NEW DIMENSIONS IN VISUAL MODELING: EXPRESSIONS, SUBNETWORKS, AND AN OBJECT-ORIENTED ENVIRONMENT

AT&T Bell Laboratories, Holmdel, NJ 07733, USA

In this paper we describe our experiences with and our approach to enhanced graphical simulation modeling involving teletraffic applications. Specifically, our approach to simulation is one of developing ready-to-use tools, as well as tool environments for constructing customized tools that can be easily programmed by the teletraffic user.

1. INTRODUCTION

Visual modeling of teletraffic systems received considerable interest when it was described and demonstrated by our colleagues at the 12th ITC in 1988 [1]. The visual modeling paradigm allows the user to both represent and study the behavior of a system graphically. Our approach to such tools is two pronged. We are adding new features and abstractions to Queueing+Analysis Software, or Q+, described below, so as to get immediate user feedback as to how these features really help the modeler. We describe the addition of expressions and subnetworks to Q+, and resulting capabilities, especially involving teletraffic applications.

The second prong in our approach involves a rethinking of both implementation issues and user interfaces to visual modeling. To do this we are creating a new environment called Eva, for building tools like and beyond Q+. We also describe this environment and illustrate its power.

2. Q+

The visual modeling tool described in [1] is called Q+ (formerly the Performance Analysis Workstation, or PAW). Q+ is now in widespread use within AT&T and at over 50 other sites around the world. The basic idea of Q+ is that the user inputs a model just as he/she would describe it to a colleague: by drawing a picture. The user then observes the behavior of the system via the animated movement of traffic entities within the network and the gradual evolution of statistics. Background on Q+ can be found in [2], teletraffic applications in [1], and educational and training experiences in [3]. During 1989 two new dimensions have been added to Q+: expressions and subnetworks.

2.1 Expressions

Previously, Q+ provided Monte Carlo simulation capabilities and interfaces to various analytical tools. The purely graphical modeling paradigm is powerful and allows many complex situations to be easily represented. However, there are cases where it is more natural to express a system's operation with program fragments, e.g. variables or functions. We wanted a way to enhance Q+ to allow this capability, but without burdening the user with all the usual responsibilities and pitfalls of textual modeling. In late 1989 we produced a trial version of Q+ that allows, among other things, users to access many of the Q+ attributes by evaluating an expression. These expressions can consist of a simple variable, a program fragment, or a whole program. The expression capability is symbolic, and indeed textual, but complements the purely visual components of Q+ in a natural way. For example, since performance models are quantitative, it might make sense for a transaction (packet, call, etc.) to carry with it a variable \( x \), which might represent length, time, history, system state, etc., and for the network to alter its treatment of this transaction based on that variable. This attribute might instead belong to the node and might even be a function that the node executes to determine how to route or service the transaction. The overall effect is a natural complement to the graphical paradigm that can easily express complex operations or protocols.

The most important aspect of the design of these new features is that they can be thought of as an applique over the current, simple Q+ paradigm. Thus, the basic Q+ entities of transactions and nodes, and the basic operation of creating/servicing/routing transactions remains unchanged. But now, where the user previously entered a constant or distribution, or chose from a menu, one can enter an expression for evaluation at runtime. This expression can be simply the evaluation of a variable or can be an entire computation that examines the state of the network. Just like Q+, these expressions are completely interpretive. A model can be stopped, an expression entered or changed, and the model rerun from its current state without any type of recompilation. The expression language is called Exp. Exp is a simple language that resembles C syntactically, but is easier to use than C in the Q+ environment, since it provides dynamic (automatic) typing, and full support for the list abstractions of Q+. However, if the user prefers, C functions can be written and simply called from Exp.

For example, a packet's service time at a node could be obtained as a function of the length of the packet and the speed of the node. Thus the service time of the packet might be entered as

\[
*\text{tvar}("length")/*\text{nvar}("speed")\;
\]

The function \( \text{tvar()} \) returns a unique variable name based on the transaction and the passed suffix, in this case "length", and is dereferenced by the \( * \) operator. Similarly, \( \text{nvar()} \) returns a unique variable name based on the node and suffix "speed". The node may also carry out a protocol processing function that increases the length of the packet, in which case it would be entered as:

\[
{*\text{tvar}("length")}=\text{header};
*\text{tvar}("length")/*\text{nvar}("speed")\;
\]
While certain state dependent routing capabilities (e.g., join the shortest queue) are built into Q+, the user can easily define other state-dependent routing, service, etc. behavior using expressions. The example given below is sometimes referred to as join the biased queue. Suppose the user wants to route customers from node n1 to either node n2 or n3 based on a function of the queue length at those nodes. We can define the following function as the routing rule of node n1 (the only class in the model is c):

```plaintext
let nl_route() == {
  local q2, q3;
  q2 = n2.N_TOTAL;
  q3 = n3.N_TOTAL;
  if (a2*q2 < a3*q3) then
    list(n2,"c")
  else
    list(n3,"c");
};
```

To implement the join the shortest queue discipline, the user would simply set a2 and a3 equal and positive. Although Q+ provides this discipline, one can implement a more general parametric mechanism by altering these values.

