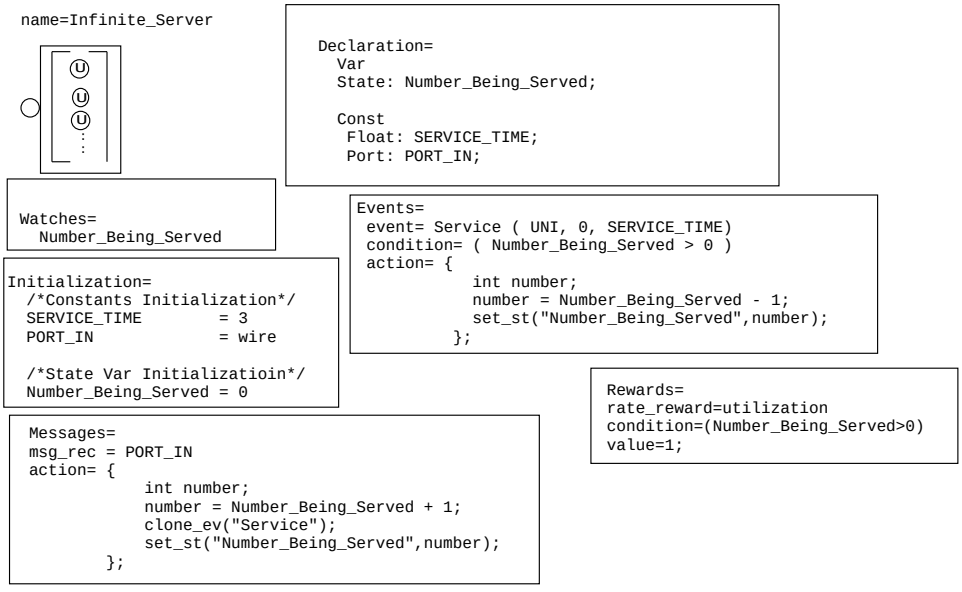


Figure 9.14: The Infinite_Server object (Event Cloning Model).

One can generate as many samples as specified during the execution of an action.

When cloning is used the simulator only generates automatically a new sample when condition (a) above is true *or* condition (b) *and* this is the last sample to execute.

It is also important to note that when the conditions for an event to occur become false, all its samples in the simulator event queue are immediately discarded.

Consider the infinite queue example in this section. When a message is received by object `Infinite_Server` and the `Infinite_Server` queue is empty, a sample of the `Service` event is generated and the event becomes enabled. Note that no other sample of `Service` is generated, since the `clone_ev()` command has no effect when the action is executed. This is because the event `Service` is not enabled when the message is received.

Assume now that the `Infinite_Server` queue has more than one packet queued, and more than one samples in the event queue. When any of the generated samples triggers, the corresponding action is executed but no other sample is automatically generated, since there are other samples in the simulator event queue. The last sample that triggers resets the state variable `serving` and the conditions for `Service` to occur become false. See also the MDk1 example.

9.8 Multiple action

Several actions may be associated with an event or a message, each having a different probability of occurrence. Of course, the probabilities must add to 1. qThe probabilities are specified using the reserved word **prob** followed by an arithmetic expression.

EXAMPLE: Assume we want to model a faulty link which corrupts a packet with probability $1 - p$. Upon arrival the packet is accepted and queued with probability p and discarded with probability $1 - p$.

```
action= {
   /* a packet with no errors has arrived */
   int queue;
   queue = Queue + 1;
   set_st("Queue", queue);
} : prob = p;
{
   /* a corrupted packet has arrived, do nothing */
   ;
} : prob = 1 - p;
```

9.8.1 Model Description

To illustrate multiple action constructs, we choose the model of Figure 9.15. In this model an arriving packet is directed to one of the four service centers. The service center is chosen

as follows: first, two out of the four queues are randomly chosen. Then the service center which has the smallest queue (from the two chosen) is selected.

In this model there are three types of objects:

1. Poisson Source - this object generates and sends packets to the split object.
2. Split - this object randomly selects two out of the four queues and sends a message to them informing that there is a new packet to be served. Note also that we tagged one of the two chosen queues with the same probability. This is used in case the queues have identical sizes.
3. Queues - There are four queues that store and serve the packets. The service centers that receive a message from the split node also exchange control messages to determine which one has the smallest queue. Note that the "data" sent by the split node indicates the center that must start the comparison process.

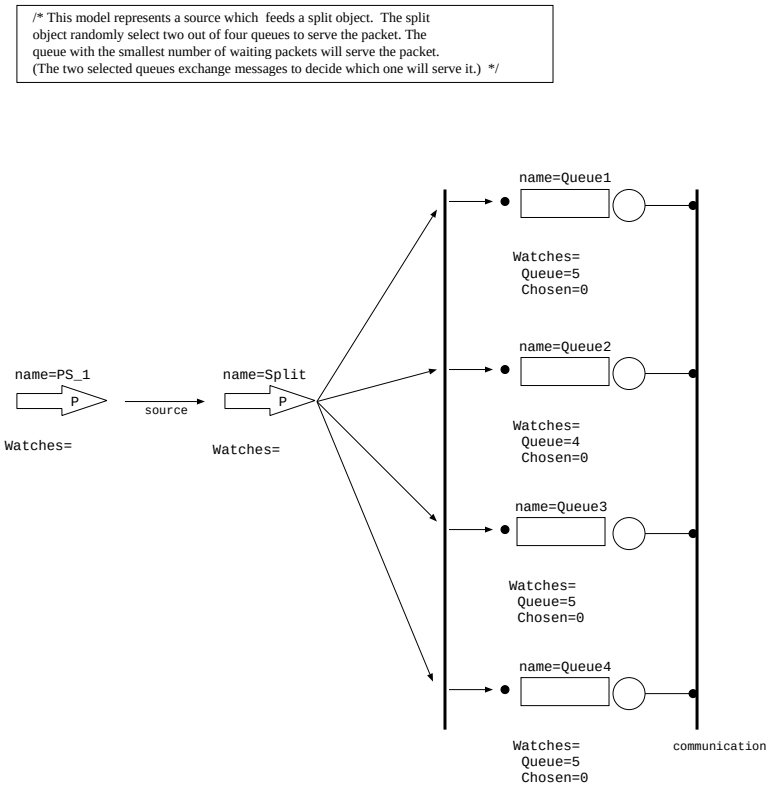


Figure 9.15: The Multiple Action Model.

The description of the objects is shown in Figures 9.16, 9.17 and 9.18.

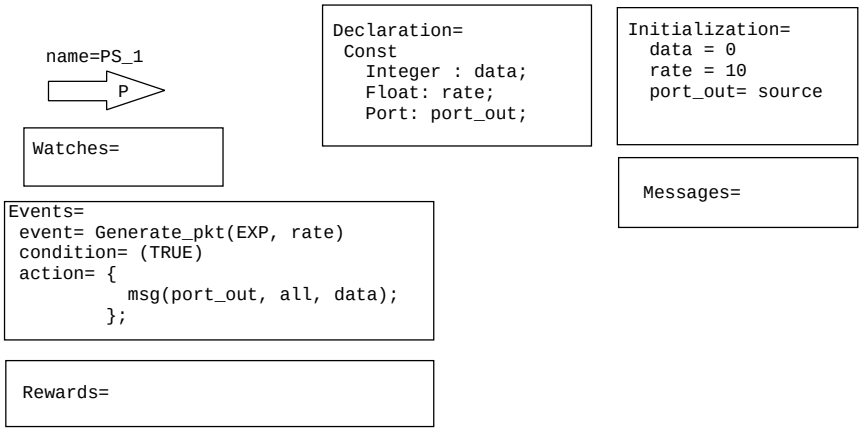


Figure 9.16: The Poisson Source object (Multiple Action Model).

9.9 Model with Symbolic Parameters

As mentioned previously, the tool supports the use of symbolic parameters. Symbolic parameters can be used in expressions that specify a distribution, expressions that specify the probabilities of an action, and expressions that define the value of a reward. For example, the rate of an exponential distribution (EXP) can be specified using an expression that contains a parameter. In the same way, the probability of an action can be specified using an expression that contains a parameter. Also, when specifying rate or s, the value of the reward can be specified using an expression that includes a parameter. Symbolic parameters cannot be used in conditions or inside actions.

Parameters are useful for sensitivity analysis . Suppose that we want to study the influence of different event rate values. Then, if we define the rate of events as a parameter, the chain will be generated only once. The transition probabilities of the chain are defined as functions of the parameters. The user can choose a value for these parameters and the transition probability of the chain is easily calculated by the tool without recomputing the Markov chain. Parameters are set in the interface of the Analytical Model Solution. In the proper option of the interface, the user can define the numerical values for the parameters and generate the chain. The rewards are also calculated after the specification of the parameters. It is necessary to specify all parameters before solving the model.

If simulation is used, no symbolic parameters can be specified. When simulating a model, the state space is not generated and the simulator calculates the samples for the events and the probabilities of the action based on the numerical values given.
IMPORTANT: A model with parameters can not be simulated!

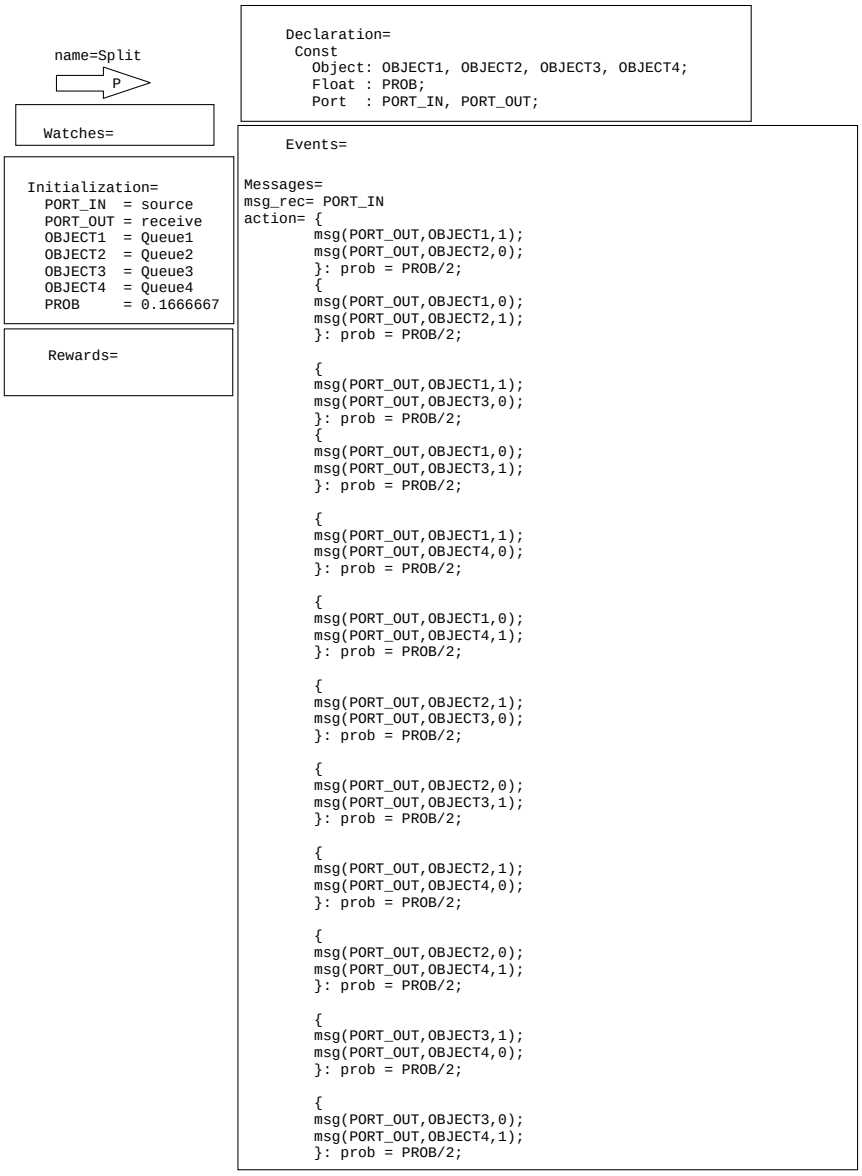


Figure 9.17: The Split object (Multiple Action Model).

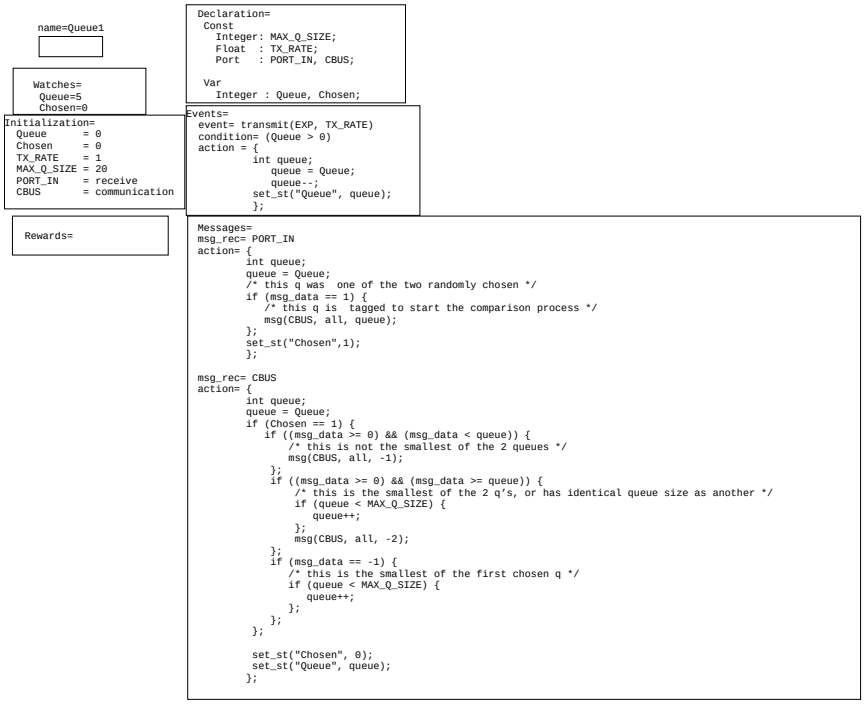


Figure 9.18: The Queue object (Multiple Action Model).

9.9.1 Model Description

We use the M/M/1/k model to exemplify the use of symbolic parameters. We only change the declaration of the exponential rates. For the two exponential events, we declare in the `Declaration` attribute

```
Object: Poisson_Source
Declaration=
  Const
    Port: PORT_OUT;
  Param
    Float: pkt_rate;

Object:Server_Queue
Declaration=
  Const
    Integer: QUEUE_SIZE;
    Port: PORT_IN;
  Var
    integer: Queue;
  Param
    Float:service_rate;
```

We must modify the `Initialization` attribute of the objects, removing the `pkc_rate` and `service_rate` initialization. Figure 9.19 shows the model.

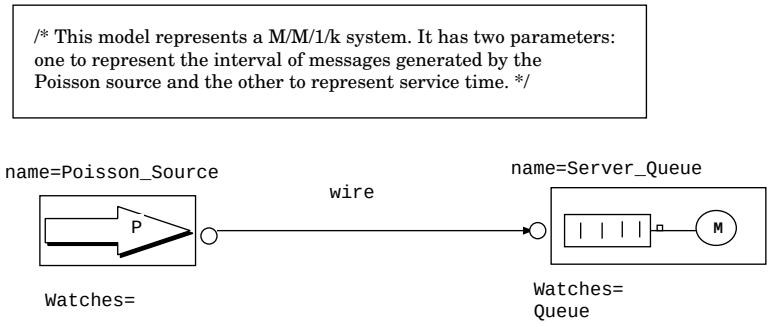


Figure 9.19: The MM1k Model with Symbolic Parameters.

9.9.2 Solving the Model

The main difference between the solution of the model with symbolic parameters is that it is necessary to give the values of all symbolic parameters before solving the model. Figure

9.20 shows the interface for setting the parameter values.

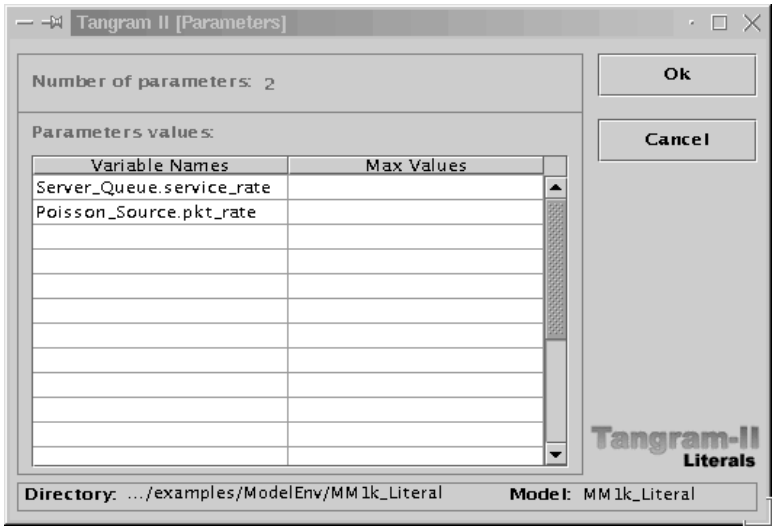


Figure 9.20: The Symbolic Parameters Window.

We can then solve the model using any analytical solution method.

9.10 Gated Queuing Vacation Model

9.10.1 Model Description

This model describes a system with a Poisson source that generates packets to a queue that serves them if and only if it has a token (generated by the `Token_Passer` object). When the queue receives the token, it closes a "gate" and serves the packets until a timeout expires. Note that the queue continues receiving packets, but they are stored behind the gate. If packets stored after the gate is closed are not served, they must wait until the next arrival of the token; see Figure 9.21.

The description of the objects is shown in Figures 9.22 and 9.23.

9.10.2 Solving the Model

To solve the model, click on the Simulation Module. For this model, we will use batch-simulation. The Stop Condition can be the `G_Queue.Service` (Event Stop Condition), for instance, with 140 triggers or Time 1200. In this case, the simulation will finish when one of the conditions is true.

An interactive-simulation can also be performed. Using interactive simulation we are able to observe the simulation step by step, and to follow the evolution of the model states.

```
/*This model describes a system with a Poisson source
that generates packets to a queue. The queue serves the packets if and
only if it has the token (generate by the Token_Passer object). When the
queue receives the token, it closes the "gate" and serves the packets until
the timeout expires (the queue continues receiving packets, but they are
stored behind the gate).  If some packets are not served, they must wait the
next token arrival.*/
```

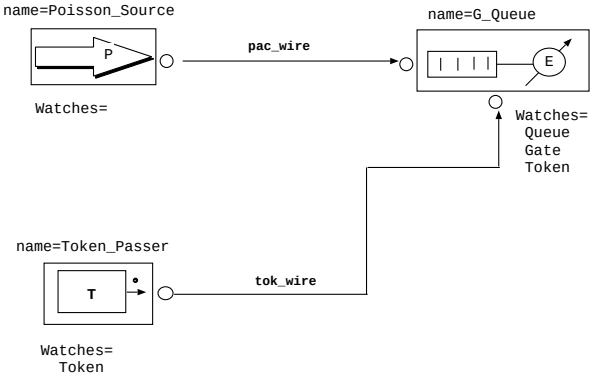


Figure 9.21: The Gated Queueing Vacation Model.

Figure 9.22: The Gated Queue object (Gated Queueing Vacation Model).

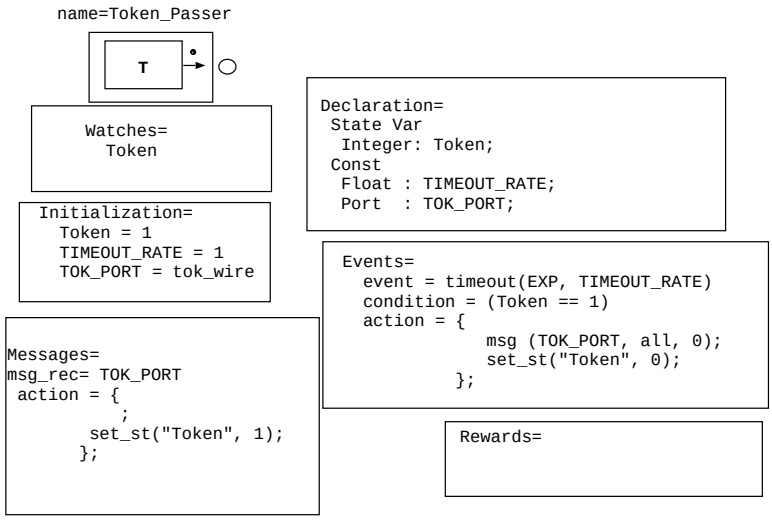


Figure 9.23: The Token Parser object (Gated Queueing Vacation Model).

To use the Interactive Simulation, choose the Interactive tabbed pane. Set, for example, `Number of transitions` to 10. So the state variables of the model can be visualized (on the TGIF screen) every 10 model transitions.

In order to visualize the state variables, the user should include in the `Watches` attribute the name of the state variables that he/she wishes to observe. If no state variables are included in the `Watches` attribute for an object, the simulator does not display any state variables for that object.

NOTES

1. There is a limitation on the total number of variables displayed during an interactive simulation. In this version, the total number of variables that can be observed at once is 200.

2. The user can change the variable that is been displayed at any time during the interactive simulation.

3. The `Watches=` attribute must exist if an interactive simulation is performed, even if no variable is associated with it.

4. The state variables are re-written on the TGIF canvas at each interactive simulation step. The color and font size and style used during state variable re-writing is that currently set in TGIF. However, they can be changed after a step is executed.

5. The interactive simulation stops after the execution of an event. If simultaneous events occur, the simulation stops after each one is executed. The user should observe the simulation clock (top left corner of canvas) to see the model evolution with time.

9.11 Vector Variable Model

Not only integer-valued state variables can be used in TANGRAM-II. This example shows how we can use a vector-variable. The vector variable is used as a state variable and the behavior is identical to the C language. When a vector variable is used we must pay attention to the following:

1. The first index of the vector is zero (If we declare a vector with five elements, the indices of these elements are (0,1,2,3,4)).

2. An integer variable, for example i, has to be used as an index of the vector state variable ($N[i] = \ldots$)

3. We can initialize the vector state variable as follows :

   (a) `N[] = 1` (all positions of the vector are equal to 1).

   (b) `N = [1,2,3,4,5]` (each position receives a specific value).

4. Note that all entries must be initialized.

5. When we want to copy a state var vector to a auxiliary variable, we must use the `get_st` function. The syntax is: `get_st(auxiliary_var,state_var)` .

6. We can specify the dimension-of-the-vector-variable using a Integer constant type. In this case, we must specify the Const part before the Var part in the Declaration attribute of the object. In the Initialization attribute, we must specify the dimension of the vector before the initialization of the vector variable. If we initialize each position with a specific value, it is not necessary to follow this order. See figure 9.26 for more details.

7. To generate the mathematical model, we must specify the maximum value of the variables (in State Space Generation Module). In the case of the vector variable, this parameter is the maximum value considering the value in each vector's position.

8. During the simulation, we are able to see the evolution of the state vector variable. For example, if we have a vector with 100 positions and we want to see all positions between 0 and 10, we have to do specify in the `Watches` attribute
   `name_vector_variable[0:10]`. If we want to see another interval, we have to do the same:
   `name_vector_variable[20:50]`.

9.11.1 Model Description

This model is very simple: there are two Poisson sources that send packets to a FIFO queue. The queue has a limited buffer with size equal to 10. We use a vector state variable with 10 positions to represent the buffer of the queue. Each position of the buffer stores the `id` of the source which generates the packet. When the service event triggers, the packet in the first position of the queue is served and all other packets are shifted. The model is shown in Figure 9.24.

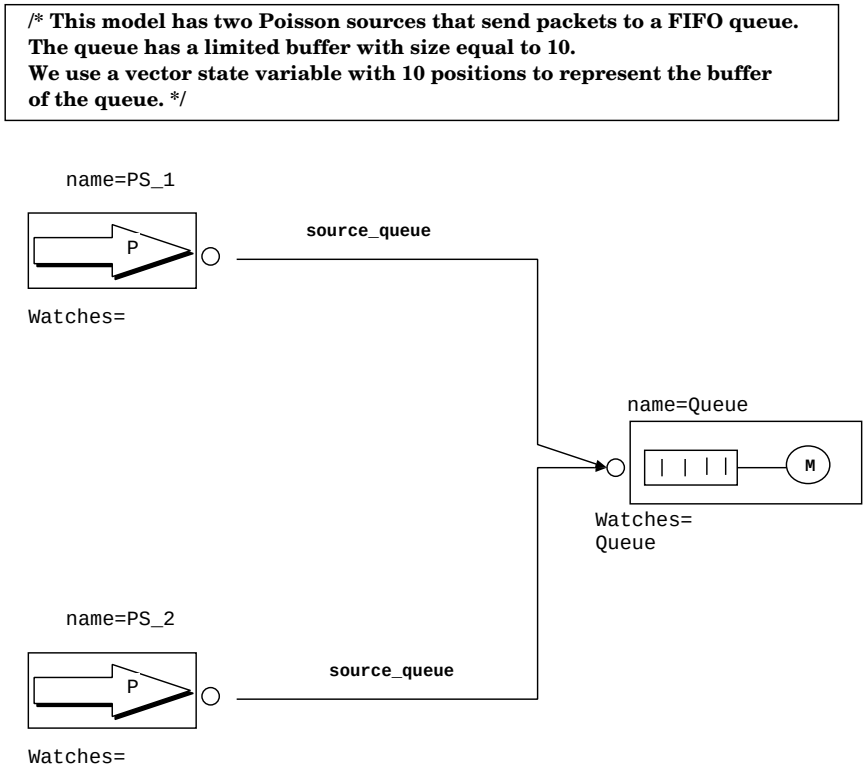


Figure 9.24: The Vector Variable Model.

It is important to note that with this kind of feature we are able to implement a service discipline which is a function of the type of packet in the queue.

The description of the objects is shown in Figures 9.25 and 9.26

9.12 Simulation Model with Animation

TANGRAM-II supports animation using TGIF commands and the animation attribute.

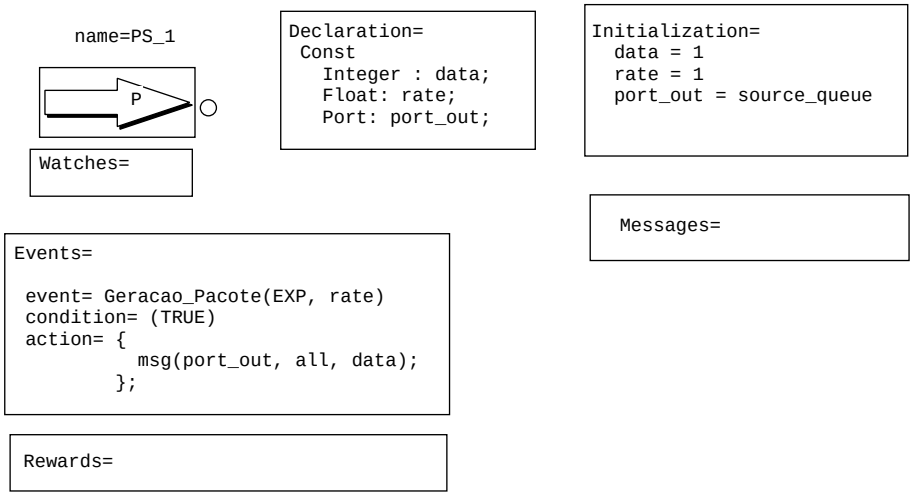


Figure 9.25: The Poisson Source object (Vector Variable Model).

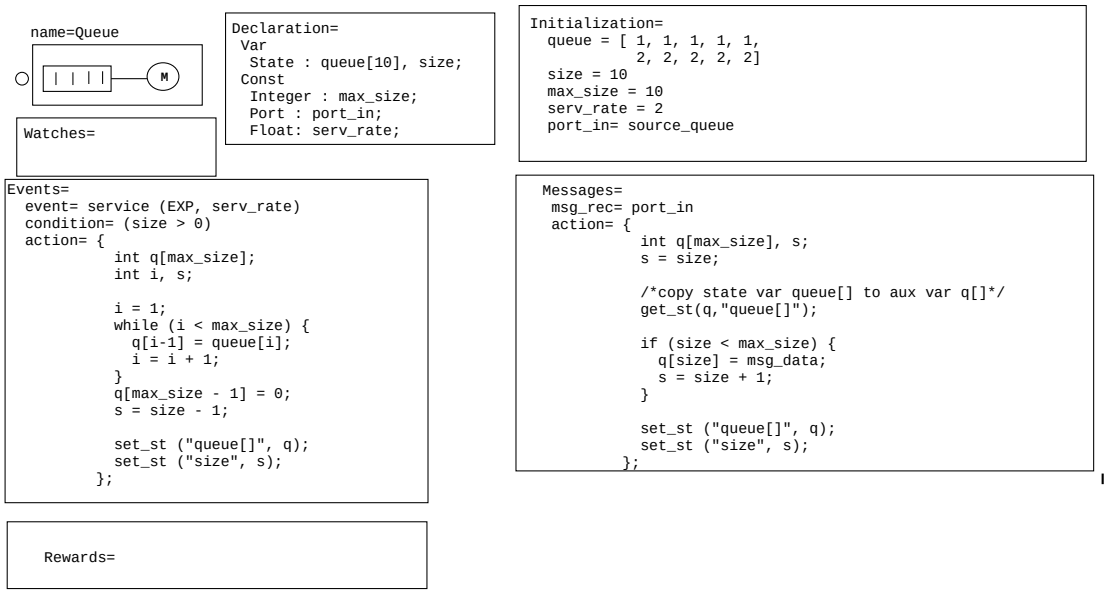


Figure 9.26: The Queue object (Vector Variable Model).

9.12.1 Model Description

We will now show an example of an animation done with the M/M/1/k model. This animation consists of two parts. The packet being sent from the source to the queue, and the increase/decrease of the queue size. The animation of the queue size is done by an rectangle object named "_BAR_". This object stretches its size to the appropriate value of its variable q. The object `Exp_Server` has the following Animation attribute:

```
Animation=
  get_line_in_attr(result1, Watches, 1);
  tokenize(result2, $(result1), "=");
  get_line_in_attr(q, result2, 2);
  assign(_BAR_.n, $(q));
  if("$(__SIMULATION_STEP__.Ani_T_Steps)"==
     "$(__SIMULATION_STEP__.Ani_C_Step)",
       _BAR_.animate, NULL);
```

This code gets the current value of the queue state variable and sets the variable n in object _BAR_ to this value. If it is the last step in the animation loop then it calls the function `_BAR_.animate` which animates the object _BAR_. This object has an animation function "animate" that is called by the object Exp_Server, that checks the size of the variable n and executes the corresponding function that resizes the BAR. This is done only at the last animation step.

To animate the movement of the packet from the source to the queue it is necessary to modify the behavior of the object Packet_Source. The animation can depend only on an object's state variables changes. In order for the animation to occur only when a packet is transmitted, a state variable change must occur. We modify the dummy state variable S to switch between states 0 and 1. Every time a packet is transmitted the state changes. By doing this, the Animation attribute can decide when to play the animation of the packet movement.

```
Animation=
get_line_in_attr(result1, Watches, 1);
tokenize(result2, $(result1), "=");
get_line_in_attr(s, result2, 2);
if("$(s)"!="$(s_ant)", _PAC_.animate, NULL);
if("$(__SIMULATION_STEP__.Ani_T_Steps)" ==
"$(__SIMULATION_STEP__.Ani_C_Step)", Reset_S, NULL);
Reset_S=
  assign(s_ant, $(s));
```

This code checks to see if the state variable S has changed its value. If so, then it calls the animation function in object _PAC_ for every animation step.

At the last animation step, it stores the new value for the state variable S. The object _PAC_, illustrated in Figure 9.27, has the animate function which moves the object horizontally to the left. At the first step, this object records its position. For every step it uses the number of the animation step to calculate the new x coordinate and shift the object to the left. At the last step of the animation it restores its original position.

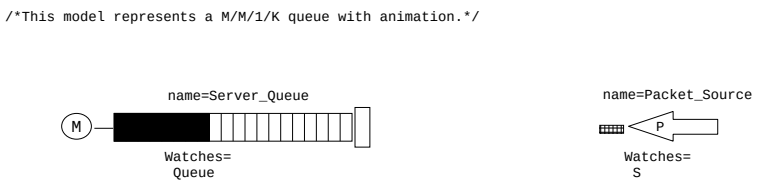


Figure 9.27: The Simulation Model with Animation.

Many other complex animations can be specified. The user can basically define any possible graphic command supported by TGIF. Sometimes the user will need to modify the behavior of the object in order to get the desired animation.

9.13 An Availability Model

9.13.1 Model Description

Consider a fault-tolerant database system which has a front end, a database, and two processing subsystems. Each processing subsystem contains a switch, a memory, and two processors. A processing subsystem is considered operational if the memory, the switch, and one of the two processors are operational. The entire system is operational if the database, the front-end, and at least one of the two processing subsystem is operational. We further assume that when a processor fails it contaminates (or fails) the database with probability (1-PROB), where PROB is the coverage probability. This model implements a simple repair policy . The database model presented in the next section model models a more complex repair policy.

There are four objects in the model:

Database Object This object has one state variable that represents if the object is operational or is not. There are two events: a FAIL event with rate equal to FAILURE_RATE, and a REPAIR event, that repairs the object with rate equal to REPAIR_RATE. In the Messages attribute, the messages received are sent by the two subsystems (when they fail, with probability PROB, they contaminate the database object). In the Rewards attribute, a reward (named database_availability) is specified. It computes the fraction of time that the database is operational.

Front End Object This object has the same behavior of the Database Object.

Subsystem Processor Objects Each Subsystem Processor Object has three state variables:
`Operational_Procs` (that represents the total number of operational processors in the subsystem), `Operational_Switch` (that represents if the switch is operational or not) and `Operational_Memory` (if the memory is operational or not).

The `Events` attribute has events that represents the failure of the memory, of the switch and of the processors and events that repair them. When the processor fail, it contaminates a database model with probability 1-PROB.

The `Rewards` attribute has a reward that computes the fraction of time that the subsystem processor is operational.

The availability of the model is specified by a Global Reward (`system-availability`). This attribute has the same syntax as the `Reward` attribute.

IMPORTANT: The conditions can be evaluated based only on the rewards of the objects. The value assigned to a global reward can be a constant, or a reward defined for an object of the model. If, for example, the object has a state variable S, we *cannot* specify the condition
`(condition= (object.S == 1))`. But if the object has a reward named `reward_R`, we can specify the condition `(condition = (object.reward_R == 1))`. In the case of the value assigned, we can specify, for example,
`value = object.reward_R_1 + object.reward_R_2`.

The model is shown in Figure 9.28.

The System_1 object is shown in Figure 9.29.

9.14 A Database Model

9.14.1 Model Description

The following example illustrates the modeling of complex failure and repair interactions among the various components of the system. The system consists of three types of components: two processors, a front end, and a database. The failure mode is used to model the fact that a processor can fail in two different modes: mode A with probability PROB and mode B with probability 1-PROB. The repair rates are different in each of the modes. Failures of a processor may affect the system in different ways. In the failure mode A no component is affected. In the failure mode B the database is affected. Using the usual concept of coverage, with probability COVERAGE the database is successfully recovered and no manual repair is needed. The recovery is assumed to be instantaneous (instantaneous coverage). With probability 1-COVERAGE the database is corrupted and a repair is necessary. The database may also fail spontaneously. In order to repair the database at least one processor must be operational.

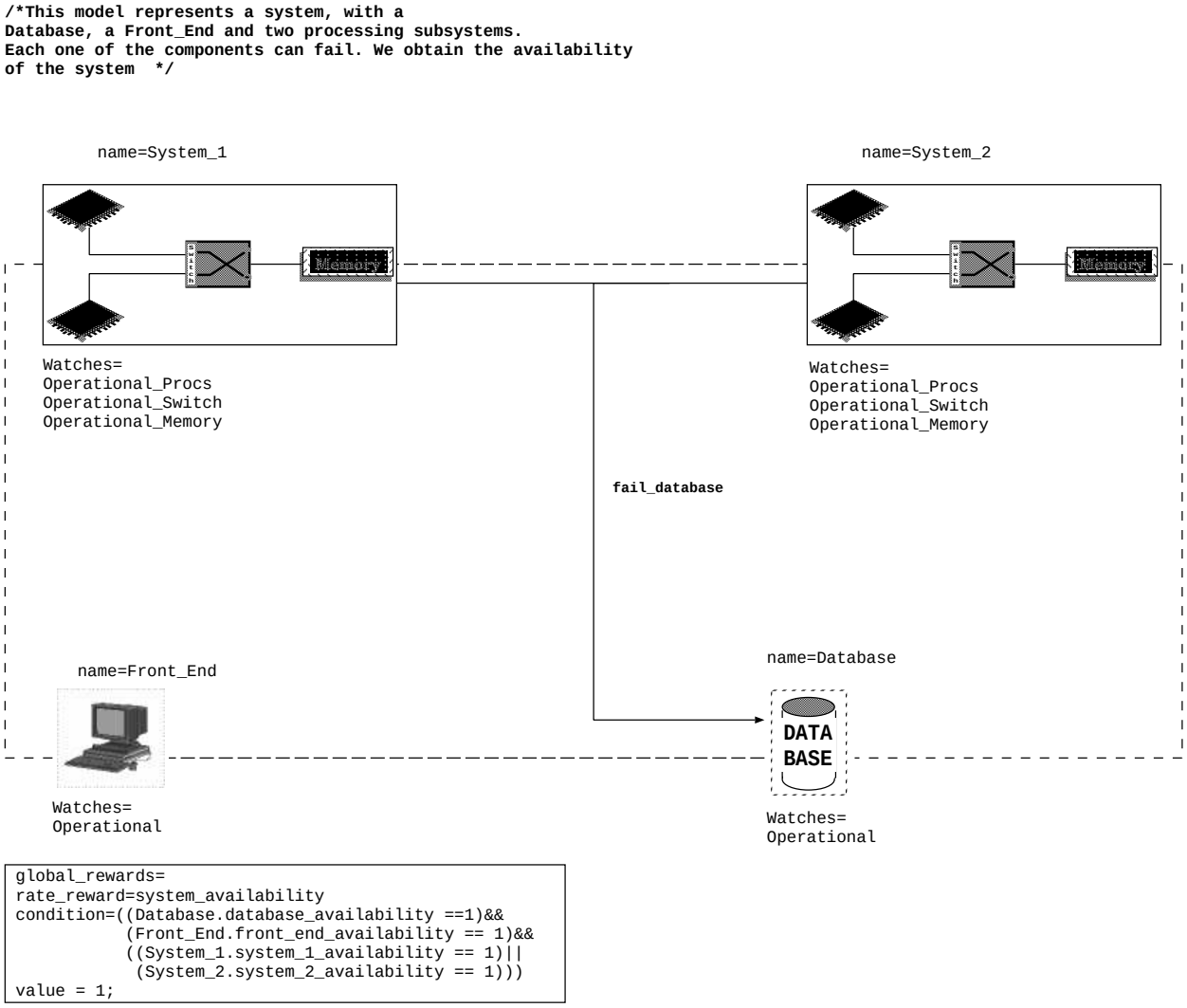
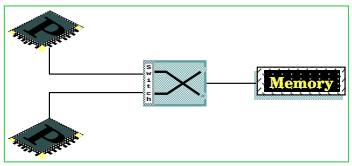


Figure 9.28: The Availability Model.




```
Watches=
Operational_Procs
Operational_Switch
Operational_Memory
```

```
Declaration=
State Var

Integer:Operational_Procs, /* Total number of operational processors */
        Operational_Switch,/* 0 if switch is not operational; 1 if switch
is operational */ Operational_Memory;/* 0 if memory is not operational; 1
if memory is operational */

Const

Integer: N_PROCS /* Total number of processors in the system*/;

Float: FAILURE_RATE,REPAIR_RATE, /* Both of switch and memory have
the same failure and repair rates */
       PROC_FAILURE_RATE, PROC_REPAIR_RATE, PROB;

Port: DATABASE /* This port sends a message that causes the fail in
database */;
```

```
Initialization=
Operational_Procs = 2
Operational_Switch = 1
Operational_Memory = 1
N_PROCS      = 2
FAILURE_RATE = 0.0004
REPAIR_RATE = 1
PROC_FAILURE_RATE  = 0.008
PROC_REPAIR_RATE    = 1
PROB  = 0.8
DATABASE = fail_database
```

```
Messages=
```

```
Rewards=

rate_reward = system_1_availability
condition   = ((Operational_Memory ==1)
               &&(Operational_Switch==1)&&
               (Operational_Procs >0))
value       = 1;
```

```
Events=
/*events related with Processor  1*/
event     = Fail_Processor(EXP,Operational_Procs*PROC_FAILURE_RATE)
condition = ( Operational_Procs > 0)
action={
          int operational_procs;
   operational_procs = Operational_Procs - 1;
          msg(DATABASE,all,0);
          set_st("Operational_Procs",operational_procs);

 }: prob=1-PROB;

       {
          int operational_procs;
   operational_procs = Operational_Procs - 1;
          set_st("Operational_Procs",operational_procs);

       }: prob=PROB;

event     =  Repair_Processor(EXP,(N_PROCS - Operational_Procs) *
PROC_REPAIR_RATE)
condition = (Operational_Procs <= 1)
action={
         int operational_procs;
         operational_procs = Operational_Procs + 1;
         set_st("Operational_Procs",operational_procs);

       };


/*events related with memory*/
event     = Fail_Memory(EXP,FAILURE_RATE)
condition = ( Operational_Memory == 1 )
action={
         ;set_st("Operational_Memory",0);
       };

event     =  Repair_Memory(EXP,REPAIR_RATE)
condition = ( Operational_Memory == 0 )
action={
         ;set_st("Operational_Memory",1);
       };


/*events related with switch*/
event     =  Fail_Switch(EXP,FAILURE_RATE)
condition = ( Operational_Switch == 1 )
action={
         ;set_st("Operational_Switch",0);
       };


event     = Repair_Switch(EXP,REPAIR_RATE)
condition = ( Operational_Switch == 0 )
action={
         ;set_st("Operational_Switch",1);
       };
```

Figure 9.29: The System_1 object (Availability Model).

The system is considered operational when at least one of each type of component is operational. No component can fail once the system is down. Finally, there is a single repair center and the highest repair priority is assigned to the front end followed by the database and then by the processors.

To model this database system, we use four objects:

Database Object This object has two state variables: `Failed`, which represents if the database is failed, and `Can_Fail`, used to enable the Fail event (remember that when the system is down, the objects cannot fail).

The single event is Fail. When this event triggers, the database sends a message to the repair center, through the FAILURE_REPAIR port (named `failure_repair_port`). This port is used by all objects to send a message when they fail.

The message received through the affected port (named `affected_port`) is sent by the processor object (in this case the processor failure mode is B). The message received through the FAILURE_REPAIR port (named `failure_repair_port`) is sent by the repair center to indicate that an object was repaired. The message received through the status port (named `sys_status_port`) is sent by the repair center when the system is down (in this case, the event FAIL becomes disabled) and when the system is up (in this case, the event FAIL becomes enabled).

Front End Object The front end object has the same behavior as the Database object. The only difference is that the front end object is not affected when the processor fails.

Processor object This object has three state variables: `Proc1_Fail`, which represents the number of processors that have failed in mode A, `Proc2_Fail`, which represents the number of processors that have failed in mode B, and the variable `Can_Fail`, similar to that of the database object.

The single event is Fail, that represents a failure of the processor object. With probability PROB, the failure is of type A, and with probability 1-PROB it is of type B.

The messages received from the repair center are similar for all objects. The only difference is that in the case of the processor object, the message data field is checked to verify the type of failure that was repaired.

Repair Center This object represents a repair center in the database model. It controls the repair priorities and determines when the system is operational or not.

The only state variable is a vector-variable, that stores in each position the status of each component. The first position is reserved to the object with the highest repair priority (in this case, the front end object). The second position is reserved to the

object with has second priority level (the database object) and the third and fourth positions, are reserved to the processors (each position corresponding to a different failure mode).

The events in the repair center object correspond to a repair performed in a system component. When the repair center repairs an object, it sends a message to the respective object (FRONT_END, DATABASE, PROCESSOR) through the `failure_repair_port`. The message that represents the system status is sent if and only if the specified condition is true (i.e. when the system is operational or not).

In the Messages attribute, the repair object receives th e messages sent by the other objects in the database model. When the message arrives, the repair object checks who sends the message (through the function `objcmp(msg_source,object)`). This is a boolean function that compares two objects. The word `msg_source` checks the object that sends the message. Then the vector-variable, which indicates the failed objects, can be updated. If the database or the front end fails, a message is sent to all objects and the system becomes down. If the number of processors failed is equal to 2, the same message is sent to all objects in the model.

The model is shown in Figure 9.30.

The Processor object and the Repair object are shown in Figures 9.31, 9.32 and 9.33.

9.15 Go Back N Protocol Model

9.15.1 Model Description

The go-back-N-protocol is used for reliable data transfer. Please refer to a good textbook in networking (e.g. [33]) for details. In this protocol, packets to be transmitted from A to B are numbered sequentially. This sequence number (SN) is sent in the packet header and it is checked by the receiver.

Our model is a simple version of the go-back-n protocol. It consists of a Sender object that transmits packets to another object and receives acknowledgments for packets correctly received. A Receiver object accepts packets from the Sender object and transmits acknowledgments for packets that are received correctly.

In order to simplify our model, we assume that the round trip time (RTT) measured from the time the sender transmitted a packet until it gets an acknowledgment back (assuming the packet was correctly received by the receiver and also that the acknowledgment was received correctly by the transmitter) is included in the "Channel" object. This assumption is useful to maintain the cardinality of the state space under reasonable size. (The user is encouraged to relax this assumption and see what happens.) We also assume that the ACK packets are small enough so that the its transmission delay is negligible. Therefore only the propagation delays are included in the channel object. Furthermore, no

/* This model represents a system with a Database, a Front_End, a Processor and a Repair Center.
The components can fail and the failure can affect the behavior of others components. We obtain the availability of the system. */

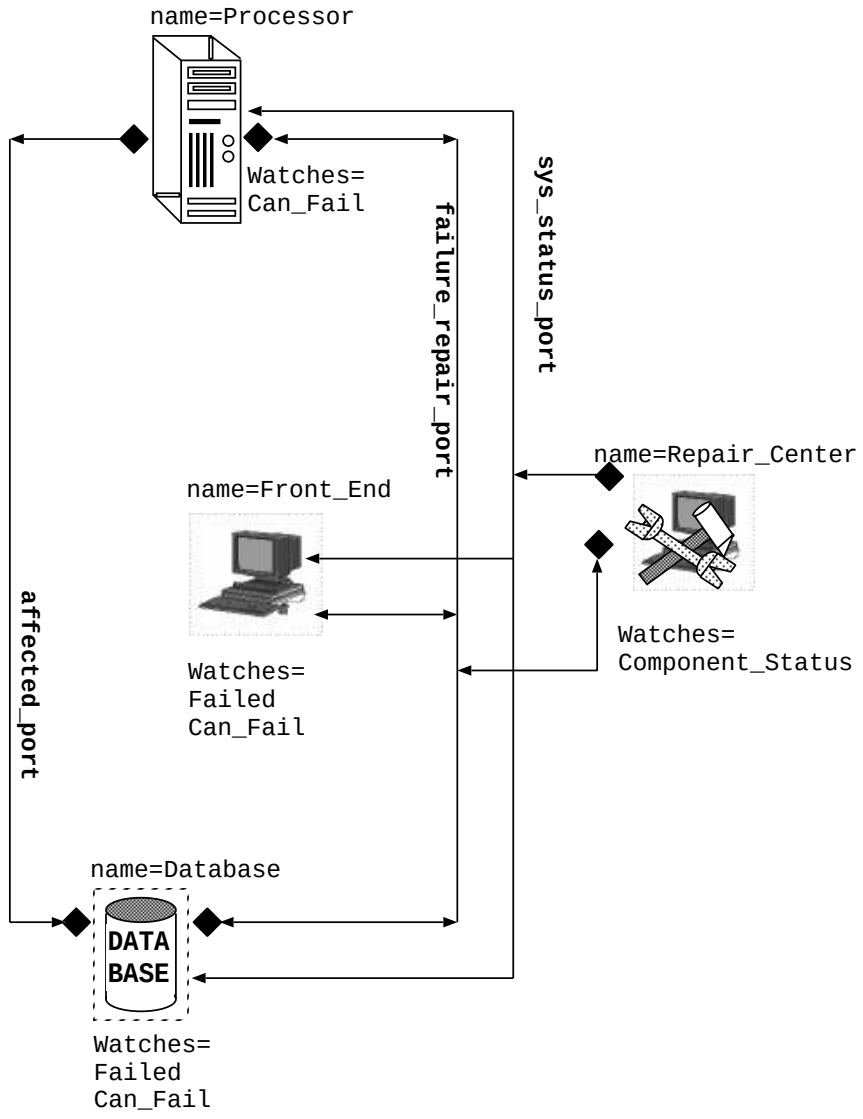


Figure 9.30: The Database Model.


```
Declaration=
State Var
Integer:  Proc1_Fail, Proc2_Fail,Can_Fail;


Const
Float:  FAILURE_RATE,PROB,COVERAGE;
Port:   FAILURE_REPAIR,STATUS,AFFECT;
Object: REPAIR_CENTER;
```

```
Initialization=
Proc1_Fail =0
Proc2_Fail =0
Can_Fail =1   /*
1 means that the object can fail
0 means that the object can NOT fail */
FAILURE_RATE   =0.0013
FAILURE_REPAIR= failure_repair_port
STATUS =sys_status_port
AFFECT =affected_port
REPAIR_CENTER= Repair_Center
PROB  =0.5
COVERAGE =0.8
```

```
Rewards=
```

```
Messages=
msg_rec= STATUS
action={
        /* Repair Center sends a
 message to all objects indicating
 the system status. If
 msg_data = 0 -> system is down so
 object can NOT fail,
 msg_data = 1 -> system is up so object
 can fail. */
            int option;
            option = msg_data;
            set_st("Can_Fail",option);
      };

msg_rec= FAILURE_REPAIR
action={
       /* Repair Center sends a message
to this port indicating that
           this object was repaired and
the fail mode. */
            int proc1_fail, proc2_fail;
            proc1_fail = Proc1_Fail;
            proc2_fail = Proc2_Fail;
             if (msg_data == 1)
                  proc1_fail = proc1_fail -1;
            else
                   proc2_fail = proc2_fail - 1;
             set_st("Proc1_Fail",proc1_fail);
             set_st("Proc2_Fail",proc2_fail);
     };
```

```
Events=
event=Fail(EXP,FAILURE_RATE)

/* A processor fails if it can fail and at least one processor
 is operational */
condition = ((Can_Fail==1) && ((Proc1_Fail + Proc2_Fail) < 2))
action=  {
                     /* Fail mode 1, just sends a message to
Repair Center */
                     int proc1_fail;
                     proc1_fail = Proc1_Fail + 1;
                     msg(FAILURE_REPAIR,REPAIR_CENTER,1);
                     set_st("Proc1_Fail",proc1_fail);
               }:  prob=PROB;

       {
            /* Fail mode 2, sends a message to Database to fail
 and to Repair Center*/
                    int proc2_fail;
                    proc2_fail = Proc2_Fail + 1;
                    msg(AFFECT,all,0);
                    msg(FAILURE_REPAIR,REPAIR_CENTER,2);
                    set_st("Proc2_Fail",proc2_fail);
         }:     prob=((1-PROB)*(1-COVERAGE));

       {
          /* Fail mode 2, just sends a message to Repair Center*/
                    int proc2_fail;
                    proc2_fail = Proc2_Fail + 1;
                    msg(FAILURE_REPAIR,REPAIR_CENTER,2);
                    set_st("Proc2_Fail",proc2_fail);
       }:       prob=((1-PROB)*COVERAGE);
```