The user can set the routing of node n1, class c in the Q+ editor, or in an external file with the function define_routing("n1","c",’nl_route()”). The quote ' delays the evaluation of nl_route until the actual routing occurs vs. when it is defined.

### 2.3 User-defined Statistics

Another aspect of modeling in which expressions arise naturally is statistics. In addition to the current repertoire of built-in Q+ statistics, there are now four new types of user-defined expression statistics. They support both "time" and "customer" average concepts, but unlike the built-in ones, they do not have to be associated with a particular node; they may be associated with multiple nodes (e.g. joint statistics) or perhaps no node at all.

For example, one can measure the distribution of the product of successive interdeparture times of a departure stream from a node with a simple expression statistic. The mean of this distribution is then the first lag serial covariance of the stream. An Exp function could be defined as:

```plaintext
let statf() == {
  if (undef(n)) n=0;
  if (!undef(t1)) t2=t1;
  if (!undef(t0)) t1=t0;
  t0 = get_panel_field(T_CURR);
  n++;
  if (n>2) (t1-t0)*(t2-t1);
};
```

To notify Q+ to collect this function, the user can use the graphical statistic editor or call the expression create_node_stat() in an external file.

In this example we use undef() to make this statistic self-initializing. Note that when statf() is called and n is less than or equal to 1, it returns nil and no statistic is collected. To display the statistic, we graphically peg any window to node J, as one would for any built-in Q+ statistic.

These expression statistics can be catalogued into libraries to facilitate procedures such as on-line experiment control and data analysis. These capabilities serve to further the basic Q+ principles of supporting interactive model building and exploratory analysis of its behavior.

### 2.4 Subnetworks

The other major new feature now provided with Q+ is subnetworks. The subnetwork capability allows and encourages the user to structure models hierarchically. Thus a model can possess submodels to any degree of nesting. This abstraction facilitates model construction via the principle of information-hiding that is well accepted in structured programming. It also supports basic operations of replicating model components, altering the graphical representation of parts of models, reading in models from libraries, etc. By structuring models both modularly and hierarchically, it encourages sharing and reuse of modeling efforts.

Q+ subnetworks allow the user to move and rescale subnetworks, and also cut and paste together Q+ models. The user can align time and disambiguate names when models are pasted together. He/she can also manipulate much larger models easily via an enhanced viewing mechanism.

### 2.5 Example of Expressions and Subnetworks

Figure 1 shows both new features in use. The model shown represents a distributed computing environment with clients, a LAN, and a server. The model shown contains several levels of subnetworks, which are either displayed (with animation and model internals visible for that subnetwork) or undisplayed. Also shown is the user currently entering an expression through the Q+ screen editor. This expression for routing evaluates a function, increments an attribute of the current transaction, and routes the transaction by returning its next node and class.

### 2.6 Dynamic Icons and Temporal Browsing

We are currently investigating several new features for Q+, including dynamic icons and temporal browsing. Currently Q+ provides standard icons for nodes, classes, transactions, and statistics that can not be changed. With dynamic user-defined icons, the user will be able to associate various icons with these entities, and even dynamically modify or change them during simulation. We are designing a Q+ icon palette for users to draw their own icons and manipulate existing ones. With this feature, it will be easy to distinguish nodes that have different functions and highlight important model characteristics.

Temporal browsing will also provide increased information on dynamic model characteristics. It allows events to be posted from outside the simulation. These events become scheduled just like traditional events do. The browser is asynchronous to Q+ and therefore allows the user to observe and change a model independently of the Q+ screen. With these abilities, many possible features could be implemented; we are investigating animation of expressions, source code debugging, and new editing methods to name a few.

### 3. OBJECT-ORIENTED ENVIRONMENT AND FUTURE DIRECTIONS

While Q+ has been delivering visual modeling and simulation capability to users over the last 7 years, object-oriented programming environments have also been progressing. To explore further the power of graphics and current object-oriented software techniques in visual modeling and simulation, we have been
concurrently developing an object-oriented software environment called **Eva**, for developing tools like Q+. The basic idea behind **Eva** is to provide a set of graphics and simulation objects and then support methods for their integration to achieve an arbitrary class of discrete-event simulations. Thus one can effectively customize a specific application by changing the behavior of simulation objects. We first give a brief overview of the architecture and design of **Eva**, and then describe an example of a prototype **EOS** (Environmental Observation System) visual simulation system, which includes an environment like Q+, but also mobile objects to allow a dynamically reconfigured communication network.

### 3.1 Overview of **Eva**

**Eva** is developed with the programming language C++ [4] on principles pioneered by **Simula** [5]. It provides two kinds of objects, **simulation objects** and **graphics objects**. A programmer can use the basic simulation objects to develop the specific simulation objects and algorithms required for a given problem, and the graphics objects to develop the graphical representation of the simulation objects. With **Eva**, the application-specific simulation and graphics objects can and should be developed independently. This decreases complexity and encourages reuse.