Figure 9.31: The Processor object (Database Model).

```
name=Repair_Center
```

```
Rewards=
```

```
Watches=
Component_Status[0:3]
```

```
Initialization=
Component_Status    = [0,0,0,0]
FRONT_END           = Front_End
DATABASE            = Database
PROCESSOR           = Processor
FAILURE_REPAIR      = failure_repair_port
STATUS                 = sys_status_port
PROC1_REPAIR_RATE      = 0.8
PROC2_REPAIR_RATE      = 1
FRONT_END_REPAIR_RATE  = 1
DATABASE_REPAIR_RATE   = 1
```

```
Declaration=
State Var
Integer : Component_Status[ 4 ];
      /* position 0:
            is the Front-end status
         position 1:
            is the Database status,
         position 2:
            is the quantity of Processors that fail in mode 1
         position 3:
            is the quantity of Processors that fail in mode 2

         value 0 means that the object is not failed
         value 1 means that the object is failed

         in the processors case it means the quantity of
         failed processors.
      */

Const

Float : PROC1_REPAIR_RATE, PROC2_REPAIR_RATE,
        FRONT_END_REPAIR_RATE, DATABASE_REPAIR_RATE;

Object: FRONT_END,DATABASE,PROCESSOR;

Port  : FAILURE_REPAIR, STATUS;
```

```
Messages=
msg_rec = FAILURE_REPAIR
action=
{
   int component_status[ 4 ], index;
   index = 0;
   while( index < 4 )
   {
      component_status[ index ] = Component_Status[ index ];
      index = index + 1;
   }
   if( objcmp( msg_source, FRONT_END ) )
   {
      component_status[ 0 ] = component_status[ 0 ] + 1;
      msg( STATUS, all, 0 );
   }
   if( objcmp( msg_source, DATABASE ) )
   {
      component_status[ 1 ] = component_status[ 1 ] + 1;
      msg( STATUS, all, 0 );
   }
   if( objcmp( msg_source, PROCESSOR ) )
   {
      if( msg_data == 1 )
         component_status[ 2 ] = component_status[ 2 ] + 1;
      else
         component_status[ 3 ] = component_status[ 3 ] + 1;
      if( component_status[ 2 ] + component_status[ 3 ] == 2 )
         msg( STATUS, all, 0 );
   }
   set_st( "Component_Status[]", component_status );
};
```

```
Events=

event=Repair_Front_End( EXP, FRONT_END_REPAIR_RATE )
/* It repairs Front-end if it is failed. */
condition= ( Component_Status[ 0 ] > 0 )
action =
{
    msg( FAILURE_REPAIR, FRONT_END, 0 );
    msg( STATUS, all, 1 );
    set_st( "Component_Status[0]", 0 );
};


event= Repair_Database( EXP, DATABASE_REPAIR_RATE )
/* It repairs Database if it is failed and if there
   is at least one operational processor.        */
condition= ( (Component_Status[1] > 0) &&
      (Component_Status[2]+Component_Status[3]<2) )
action =
{
    msg( FAILURE_REPAIR, DATABASE, 0 );
    msg( STATUS, all, 1 );
    set_st( "Component_Status[1]", 0 );
};

event= Repair_Proc1( EXP, PROC1_REPAIR_RATE )
/* It repairs the Processor that fail in mode 1 if:
   there is at least 1 processor that fails in mode
    and none of Front-end and Database is failed
   OR
   there is at least 1 processor that fails in mode
   1 and both processors are failed and database is
   failed too (if both processors are failed, the
   Repair needs to repair a processor before to
   repair the Database).*/

condition= (((Component_Status[2]>0) &&
(Component_Status[0]==0)  &&
(Component_Status[1]==0)) ||
((Component_Status[2]>0)  &&
(Component_Status[2]+Component_Status[3] == 2) &&
(Component_Status[1]==1)))
action =
{
    int component_status;
    msg( FAILURE_REPAIR, PROCESSOR, 1 );
    component_status = Component_Status[ 2 ] - 1;
    if( Component_Status[ 1 ] == 0 )
         /*All objects can fail. */
         msg( STATUS, all, 1 );
    set_st("Component_Status[2]", component_status);
};


event= Repair_Proc2( EXP, PROC2_REPAIR_RATE )
/* It repairs the Processor that fail in mode 2 if:
   there is at least 1 processor that fails in mode
   2 and none of Front-end and Database is failed
   OR
   there is at least one processor that fails in
   mode 2 and both processors are failed and
   database is failed too (if both processors
   are failed, the Repair needs to repair a
   processor before to repair the Database).*/

condition= (((Component_Status[3]>0)&&
(Component_Status[0]==0)  &&
(Component_Status[1]==0)) ||
((Component_Status[3]>0)  &&
(Component_Status[2]+Component_Status[3] == 2) &&
(Component_Status[1]==1)))

action =
{
    int component_status;
    msg( FAILURE_REPAIR, PROCESSOR, 2 );
    component_status = Component_Status[ 3 ] - 1;
    if( Component_Status[ 1 ]== 0 )
         /*All objects can fail. */
         msg( STATUS, all, 1 );
    set_st("Component_Status[3]", component_status);
};
```

Figure 9.32: The Repair object (Database Model).


```
name=Database

Watches=
Failed
Can_Fail

Declaration=
State Var
Integer: Failed,Can_Fail;

Const
Float: FAILURE_RATE;
Port:  AFFECT,FAILURE_REPAIR,STATUS;
Object:REPAIR_CENTER;

Rewards=
rate_reward = Data_avail
condition= (Failed == 0)
value = 1;

Initialization=
Failed          = 0
Can_Fail        = 1
FAILURE_RATE    = 0.0003
AFFECT          = affected_port
FAILURE_REPAIR  = failure_repair_port
STATUS          = sys_status_port
REPAIR_CENTER   = Repair_Center

Events=
event = Fail (EXP,FAILURE_RATE)
/* Database fails if it can fail and it is not failed. */
condition = (( Can_Fail == 1 ) && ( Failed == 0 ))
action={
                msg(FAILURE_REPAIR,REPAIR_CENTER,0);
                set_st("Failed",1);
       };

Messages=
msg_rec = AFFECT
action = {
             /* Processor affects the database.*/
             if( (Can_Fail == 1) && (Failed == 0) )
                 msg( FAILURE_REPAIR, REPAIR_CENTER, 0);
             set_st( "Failed", 1 );
        };

msg_rec=FAILURE_REPAIR
action={
           /* Repair Center sends a message to this port indicating that
           this object was repaired. */
           ;set_st( "Failed", 0 );
      };


msg_rec=STATUS
action={
                  /* Repair Center sends a message to all objects indicating
                    the system status. If msg_data = 0 -> system is down so
                    object can NOT fail, msg_data = 1 -> system is up so object
                    can fail. */
                    int option;
                    option = msg_data;
                    set_st( "Can_Fail", option );
      };
```

Figure 9.33: The Database object (Database Model).

sender timeout is modeled, and when the sender transmits all packets in a window it "goes back" and retransmits from the beginning of the window. In order to obtain a Markov model, all random variables are assumed to be exponentially distributed.

In the first model we present we assume that both packets and acknowledgments may be lost. *However*, the receiver only sends ACK packets back to the transmitter when it accepts a packet. The Sender object has two state variables: one, `Win_begin`, indicates the beginning of the transmitter window, and the other, `SN`, points to the sequence number of the next packet to be transmitted. The Receiver object has only one state variable `RN` that indicates the sequence number the receiver is expecting. That is, a packet with sequence number equal to `RN` is accepted, if received correctly. The Channel object has two variables. The first `ACK_RN` indicates the serial number of the *last* ACK that was sent to the receiver. (Note that this last ACK could have been lost.) The second variable `N_acks` indicates the number of ACK packets that are in transit in the channel. The model is shown in Figure 9.34.

The TANGRAM-II description of the objects is shown in Figures 9.35, 9.36 and 9.37.

We encourage the user to first try the model using `p_losing_ack=0`, that is, no ACK packets are lost. This model has 18 states, considering the transmission window equal to 2 as in the figures. (WARNING: the transmission window must be identical in all objects since their specification uses this parameter.)

The impulse-reward in the Channel object marks the transitions that represent a packet accepted by the receiver (in this model, this event happens when an ACK is sent to the channel). Using the marked transitions and the steady state results, the user can easily compute the system throughput.

Now lets change `p_losing_ack` to the value in Figure 9.36, and solve it again. The model now has 36 states. However, the Markov chain generated is not ergodic. Why? Because we assumed the sender only sends ACKs back whenever it accepts a packet. Since ACK packets can be lost, the sender may never receive an ACK for packets it sent. The user should verify the following set of states: $\{14, 20\}, \{28, 33\}$ and $\{34, 36\}$. Once the model reaches any of these states, it remains in the corresponding set. You should also try the interactive simulation and observe that the model reaches a deadlock state!

The tool, in this present version, does not check for ergodicity, and so the user must be careful when specifying a model. Also some of the solvers cannot be used when the model is not ergodic. For instance, the GTH solver will not produce the correct result. However, the Power method can be used. You should verify the results with the Power method, allowing sufficient iterations until convergence is reached.

In order to obtain a correct working protocol, the receiver must send ACK packets whenever it receives a packet, not only when it accepts a packet. In our model, we also modify the Channel object so that: (a) a duplicate ACK is discarded when the channel has still ACKs to deliver and; (b) a duplicate ACK is not discarded if all ACK packets have been delivered.

Following the new specification, the message attribute of the Receiver object is changed

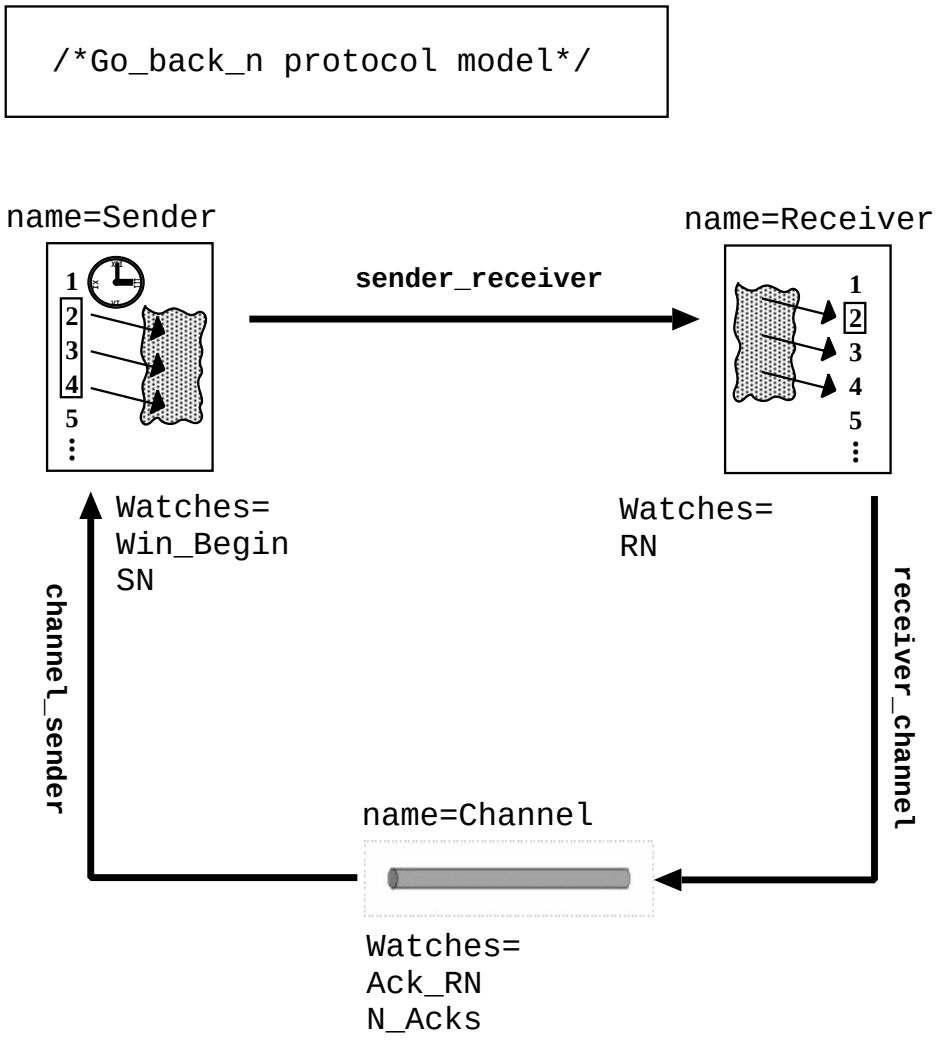
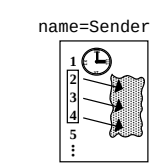


Figure 9.34: The Go Back N Model.




```
Declaration=
State Var
  Integer:Win_Begin, SN;
Const
  Integer:TRANS_WIN;
  Float:TRANSMISSION_RATE;
  Port: ACK_PORT, PKT_PORT;
```

```
Watches=
Win_Begin
SN
```

```
Initialization=
Win_Begin = 0
SN = 0
TRANS_WIN = 2
PKT_PORT = sender_receiver
ACK_PORT = channel_sender
TRANSMISSION_RATE = 1.0
```

```
Rewards=
```

```
Events=
event=SN_Transmission(EXP,TRANSMISSION_RATE)
condition=(TRUE)
action={
           int s, module, win_end;
           s = SN;
           module = TRANS_WIN + 1;
           win_end = (Win_Begin + TRANS_WIN - 1)%module;
           msg(PKT_PORT,all,SN);
           if (SN != win_end )
           /* increment SN since we haven't reached the end
              of the windown */
           {
               s = (SN + 1)%module;
           }
           else
           /* Since we transmitted the last packet in the window
              Go Back to the beginning of the window */
           {
               s = Win_Begin;
           }
           set_st("SN",s);
       };
```

```
Messages=
msg_rec = ACK_PORT
action = {
            int w_b, w_e, s, module, diff,
                order_of_SN_in_window,order_of_RN_in_window;

            w_b = Win_Begin;
            module = TRANS_WIN + 1;
            s = SN;

            /* The key steps below are used to identify the order
               of RN and SN inside the window. */

            diff = module - Win_Begin;
            order_of_SN_in_window = ( s + diff ) % module;
            order_of_RN_in_window = (int)(msg_data + diff ) % module;

            w_b = (int)(msg_data) % module;

            if ( order_of_RN_in_window > order_of_SN_in_window )
            {
                s = (int)(msg_data) % module;
            }

            set_st("Win_Begin",w_b);
            set_st("SN",s);
        };
```

Figure 9.35: The Sender object (Go Back N Model) Model 1.

```
name=Channel
```

```
Watches=
Ack_RN
N_Acks
```

```
Declaration=
State Var
  Integer: Ack_RN, N_Acks;
Const
  Integer: TRANS_WIN;
  Float:DELAY_RATE, P_LOSING_PKT;
  Port: SEND_ACK_PORT, RECV_ACK_PORT;
```

```
Initialization=
Ack_RN = 0
N_Acks = 0
TRANS_WIN = 2
DELAY_RATE = 0.5
P_LOSING_PKT = 0.1
SEND_ACK_PORT = channel_sender
RECV_ACK_PORT = receiver_channel
```

```
Rewards=
impulse_reward= packet_accepted
Port = RECV_ACK_PORT,1
value= 1;
```

```
Events=
event     = Delayer(EXP, N_Acks * DELAY_RATE)
condition = (N_Acks > 0)
action = {
             /* Sends the ack and
                sets ACK_RN and N_acks variables. */
             int n_acks,ack_rn;
             ack_rn = (Ack_RN + 1)%(TRANS_WIN + 1);
             msg(SEND_ACK_PORT,all,ack_rn);
             n_acks = N_Acks - 1;
             set_st("N_Acks",n_acks);
             set_st("Ack_RN",ack_rn);
         }:prob = 1 - P_LOSING_PKT;
         {
             /* Just sets ACK_RN and N_acks variables. */
             int ack_rn, n_acks;
             ack_rn = (Ack_RN + 1)%(TRANS_WIN + 1);
             n_acks = N_Acks - 1;
             set_st("N_Acks",n_acks);
             set_st("Ack_RN",ack_rn);
         }:prob = P_LOSING_PKT;
```

```
Messages=
msg_rec = RECV_ACK_PORT
action={
           int n_acks; /* number of acks stored in channel */
           int ack_rn; /* serial number of the last ack transmitted from channel.
                          Note this last ack may have been lost */
           int module;
           int sn_last_ack; /* serial number of the last ack in channel to be
                               or last ack sent if n_acks=0 */

           module = TRANS_WIN + 1;
           n_acks = N_Acks;
           ack_rn = Ack_RN;
           sn_last_ack = (ack_rn + n_acks)%module;
           if (msg_data == (sn_last_ack+1)%module)
           /* this is a new ack that is entering the channel */
           {
               n_acks = N_Acks + 1;
           }
           else
           /* this is a duplicate ack */
           {
               if (n_acks == 0)
           /* acks in channel have been lost, so send this duplicate ack */
               {
                   n_acks = N_Acks + 1;
                   /* ack_rn = ack_rn - 1, in Module */
                   ack_rn = (ack_rn + TRANS_WIN)%module;
               }
           }
           set_st("N_Acks",n_acks);
           set_st("Ack_RN",ack_rn);
       };
```

Figure 9.36: The Channel object (Go Back N Model). Model 1

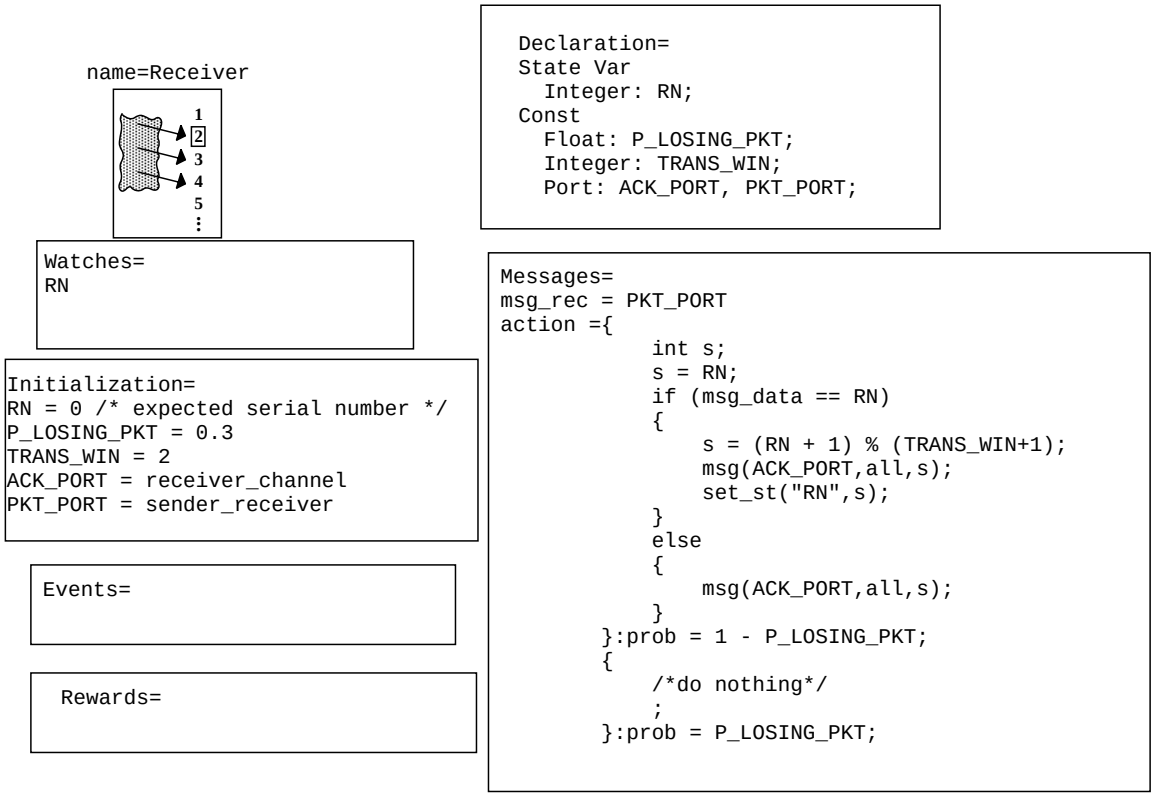


Figure 9.37: The Receiver object (Go Back N Model) Model 1.

to:

```
msg_rec = PKT_PORT
action ={
            int s;
            s = RN;
            if (msg_data == RN)
            {
                s = (RN + 1) % (TRANS_WIN+1);
                msg(ACK_PORT,all,s);
                set_st("RN",s);
            }
            else
            {
                msg(ACK_PORT,all,s);
            }
        }:prob = 1 - P_LOSING_PKT;
        {
            /*do nothing*/
            ;
        }:prob = P_LOSING_PKT;
```

The message attribute of the Channel object is changed to:

```
msg_rec = RECV_ACK_PORT
action={
           int n_acks; /* number of acks stored in channel */
           int ack_rn; /* serial number of the last ack transmitted from channel.
                       Note that this last ack may have been lost */
           int module;
           int sn_last_ack; /* serial number of the last ack in channel to be
                            or last ack sent if n_acks=0 */

           module = TRANS_WIN + 1;
           n_acks = N_Acks;
           ack_rn = Ack_RN;
           sn_last_ack = (ack_rn + n_acks)%module;
           if (msg_data == (sn_last_ack+1)%module)
           /* this is a new ack that is entering the channel */
           {
               n_acks = N_Acks + 1;
           }
           else
           /* this is a duplicate ack */
           {
               if (n_acks == 0)
               /* acks in channel have been lost, so send this duplicate ack */
               {
                   n_acks = N_Acks + 1;
                   /* ack_rn = ack_rn - 1, in Module */
                   ack_rn = (ack_rn + TRANS_WIN)%module;
               }
           }
           set_st("N_Acks",n_acks);
           set_st("Ack_RN",ack_rn);
       };
```

One final observation: note that the *impulse reward* attribute in this last model marks all the transitions corresponding to an ACK packet being received, and no distinction is

made between a duplicate ACK or an ACK for a new packet accepted by the receiver. The user is encouraged to find out how to calculate the system throughput in this new model.

9.16 Multiplex Channel

9.16.1 Model Description

This example shows two sources sharing a channel using time-division multiplexing (TDM). In the objects q_1 and q_2 packets are generated and served if the object has the token. The token stays with q_1 or q_2 until one of the following two events occurs: (1) a timeout expires (the timeout is the upper bound for the time an object can remain with the token) or (2) the queue is empty. The Channel object manages the token. When there is no packets to serve in the system, the token remains with the Channel object until a packet is generated in q_1 or q_2.

There are three objects in the model:

Channel this object sends the token to q_1 and q_2. If the queues do not have packets, the token remains with the channel.

q_1 **and** q_2 these objects have three events:

1. Packet_Arrive: generation of packets at exponentially-distributed intervals;
2. Service: transmission of the first packet in the queue if the object has the token;
3. Timeout: sends the token to the channel.

The model is shown in Figure 9.38.

The TANGRAM-II description of the queue object is shown in Figure 9.39.

9.16.2 Solving the Model

For this model, we will give special attention on how to use an approximation technique to calculate the Expected Cumulative Rate Reward (ESRA). For more references about this method see [22].

We have to specify a measure of interest using rewards. In this example, we can specify the total time an object holds the token, using a rate reward in the Reward's attribute:

```
rate_reward  = token_queue_1
   condition = (token == 1)
   value     = 1;
```

It is possible to compute the expected cumulative rate reward for a subset of states. The subset of states has to be specified using the global reward object. Only the objects that satisfy the global reward condition will be included in the subset. In this example, we want to consider all states so the syntax of the global reward is the following:

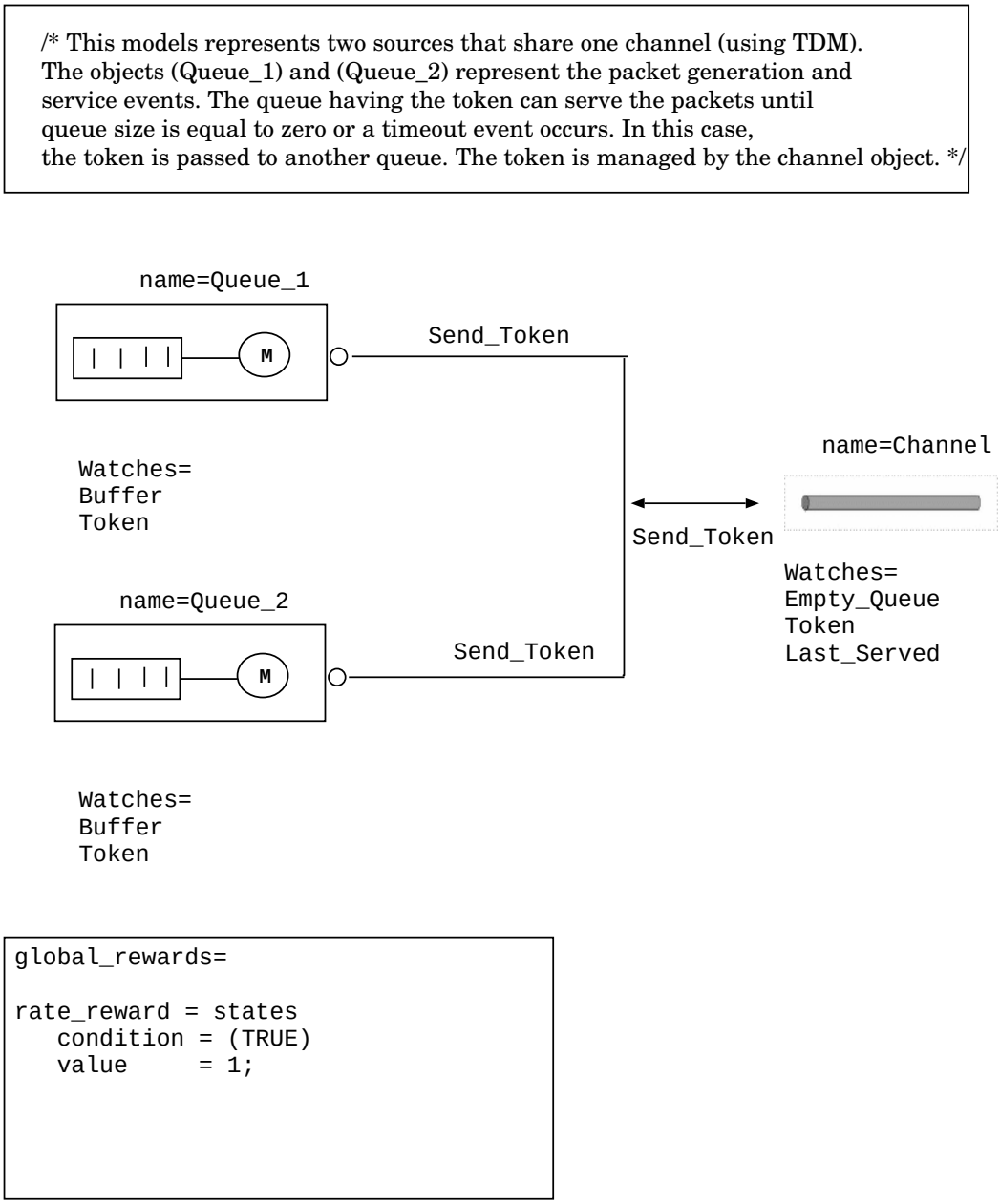
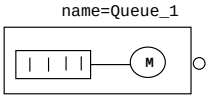


Figure 9.38: The Multiplex Channel Model.




```
Declaration=
Var
 State:Buffer,Token;

Const
Integer:MAX_BUFFER;
Float: PACKET_RATE, SERVICE_RATE,TIMEOUT_RATE;
Port:SEND_TOKEN, FIRST_PACKET;
Object:OBJ;
```

```
Watches=
Buffer
Token
```

```
Initialization=
/*Constant Initialization*/
PACKET_RATE  = 0.75
SERVICE_RATE = 3.75
TIMEOUT_RATE = 0.5
MAX_BUFFER   = 2
SEND_TOKEN   = Send_Token
FIRST_PACKET = First_Packet
OBJ          = Channel

/*State Vars Initialization*/
Buffer       = 0
Token        = 0
```

```
Rewards=
rate_reward  = prop_token
   condition = (Token == 1)
   value     = 1;
```

```
Events=
event     =Packet_Arrive( EXP, PACKET_RATE )
condition =( Buffer < MAX_BUFFER )
action    ={
               int buffer;
               buffer = Buffer;
               if( ( buffer == 0 ) && ( Token == 0 ) )
                   msg( FIRST_PACKET,all,1 );
               buffer++;
               set_st ( "Buffer",buffer );
           };

event     =Service( EXP, SERVICE_RATE )
condition =( Token == 1 )
action    ={
                int buffer, token;
                buffer = Buffer;
                token  = Token;
                buffer--;
                if( buffer == 0 )
                {
                    msg( SEND_TOKEN,all,0 );
                    token = 0;
                }
                set_st( "Buffer", buffer );
                set_st( "Token",token );
            };


event     =Timeout(EXP, TIMEOUT_RATE)
condition =(Token == 1)
action    ={
               msg(SEND_TOKEN, all, 1);
               set_st ("Token", 0);
           };
```

```
Messages=
msg_rec =SEND_TOKEN
action  ={
            int token;
            token  = Token;
            if( objcmp( msg_source, OBJ ) )
            {
                if( ( token == 0 ) &&
                ( msg_data == 1 ) )
                {
                    if( Buffer == 0 )
                        msg(SEND_TOKEN,all, 2);
                    else
                        token = 1;
                }
            }
            set_st( "Token",token );
        };
```

Figure 9.39: The Queue Object.

```
global_rewards=
rate_reward  = states
   condition = (TRUE)
   value     = 1;
```

To solve the model, click on the Analytical Model Solution button. Choose the Transient → Expected Values → ESRA → Direct Technique.

In the next step, we input the following parameters: Initial Probability, Time Intervals, Total number of Erlang Stages, Block Set (in this method the matrix is block-partitioned) and the Measure of Interest.

```
Initial Probability : Initial State
Time Intervals      = 5 1
Erlang Stages       = 10
Block Set           = 1 29 1
Measure of Interest = Set Reward
Reward Name         =
   multiplex_channel.rate_reward.Queue1.token_queue_1
```

(See [22] for more information about the *Erlang Stages* parameter.) The result is printed in the file
`Multiplex_Channel.TS.DIRECT.Cumulative_SET_Reward`.
This file has some information about the parameters considered in the solution and the expected rate reward accumulated at each time interval in the subset of states. In this example, the rate reward is the total time queue 1 holds the token.

9.17 The Geometric-sized Bulk Arrivals Model

9.17.1 Model Description

This model represents a $M^{[X]}/M/1/K$ queueing system where X is a geometric random variable. It is basically made of the same objects as the $M/M/1/K$ model (Poisson source and Exponential server), with an additional object acting as an interface between the two, as shown in figure 9.40.

The Geometric_Bulk object receives the Packet_Generation events from the Poisson source and generates a number of packets for the exponential server that is geometrically-distributed with parameter P, where P is a constant declared in the object. In this example, we are not very concerned with taking measures from the system but instead we will focus on the modeling technique employed in the Geometric_Bulk object.

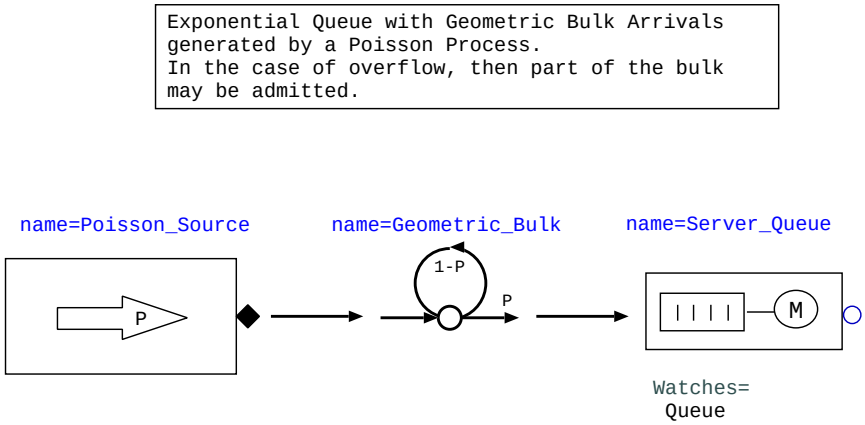


Figure 9.40: The Geometric bulk arrivals model.

9.17.2 Recursion with Tangram messages

Imagining the Geometric_Bulk object from figure 9.41 as a closed black box, all one can see is that it transforms a single message received by it into a new set of X messages, where X is a strictly positive geometric random variable. In order to accomplish that, it needs to perform a few tricks.

The object has a reference to itself, denoted by the constant `MYSELF` declared with the `Object` type. Upon receiving a message at port PORT_IN, the object executes one of two possible actions:

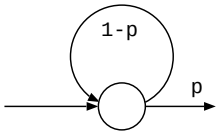
- With probability P, it sends the final message in the bulk through PORT_OUT and stops.

- With probability $1 - P$, it sends a message to the server and another one to itself at PORT_IN, which regenerates the process.

The recursive-messages keep ocurring until the state with the queue full is reached. From there, when a new message is generated by the bulk object, the model remains at the same state and Tangram stops the recursion.

If the server queue has a maximum size of N customers, then the Tangram-II Markov chain generator explores the state space as depicted in figure 9.42.

In conclusion, we may notice that with the idea of recursive messages, Tangram-II can generate state spaces that, at first, may not be obvious how to model.




```
name=Geometric_Bulk

Declaration=
    Const
        Port: PORT_IN, PORT_OUT;
        Float: P;
        Object: MYSELF;

Initialization=
    /* Geometric distribution */
    P = 0.9

    /* Ports */
    PORT_IN  = wire1
    PORT_OUT = wire2

    /* Pointer to self */
    MYSELF = Geometric_Bulk
```

```
Messages=
    msg_rec = PORT_IN
    action =
    {
        /* With probability (1-P)
           send one packet and
           recursively send the message
           to itself once more */
        msg( PORT_OUT, all   , 0 );
        msg( PORT_IN , MYSELF, 0 );
    } : prob = 1.0 - P;
    {
        /* With probability P,
           send the final packet
           from the bulk */
        msg( PORT_OUT, all , 0 );
    } : prob = P;
```

Figure 9.41: The Geometric_Bulk object.

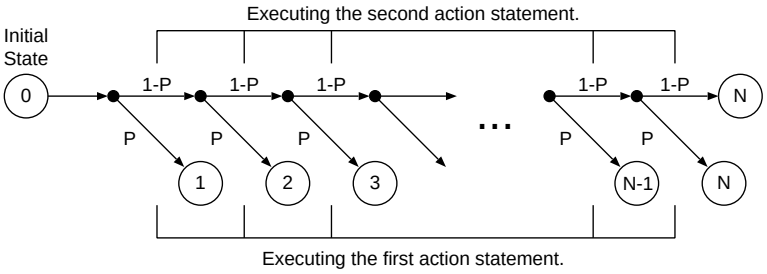


Figure 9.42: The recursion tree generated by Geometric_Bulk object from the initial state.

9.18 The Binomial-sized Bulk Arrivals Model

9.18.1 Model Description

As a sequel to the previous example, this model also represents a $M^{[X]}/M/1/K$ queueing system, though here X is a binomially-distributed random variable. Again, the only difference between this model and the basic $M/M/1/K$ model is the additional object that generates binomially-distributed bulks from the Poisson process generated by the source. The model is depicted in figure 9.43.

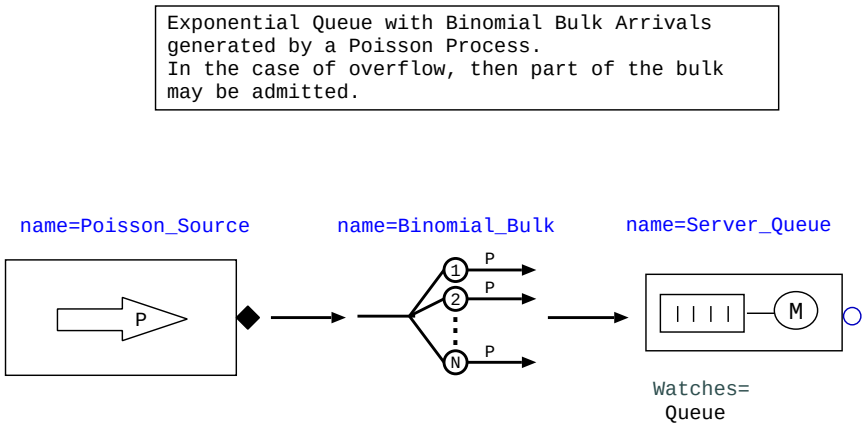


Figure 9.43: The Binomial bulk arrivals model.

9.18.2 Limited Recursion and Vanishing States

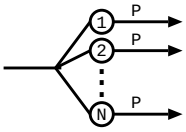
In this case recursion is also used to generate the bulks of messages, but it has an fixed limit defined by the positive integer constant N, which along with the constant probability P, form the set of parameters for the binomial distribution.

As figure 9.44 shows, whenever the Binomial_Bulk object receives a message, it can be forwarded to the server if the action with probability P is executed, and then, in either one of the action codes, it will check whether the state variable $TempN$ is greater than zero. In the positive case, it will decrease $TempN$ by one and send another message to itself. Otherwise it will reset $TempN$ to N and finish the recursion.

It is clear from the previous explanation and from the code, that the recursion in the Binomial_Bulk object takes place at most N consecutive times. It can happen less than N times if the process reaches a state where the queue is full.

The important thing to notice in this example is that, in order to generate a fixed size recursion, we need to add a new state variable to the model for keeping track of how deep the recursion is so far.

```
name=Binomial_Bulk
```




```
Declaration=
  Const
    Port: PORT_IN, PORT_OUT;
    Float: P;
    Integer: N;
    Object: MYSELF;

  Var
    Integer: TempN;


Initialization=
   /* Binomial distribution */
   P = 0.5
   N = 10

   /* Initialized to N */
   TempN = 10

   /* Ports */
   PORT_IN=wire1
   PORT_OUT=wire2

   /* Pointer to self */
   MYSELF=Binomial_Bulk
```

```
Messages=
msg_rec=PORT_IN
action=
{
    int temp_n;

    /* With probability P, send another packet */
    msg( PORT_OUT, all, 0 );

    if( TempN > 0 )
    {
         msg( PORT_IN, MYSELF, 0 );
         temp_n = TempN-1;
    }
    else
    {
         /* Reset the binomial distribution */
         temp_n = N;
    }

    set_st( "TempN", temp_n );
} : prob=P;
{
    int temp_n;

    /* With probability 1-P, skip this packet */

    if( TempN > 0 )
    {
         msg( PORT_IN, MYSELF, 0 );
         temp_n = TempN-1;
    }
    else
    {
         /* Reset the binomial distribution parameter */
         temp_n = N;
    }

    set_st( "TempN", temp_n );
} : prob=1.0-P;
```

Figure 9.44: The Binomial_Bulk object.

In the previous example, the model did not require this, because it refered to a geometric distribution, which has the memoryless property. This way, each step of the recursion can be determined without the use of any memory (state variables).

Since the geometric distribution is also the only discrete memoryless distribution, any other kind of batch size distribution (including the binomial case) will have to add new state variables to allow Tangram-II to generate the model's state space and transition rates.

At first, one might think this is bad, since adding a new state variable might increase the state space, but it turns out this does not happen. To explain why, we need to consider an example of what happens during the chain generation procedure for this model.

Suppose the model is set initially to the state where the queue is empty, and let our state tuple be defined as $(Server\_Queue.Queue, Binomial\_Bulk.TempN)$. Since the $Binomial\_Bulk$ object requires that $TempN$ be initialized to N, and having $N = 3$ in this example, our initial state will be $(0, 3)$. We further assume that the maximum number of customers in the queueing system is 10.

When an arrival event happens, while the model is in the initial state, the sequence of execution in the model can be illustrated by the diagram shown in figure 9.45.

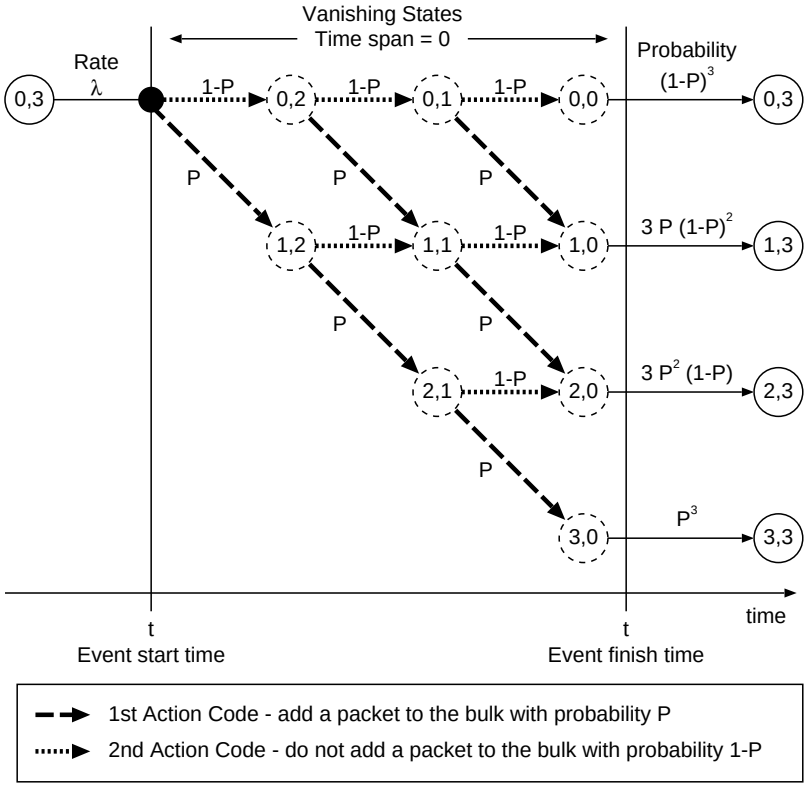


Figure 9.45: The Binomial bulk generation process from the initial state.

Notice that the recursion ends when $TempN$ reaches zero, being reset to $N = 3$. All the message exchanging happens in zero time, with the generation of several intermediate states, in which the value of $TempN$ can be 0, 1 and 2, but after the recursion the model always ends up with $TempN = 3$. Because of this, all the intermediate states do not show up in the final state space (as shown in figure 9.46), and so are called vanishing states.

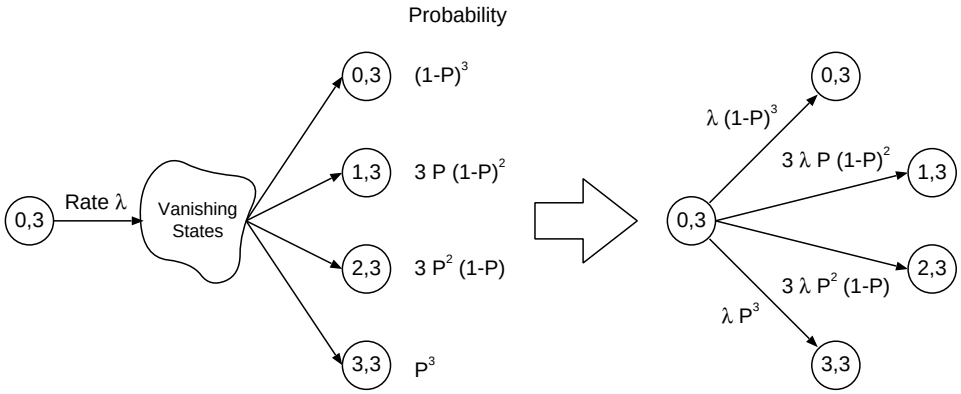


Figure 9.46: Transitions generated by the arrival event at the initial state.

After generating the state space for this model, all the states will have $TempN = N$. The main lesson to learn here is that even though the $TempN$ variable is not necessary to describe the state space (since it equals the constant N in all states), it is essential to the state generation procedure.

Chapter 10

Whiteboard

10.1 Introduction

This section describes a whiteboard tool built on top of TGIF: TGWB (Tangram Whiteboard). TGIF (Tangram2 Graphic Interface Facility) is a Xlib based interactive 2-D drawing facility under X11. The tgif tool is a powerful vector based drawing tool. The user draws objects, i.e., rectangles, lines, circles and splines, over a drawing area. Objects may be transformed - for instance, rotated, translated and flipped. New objects may be constructed by grouping other objects. The tgwb allows simultaneous modifications in drawings by users in a group. It is a versatile multicast distributed tool. TGWB interface is shown in Figure 10.1, where two users are editing a Tangram-II model.

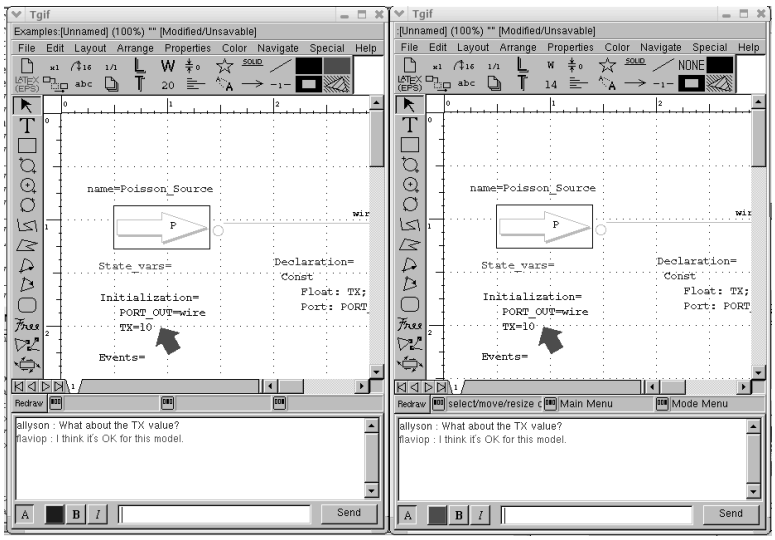


Figure 10.1: TGWB: Tangram Whiteboard interface.

Distributed whiteboards must ensure that every member in a session has the same view of the drawings. A Reliable Multicast Library and a total ordering mechanism was developed to allow reliable multicast transmission and member view consistency. Further information about those issues can be found at [35].

10.2 Using TGWB

10.2.1 Environment

The TGWB tool was develop to be used in a IP Multicast network. If Multicast is not available one can use the *mcastproxy* program to connect the whiteboards. More information about *mcastproxy* can be found in section 10.2.3.

10.2.2 TGWB Configuration

The reliable multicast transmission was implemented as a function library called RML (Reliable Multicast Library) [35]. When tgwb is started the user is prompted for configuration options, such as multicast address and port and whether or not the *mcastproxy* (see section 10.2.3) program should be started. Another way to customize RML options is editing the configure file **tgwb.conf**, located at *.tgwb* directory in the user's home.

```
#Reliable Multicast Library configuration file

#Reliable Multicast Library version
RM_VERSION=1.0

#Transmission mode: 0 multicast (default), 1 unicast
TRANSMISSION_MODE=0

#Multicast or Unicast IP address to send data (destination IP)
DEST_IP=225.2.2.10

#Multicast or Unicast port to send data (destination port)
DEST_PORT=5151

#Time to live for the packets setting (1 indicates local network)
TTL=1

#Inter-packet sleep timer:
#Time between packet transmissions ( choose from 0 to 65535 microseconds)
MICROSLEEP=10

#Log file path - NULL disable logging (default)
LOG_FILE=NULL

#Random Timers Distribution: 0 uniform 1 exponential
TIMER_DISTRIBUTION=0

#Timer parameters
# Timers values are obtained in the intervals:
```

```
# (TIMER_PARAM_A*T,(TIMER_PARAM_A+TIMER_PARAM_B)*T)
#   for NAKs
# (TIMER_PARAM_C*T,(TIMER_PARAM_C+TIMER_PARAM_D)*T)
#   for wait for retransmitions
# (TIMER_PARAM_E*T,(TIMER_PARAM_C+TIMER_PARAM_F)*T)
#   for for retransmitions
# Where
#  TIMER_PARAM_A, TIMER_PARAM_B, TIMER_PARAM_C,
#  TIMER_PARAM_D, TIMER_PARAM_E and
#  TIMER_PARAM_F are integer constants
#  T is the estimated one-way delay to the senders
TIMER_PARAM_A=2
TIMER_PARAM_B=2
TIMER_PARAM_C=5
TIMER_PARAM_D=2
TIMER_PARAM_E=2
TIMER_PARAM_F=2

#Host related parameters and timers:
# Must contain exactly the following lines:
#   HOSTS_IDENTIFIED=0
#   DEFAULT <AVERAGE_ESTIMATED_DELAY>
#   host1   <ESTIMATED_ONE-WAY_DELAY_TO_host1>
#   host2   <ESTIMATED_ONE-WAY_DELAY_TO_host2>
#   ...
#   hostN   <ESTIMATED_ONE-WAY_DELAY_TO_hostN>
# If HOSTS_IDENTIFIED=0 then we will read only the DEFAULT
# estimated delay.
HOSTS_IDENTIFIED=0
DEFAULT 300

#Max number of naks that can be sent for each packet. 100 (default)
MAX_NAK=100

# We will be able to retransmit the last MAX_MEMBER_CACHE_SIZE packets
# from each member of the multicast group, i.e., we will store the last
# MAX_MEMBER_CACHE_SIZE PACKETS from each member of the multicast group
# in the cache. 4000 (default)
#
# WARNING: if you set MAX_MEMBER_CACHE_SIZE to low values the protocol
# may fail!
#
MAX_MEMBER_CACHE_SIZE=4000

#Enable support for new members 1 enabled (default), 0 disabled
NEW_MEMBER_SUPPORT=0

#Show transmission statistics: 0 disabled (default) 1 enabled
STATISTICS=0

#Time between transmission of refresh messages (seconds)
REFRESH_TIMER=10

#Loss simulation: 0 disabled (default) any float number > 0 enabled
#
# A note about loss simulation:
# When loss simulation is enabled (LOSS_PROB > 0) we always loose the
# first 10 received packets, and the first received data packet -
```

```
# that is, the first burst of received packets.
# After that, packets are lost according to LOSS_PROB.
# Example: LOSS_PROB=30
# The first 10 received packets will be lost.
# Then, 30% of the packets will be lost
LOSS_PROB=0

# Time to wait, in microseconds, before leaving the multicast group.
LEAVE_GROUP_WAIT_TIME = 500000

# Size of the buffer of the receiver host
# (maximum size of a message that may be processed by the receiver host).
RCV_BUFFER_SIZE = 10000
```

10.2.3 mcastproxy

When IP Multicast is not available the *mcastproxy* program can be used to connect hosts. Suppose that students from four different campi want to use TGWB. Suppose also that multicast is available in the local networks but it is not available among the campi. In that scenario, the *mcastproxy* program can be used to connect each campi. As shown in Figure 10.2, there is one instance of *mcastproxy* in each campus. Each *mcastproxy* instance can listen to local multicast traffic and send it unicast to each other campus. The packets received through unicast are sent through multicast to local users.

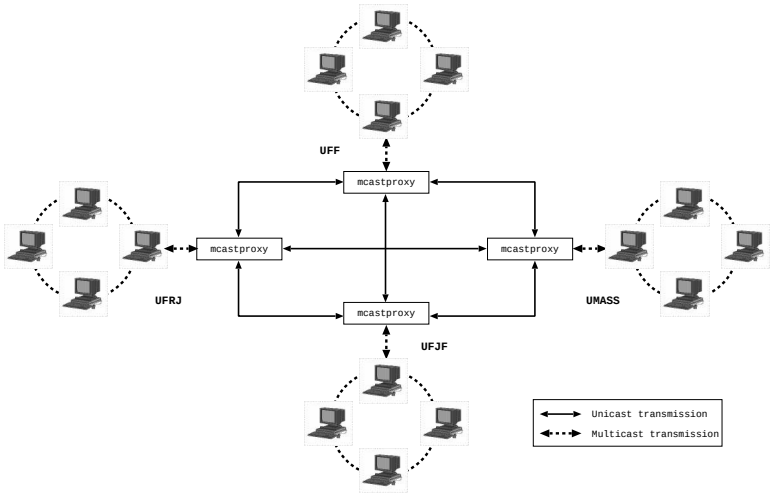


Figure 10.2: *mcastproxy* environment example

mcastproxy program uses a configuration file, **mcastproxy.conf** from the *.tgwb* directory located at user's home. Next is presented an example of this file.

```
#Multicast group address
GROUPADDR=225.1.2.3
```

```
#Number of hosts to send unicast packets
NADDR=2
#IP Addresses list to send unicast packtes
ADDRLIST
192.168.1.2
10.0.0.1
# Time to live for multicast packets
TTL=1
REUSEADDR=0
# Loopback: 1 enable, 0 disable
LOOPBACK=1
# Unicast port to use
UCASTPORT=32566
# Multicast port to use
MCASTPORT=5151
```

Chapter 11

Modeling Tool Kit

11.1 Introduction

The Modeling Tool Kit (MTK) is a programming framework, for TANGRAM-II, where users can develop different types of mathematical models and algorithms, and use them together in a single environment. It offers its users the ease of working, at the same time, under the same tool, with distinct models, of different types, apply them to the same problem, and compare their results. It also allows users to expand the set of available types of models, by programing and inserting new, customized ones, into the tool.

MTK is build up of, basically, two parts: a main structure, composed by the MTK interface, and a set of different **plugins**, that can be added or removed by the user. Figure 11.1 shows this architecture. The MTK interface resides in the highest layer level and, through it, users can interact with the available plugins. The interface/plugin communication is done through a library, called `libmtk`, which, therefore, resides in the intermediate layer of the hole structure. Finally, the plugins, which are, in fact, a blueprint from which different models are created, reside in the lowest layer level.

Each plugin works as black box, and defines a specific type of mathematical model. Its code must be self-contained, in such a way that its implementation cannot depend on other plugins. By this, it is guaranteed that a plugin removal (or addition) will not compromise the entire tool or the other plugins. Just like a *class* in the object oriented paradigm, each plugin is composed of attributes and methods, which all models created from that plugin share, and which can be accessed or executed by the user.

In order to enable plugin communication with the MTK structure and with other plugins, the types of models implemented in the MTK framework must follow a programming template. This fact should not be viewed as a limitation, since language freedom is not attenuated, and the programmer has only to adjust the way he or she implements the code. From the users point of view, it is much more comfortable to work with all models in a single environment than having many standalone programs, with distinct input and output

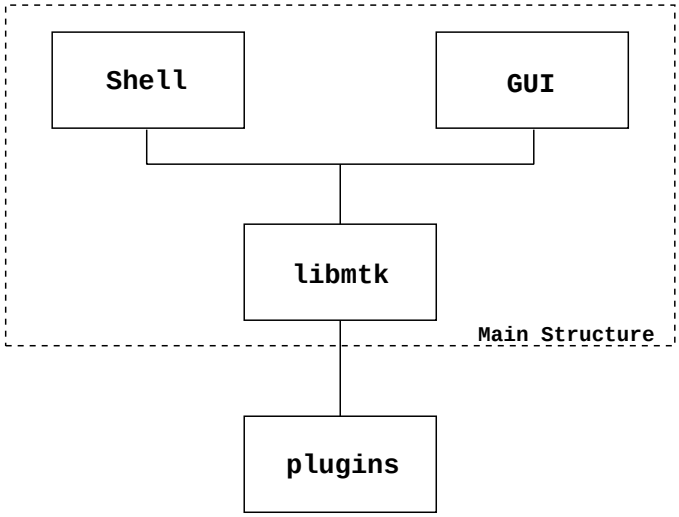


Figure 11.1: MTK block architecture

standards.

The MTK framework was develop during the work of [44], as the author realized the similarity between the studied models, and needed to compare their results using the same input data. Since then, some other authors have contributed with new plugins.

11.2 Getting Started

In this section, we will take you through a short tutorial, to help you get acquainted with the main features of MTK. If you are already familiar with them, and are looking for detailed information on a specific plugin, we recommend you jump right to section 11.5, were we describe, individually, each of the available plugins. Our goal, here, will be to build a hidden Markov model to adequately represent a coin-tossing game, described below, and use it to forecast its future outcomes.

Suppose you are in a casino and, while walking through the main floor, you notice that there is a new strange game available, called *Thank Paty the Parrot!*. Curious, you walk up to the table, which has a green parrot standing on top of it, and ask the dealer about the game. Its rules are very simple. There are two coins inside a small basket, over the table, in front of the parrot. In each round, the parrot randomly chooses one, picking it up with his beak, and hands it over to the dealer. The dealer, then, flips the coin into the air. If it lands showing tails, you win 1 dollar for every dollar bet, doubling your money. Otherwise, if it lands showing heads, the casino keeps all the money you bet. After the bets are payed, the dealer puts the coin back into the basket, shuffles it, and places it, again, in front of the parrot.

When asked about the coins, the dealer says that, despite being visually identical, they have different biases. One has a significant greater probability of showing heads, and the other has a significant probability of showing tails. So your chances of winning are conditioned on the parrots choice, thus the name *Thank Paty the Parrot!*. Paty, as you, can't tell the difference between both coins, but knows that every time he chooses one, he gets a nice pat on the head.

After hearing the explanation, you remember having read, just last week, a paper on hidden Markov models (HMM) [39], and how they can be applied to situations just like the one described to you. Feeling that, finally, all that studying might pay off, you convince yourself that this is the game that's going to get you some money!

But before playing, you have to build your HMM model. To do this, you must first collect some observations, that will be used to estimate the parameters of the model. So you decide to observe others playing, and take notes on the outcomes of the coins. Your observations are recorded in the file *trace.txt*[1] in which a 1 represents tails, and a 0 represents heads. After collecting 1000 observations[2], you head home and, using MTK, start building your HMM model.

11.2.1 Setting Up MTK

Before running MTK, you must first specify the directory were the plugins - the <**plugin_name**>**.pgn** files - are located. This can be done by setting a **MTK_PATH** environmental variable to point to this location. Since these plugins will usually be located in the *$TANGRAM2_HOME/lib/mtk_plugins/* directory, all you have to do is insert a line, similar to the one bellow, in your *.bashrc* file.

```
export MTK_PATH=$TANGRAM2_HOME/lib/mtk_plugins/
```

If, for some reason, you have other plugins located elsewhere, or just want to specify a new directory without changing the MTK_PATH variable, you can use the `-p <directory>` option, in the command line, when running mtk.

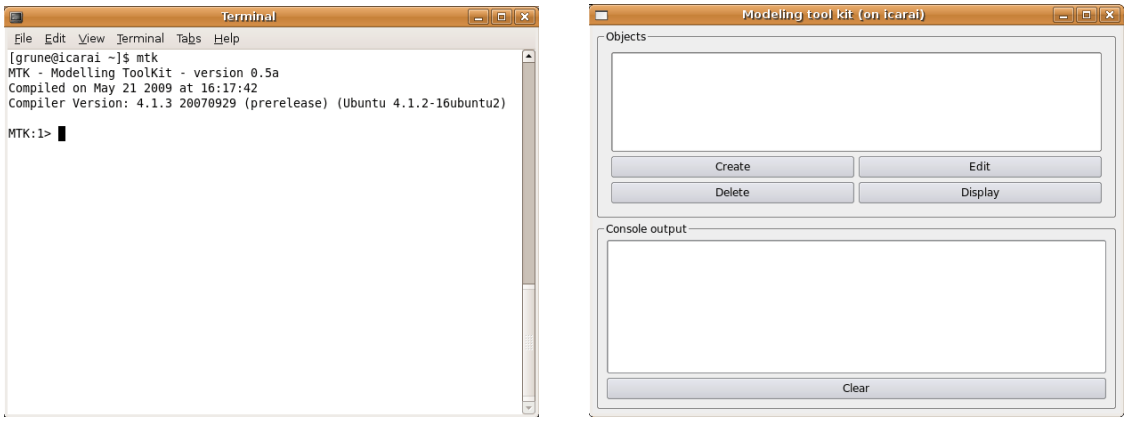
11.2.2 Starting MTK

Start MTK by typing `mtk` on the command line. The command line interface will appear, as shown in Figure 11.2(a). MTK has also a graphical interface, shown in Figure 11.2(b), that can be called upon by using the `-g` option: `mtk -g`.

In the rest of this manual, we will focus on the command line interface, since the graphical one is intuitive.

[1] This file is part of MTK, and can be found in <diretório X>.

[2] OK, so this is an unreal number of observations to collect in a scenario like this, but so is a parrot standing on top of a casino table, right?



(a) (b)

Figure 11.2: MTK interfaces: (a) shell interface; (b) graphical interface.

11.2.3 First Steps

After we have started MTK, our first step will be to check which plugins are available, and if we have one that allows us to work with a hidden Markov model. This can be done with the `list` command:

Example:

```
MTK:1> list plugins
-----------------
Available Plugins
-----------------
           example - Example plugin
        floatvalue - Float Value Samples
              ghmm - Hierarchical Gilbert Hidden Markov Model
               hmm - Hidden Markov Model
         hmm_batch - Hierarchical General Hidden Markov Model - Fixed Batch
hmm_batch_variable - Hierarchical General Hidden Markov Model - Variable Batch
          intvalue - Integer Value Samples
```

To obtain information about any plugin, or even about some plugin method, use the `help` command:

`help` <*plugin_name*>
or
`help` <*plugin_name*>.<*method_name*>

Example:

```
MTK:2> help hmm
This plugin defines a discrete-time, discrete-space hidden Markov
model. Both observations and hidden states are non-negative 32-bit
integers.
    Constructors: hmm( )
                  hmm( <N>, <M> )
    Where:
        <N> - number of hidden states.
        <M> - number of observation symbols.

--------------------
Available attributes
--------------------
         N - number of hidden states
         M - number of observation symbols
     pi[i] - initial probability for the i-th state
   A[i][j] - transition probability from state i to state j
   B[i][j] - probability of symbol j at state i
 result[i] - result array of last executed command

------------------
Available displays
------------------
 all - model parameters
  pi - initial state distribution
   A - state transition matrix
   B - observation probability matrix

-----------------
Available methods
-----------------
                  load                  save             normalize
              training              simulate               viterbi
            likelihood              forecast   import_from_tangram
               set_full            set_coxian               set_qbd
            set_gilbert              fix_full            fix_coxian
                fix_qbd           fix_gilbert           set_epsilon
          symb_sum_dist             symb_tavg            state_prob
```

So far so good! MTK has a plugin that allows us to work with a hidden Markov model - the `hmm` plugin. Now let's get to work!

11.2.4 Creating and Working with Objects

As we said in section 11.1, each plugin is a blueprint from which models are created, just like a *class* in the object oriented paradigm. Using this same analogy, we will call each model created, from a specific plugin, an *object*. MTK allows you to create, and work, at the same time, with different objects, created or not from the same plugin. This is very useful when we want to test the same type of model, only with different parameters, to the same problem.

This said, its time to create our HMM model. Any object, on MTK, can be created with the `new` command:

<*object_name*> `=` `new` <*plugin_name*> `(` <*parameters*> `)`

Example:

```
MTK:3> game_model = new hmm( 2, 2 )
hmm object was successfully created with name game_model
Parameters were initialized with random values.
Model error tolerance is epsilon = 0.000010
```

We have chosen to create a HMM with two states and two symbols for obvious reasons: there are only two coins, hence, we will use one state represent each; there are, also, only two possible outcomes, heads or tails, and thus, we need only two symbols.

After creating our HMM model, we need to load, into MTK, the observations collected, stored in the *trace.txt* file, which we will use to adjust the parameters of our model. MTK has two plugins that reproduce an array of elements, and can be used to work with observation samples: the `intvalue` plugin, and the `floatvalue` plugin. The first one handles only integer value data, and the second, float value data. Since our observations are composed of, only, 0's and 1's, we will use the `intvalue` plugin. This said, let's create and load our trace file into MTK.

Example:

```
MTK:4> trace = new intvalue( )
MTK:5> trace.load( "trace.txt" )
```

A plugin usually implements some `display` function, which shows, on screen, some of the objects attributes. Specifically, the `intvalue` plugin has a display called *stats*, which calculates and shows some statistics of the object.

Example:

```
MTK:6> trace.display( stats )
'stats' at 'trace'

Minimum:  0
Maximum:  1
Mean:     0.48
Variance: 0.24985
```

We are now ready to adjust our HMM model parameters. This can be accomplished with the `training` command, present in the `hmm` plugin. It estimates the model parameters by maximizing the likelihood of a sample, given as input, using the Baum-Welch algorithm[5]. To obtain information on this, or any other method, from any other plugin, you can use the `help` command:

`help` <*plugin_name*>.<*method_name*>

Example:

```
MTK:7> help hmm.training
Estimates the model parameters by maximizing the likelihood of a
given observation sample. In case this sample is composed of
incomplete data (observations only) the Baum-Welch algorithm is
used. Multiple observation samples can also be used.
    Usages: training( <it>, [<thr>], <object1> [, ... ] )
            training( <object1>, <object2> )
    Where:
        <it>         - number of iterations to perform in the
                       Baum-Welch algorithm.
        <thr>        - log-likelihood threshold to stop training.
        <object1>    - the object containing the observations.
        <object2>    - the object containing the states path.
```

As it is well known, the reestimation equations, used by the Baum-Welch algorithm, give values of the HMM parameters which correspond to a local maximum of the likelihood function[39], but they do not guarantee that this maximum is also the global maximum. Therefore, the initial estimates of these parameters, chosen by the user, play, usually, an important role on the estimation. By default, MTK initializes the parameters of a `hmm` object with random values. Hence, in order to try and get a better estimation, we will, before training our model, change these initial values.

Every attribute, from every plugin, can be edited by the user, at any time. What we want to do, is to change the values of the `hmm` attributes `pi`, `A` and `B`, which correspond, respectively, to: the initial state distribution, the state transition probability matrix, and

the symbol emission probability matrix. We know a bit about the symbol emission probability. Recall that, despite not telling us the exact bias of each coin, the dealer told us that it is significant. The problem is, what does he mean by "significant"? Let's take a guess. Say each coin has a bias of 0.6 probability. Since we don't know anything about the way the parrot chooses the coins, let's assume that he is totally unbiased. With these assumptions made, it is time to change the desired attributes of our model. This can, normally, be accomplished using the following syntax:

<object_name>.<attribute_name>`[`*<index>*` ] = `*<value>*

Example:

```
MTK:8>game_model.pi[0] = 0.5; game_model.pi[1] = 0.5;

MTK:9>game_model.A[0][0] = 0.5; game_model.A[0][1] = 0.5;
      game_model.A[1][0] = 0.5; game_model.A[1][1] = 0.5;

MTK:10>game_model.B[0][0] = 0.6; game_model.B[0][1] = 0.4;
       game_model.B[1][0] = 0.4; game_model.B[1][1] = 0.6;
```

After setting the initial parameter values, and reading the help information above, on the `training` method, estimating the parameters of our model should be straightforward. We will use 1000 iterations.

Example:

```
MTK:11> game_model.training( 10000, trace )
#  iteration       log-likelihood           likelihood
           0    -6.9314718056e+02    9.3326361852e-302
           1    -6.9237923384e+02    2.0114968500e-301
           2    -6.9233799381e+02    2.0961853190e-301

         ...           ...                  ...

         998    -6.9073270968e+02    1.0437481465e-300
         999    -6.9073257832e+02    1.0438852618e-300
        1000    -6.9073244705e+02    1.0440222957e-300
```

Let's take a look at the estimated parameters. This can be easily done using the `hmm`'s `display` function:

Example:

```
MTK:12> game_model.display( all )
'all' at 'game_model'

Number of states:  2
Number of symbols: 2

Initial state distribution:
 [ 0.00000e+00 1.00000e+00 ]

State transition probabilities:
 [ 4.72127e-01 5.27873e-01 ]
 [ 5.54976e-01 4.45024e-01 ]

Symbol observation probabilities:
 [ 8.86456e-01 1.13544e-01 ]
 [ 1.33456e-01 8.66544e-01 ]
```

Now that we have our model at hand, it is a good idea to save it in a file, so we can use it in the future. This can be done with the `save` command, available to most plugins:

Example:

```
MTK:13> game_model.save( "game_model_parameters.txt", "all" )
Hmm was successfully saved in file game_model_parameters.txt.
```

So, its finally time to make some money! In order to accomplish this, we will try and forecast the next outcome of the game, given the last 3, and, observing the forecasted values, decide if it is time to bet our money, or hold off a bit more. Fortunately, the `hmm` plugin has a function, called `forecast`, that does just that! Given some previous sample, it calculates the probability of each symbol, in each future time step. Thus, as a result, we will have the probability, based on our model, of showing heads or showing tails in the next round.

Say the most recent 3 outcomes where all heads, and are stored in the *recent_outcome.txt* file. So, first, we load them into MTK, using a new `intvalue` object, and then use the `forecast` function:

Example:

```
MTK:14> recent_obs = new intvalue()
MTK:15> recent_obs.load( "recent_outcome.txt" )
Intvalue was successfully loaded.
```

```
MTK:16> game_model.forecast( 1, recent_obs )
(Time Step 1): distribution: [ 4.9768404831e-01 5.0231595169e-01 ];
               most probable symbol: 1;
               entropy (in bits): 9.9998452377e-01
```

As we can see, given these most recent 3 observations, there is a slight bigger chance of, in the next round, showing tails than showing heads. If you are risk tolerant, then it might be the time to bet some money!

The `hmm` plugin has also another method that can be used for forecasting, called `symb_sum_dist`, which calculates the probability distribution of the sum of the symbol's values emitted in a time window of size F, given some previous history. With it, we can calculate the probability of, in the next 5 rounds, for example, showing one tail, two tails, etc., given the 3 most recent outcomes.

Example:

```
MTK:17> game_model.symb_sum_dist( recent_obs, 5 )
# sum(Obs) P[sum(Obs) after 5 time steps]
     0            3.0514042169e-02
     1            1.6531242933e-01
     2            3.3398035017e-01
     3            3.1275828032e-01
     4            1.3558484787e-01
     5            2.1850050151e-02

# estimated average of sum(Obs): 2.4231376128e+00
```

By looking at the result above, we can notice that, in the next five consecutive rounds, the overall chances of the casino winning are better than the chances of a player winning.

This concludes our short tutorial. By now, you should feel more comfortable using MTK, and exploring its features. In the next two sections, we describe, in details, all of MTK's main commands, and every available plugin.

11.3 MTK's Main Commands

In the section, we describe the basic, plugin independent, commands of MTK.

11.3.1 `Help`

Displays help messages for plugins, plugin methods, or MTK commands.

Usage: `help` <*keyword*>

where <*keyword*> may be either a `<plugin_name>`, a MTK `<command_name>`, or a plugin method, in which case, the input should be `<plugin_name>.<method_name>`.

11.3.2 `List`

Lists either the available plugins or the instantiated objects.

Usage: `list` <*option*>

where <*option*> can be either `plugins`, if the available plugins are to be listed, or `objects` if the instantiated objects are to be listed.

11.3.3 `Set`

This commands serves three purposes: it activates an instantiated object; it sets the MTK's output terminal; and, in case the output terminal is set to a file, it can be used to specify this file. In the following paragraphs, we, individually, describe each of these uses.

It was shown in section 11.2 that, any method, of any object, can be executed by typing, in the command line, `<object_name>.<method_name>(<parameters>)`. However, it seems cumbersome to have to, every time, type the object's name. So MTK allows the user to *activate* an object. Whenever an object is *active*, its methods can be called upon by typing only their names; the objects name, in this case, can be omitted. When there is only one object instantiated, it will, evidently, be active at all times. But when there are two or more objects instantiated, the user can activate one or the other by using the `set` command.

Besides activating objects, `set` can be used to set MTK's output terminal, to either the screen or a file. If the terminal is set to 'file', then this command can be used, again, to specify the name of the file. This feature allows the user to save the desired outputs of his experiment, in order to study them later, or in a more careful way.

Usage: `set` <*attribute*> <*value*>

where <*attribute*> can be either `active`, in which case <*value*> should be the name of the object to be activated; `terminal`, in which case <*value*> should be either `file` or `screen`; or `output`, if terminal is set to 'file', in which case <*value*> should be the name (or path) of the destination file.

11.3.4 `Show`

Shows the value of any MTK attribute. These attributes are those set with the `set` command, whose purposes where explained above.

Usage: `show` <*attribute*>

where <*attribute*> can be either `active`, `terminal`, `output`, or `version`, in which case MTK's version number is printed on screen.

11.3.5 `Quit`

Quits the MTK's session. This can also be done typing `Ctrl+d`.

Usage: `quit`

11.4 Creating and Deleting Objects

As was said in section 11.2.4, each plugin is a blueprint from which objects are created. There is no limit on the number of objects that can be instantiated, during the same session, from the same plugin. This allows the user to work with several instances of the same model, but with different parameters, and compare their results, when applied to the same problem.

Any object, on MTK, can be created with the `new` command:

Usage: <*object_name*> `=` `new` <*plugin_name*>*(*<*parameters*>*)*

where <*object_name*> is the name given to the object; <*plugin_name*> is the name of the plugin from which the object will be created; <*parameters*> are the parameters requested by each plugin, which can be found on the plugin's help message.

Deleting an object is even easier than creating it. To delete any object, on MTK, just use the `delete` command:

Usage: `delete` <*object_name*>

where <*object_name*> is the name of the object to be deleted.

11.5 Available Plugins

A plugin is a blueprint from which objects are created, just like a *class* in the object oriented paradigm. It is composed of **attributes**, **displays** and **methods**. The **attributes**

describe the characteristics of a plugin, and thus, each attribute usually represents some specific parameter. The **displays** shows, on screen, the values of each **attribute**. Finally, a **method** is a group of actions that are executed to some specific purpose, like generating forecasts, or estimating parameters.

In this section we describe, individually, and in details, the currently implemented plugins. Our approach intends to be complete, but not exhaustive. For any additional information, the user is advised to use the `help` command, available to each plugin.

In order to make our exposure as comprehensible as possible, we have decided to separate the plugins into two main categories, according to their use: *data manipulation plugins* and *mathematical model plugins*.

Data Manipulation Plugins

The following plugins allow the user to manipulate data samples, and can be used to store the results produced by some of the other plugin's methods.

11.5.1 Intvalue Plugin

The `intvalue` plugin implements an array of integer value elements. It behaves, essentially, like an ordinary array of elements, in which every integer is stored in a specific position, and is assigned a unique subscript. It is commonly used to handle data traces, and can be used as the input or output parameter for other plugin's methods.

Constructors

This plugin has two different constructors:

`intvalue( )`
`intvalue( <`*file_name*`> )`

where the first one creates an empty array, and the second, an array whose elements are loaded from the <*file_name*> file.

Attributes

1. `lower`: Specifies a lower bound for the elements.
2. `upper`: Specifies an upper bound for the elements.
3. `size`: Specifies the array's size.
4. `data[i]`: Returns the i-th element of the array.

Displays

1. `sample`: Prints all the elements of the array.

2. `bounds`: Prints the array's lower and upper bounds.

3. `stats`: Prints the following statistics of the array: minimum value, maximum value, mean and variance.

Methods

11.5.1.1 `Load`

Loads a set of elements from a file.

Usages: `load(`<*file_name*>`)`
: `load(`<*file_name*>, <*offset*> *[,* <*size* >*]*`)`

where <*file_name*> is the name of the file from which the elements are loaded; <*offset*> is the number of elements to skip, from the file, before loading; and *[,* <*size*>*]* is an optional parameter, which specifies the number of elements to be loaded.

11.5.1.2 `Save`

Saves the elements of the `intvalue` array in a file.

Usages: `save(`<*file_name*>`)`
: `save(`<*file_name*>, <*offset*> *[,* <*size* >*]*`)`

where <*file_name*> is the name of the file into which the elements are saved; <*offset*> is the number of elements to skip, from the `intvalue` array, before saving; and *[,* <*size*>*]* is an optional parameter, which specifies the number of elements to be saved.

11.5.1.3 `Truncate`

Truncates the elements of the the `intvalue` array between two values.

Usage: `truncate(`<*lower*>, <*upper*>`)`

where <*lower*> is the lower bound to which smaller elements are truncated, and <*upper*> is the upper bound to which bigger elements are truncated.

11.5.1.4 `Autocorrelation`

Calculates the autocorrelation of the elements from the `intvalue` array.

Usage: `autocorrelation(`<*lag*> *[,*<*start*>*]*`)`

where <*lag*> is the maximum autocorrelation lag that will be computed and *[,*<*start*>*]* is an optional parameter that specifies the position, in the `intvalue` array, from which to start calculating the autocorrelation. If this parameter is not specified, the calculation will start from the beginning of the array.

Input and Output File Format

Every file read (written) by the `intvalue` plugin must be in the following format: the first line contains the number of elements in the file, and each subsequent line contains one, and only one, element.

Example:

```
5
0
1
0
1
0
```

11.5.2 Floatvalue Plugin

The `floatvalue` plugin implements an array of floating point elements. It behaves, essentially, like an ordinary array of elements, in which every float is stored in a specific position, and is assigned a unique subscript. It is commonly used to handle data traces, and can be used as the input or output parameter for other plugin's methods.

Constructors

This plugin has two different constructors:

```
floatvalue( )
floatvalue( <file_name> )
```

where the first one creates an empty array, and the second, an array whose elements are loaded from the <*file_name*> file.

Attributes

1. `lower`: Specifies a lower bound for the elements.
2. `upper`: Specifies an upper bound for the elements.
3. `max_bound`: Specifies the maximum absolut value that an element may have. Thus, the element's value range is [−max_bound,max_bound].
4. `size`: Specifies the array's size.
5. `data[i]`: Returns the i-th element of the array.

Displays

1. `sample`: Prints all the elements of the array.
2. `bounds`: Prints the array's lower and upper bounds.
3. `stats`: Prints the following statistics of the array: minimum value, maximum value, mean and variance.

Methods

11.5.2.1 `Load`

Loads a set of elements from a file.

Usages: `load(`<*file_name*>`)`
: `load(`<*file_name*>, <*offset*> *[,* <*size* >*]*`)`

where <*file_name*> is the name of the file from which the elements are loaded; <*offset*> is the number of elements to skip, from the file, before loading; and *[,* <*size*>*]* is an optional parameter, which specifies the number of elements to be loaded.

11.5.2.2 `Save`

Saves the elements of the `floatvalue` array in a file.

Usages: `save(`<*file_name*>`)`
: `save(`<*file_name*>, <*offset*> *[,* <*size* >*]*`)`

where <*file_name*> is the name of the file into which the elements are saved; <*offset*> is the number of elements to skip, from the `floatvalue` array, before saving; and *[,* <*size*>*]* is an optional parameter, which specifies the number of elements to be saved.

11.5.2.3 `Truncate`

Truncates the elements of the the `floatvalue` array between two values.

Usage: `truncate(`<*lower*>, <*upper*>`)`

where <*lower*> is the lower bound to which smaller elements are truncated, and <*upper*> is the upper bound to which bigger elements are truncated.

11.5.2.4 `Autocorrelation`

Calculates the autocorrelation of the elements from the `floatvalue` array.

Usage: `autocorrelation(`<*lag*> *[,*<*start*>*]*`)`

where <*lag*> is the maximum autocorrelation lag that will be computed; and *[,*<*start*>*]* is an optional parameter that specifies the position, in the `floatvalue` array, from which to start calculating the autocorrelation. If this parameter is not specified, the calculation will start from the beginning of the array.

Input and Output File Format

Every file read (written) by the `floatvalue` plugin must be in the following format: the first line contains the number of elements in the file, and each subsequent line contains one, and only one, element.

Example:

```
5
0.6
1.0
0.3
1.2
0.1
```

Mathematical Model Plugins

The following plugins allow users to work with several, distinct, mathematical models. Each one implements a specific model, and supplies the user with various algorithms associated with it. Next, we describe, individually, each of these plugins.

11.5.3 Hidden Markov Model Plugin

The Hidden Markov Model plugin implements a discrete-time, discrete-space hidden Markov model (HMM)[39]. Its allows users to build a HMM, estimate its parameters (given some observation sample), produce forecasts, etc. Its basic code was implemented during the work [24] and, later, was adapted to the MTK paradigm by [44].

Constructors

The `hmm` plugin has two different constructors:

`hmm( )`
`hmm(` <*N*>, <*M*> `)`

where the first one creates an empty hidden Markov model with 0 states and 0 observation symbols (it assumes its parameters will be loaded hereafter); and the second one creates a HMM with <*N*> hidden states and <*M*> observation symbols, and initializes its parameters with random values, satisfying the stochastic constrains. Each of the N states is associated with a unique integer number, ranging from 0 to $N-1$, which identifies the state. The same is true for the observation symbols. Each of the M symbols is represented by a unique integer number, whose value can range from 0 to $M-1$.

Attributes

1. `N`: Number of hidden states.
2. `M`: Number of observation symbols.
3. `pi[i]`: Initial probability of the i-th hidden state.
4. `A[i][j]`: Transition probability from hidden state i to hidden state j.
5. `B[i][j]`: Probability of symbol j at hidden state i.
6. `result[i]`: Resulting array of last executed method.

Displays

1. `all`: Prints all model parameters.
2. `pi`: Prints only the initial hidden state distribution.
3. `A`: Prints only the hidden state transition matrix.
4. `B`: Prints only the symbol emission probability matrix.

Methods

11.5.3.1 `Load`

Loads a model description (its parameters) from a file.

Usage: `load(`<*file_name*> *[,* <*attribute*>*]*`)`

where <*file_name*> is the name of the file from which the description is loaded; and <*attribute*> is an optional parameter, which allows users to load only one specific parameter. It may be {`pi`, `A`, `B`, `all`}, if the parameter that is to be loaded is pi, A, B, or all of the previous, respectively.

Warning: When loading only specific parameters, like `A`, make sure that the file from which it will be loaded has only that parameter stored in it, and is in the format specified at the end of this section.

11.5.3.2 `Save`

Saves a model description (its parameters) to a file.

Usage: `save(`<*file_name*> *[,* <*attribute*>*]*`)`

where <*file_name*> is the name of the file into which the description will be saved; and <*attribute*> is an optional parameter, which allows users to save only one specific parameter. It may be {`pi`, `A`, `B`, `all`}, if the parameter that is to be saved is pi, A, B, or all of the previous, respectively.

11.5.3.3 `Normalize`

Normalizes the model's parameters to satisfy the stochastic constrains. The normalization is done by dividing every element of a probability vector by the sum of the elements of this vector.

Usage: `normalize()`

11.5.3.4 `Model Parameter Estimation`

Estimates the HMM's parameters by maximizing the likelihood of a given observation sample. In case this sample is composed of incomplete data (observations only) the Baum-Welch algorithm[5] is used. Multiple observation samples may also be used.

Usages: `training(` <*it*> *[,*<*thr*>*],* <*object_1*> *[, ...]*`)`

: `training(` <*object_1*>, <*object_2*> `)`

where <*it*> is the maximum number of iterations that the Baum-Welch algorithm will perform before stopping; <*thr*> is the log-likelihood gain[3] threshold that, when reached, will stop the estimation algorithm; <*object_1*> is the MTK object containing the observation sample; and <*object_2*> is the MTK object containing the state path that generated the observations in <*object_1*>.

Output: Shows the progress of the estimation, by printing the current iteration and the likelihood of the observation sample used in the training, given the current parameter estimates. The values estimated for the parameters are automatically stored in the `hmm` object's attributes.

11.5.3.5 `Likelihood`

Calculates and displays the probability that a given observation sample was generated by the current model parameters.

Usage: `likelihood(` <*object_1*> *[,*<*object_2*>*]* `)`

where <*object_1*> is the MTK object containing the observation sample; and <*object_2*>, an optional parameter, is the MTK object containing the state path that generated the observations.

Output: Prints both the likelihood and the log-likelihood of the observation sample in <*object_1*>.

11.5.3.6 `Viterbi`

Implements the Viterbi algorithm[39]. Given an observation sample, determines the sequence of hidden states, known as *state path*, that is most likely to have generated it.

Usage: `viterbi(` <*object_1*> ,<*object_2*> `)`

where <*object_1*> is the MTK object containing the observation sample; and <*object_2*> is the MTK object used to store the state path evaluated.

Output: Prints the initial and the last state of the state sequence determined, and the log-likelihood (under the name **Score**) that the observation sample was generated by that

[3] By log-likelihood gain we mean the difference between the log-likelihood in iteration i and iteration $i-1$.

particular sequence of states. As mentioned above, the complete state path is stored in *<object_2>*.

11.5.3.7 `Simulate`

Generates a random sample path from the model.

Usage: `simulate(` *<sample_size>, <object_1> [,<object_2>]*`)`

where *<sample_size>* is number of samples that will be generated; *<object_1>* is the MTK object used to store the observations generated; and *[<object_2>]*, an optional parameter, is the MTK object used to store the state path from which the observations where generated.

11.5.3.8 `Forecast`

Calculates and displays the symbol's probability distribution at each time unit, in a forecasting time interval, given some previous observation history.

Usage: `forecast(` *<F>, <object_1> [,<object_2>]*`)`

where *<F>* is the size, in time units, of the forecasting time interval; *<object_1>* is the MTK object containing the observation history; and *[<object_2>]*, an optional parameter, is the MTK object used to store the most likely symbol of each distribution of each time unit.

Output: Prints, for each time unit, its symbol probability distribution, the most probable symbol of this distribution, and the entropy[8], in bits, of the distribution.

11.5.3.9 `Symbol Value Time Average`

Recall that each symbol in the HMM is represented by a unique integer-valued number. This method assumes the system is in steady state, and calculates the long term time average (when $t \to \infty$) of the values of the emitted symbols. In other words, it calculates the expected symbol value that will be observed in one symbol emission.

Usages: `symb_tavg()`

Output: Prints the symbol's average value emitted, over time, when $t \to \infty$.

11.5.3.10 `Symbol Value Sum Distribution`

Given a time window of size F, and an observation history H, calculates and displays the distribution of the sum of the emitted symbol's values, in F, given H. That is, it evaluates $P\left[\sum_{t=1}^{t=F} O_t|H\right]$ for every possible sum $O_1+O_2+\cdots+O_F$. Evidently, this method assumes the symbols can be added one to another.

Note: To clarify things, let's look at an example. Suppose you are working with a model which has only two symbols, whose values are 0 and 1. In a time interval of $F = 2$ time units, the possible symbol outcomes are $\{(0,0),(0,1),(1,0),(1,1)\}$. Thus, there are three possibilities for the sum of values of the symbols emitted: $\{0,1,2\}$. The `symb_sum_dist()` method will calculate the probability of each of these three outcomes.

Usages: `symb_sum_dist(` <*object_src*>, <F> `)`

where <*object_src*> is the MTK object that containing the observation history; and <F> is the size, in time units, of the time interval considered.

Output: Prints all possible symbol sum values with their respective probability, and their average value calculated from this distribution.

11.5.3.11 `State Probability`

Evaluates, for each hidden state, the probability of finding the system in that hidden state, i.e., evaluates the hidden states probability distribution. If no parameters are passed, it calculates the steady state probability distribution. If an observation sample is passed as parameter, it calculates the probabilities based on this history sample, thus assuming the model is in transient state.

Usages: `state_prob()`
: `state_prob(` <*object_src*> `)`

where <*object_src*> is the MTK object that containing the observation sample.

Output: Prints the probability of each hidden state.

11.5.3.12 `Set Full Structure`

Assigns random values for the hidden state transition matrix, `A`, the symbol emission probability matrix, `B`, and the initial hidden state probability vector, `pi`, satisfying the stochastic constrains.

Usage: `set_full(`<*N*>, <*M*>`)`

where <*N*> is the number of hidden states; and <*M*> is the number of observation symbols of the hmm model.

11.5.3.13 `Set Coxian Structure`

Assigns random values for the hidden state transition matrix, `A`, the symbol emission probability matrix, `B`, and the initial state probability vector, `pi`, satisfying the stochastic constrains, but sets the Markov chain to a coxian structure.

Usage: `set_coxian(`<*N*>, <*M*>`)`

where <*N*> is the number of hidden states; and <*M*> is the number of observation symbols of the hmm model.

11.5.3.14 `Set Quasi Birth-Death Structure`

Assigns random values for the hidden state transition matrix, `A`, the symbol emission probability matrix, `B`, and the initial state probability vector, `pi`, satisfying the stochastic constrains, but sets the Markov chain to a quasi birth-death structure.

Usage: `set_qbd(` <*G*>, <*E*>, <*M*> [, <*BACK*> [, <*FWD*>]] `)`

where <*G*> in the number of groups of states; <*E*> in the number of states (elements) per group[4]; <*M*> is the number of symbols; <*BACK*> is the number of reachable back groups; and <*FWD*> is the number of reachable forward groups.

11.5.3.15 `Set Gilbert Structure`

Transforms the `hmm` object into a Gilbert HMM model[28, 26], no matter what previous structure the object had.

Usage: `set_gilbert()`

11.5.3.16 `Fix Full Structure`

Adds the value `epsilon`, which is defined in section 11.5.3.20, to every element of the hidden state transition matrix, `A`, the symbol emission probability matrix, `B`, and the initial hidden state probability vector, `pi`, and normalizes them to satisfy the stochastic constrains. This method can be used as an easy way of changing any parameter whose value

[4] $N = G \cdot E$

might have been set to zero (by a parameter estimation, for example) and setting it to a really small value.

Usage: `fix_full(`<*N*>, <*M*>`)`

where <*N*> is the number of hidden states; and <*M*> is the number of observation symbols of the hmm model.

11.5.3.17 `Fix Coxian Structure`

Assuming that the hidden states form a coxian structure, adds the value `epsilon`, which is defined in section 11.5.3.20, to every element of the hidden state transition matrix, `A`, the symbol emission probability matrix, `B`, and the initial hidden state probability vector, `pi`, and normalizes them to satisfy the stochastic constrains.

Usage: `fix_coxian(`<*N*>, <*M*>`)`

where <*N*> is the number of hidden states; and <*M*> is the number of observation symbols of the hmm model.

11.5.3.18 `Fix Quasi Birth-Death Structure`

Assuming that the hidden states form a quasi birth-death structure, adds the value `epsilon`, which is defined in section 11.5.3.20, to every element of the hidden state transition matrix, `A`, the symbol emission probability matrix, `B`, and the initial hidden state probability vector, `pi`, and normalizes them to satisfy the stochastic constrains.

Usage: `fix_qbd(` <*G*>, <*E*>, <*M*> [, <*BACK*> [, <*FWD*>]] `)`

where <*G*> in the number of groups of states; <*E*> in the number of states (elements) per group[5]; <*M*> is the number of symbols; <*BACK*> is the number of reachable back groups; and <*FWD*> is the number of reachable forward groups.

11.5.3.19 `Fix Gilbert Structure`

Assuming that the hidden states form a Gilbert structure, adds the value `epsilon`, which is defined in section 11.5.3.20, to every element of the hidden state transition matrix, `A`, the symbol emission probability matrix, `B`, and the initial hidden state probability vector, `pi`, and normalizes them to satisfy the stochastic constrains.

[5] $N = G \cdot E$

Usage: `fix_gilbert()`

11.5.3.20 `Set Error Tolerance Value - Epsilon`

Sets the objects error tolerance value, `epsilon`. Default value is 0.00001.

Usage: `fix_gilbert(` <*float_value*> `)`

where <*float_value*> is the desired error tolerance value.

11.5.3.21 `Import From Tangram-II`

Imports a Markov chain structure from a Tangram-II model. This method is further described in section 8.2.

Usage: `import_from_tangram(`<*obj_name*>, <*num_hidden_vars*>, <*system_variable_list*>`)`

where <*obj_name*> is the tangram-II model's description file (the *.obj file*); <*num_hidden_vars*> is the number of system states which represent the hidden states; and <*system_variable_list*> is a comma separated list with the Tangram-II's model variable names, in the format 'Object.StateVariable'. This list must have <*num_hidden_vars*> elements

Input and Output File Format

Every file read (written) by the `hmm` plugin must be in the following format:

$< N >$
$< M >$

$< pi_{(N \times 1)} >$

$< A_{(N \times N)} >$

$< B_{(N \times M)} >$

where <*N*>, <*M*>, <*pi*>, <*A*> and <*B*> are the `hmm` attributes specified previously.

Example:

```
2
3
```

```
1.0
0.0

0.3 0.7
0.5 0.5

0.2 0.2 0.6
0.3 0.6 0.1
```

Note: The `load` and `save` methods can, as previously described, load/save specific attribute descriptions. In this case, the file which is read/written by these methods should have the same format described above, but include only the specific attribute which is being loaded/saved, i.e, the other attributes should be omitted from the file.

11.5.4 Hierarchical Gilbert Hidden Markov Model Plugin

The Hierarchical Gilbert Hidden Markov Model (GHMM) plugin defines a discrete-time, 0-1 observation, hierarchical HMM[44], in which a Markov chain (in this case, a gilbert model) is associated with each hidden state, and is responsible for the symbol emissions. Figure 11.3 illustrates the structure of such a two-level hierarchical model. After each

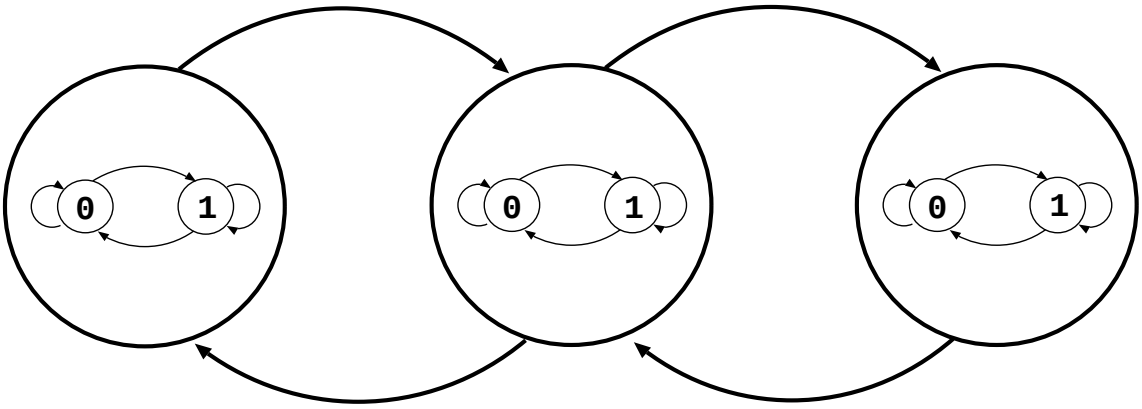


Figure 11.3: Example of a hierarchical HMM with 3 hidden states, in which a Gilbert Markov chain is associated with each hidden state.

transition (S_i, S_j) in the the hidden chain (the upper-level in this two-level hierarchy), the lower-level model associated with S_j emits symbols by transiting for a determined number of steps, called a **batch**, before the upper-level chain makes another transition[6]. A symbol

[6]The initial state in the lower-level chain is chosen according to a initial state probability distribution associated with this lower-level Markov chain. Keep in mind that after **every** transition in the upper-level

is emitted every time the lower-level Markov chain makes one transition, thus reaching one of its states. In case this lower-level chain is a gilbert model, as is the case in this plugin, symbol 0 is emitted every time the lower-level state I_0 is reached, and symbol 1 is emitted every time the lower-level state I_1 is reached, where I_k, with $k = \{0, 1\}$, defines a lower-level state. Each lower-level state emits one, and only one, symbol, and different lower-level states, inside a same hidden state, cannot both emit the same symbol. This plugin is a contribution of [44].

Constructors

The `ghmm` plugin has two different constructors:

`ghmm( )`
`ghmm(` <*N*>, <*B*> `)`

where the first one creates an empty hierarchical gilbert hidden Markov model (it assumes its parameters will be loaded hereafter); and the second one creates a GHMM with <*N*> hidden states, whose observation batch size is <*B*>, and initializes its parameters with random values, satisfying the stochastic constrains. Each of the N states is associated with a unique integer number, ranging from 0 to $N - 1$, which identifies the state. The observations emitted by the model are either 0 or 1.

Attributes

1. `N`: Number of hidden states.
2. `B`: Observation batch size.
3. `pi[i]`: Initial probability of the i-th hidden state.
4. `A[i][j]`: Transition probability from hidden state i to hidden state j.
5. `p[i]`: Transition probability from the state associated with symbol 0 to that associated with symbol 1, in hidden state i.
6. `q[i]`: Transition probability from the state associated with symbol 1 to that associated with symbol 0, in hidden state i.
7. `r[i]`: Initial probability of the state associated with symbol 1, in hidden state i.
8. `result[i]`: Resulting array of last executed method.

chain, which includes transitions to the same state, a new initial lower-level state is chosen, and the symbol emission process statrs from that state.

Displays

1. `all`: Prints all model parameters.
2. `pi`: Prints only the initial hidden state distribution.
3. `A`: Prints only the hidden state transition matrix.
4. `obs`: Prints, for every hidden state, the values of the attributes p, q and r, defined above.

Methods

11.5.4.1 `Load`

Loads a model description (its parameters) from a file.

Usage: `load(`<*file_name*> *[,* <*attribute*>*]*`)`

where <*file_name*> is the name of the file from which the description is loaded; and <*attribute*> is an optional parameter, which allows users to load only one specific parameter. It may be {`pi`, `A`, `obs`, `all`}, if the parameter that is to be loaded is pi, A, obs, or all of the previous, respectively.

Warning: When loading only specific parameters, like `A`, make sure that the file from which it will be loaded has only that parameter stored in it, and is in the format specified at the end of his section.

11.5.4.2 `Save`

Saves a model description (its parameters) to a file.

Usage: `save(`<*file_name*> *[,* <*attribute*>*]*`)`

where <*file_name*> is the name of the file into which the description will be saved; and <*parameter*> is an optional parameter, which allows users to save only one specific parameter. It may be {`all`, `pi`, `A`, `B`}, if the parameter that is to be saved is all, pi, A or B, respectively.

11.5.4.3 `Normalize`

Normalizes the model's parameters to satisfy the stochastic constrains. The normalization is done by dividing every element of a probability vector by the sum of the elements of this vector.

Usage: `normalize()`

11.5.4.4 `Model Parameter Estimation`

Estimates the GHMM's parameters by maximizing the likelihood of a given observation sample, using a variation of the Baum-Welch algorithm[5]. The reestimation equations used can be found in [44].

Usage: `training(` *<it> [,<thr>], <object>* `)`

where *<it>* is the maximum number of iterations that the Baum-Welch algorithm will perform before stopping; *<thr>*, an optional parameter, is the log-likelihood gain[7] threshold that, when reached, will stop the estimation algorithm; and *<object>* is the MTK object containing the observation sample.

Output: Shows the progress of the estimation, by printing the current iteration and the likelihood of the observation sample used in the training, given the current parameter estimates. The values estimated for the parameters are automatically stored in the `ghmm` object's attributes.

Optimized Parameter Estimation: Besides the `training()` method, the GHMM plugin also implements another parameter estimation method, called `training_fast()`. The difference between both is that while the first one calculates, for each observation in the sample, a new set of parameters for the model, the second one calculates, in a single step, these new parameters for a hole observation batch, thus improving its estimation speed by a factor of B[44]. The parameters taken by this method, and its output, are identical to those of the `training()` method:

Usage: `training_fast(` *<it> [,<thr>], <object>* `)`

11.5.4.5 `Likelihood`

Calculates and displays the probability that a given observation sample was generated by the current model parameters.

Usage: `likelihood(` *<object1>* `)`

where *<object>* is the MTK object containing the observation sample.

[7] By log-likelihood gain we mean the difference between the log-likelihood in iteration i and iteration $i-1$.

Output: Prints both the likelihood and the log-likelihood of the observation sample in <*object*>.

Optimized Likelihood Calculation: As the `training` method, the `likelihood` method has an optimized version, which works, basically in the same way as `training_fast`.

Usage: `likelihood_fast(` <*object1*> `)`

11.5.4.6 `Viterbi`

Implements the Viterbi algorithm[39]. Given an observation sample, determines the sequence of hidden states, known as *state path*, that is most likely to have generated it.

Usage: `viterbi(` <*object_1*> ,<*object_2*> `)`

where <*object_1*> is the MTK object containing the observation sample; and <*object_2*> is the MTK object used to store the state path evaluated.

Output: Prints the initial and the last state of the state sequence determined, and the log-likelihood (under the name **Score**) that the observation sample was generated by that particular sequence of states. As mentioned above, the complete state path is stored in <*object_2*>.

11.5.4.7 `Simulate`

Generates a random sample path from the model.

Usage: `simulate(` <*sample_size*>, <*object_1*> *[,*<*object_2*>*]*`)`

where <*sample_size*> is number of samples that will be generated; <*object_1*> is the MTK object used to store the observations generated; and *[*<*object_2*>*]*, an optional parameter, is the MTK object used to store the state path from which the observations where generated.

11.5.4.8 `Symbol Value Time Average`

Recall that the symbols in the GHMM are represented by either a 0 or a 1. This method assumes the system is in steady state, and calculates the long term time average (when $t \to \infty$) of the values of the emitted symbols. In other words, it calculates the expected symbol value that will be observed in one symbol emission.

Usages: `symb_tavg()`

Output: Prints the symbol's average value emitted, over time, when $t \to \infty$.

11.5.4.9 `Symbol Value Sum Distribution`

Given a time window of size F, and an observation history H, calculates and displays the distribution of the sum of the emitted symbols values, in F, given H. That is, it evaluates $P\left[\sum_{t=1}^{t=F} O_t|H\right]$ for every possible sum $O_1+O_2+\cdots+O_F$. Evidently, this method assumes the symbols can be added one to another.

Note: To clarify things, let's look at an example. Suppose you are working with a model which has only two symbols, whose values are 0 and 1. In a time interval of $F = 2$ time units, the possible symbol outcomes are $\{(0,0),(0,1),(1,0),(1,1)\}$. Thus, there are three possibilities for the sum of the symbols emitted: $\{0,1,2\}$. The `symb_sum_dist()` method will calculate the probability of each of these three outcomes.

Usages: `symb_sum_dist(` <*object_src*>, <*F*> `)`

where <*object_src*> is the MTK object that containing the observation history; and <*F*> is the size, in time units, of the time interval considered.

Output: Prints all possible symbol sum values with their respective probability, and their average value calculated from this distribution.

11.5.4.10 `State Probability`

Evaluates the hidden state probability distribution, be it on steady state, or, if an observation sample is passed as parameter, on transient state.

Usages: `state_prob()`
: `state_prob(` <*object_src*> `)`

where <*object_src*> is the MTK object that containing the observation sample.

Output: Prints the probability of each hidden state.

11.5.4.11 `Set Full Structure`

Assigns random values for the hidden state transition matrix, `A`, the symbol emission probability vectors, `p,q` and `r`, and the initial hidden state probability vector, `pi`, satisfying the stochastic constrains.

Usage: `set_full(`<*N*>, <*B*>`)`

where <*N*> is the number of hidden states; and <*B*> is the observation batch size.

11.5.4.12 `Set Coxian Structure`

Assigns random values for the hidden state transition matrix, `A`, the symbol emission probability vectors, `p,q` and `r`, and the initial hidden state probability vector, `pi`, satisfying the stochastic constrains, but sets the hidden Markov chain to a coxian structure.

Usage: `set_coxian(`<*N*>, <*B*>`)`

where <*N*> is the number of hidden states; and <*B*> is the observation batch size.

11.5.4.13 `Set Quasi Birth-Death Structure`

Assigns random values for the hidden state transition matrix, `A`, the symbol emission probability matrices, `p,q` and `r`, and the initial state probability vector, `pi`, satisfying the stochastic constrains, but sets the hidden Markov chain to a quasi birth-death structure.

Usage: `set_qbd(` <*G*>, <*E*>, <*B*> [, <*BACK*> [, <*FWD*>]] `)`

where <*G*> in the number of groups of states; <*E*> in the number of states (elements) per group[8]; <*B*> is the observation batch size; <*BACK*> is the number of reachable back groups; and <*FWD*> is the number of reachable forward groups.

11.5.4.14 `Fix Full Structure`

Adds the value `epsilon`, which is defined in section 11.5.4.17, to every element of the hidden state transition matrix, `A`, the symbol emission probability matrices, `p,q` and `r`, and the initial hidden state probability vector, `pi`, and normalizes them to satisfy the stochastic constrains. This method can be used as an easy way of changing any parameter whose value might have been set to zero (by a parameter estimation, for example) and setting it to a really small value.

Usage: `fix_full(`<*N*>, <*B*>`)`

where <*N*> is the number of hidden states; and <*B*> is the observation batch size.

[8] $N = G \cdot E$

11.5.4.15 `Fix Coxian Structure`

Assuming that the hidden states form a coxian structure, adds the value `epsilon`, which is defined in section 11.5.4.17, to every element of the hidden state transition matrix, `A`, the symbol emission probability matrices, `p,q` and `r`, and the initial hidden state probability vector, `pi`, and normalizes them to satisfy the stochastic constrains.

Usage: `fix_coxian(`<*N*>, <*B*>`)`

where <*N*> is the number of hidden states; and <*B*> is the observation batch size.

11.5.4.16 `Fix Quasi Birth-Death Structure`

Assuming that the hidden states form a quasi birth-death structure, adds the value `epsilon`, which is defined in section 11.5.4.17, to every element of the hidden state transition matrix, `A`, the symbol emission probability matrices, `p,q` and `r`, and the initial hidden state probability vector, `pi`, and normalizes them to satisfy the stochastic constrains.

Usage: `fix_qbd(` <*G*>, <*E*>, <*B*> *[,* <*BACK*> *[,* <*FWD*> *]]* `)`

where <*G*> in the number of groups of states; <*E*> in the number of states (elements) per group[9]; <*B*> is the observation batch size; <*BACK*> is the number of reachable back groups; and <*FWD*> is the number of reachable forward groups.

11.5.4.17 `Set Error Tolerance Value - Epsilon`

Sets the objects error tolerance value, `epsilon`. Default value is 0.00001. **Usage**: `fix_qbd(` <*float_value*> `)`

where <*float_value*> is the desired error tolerance value.

11.5.4.18 `Import From Tangram-II`

Imports a Markov chain structure from a Tangram-II model. This method is further described in section 8.2.

Usage: `import_from_tangram(`<*obj_name*>, <*num_hidden_vars*>, <*system_variable_list*>`)`

where <*obj_name*> is the Tangram-II model's description file (the *.obj file*); <*num_hidden_vars*> is the number of system states which represent the hidden states; and <*system_variable_list*>

[9] $N = G \cdot E$

is a comma separated list with the Tangram-II's model variable names, in the format 'Object.StateVariable'. This list must have <*num_hidden_vars*> elements

Input and Output File Format

Every file read (written) by the `ghmm` plugin must be in the following format:

$< N >$
$< B >$

$< pi_{(N \times 1)} >$

$< A_{(N \times N)} >$

$< obs_{(N \times 3)} >$; where $obs[i][0] = r[i]$,
$obs[i][1] = p[i]$,
$obs[i][2] = q[i]$

where <*N*>, <*B*>, <*pi*>, <*A*> and <*r*>, <*p*>, <*q*> are the `ghmm` attributes specified previously.

Example:

```
2
5

1.0
0.0

0.3 0.7
0.5 0.5

0.8 0.1 0.5
0.9 0.6 1.0
```

Note: The `load` and `save` methods can, as previously described, load/save specific attribute descriptions. In this case, the file which is read/written by these methods should have the same format described above, but include only the specific attribute which is being loaded/saved, i.e, the other attributes should be omitted from the file.

11.5.5 Hierarchical General Hidden Markov Model Plugin - Fixed Batch

The Hierarchical General Hidden Markov Model - Fixed Batch (HMM-Batch) plugin is a generalized version of the GHMM one, and was implemented during the work of [46]. Inside each state, instead of a simple Gilbert model, the user can define a general Markov chain, which, as in the case of the GHMM, is responsible for the symbol emissions. Figure 11.4 illustrates the structure of such a hierarchical model. After each transition (S_i, S_j) in

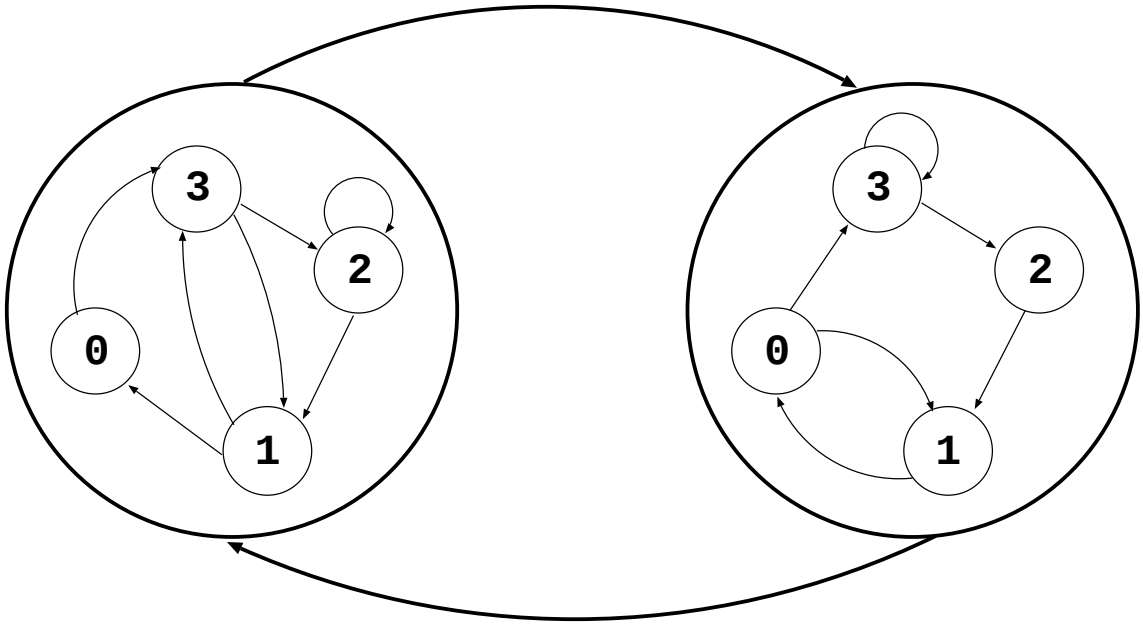


Figure 11.4: Example of a hierarchical general HMM with 2 hidden states and 4 observation symbols.

the the hidden chain (the upper-level one), the lower-level model associated with S_j emits symbols by transiting for a fixed number of steps, called a **batch**, before the upper-level chain makes another transition[10]. Every time the lower-level Markov chain makes one transition, thus reaching one of its states, the model emits one symbol, which is the one associated with the lower-level state just reached. Hence, symbol k is emitted every time the lower-level state I_k is reached, where I_k, with $k = \{0, \ldots, M-1\}$, defines a lower-level state inside the hidden state S_j. Each lower-level state emits one, and only one, symbol, and different lower-level states, inside a same hidden state, cannot both emit the same symbol.

[10]The initial state in the lower-level chain is chosen according to a initial state probability distribution associated with this lower-level Markov chain. Keep in mind that after **every** transition in the upper-level chain, which includes transitions to the same state, a new initial lower-level state is chosen, and the symbol emission process statrs from that state.

Constructors

The `hmm_batch` plugin has two different constructors:

`hmm_batch( )`
`hmm_batch(` <*N*>, <*M*>, <*B*> `)`

where the first one creates an empty hierarchical general hidden Markov model (it assumes its parameters will be loaded hereafter); and the second one creates a HMM-Batch with <*N*> hidden states, <*M*> observation symbols, whose observation batch size is <*B*>, and initializes its parameters with random values, satisfying the stochastic constrains. Each of the N states is associated with a unique integer number, ranging from 0 to $N-1$, which identifies the state. The observations symbols, emitted by the model, range from 0 to $M-1$, and each lower-level state, inside a hidden state S_j, is associated with one, and only one, symbol.

Attributes

1. `N`: Number of hidden states.
2. `M`: Number of observation symbols.
3. `B`: Observation batch size.
4. `pi[i]`: Initial probability of the i-th hidden state.
5. `A[i][j]`: Transition probability from hidden state i to hidden state j.
6. `p[i][j][k]`: Transition probability of symbol j to symbol k in hidden state i.
7. `r[i][j]`: Initial probability of symbol j in hidden state i.

Displays

1. `all`: Prints all model parameters.
2. `pi`: Prints only the initial hidden state distribution.
3. `A`: Prints only the hidden state transition matrix.
4. `obs`: Prints, for every hidden state, the values of the attributes p and r, defined above.

Methods

The methods of the HMM-Batch plugin are identical to those of the GHMM one, with the exception of the `symb_tavg` and `symb_sum_dist` methods, which are not implemented in this plugin. For this reason, we will not describe them here; the user who wishes to obtain more information on any of its methods is advised to use MTK's `help` command, or read section 11.5.4 of this manual.

Input and Output File Format

Every file read (written) by the `hmm_batch` plugin must be in the following format:

$< N >$
$< M >$
$< B >$

$< pi_{(N \times 1)} >$

$< A_{(N \times N)} >$

$< obs_{(N \times (M+(M \cdot M)-1))} >$; where $obs[i][j] = r[i][j]$, for $0 \leq j < M$;
$obs[i][M] = p[i][0][0]$;
$obs[i][M+1] = p[i][0][1]$;
$\ldots$
$obs[i][M+(M-1)] = p[i][0][M-1]$;
$obs[i][M+M] = p[i][1][0]$;
$\ldots$
$obs[i][M+(M \cdot M)-1] = p[i][M-1][M-1]$;

where $<N>$, $<M>$, $<B>$, $<pi>$, $<A>$ and $<r>$, $<p>$ are the `hmm_batch` attributes specified previously.

Example:

```
2
3
5

1.0
0.0
```

```
0.3 0.7
0.5 0.5

# r[i][j]   ## p[i][0][k]  ## p[i][1][k]  ## p[i][2][k]
0.1 0.1 0.8    0.2 0.2 0.6    0.9 0.1 0.0    0.4 0.4 0.2
0.5 0.0 0.5    0.8 0.0 0.2    0.3 0.3 0.4    0.7 0.2 0.1
```

Note: The `load` and `save` methods can, as previously described, load/save specific attribute descriptions. In this case, the file which is read/written by these methods should have the same format described above, but include only the specific attribute which is being loaded/saved, i.e, the other attributes should be omitted from the file.

11.5.6 Hierarchical General Hidden Markov Model Plugin - Variable Batch

The Hierarchical General Hidden Markov Model - Variable Batch (HMM-VarBatch) plugin is a generalized version of the HMM-Batch one, and was implemented during the work of [46]. Inside each hidden state, the user defines a general Markov chain with an **absorbing state**, which is responsible for the symbol emissions. Figure 11.5 illustrates the structure of such a hierarchical model. After each transition (S_i, S_j) in the the hidden chain

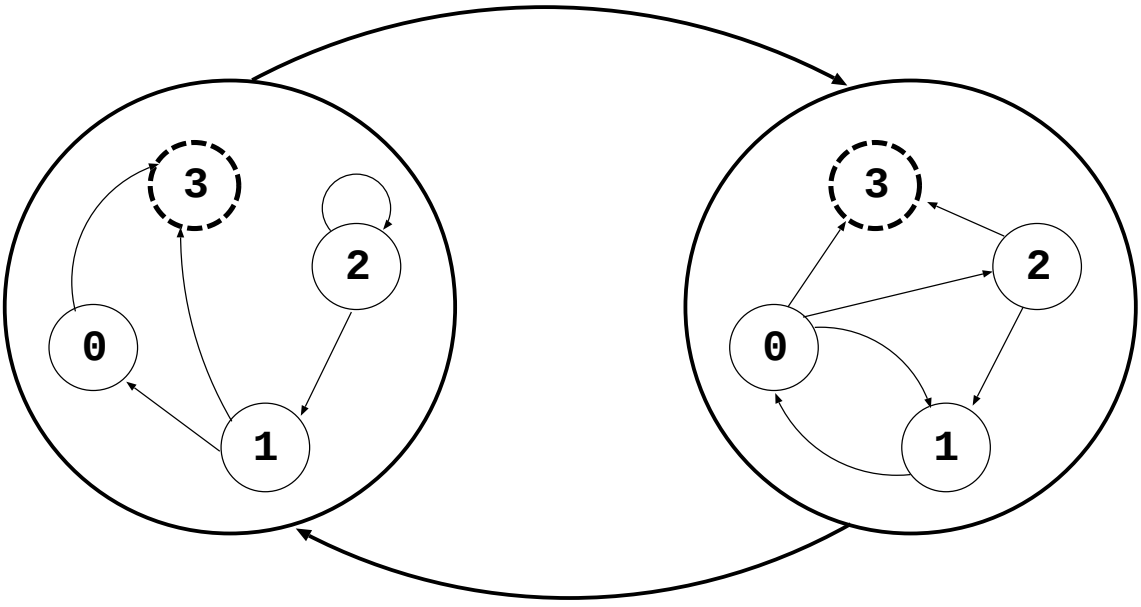


Figure 11.5: Example of a hierarchical general HMM with variable batch size, 2 hidden states and 4 observation symbols. Notice the absorbing state, which is identified by the dashed dark circle.

(the upper-level one), the lower-level model associated with S_j emits symbols by transiting through its states, **until it reaches the absorbing state**, which, then, causes the upper-level chain to make another transition[11]. Every time the lower-level Markov chain makes one transition, thus reaching one of its states, the model emits one symbol, which is the one associated with the lower-level state just reached. Hence, symbol k is emitted every time the lower-level state I_k is reached, where I_k, with $k = \{0, \ldots, M-1\}$, defines a lower-level state inside the hidden state S_j. Each lower-level state emits one, and only one, symbol, and different lower-level states, inside a same hidden state, cannot both emit the same symbol.

Important: It is crucial to keep in mind that the absorbing state also emits one symbol, which we call the **end of batch symbol**. This symbol is part of the observation symbols, just like every other, and, thus, must be part of the observations collected. All trace files used with this plugin **must** end with this symbol, since it represents the end of a batch of observations within some hidden state.

Constructors

The `hmm_batch_variable` plugin has two different constructors:

`hmm_batch_variable( )`
`hmm_batch_variable(` <*N*>, <*M*> `)`

where the first one creates an empty hierarchical general hidden Markov model (it assumes its parameters will be loaded hereafter); and the second creates a HMM-VarBatch with <*N*> hidden states and <*M*> observation symbols, and initializes its parameters with random values, satisfying the stochastic constrains. MTK **always** assumes that the last symbol, i.e., the $(M-1)$-th symbol, is the end-of-batch symbol. Don't forget this! Each of the N states is associated with a unique integer number, ranging from 0 to $N-1$, which identifies the state. The observations symbols, emitted by the model, range from 0 to $M-1$, and each lower-level state is associated with one, and only one, symbol. As said previously, the symbol $M-1$ will **always** be the end-of-batch symbol.

Attributes

1. `N`: Number of hidden states.
2. `M`: Number of observation symbols.

[11] The initial state in the lower-level chain is chosen according to a initial state probability distribution associated with this lower-level Markov chain. Keep in mind that after **every** transition in the upper-level chain, which includes transitions to the same state, a new initial lower-level state is chosen, and the symbol emission process statrs from that state.

3. `pi[i]`: Initial probability of the i-th hidden state.
4. `A[i][j]`: Transition probability from hidden state i to hidden state j.
5. `p[i][j][k]`: Transition probability of symbol j to symbol k in hidden state i.
6. `r[i][j]`: Initial probability of symbol j in hidden state i.

Displays

1. `all`: Prints all model parameters.
2. `pi`: Prints only the initial hidden state distribution.
3. `A`: Prints only the hidden state transition matrix.
4. `obs`: Prints, for every hidden state, the values of the attributes p and r, defined above.

Methods

The methods of the HMM-VarBatch plugin are identical to those of the HMM-Batch one. For this reason, we will not describe them here; the user who wishes to obtain more information on any of its methods is advised to use MTK's `help` command, or read the HMM-Batch section of this manual.

Input and Output File Format

The `hmm_batch_variable` plugin file format is identical to the `hmm_batch` format, excluding only, obviously, the batch size specification:

Example:

```
2
3

1.0
0.0

0.3 0.7
0.5 0.5

# r[i][j]   ## p[i][0][k]  ## p[i][1][k]  ## p[i][2][k]
0.1 0.1 0.8    0.2 0.2 0.6    0.9 0.1 0.0    0.0 0.0 0.0
0.5 0.0 0.5    0.8 0.1 0.1    0.3 0.3 0.4    0.0 0.0 0.0
```

Note: The `load` and `save` methods can, as previously described, load/save specific attribute descriptions. In this case, the file which is read/written by these methods should have the same format described above, but include only the specific attribute which is being loaded/saved, i.e, the other attributes should be omitted from the file.

11.6 Creating Your Own Plugin

The MTK distribution comes with an `example` plugin[12], which implements a sum of two integer parameters. You can start creating your own new MTK plugin from this template class.

All plugins are created using a single template class, called `ObjectBase`. In order to implement a new plugin, take the following steps:

i) Copy the `example` directory (and all its files) to a new one, whose name will be identical to that you will choose for your new plugin.

ii) Replace the `Example` class name inside the implementation and its header files with the name of your new plugin;

iii) Define the attributes of your new plugin;

iv) Write plugin constructors, set and get methods, displays and methods;

v) Write plugin help messages, defined in the header file;

vi) Use `addConstructor`, `addOption`, `addDisplay`, and `addCommand` methods, inherited from `ObjectBase`, to register plugin information into the MTK framework.

And you're done! It is as simple as that!

11.7 Integration with TANGRAM-II

Up to now, we have shown how the MTK tool can be used as a standalone application. However, recently, MTK was incorporated into TANGRAM-II. As a result of this incorporation, it is now possible to use MTK's plugins in a Tangram-II simulation. In this section, we describe, in details, how MTK can be used inside TANGRAM-II.

11.7.1 Using MTK in a Tangram-II Simulation

When creating a model in TANGRAM-II, it is imperative to define the events that characterize the system that is being modeled, and the actions that take place whenever these

[12] Which can be found in the *$MTK_HOME/plugins* directory.

events occurs. With MTK's integration with TANGRAM-II, it is now possible to, inside these events, work with MTK's plugin methods, as if they where just another action taking place inside the event. Therefore, when simulating a system, users can now, for example, collect and save traces obtained during the simulation, or use custom models (implemented as MTK plugins) to analyze the system during its simulation in TANGRAM-II.

TANGRAM-II views MTK as a **black box**, from which it can interact only through its plugin's attributes and methods. To achieve this, a new Tangram-II type, called `MTKObject`, was created. Each MTK object, that will be used in a Tangram-II model, will be viewed by TANGRAM-II as a variable of this type, and must be declared in `Declaration` section, just like any other variable in TANGRAM-II. Once created, the manipulation of these objects can be done through six primitives, listed below, in any point of the action code of a message or event. Through these primitives, users can access the object's attributes and execute their methods.

Each of these primitives is translated into the `user_code.c` file as a set of commands, which perform the communication between the Tangram-II simulator and the `libmtk` library, in order to execute the requested MTK actions during the simulation. Next, we describe, individually, each of the six primitives.

TangramII-MTK primitives

The following primitives are used to manipulate the MTK objects. Only through them, can the users access the object's attributes and execute their methods.

11.7.1.1 mtk_create: Creates Objects

This primitive creates a MTK object. By declaring a `MTKObject` variable, all you are doing is saying to TANGRAM-II that this variable will represent a MTK object; but nothing is being said about the object's class (if it is a `hmm` object, an `intvalue` object, etc.). To actually create the object, you must use the `mtk_create` primitive.

Usage: `mtk_create(`*<mtk_object_name>*`, "`*<plugin_name>*`"`*[, <args_list>]*`)`

where *<mtk_object_name>* is the name given to the `MTKObject` variable, by the user, in the `Declaration` section; "*<plugin_name>*" (must be between quote marks!) is the plugin name from which the object will be created, i.e., the plugin that this object correspond to; and *[, <args_list>]* is the set of parameters taken by the newly created object.

Example: Creating a HMM object

```
Declaration=
Const
   ...
```

```
Var
    MTKObject : Predictor;
    ...

Events=
event = event( TIME )
condition = ( TRUE )
action =
{
    ...
    mtk_create( Predictor, "hmm", NUM_STATES, NUM_SYMBOLS );
    ...
};
```

11.7.1.2 mtk_run: Executes Objects Methods

This primitive allows the user to execute a method from a `MTKObject`.

Usage: `mtk_run(`*<mtk_object_name>*`, "` *<method_name>*`"` *[, <args_list>]* `)`

where *<mtk_object_name>* is the name given to the `MTKObject` variable, by the user, in the `Declaration` section; "*<method_name>*" (must be between quote marks!) is the name of the method to be executed; and *[, <args_list>]* is the set of parameters taken by the method.

Example: Training the HMM

```
Events=
event = event( TIME )
condition = ( TRUE )
action =
{
    ...
    mtk_run( Predictor, "training", TRAINING_ITERATIONS, TRACE_OBSERVATIONS );
    ...
};
```

11.7.1.3 mtk_get: Gets Object Attribute Value

Gets the value of an attribute from a `MTKObject`, and stores it in a Tangram-II variable. With it, users can acquire and store, in a Tangram-II variable, any parameter value (attribute) or output of a `MTKObject`.

Usage: `mtk_get(`<*tangram2_dest_variable*>, <*mtk_object_name*>,
<*object_attribute*> [,<*index_list*>])

where <*tangram2_dest_variable*> specifies the Tangram-II's destination variable, in which the attribute value will be stored; <*mtk_object_name*> specifies the object's name, from which we will get the attribute; <*object_attribute*> specifies the object's attribute whose value we are interested in; and [,<*index_list*>] specifies the position of the attribute (if the attribute happens to be a vector, or matrix) in which we are interested in.

Example: Acquiring the new PI values, after a HMM training

```
Events=
event = event( TIME )
condition = ( TRUE )
action =
{
    float param_PI[2];
    ...
    /* Training the HMM */
    mtk_run( Predictor, "training", TRAINING_ITERATIONS, TRACE_OBSERVATIONS );

    /* Acquiring new PI values */
    mtk_get( param_PI[0], Predictor, "pi", 0 );
    mtk_get( param_PI[1], Predictor, "pi", 1 );
    ...
};
```

11.7.1.4 mtk_set: Sets Object Attribute Value

It is, essentially, the opposite of the `mtk_get` primitive. It allows the user to change the value of a `MTKObject's` attribute, setting it to a specific value.

Usage: `mtk_set(`<*tangram2_source_variable*>, <*mtk_object_name*>,
<*object_attribute*> [,<*index_list*>])

where <*tangram2_source_variable*> specifies the Tangram-II variable that holds the new value; <*mtk_object_name*> is the `MTKObject` whose attribute will be changed; <*object_attribute*> the attribute's name; and [,<*index_list*>] specifies the position of the attribute (if the attribute happens to be a vector, or matrix) whose value will change.

Example: Changing the HMM's initial state probability vector (PI).

```
Events=
event = event( TIME )
condition = ( TRUE )
action =
{
    float new_PI[2];
    ...
    new_PI[0] = 0.5;
    new_PI[1] = 0.5;
    ...
    /* Changing PI values */
    mtk_set( new_PI[0], Predictor, "pi", 0 );
    mtk_set( new_PI[1], Predictor, "pi", 1 );
    ...
};
```

11.7.1.5 mtk_copy: Copies' Objects

Copies' one `MTKObject` to another.

Usage: `mtk_copy(`<*dest_mtk_object*>, <*src_mtk_object*>`)`

where <*dest_mtk_object*> is the destination object, which will hold the new copy; and <*src_mtk_object*> is the source object, the one which will be copied.

Example: Copying the HMM object ***Predictor****.*

```
Declaration=
Const
   ...

Var
    MTKObject : Predictor_Copy;
    ...
Events=
event = event( TIME )
condition = ( TRUE )
action =
{
    ...
    mtk_copy( Predictor_Copy, Predictor );
    ...
```

```
};
```

11.7.1.6 mtk_delete: Deletes Created Objects

Deletes any created `MTKObject`.

Usage: `mtk_delete(`<*mtk_object_name*>`)`

where <*mtk_object_name*> is the name of the `MTKObject` variable to be deleted.

Example: Deleting the HMM object

```
Events=
event = event( TIME )
condition = ( TRUE )
action =
{
    ...
    mtk_delete( Predictor );
    ...
};
```

11.7.2 Initializing MTK Parameters

Motivated by the need to initialize the MTK objects before the start of the simulation, a special event parameter, called `INIT`, was created. Any event whose parameter is set to `INIT` is scheduled just one time, when the simulation time is equal to zero. This allows the user to execute an action, or a group of actions, before the simulation actually starts.

The `INIT` keyword is used in the same way as any event distribution name keyword is:

Example:

```
Events=
event = event_name( INIT )
condition = ( TRUE )
action =
{
    ...
    /* Actions */
    ...
};
```

The only difference is that each Tangram-II object can have **only one** event of this type (with this parameter).

Special care should be taken with the messages sent during these events, since no assumptions can be made on the order of their execution and, consequently, no order of message delivery is guaranteed. Finally, even if a user defines an event of this kind, he is still obeyed to initialize the model variables and constants in `Initialization` section, but nothing hinders the user from overwriting these values inside the event code.

As stated above, an event of this type is useful to bootstrap a MTK object before simulation starts, reading a trace or setting its parameters. It is also helpful to initialize a long Tangram-II vector, eventually with a special logic, and it allows users to initialize model variables with mathematical expressions, possibly based on other model parameters, which cannot be done in `Initialization` section.

11.7.3 Compilation Directives

11.7.3.1 `#ifdef`

The Tangram-II modeler can restrict the parsing of a given portion of code using the compilation directives `SIMULATION` and `CHAIN_GENERATION`. Using that, the user can create an hybrid model, where some of its parts are executed according to the compilation context.

To define a portion of code which will be parsed and executed only when the model is simulated, the modeler should write:

```
(...)
#ifdef SIMULATION
/* Simulation code here */
#endif
(...)
```

The usage of the directive `CHAIN_GENERATION` is analog. Note that any portion of code can be involved by the `ifdef` environment, from a simple statement to a set of events.

11.7.3.2 `#include`

This directive allows a Tangram-II user to include a piece of code written into another file, instead of the TGIF interface. This feature can improve code reuse, since the same file would be included from distinct portions of code. Moreover, this allows the users to modify its model parameters without having to use the TGIF graphical interface, which can be useful in case of remote usage.

11.7.4 TGIF Multi-Page Model

A Tangram-II model can be defined by placing their objects into a multi-page TGIF file. This is specially useful to define complex models with many objects. Note that a user

cannot create more than one object with the same name, even if they are disposed into different pages.

For this feature to work, it is necessary to have TGIF tool version 4.1.45.4 or later.

Chapter 12

FreeMeeting

FreeMeeting is a real-time multimedia communication tool, that allows users to transmit audio and video over the Internet. It started as a voip tool, called VivaVoz, which was implemented in 1999. Later, in 2005, VivaVoz was integrated with the Comitvideo tool, which transmitted video over the Internet. As a result of this integration, Freemeeting was created.

For further information, please refer to FreeMeeting's website at http://www.land.ufrj.br/tools/tools.html.

Appendix A

Output File Formats

A.1 Introduction

The main purpose of this appendix is describe all files that are generated by the TANGRAM-II tool.

A.2 Model Environment Module

The following files are generated by the Model Environment Module:

1. $\langle name\_of\_the\_model \rangle.obj$ - This file is generated by the TGIF tool. This is a ASCII text file, that has a graphic representation of the model and is composed by the TGIF functions.

2. $\langle name\_of\_the\_model \rangle.parser$ - This file is a C program text. All objects in the model are described, with its attributes, events and so on. This file is the output for the grammar.c program.

3. $\langle name\_of\_the\_model \rangle.events$ - This file lists all events in the model. The format is:

```
name of object.name of event
```

4. $\langle name\_of\_the\_model \rangle.states$ - The entire state space is described here. The format is:

   ```
   state number   all state variables of the model
   ```

   The state variables are listed in the same order as in the $< name\_of\_the\_model >$. $< vstat >$ file.

5. ⟨*name_of_the_model*⟩.*vstat* - This file has all state variables. The format is:

```
name of object.state_variable
```

6. ⟨*name_of_the_model*⟩.*state_variable*

7. ⟨*name_of_the_model*⟩.*maxvalues* - This file has the state variables with their respectives maximum values (these values are relationed with the specified model). The format is:

```
name of object.state   maximum value
```

8. ⟨*name_of_the_model*⟩.*user.code.c* - This file is generated automatically. It contains the user code for actions, messages and expressions.

9. ⟨*name_of_the_model*⟩.*parameter* - This file is always generated. But it is fill in only if the model has the literal parameter. There is a association between a parameter and a letter.The format is:

```
letter  name of object.parameter
```

10. ⟨*name_of_the_model*⟩.*generator_mtx* - This file represents the Q generator matrix of the model. The format is:

```
previous state   actual state   transition probability
```

11. ⟨*name_of_the_model*⟩.*generator_mtx_expr* - This file is always generated. But it is fill in only if the model has the literal parameter.The format is:

```
total number of the parameters
letter name of object.parameter (association)
number letter (association)
```

12. ⟨*name_of_the_model*⟩.*generator_mtx_param* - This file represents the Q generator matrix of the model, but intead of the absolute value, the transition probabilities are the parameters specified in the model.

13. ⟨*name_of_the_model*⟩.*NM.st_trans_prob_mtx* - This file represents the transition probabilities matrix and it is generated only for Non Markovian Models. The format is:

```
previous state   actual state   probability
```

14. ⟨*name_of_the_model*⟩.*tables_dump*

15. ⟨*name_of_the_model*⟩.*rate_reward*.⟨*name_of_object.name_of_the_reward*⟩ - This file is generated by the Mathematical Model Module and has the rate reward in each state, when the condition of the reward is true. The format is:

    ```
    state  value of reward
    ```

16. ⟨*name_of_the_model*⟩.*impulse_reward*.⟨*name_of_object.name_of_the_reward*⟩ - This file is generated by the Mathematical Model Module and has the impulse reward.

17. ⟨*name_of_the_model*⟩.*rate_reward.GlobalReward*.⟨.*name_of_the_reward*⟩ - This file is generated by the Mathematical Model Module and has the global rate reward.

18. ⟨*name_of_the_model*⟩.*rate_reward.expr* - This file has the expression of the reward.

19. ⟨*name_of_the_model*⟩.*reward_levels*.⟨*name_of_the_reward*⟩

20. ⟨*name_of_the_model*⟩.*absorb_st* - This file has all absorbing states in the model.

21. ⟨*name_of_the_model*⟩.*config* - This file is generated by the TANGRAM-II interface (menu → config). These settings are used to set some parameters in TANGRAM-II tool.

22. ⟨*name_of_the_model*⟩.*OUT*.⟨*name_of_the_output_file*⟩

23. ⟨*name_of_the_model*⟩.*uniform_rate* - This file has the uniformization rate of the Continuos Markov Chain. The format is:

    ```
    uniformization_rate
    ```

24. The files below are generated only for non-Markovian models:

    (a) ⟨*name_of_the_model*⟩.*NM.st_trans_prob_mtx*: This file is generated only for Non Markovian Models. It includes the transition probabilities of the uniformized Markov chain that is obtained when all the events in the model are assumed to have exponential rates. The format is:

    ```
    previous state  actual state  probability
    ```

    (b) ⟨*name_of_the_model*⟩.*NM.chns_betw_embed_pnts*: This file contains, for each deterministic event, the Markov chains that are obtained between embedded points (details in the solution technique used). The following information is included:

i. The uniformization rate (taken from the exponential matrix considering all events exponential).

ii. event; id of the deterministic event; number of independent chains; boolean variable (1): there is an absorbing state; (0) otherwise; inverse of the deterministic rate.

iii. -1; id of the state variable; id of the independent chain; -1 0 0: delimiter

iv. Definition of each independent chain for the event: chain; id of the independent chain; number of states; uniformization rate for the independent chain; state state probability (the state id is already properly renumbered); -1 0 0

(c) $\langle name\_of\_the\_model \rangle.NM.embedded\_points$: This file contains all the transitions (from state, to state) that are associated with the firing of a deterministic event, i.e., all the embedded points considering only the execution of a deterministic event.

(d) $\langle name\_of\_the\_model \rangle.NM.embedded\_points\_expr$

(e) $\langle name\_of\_the\_model \rangle.NM.states\_det\_ev$: This file contains all the states in which a deterministic event is enabled.

(f) $\langle name\_of\_the\_model \rangle.NM.embedded\_chain\_mapping$: This is the mapping from the state id of the Markov chain considering all events with exponential rates to the state id of the resulting embedded chain.

(g) $\langle name\_of\_the\_model \rangle.NM.embedded\_chain$

(h) $\langle name\_of\_the\_model \rangle.NM.emb\_points\_st\_probs$: The steady-state probabilities for the embedded chain.

(i) $\langle name\_of\_the\_model \rangle.NM.expected\_cycle\_length$: The expected length of the intervals between embedded points.

(j) $\langle name\_of\_the\_model \rangle.NM.interest\_measures$: Indicates which state variables are considered for the calculation of the marginal probabilities. Note: although not included in the interface, joint probabilities can be calculated as well using this file.

(k) $\langle name\_of\_the\_model \rangle.NM.marginal\_probs$: The resulting marginal probabilities obtained for the non-Markovian models. The state variables used for calculating the marginal probabilities are specified by the user in the interface.

25. $\langle name\_of\_the\_model \rangle.partition$ - This file lists all partitions used in the GTH block method.

26. $\langle name\_of\_the\_model \rangle.SS.gth$ - This file is generated by the Analytical Solution Module - Stationary State GTH Solution Method, and has the vector probabilities in steady state. The format is:

```
state   probability
```

27. ⟨*name_of_the_model*⟩.*SS.gthb* - This file is generated by the Analytical Solution Module - Stationary State GTH block version Solution Method, and has the vector probabilities in steady state. The format is

```
state  probability
```

28. ⟨*name_of_the_model*⟩.*SS.jacobi* - This file is generated by the Analytical Solution Module - Stationary State Jacobi Solution Method, and has the vector probabilities in steady state. The format is:

```
number of iterations
state  probability
```

29. ⟨*name_of_the_model*⟩.*SS.gauss* - This file is generated by the Analytical Solution Module - Stationary State Gauss-Seidel Solution Method, and has the vector probabilities in steady state. The format is:

```
number of iterations
state  probability
```

30. ⟨*name_of_the_model*⟩.*SS.power* - This file is generated by the Analytical Solution Module - Stationary State Power Solution Method, and has the vector probabilities in steady state. The format is:

```
number of iterations
state  probability
```

31. ⟨*name_of_the_model*⟩.*SS.sor* - This file is generated by the Analytical Solution Module - Stationary State Succesive Over Relaxation Solution Method - and has the vector probabilities in steady state. The format is:

```
number of iterations
state  probability
```

32. ⟨*name_of_the_model*⟩.*TS.pp.*⟨*TIME*⟩

33. ⟨*name_of_the_model*⟩.*TS.brew.cumulat_distrib*

34. ⟨*name_of_the_model*⟩.*TS.brew.expected_period*

35. ⟨*name_of_the_model*⟩.*TS.exptr*

36. ⟨*name_of_the_model*⟩.*TS.operational_time*

37. ⟨*name_of_the_model*⟩.*SIMUL.*⟨*name_of_the_simulation_result*⟩ - This file is generated by the Simulation Solution Module - Batch Simulation and Rare Simulation - and has all informations about the rewards specified in the model (Rewards part).

38. ⟨*name_of_the_model*⟩.*threshold* - This file is generated by the Simulation Solution Module - Rare Simulation. The format is:

```
<name of object>.<state variable used in rare simulation>
threshold  splits
```

39. ⟨*name_of_the_model*⟩.*INTSIMUL.*⟨*name_of_the_simulation_result*⟩

40. ⟨*name_of_the_model*⟩.*IM.*⟨*name_of_measure_of_interest*⟩ - This file is generated by the Measures of Interest Module. The format depends of the kind of the measure calculated:

    (a) PMF of one or more state variables
    Without Conditional:

    ```
    name of the measure
    expected value of the measure
    probability mass function (PMF) of the state variable
    choosen
    ```

    With Conditional:

    ```
    name of the measure
    expected value of the measure
    probability of the condition is true in the model
    list of all states when the Conditional is true
    ```

    (b) Function of state variables
    Without Conditional:

    ```
    name of the measure
    function
    expected value of the function
    0.0 - probability of the function is false in the model
    1.0 - probability of the function is true in the model
    ```

    With Conditional:

    ```
    name of the measure
    function
    ```

```
condition
probability of the condition is true in the model
condition expected value of the function
0.0 - probability of the conditional function is false in
the  model
1.0 - probability of the condtional function is  true  in
the model
```

(c) Probability of a set
Without Conditional:

```
name of the measure
set description
set probability
1 - set probability
```

With Conditional

```
name of the measure
set description
conditional
conditional set probability
```

The following files are generated by the Traffic Modeling Module, in the Model Environment Module:

1. Markovian Models:

   (a) ⟨*name_of_the_model*⟩.*intervals* - This file is generated by the interface. The format is:

   ```
   number of intervals
   initial observation time   final observation time   number
   of points
   ```

   (b) ⟨*name_of_the_model*⟩.*init_prob* - This file is generated by the interface and has the initial probability. The format is:

   ```
   initial state   probability
   ```

   (c) ⟨*name_of_the_model*⟩.*idc* - This file has the idc measure. The format is:

   ```
   observation time  IDC  mean E[N(t)]  variance Var[N(t)]
   second moment E[N2(t)]
   ```

   (d) ⟨*name_of_the_model*⟩.*autocovariance* - This file has the autocovariance measure. The format is:

```
observation time    Cov[X(t),X(t+time)]
```

(e) ⟨*name_of_the_model*⟩.*autocorrelation* - This file has the autocorrelation measure. The format is:

```
observation time    Cor[X(t),X(t+time)]
```

(f) ⟨*name_of_the_model*⟩.*stationary_descriptors* - This file has the stationary descriptors measures: mean, second moment, variance, burtiness and peak value. The format is:

```
expected value
second moment
variance value
peak value
burstiness
```

2. **Traces:**

(a) ⟨*name_of_the_model*⟩.*seq_idc* - This file has the idc measure. The format is:

```
observation time  IDC  mean E[N(t)]  variance Var[N(t)]
 second moment E[N2(t)]
```

(b) ⟨*name_of_the_model*⟩.*seq_autocovariance* - This file has the autocovariance measure. The format is:

```
observation time    Cov[X(t),X(t+time)]
```

(c) ⟨*name_of_the_model*⟩.*seq_autocorrelation* - This file has the autocorrelation measure. The format is:

```
observation time    Cor[X(t),X(t+time)]
```

(d) ⟨*name_of_the_model*⟩.*stationary_descriptors* - This file has the stationary descriptors measures. The format is:

```
min rate
max rate
expected value   E[trace]
variance value   E[trace]
```

Appendix B

How to Create a New Object

B.1 Introduction

The main purpose of this appendix is describe how we can create a new object that can be used in other models.

B.2 Creating a New Object

To create a new object, the steps that must be followed are:

1. In the TANGRAM2_OBJECTS domain choose `obj_template.sym`;
2. Now create a new figure for the object;
3. Select the new figure and in the Edit Menu choose `cut` or `copy`;
4. Select the object-template;
5. In the Special Menu, choose "Replace Graphic" . The old graphic is replaced by the new figure in the cut buffer;
6. You can choose "yes" or "no" in the menu that opens. If you choose "no", a new symbol is created and stored using the file name just given.

To store a new object in the library, the steps that must be followed are:

1. In the Special Menu, choose "Make Symbolic" .
2. We can store the new object in the TANGRAM2_OBJECTS domain , or we can create a new domain. To create a new Domain , we must create a new directory (named `Mydomain`, for example) and in the `.Xdefaults` file we must specify the path of this new domain.

```
Tgif*MaxDomains:            total number of domains
Tgif*DomainPath(#of domain):MYDOMAIN:/home/nameofuser/Mydomain
```

3. Choose the new domain (Special Menu/Change Domain);
4. In the Special Menu, choose "Save Sym in Library" ;
5. Give the new name to the object and then select the directory.

This object can now be used in other models.

NOTE: You can create a new object symbol and store it in the domain of your choice.

B.3 Creating a New Model

1. In the Modeling Environment press "File" and then "New";
2. Specify a new filename and press "create";
3. In the TGIF interface, choose "Special" and them "Domain". Afterwards choose "Change Domain";
4. Choose the `TANGRAM2_OBJECTS` domain ;
5. Instantiate `obj_template.sym` or any other object symbol available in the domain.

Appendix C

How to Connect Ports

C.1 Introduction

All ports in our model can be set automatically. To do this, we must use some features of the TGIF tool. The main purpose of this appendix is to describe the necessary steps to perform this task.

C.2 Connecting two ports

To connect two ports in the model, the steps that must be followed are:

1. Choose an object (as shown in 2);

2. Choose "port.sym" in the TANGRAM2_OBJECTS domain;

3. In the "port.sym", instantiate the `name` attribute. This name must be the same of the type Port of the object (in the declaration attribute) where the port will be included. If the object has n ports, you must instantiate n "port.sym";
   Obs: Port (in the declaration attribute): port_A;
   name (attribute of port.sym): port_A;

4. Now, you must attach all ports to the object. Select all ports and the corresponding object. In the Special menu, choose "Ports and Signals". Then choose "Merge Ports with an Object" ;

5. Next, you must transform this object into a symbol. In the Special menu, choose "Make Symbolic" . If you want, you can save this new symbol in a library. Now you can instantiate this object port-enable whenever you want;

6. Repeat all steps to the other object;

7. Now you must connect two ports. In the Special Menu, choose **Ports and Signals**. Then, choose **Connect two Ports by Wire** . When the mouse pointer is located over a port, the feature it is highlighted. Click the left mouse button and connect the wire to another port, and then choose a name of the connection. If you look at the `Initialization` attribute of each object, you will observe that the chosen port has its name set to the connection name.

NOTE: Note that you can create new icons, with ports, for an object and replace then for an old icon using the steps in B-2.
NOTE: If when you move any object connected by a wire to another object, and you want the connecting wire to follow the movement, click the "constrained move" icon in the TGIF pallete.

To move the name of a wire

1. Selecting the wire;
2. Type <ALT> + <M>;
3. Click the mouse left button and select **signal_name**;

To change the name of a wire connection:

1. The corresponding objects must be symbols;
2. Select **special - ports and signals - clear signal name for a port** and then click in all ports connected by the wire;
3. Make a new wire connection.

NOTE: To save the model, you must **UnMake symbolic** all objects;

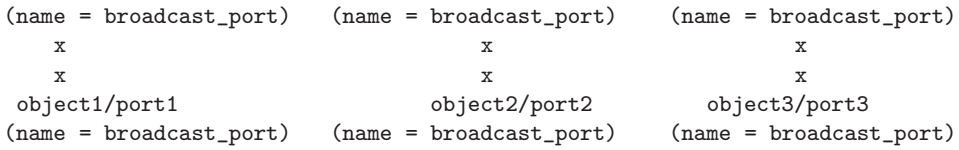
C.3 Connecting more than two ports by a broadcast link

In some cases, is necessary to connect various ports to the same broadcast link. This is possible using a special TGIF features. To connect more than two ports to the same broadcast link, the steps that must be followed are:

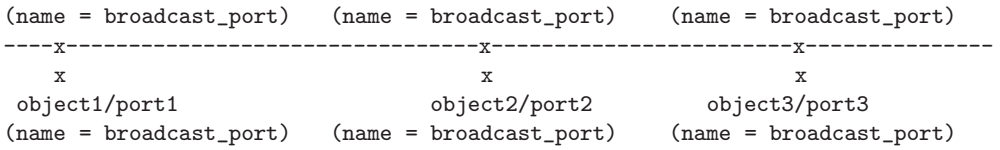
1. First, you must have the objects with their own ports instantiated (see C-2, steps 1 to 6).

```
          x                          x                          x
   object1/port1              object2/port2              object3/port3
(name = broadcast_port)   (name = broadcast_port)   (name = broadcast_port)
```

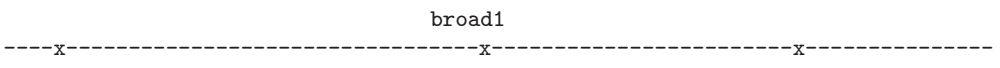
2. Now, create and as instantiate one port for each object. The port must have the same name the ones in the objects:



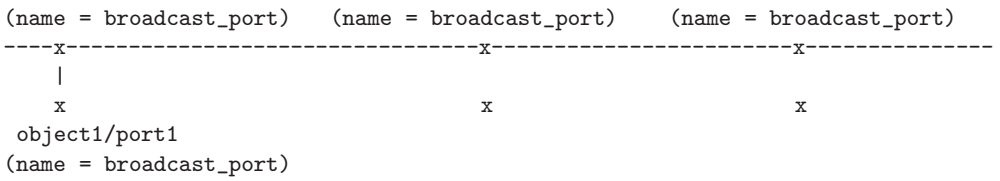
3. Draw a line (use the draw tool of TGIF) connecting the ports:



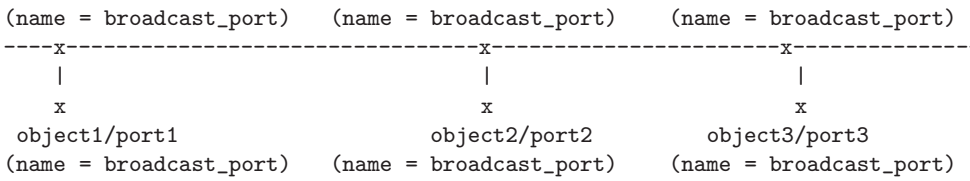
4. Select the line and ports then choose **Connect Ports to a Broadcast Wire** from the **Ports and Signals** submenu in the Special menu. Fill the dialog box eiht a name for the broadcast connection, e.g., broad1.



5. Choose **Connect Two Ports by a Wire** from the Ports and Signals submenu in the Special menu, then connect the first port from the first object to the broad1 line. Fill the dialog box with broad1 or other that you want.



6. To connect the other ports use **Repeat Connect Two Ports by a Wire**



Appendix D

The Syntax Used in The Models

D.1 Introduction

The main purpose of this appendix is describe briefly the syntax used in the models specified in TANGRAM-II tool. The syntax is more detailed in the examples section.

D.2 Syntax

D.2.1 Attributes

```
object=
    <Declaration attribute>
    <Initialization attribute>
    <Events attribute>
    <Messages attribute>
    <Rewards attribute>

<Declaration attribute>=
     Declaration=
     <Declaration definition>

     <Declaration definition>=
         <Var definition>
         <Const definition>
         <Param definition>

         <Var definition>=
              State          : <identifier list> ;
              Float          : <identifier list> ;
```

```
              Integer        : <identifier list> ;
              FloatQueue     : <identifier list> ;
              IntegerQueue   : <identifier list> ;


         <Const definition>=
              Integer        : <identifier list> ;
              | Float        : <identifier list> ;
              | Object       : <identifier list> ;
              | Port         : <identifier list> ;

         <Param definition>=
              Integer        : <identifier list> ;
              | Float        : <identifier list> ;

<Initialization attribute>=
    <state identifier>        = <integer number>
    |<integer identifie>      = <integer number>
    |<float identifier>       = <float number>
    |<object identifier       = <object name>
    |<port identifier>        = <port name>


<Events attribute>=
    Events=
    <Event description>

    <Event description>=
    <event definition>
    <event condition definition>
      <action definition>

    <event definition>=
        event = <event name>(<distribution type>, <expression>)
            <distribution type> = exp | det
        <event condition definition>=
            condition = (<boolean expression>)
        <action definition>=
            action   = <action code>;
            | action = <action prob list>;
```

```
            <action code>=

                  <variable declaration attribute>
                  <statement sequence>

            <action prob list>=
                <action code>;  <prob definition>
                <prob definition>=
                    prob = <expression>;

<Message attribute>=
    Messages=
    <Message description>

    <Message description>=
        <message definition>
        <action definition>

        <message definition>=
            msg_rec = <port identifier>

<Rewards attribute>=
    Rewards=
    <reward definition>

    <reward definition>=
        <reward rate header> <reward cond value list>
        | <reward impulse header> <reward event value list>

        <reward rate header>=
            rate_reward = <reward identifier>

        <reward impulse header>=
            impulse_reward = <reward identifier>

        <reward cond value list>=
            condition = ( <boolean expression> )
            value     = <expression> ;

        <reward event value list>=
            event     = <event name>, <triggers>
```

```
            value     = <expression> ;
```

D.2.2 C statements

The following C statements can be used :

```
<if statement>=
    if ( <boolean expression> ) <statement>

<if else statement>=
    if ( <boolean expression> ) <statement> else <statement>

<while statement>=
    while ( <boolean expression> ) <statement>

<switch statement>=
    switch (<identifier>) <case sequence>

   <case sequence> = <case sequence> <case statement> | <case statement>
   <case statement> = case <integer>: <statement sequence> break;
       | case <integer>: <statement sequence>
       | default: <statement sequence>

<for statement>=
   for (<assignment sequence>; <boolean_expression>; <for assignment statement>)
      <statement>

      <for assignment statement>=
         <assignment identifier> = <expression>
      <assignment sequence>=
         <assignment sequence>, <for assignment statement>
         | <for assignment statement>
```

D.2.3 Other statements

```
<msg statement>=
   msg ( <message port identifier>, <object identifier>, <expression> )

<objcmp statement>=
   objcmp (<object identifier>, <object identifier>)
```

D.2.4 Functions

```
<power function>=
   pow (<first expression>, <second expression>)

<sqrt function>=
   sqrt (<expression>)

<get function>=
   get_st (<auxiliary var>, <state var>) |
   get_st_float (<auxiliary var>, <state var>)

<set function>=
   set_st (<state var>, <auxiliary var>) |
   set_st_float (<state var>, <auxiliary var>)
```

D.2.5 Some reserved words

msg_source The object identifier that sent the received message in the port.

msg_data The last parameter in the `msg` statement. Represents the information that can be sent in the message. It is an integer value in this version.

Bibliography

[1] Werner Almesberger. URL ftp://lrcftp.epfl.ch/pub/linux/atm/. ATM on Linux release 0.4.

[2] Werner Almesberger. URL http://lrcwww.epfl.ch/linux-atm/. ATM support for Linux.

[3] Jan Beran, Robert Sherman, Murad S. Taqqu, and Walter Willinger. Long-Range Dependence in Variable-Bit-Rate Video Traffic. *IEEE Transactions on Communications*, 43(2/3/4):1566–1579, 1995.

[4] S. Berson, E. de Souza e Silva, and R.R. Muntz. An object oriented methodology for the specification of Markov models. In *Numerical Solution of Markov Chains*, pages 11–36. Marcel Dekker, Inc., 1991.

[5] Bilmes, Jeff A. A Gentle Tutorial on the EM Algorithm and its Application to Parameter Estimation for Gaussian Mixture and Hidden Markov Models. Technical report, University of Berkeley, 1997.

[6] R.M.L.R. Carmo, L.R. de Carvalho, E. de Souza e Silva, M.C. Diniz, and R.R. Muntz. Performance/Availability Modeling with the TANGRAM-II Modeling Environment. *Performance Evaluation*, 33:45–65, 1998.

[7] W. Chia-Whei Cheng. *The TANGRAM graphical interface facility (TGIF) manual*. TGIF WWW at http://bourbon.cs.ucla.edu:8801/tgif/.

[8] Cover, Thomas M. and Thomas, Joy A. *Elements of information theory*. John Wiley and Sons, Inc., 1991.

[9] C.E.F. de Brito, R.S. de Moraes, D.J. Oliveira, and E. de Souza e Silva. Comunicação Multicast Confiável na Implementação de uma Ferramenta Whiteboard. In $17^{\underline{th}}$ *Simpósio Brasileiro de Redes de Computa dores*, pages 222–237, Maio 1999.

[10] C.E.F. de Brito, E. de Souza e Silva, and W. Cheng. Reliable multicast communication and the implementation of TGWB, a shared vector-based whiteboard tool. Technical report, UFRJ, 2000.

[11] C.E.F. de Britto, E. de Souza e Silva, M.C. Diniz, and R.M.M. Leão. Análise Transiente de Modelos de Fonte Multimídia. In $18^{\underline{th}}$ *Simpósio Brasileiro de Redes de Computadores*, pages 519–534, May 2000.

[12] E. de Souza e Silva and H. Richard Gail. Calculating Cumulative Operational Time Distributions of Repairable Computer Systems. *IEEE Transactions on Computers*, c-35(4):322–332, 1986.

[13] E. de Souza e Silva and H. Richard Gail. The Uniformization Method in Performability Analysis. Technical report, IBM Research Division, Thomas J. Watson Research Center - Yorktown Heights, NY 10598, U.S.A., 2 1996.

[14] E. de Souza e Silva and H.R. Gail. Calculating availability and performability measures of repairable computer systems using randomization. *Journal of the ACM*, 36(1):171–193, 1989.

[15] E. de Souza e Silva and H.R. Gail. Analyzing scheduled maintenance policies for repairable computer systems. *IEEE Trans. on Computers*, 39(11):1309–1324, 1990.

[16] E. de Souza e Silva and H.R. Gail. Performability analysis of computer systems: from model specification to solution. *Performance Evaluation*, 14:157–196, 1992.

[17] E. de Souza e Silva and H.R. Gail. An algorithm to calculate transient distributions of cumulative rate and impulse based reward. *Stochastic Models*, 14(3):509–536, 1998.

[18] E. de Souza e Silva and H.R. Gail. Transient Solutions for Markov Chains. In W. Grassmann, editor, *Computational Probability*, pages 44–79. Kluwer, 2000.

[19] E. de Souza e Silva, H.R. Gail, and R.R. Muntz. Efficient solutions for a class of non-Markovian models. In *Computations with Markov Chains*, pages 483–506. Kluwer Academic Publishers, 1995.

[20] E. de Souza e Silva and R.M.M. Leão. The Tangram-II Environment. In *Computer Peformance Evaluation - Modelling Techniques and Tools - $11^{\underline{th}}$ International Conference (TOOLS2000)*, volume 1786, pages 366–369. Springer, Março 2000.

[21] E. de Souza e Silva, R.M.M. Leão, and M.C. Diniz. Transient analysis applied to traffic modeling. In *Workshop on Mathematical performance Modeling and Analysis (MAMA) 2000*. June 2000.

[22] E. de Souza e Silva, R.M.M. Leão, and R. Marie. An efficient approximation technique for calculating transient reward measures. In *Proceedings of The Fourth International Workshop on Performability Modeling of Computer and Communication Systems (PMCCS4)*, pages 16–19, Williamsburg,USA, September 1998.

[23] E. de Souza e Silva and R.R. Muntz. *Métodos Computacionais de Solução de Cadeias de Markov: Aplicações a Sistemas de Computação e Comunicação.* VIII Escola de Computação, Brasil, 1992.

[24] Duarte, Flávio P. and de Souza e Silva, Edmundo A. and Towsley, Don. An adaptive FEC algorithm using hidden Markov chains. *SIGMETRICS Perform. Eval. Rev.*, 31(2):11–13, 2003.

[25] R. M.M.Leão E. de Souza e Silva and R. Marie. Efficient solutions for an approximation technique for the tr ansient analysis of markovian models. Technical report, Institut National de Recherche en Informatique et en Automat ique, Centre de Diffusion, INRIA, 1996.

[26] Elliott, E. O. A model of the switched telephone network for data communications. *Bell Systems Technical Journal*, 44:89–109, January 1965.

[27] D.R. Figueiredo and E. de Souza e Silva. Efficient Mechanisms for Recovering Voice Packets in the Internet. In *Proceedings of IEEE/Globecom'99, Global Internet: Application and Technology Symposium*, pages 1830–1837, Dezembro 1999.

[28] Gilbert, E. N. Capacity of a burst-noise channel. *Bell Systems Technical Journal*, 39:1253–1265, September 1960.

[29] W.K. Grassmann and D.P. Heyman. Equilibrium disrtibution of block-structured markov chains with repeating rows. *J. App. Prob.*, 27:557–576, 1990.

[30] W.K. Grassmann, M.I. Taksar, and D.P. Heyman. Regenerative analysis and steady state distributions for Markov chains. *Operations Research*, 33(5):1107–1116, 85.

[31] R. H. A. Guérin and M. Naghshineh. Equivalent Capacity and Its Application to Bandwidth Allocation in High-Speed Networks. *IEEE Journal on selected areas in communications*, 9(7):968–981, 1991.

[32] A. Jensen. Markoff chains as an aid in the study of Markoff processes. *Skandinavsk Aktuarietidskrift*, 36:87–91, 1953.

[33] J.F. Kurose and K.W. Ross. *Computer Networking. A Top Down Approach Featuring the Internet.* Addison Wesley, 2001.

[34] R.M.M. Leão, E. de Souza e Silva, and Sidney C. de Lucena. A Set of Tools for Traffic Modeling, Analysis and Experimentation. In *Computer Peformance Evaluation - Modelling Techniques and Tools - $11^{\underline{th}}$ International Conference (TOOLS2000)*, volume 1786, pages 40–55. Springer, Março 2000.

[35] Reliable Multicast Library. URL http://www.land.ufrj.br/tools/rmcast. Reliable Multicast Library site.

[36] D.D. Loung and J. Biro. Needed Services for Network Performance Evaluation. In *IFIP Workshop on Performance Modeling and Evaluation of ATM Networks*, Inglaterra, Julho 2000.

[37] M. Martinello and E. de Souza e Silva. A Testbed tool for Network Performance Evaluation and its Application to Connection Admission Control Algorithms. In $18^{\underline{th}}$ *Simpósio Brasileiro de Redes de Computadores*, pages 30–46, May 2001.

[38] Elwalid D. Mitra and Robert H. A New Approach for Allocating Buffers and Bandwidth to Heterogeneous, Regulated Traffic in an ATM Node. *IEEE Journal on Selected Areas in Communications*, 13(6):1115–1127, 1995.

[39] Rabiner, Lawrence R. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, Feb 1989.

[40] A.A.A. Rocha. Medições ativas na Internet: algoritmos baseados em retardo fim-a-fim e experimentos. Master's thesis, UFRJ-COPPE/Sistemas, Agosto 2003.

[41] A.A.A. Rocha, R.M.M. Leão, and E. de Souza e Silva. A Methodology to Estimate One-way Delay and Internet Experiments (in portuguese). In *XXII Brazilian Computer Network Symposium(SBRC'04)*, Gramado, Brazil, May 2004.

[42] A.A.A. Rocha, R.M.M. Leão, and E. de Souza e Silva. A New Technique to Select Packet Pairs to Estimate Bottleneck Link Capacity (in portuguese). In *III WPerformance/XXIV SBC*, Salvador, Brazil, August 2004.

[43] S. M. Ross. Approximation transition probabilities and mean occupation times in continuos-time markov chains. *Probability in the Engineering and Informational Sciences*, 1987.

[44] Silveira Filho, Fernando and de Souza e Silva, Edmundo A. Modeling the short-term dynamics of packet losses. In *Performance Evaluation Review*, volume 34, pages 27–29, December 2006.

[45] W.J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[46] Vielmond, Carolina C. L. B. de and Leão, Rosa M. M. and de Souza e Silva, Edmundo A. Um modelo HMM hierárquico para usuários interativos acessando um servidor multimídia. In *Simpósio Brasileiro de Redes de Computadores, 2007*.

[47] L. Zhang, Z. Liu, and C.H. Xia. Clock Synchronization Algorithms for Network Measurements. In *IEEE/Infocom*, pages 160–169, New York, USA, Junho 2002.