These two types of objects are combined together with the **multiple inheritance** feature of C++. Consider a typical object-oriented simulation application. Suppose that we have available types **Customer** and **Queue**. Customers enter the queue and wait until a server is ready to service them and at that point they exit from the queue. Suppose further that we have developed icon classes **Customer Icon** and **Queue Icon**. We want to represent the simulation objects with icons and the object interaction with graphical actions (e.g., animation) in order to make the simulation process easier to follow. With multiple inheritance we can define a class that combines the properties of both the simulation objects and icon objects in a single object. We can derive the class **Iconed Customer** from **Customer** and **Customer Icon** and the class **Iconed queue** from **Queue** and **Queue Icon**.

In the following we will discuss the design and usage of the simulation and graphics objects.

### 3.2 Simulation Objects

Every discrete event simulation process can be implemented with number of co-operating agents, which communicate with each other, receive and service customers, route customers, allocate resources, wait until resources become available, etc. In **Eva**, **Nodes** are objects designed to facilitate the implementation of these agents. Nodes have their own data and functions and they can communicate by sharing data objects or by exchanging messages. Nodes have also their own private stack so that they may execute their functions independently of each other in a quasi-parallel mode similar to the way processes execute in an operating system. In other words, each node has its own locus of control. Unlike operating system processes, nodes share the same address space (within one process) and their scheduling is nonpreemptive, which means that a node will continue to execute until it decides to relinquish control. One reason for relinquishing control occurs when a node waits for an event before it can proceed with any further processing. For example, suppose that a node receives a customer and puts it in service. Then the node has to wait for two events to occur: either the customer completes service or a new customer arrives. While the node waits for either event, other nodes get a chance to execute.

Note that a simulation submodel is simply a special node that consists of a set of nodes. Therefore we have a uniform way of treating hierarchies.

A class **wait-object** is provided to facilitate the programming of the conditions for which a node may wait. A wait-object can be in either of two states, **ready** or **pending**. A node can execute a **wait**(ob) call on a wait-object ob. If ob is ready, then control returns immediately, but if ob is pending then the node is blocked until ob becomes ready. A member function of a wait-object, called **awr()** for action-when-ready, is provided to implement the semantics of the wait-object and is called by the node when the wait-object is ready. Examples of wait-object include a communication line and a service position. The former connects two nodes, the source-node and the destination-node and is ready when a customer is put on it by the source-node. Its **awr()** function delivers the customer to the destination-node. On the other hand, a service-position upon receiving a customer starts a timer for the duration of the service requirements of that customer. When the timer expires its state becomes ready. Its **awr()** function routes the customer to its destination.

An auxiliary class **customer** is also provided in **Eva**, which may be used as a base class of all types of customers. This class is not fundamentally as important as the nodes and wait-objects but nevertheless it can add convenience to the development. Customers are objects which are moved between nodes. At each node they are processed based on their specific needs and the capabilities of the hosting node. Each customer has its own data and a number of virtual (user redefinable) functions which are called when significant events occur. Examples of these events are: entering a node, joining a queue, entering service, exiting a node etc. If these functions are appropriately used, a customer can become very active in determining its own service requirements, routing, statistics gathering, etc.

### 3.3 Icon Objects

The main purpose of an icon is to provide visual information of the state of simulation data objects. It is displayed in some shape on the screen which is implemented with bitmap operations in a graphics workstation. Since icons occupy a rectangular space on the screen we need a coordinate system to express rectangles. Two coordinate systems are in fact maintained in **Eva**: the absolute and relative coordinate systems. The absolute coordinate system defines the absolute position of a point on screen, while the relative coordinate system defines a virtual square with fixed length dimensions (1024 x 1024) where an icon resides. Therefore one can define and discuss the property of an icon without concern of the absolute position of a point in the icon. For instance, in relative coordinates the center of an icon is invariably the point (512,512). To achieve the desired visual effect, icons support transformations from relative to absolute coordinates and vice versa. These transformations are icon dependent. Thus, icons can introduce a variety of visual effects just by adjusting these transformations.

Icons are objects. Thus all icons support a basic set of operations, called their interface. These operations allow icons to display/undisplay, init/kill, reshape, move, change color or appearance etc. All these operations have default implementations which are members of the base class **Icon** in the terminology of object-oriented programming. New types of icons are derived from the base class **Icon**. Therefore, icons may overwrite some of the default implementations of their members or introduce new functionality. They may also become base classes for other derivations allowing hierarchies to form which are specialized to various applications.
Icons are organized via two relations child and peer. When an icon $I_1$ becomes a child of icon $I_2$ (or $I_1$ is inserted into $I_2$) then $I_1$ is included in the space reserved for $I_2$. In other words, $I_1$ can not overlap or be disjoint from $I_2$. Also if $I_1$ is moved, its movements will be constrained be the contour of $I_2$. This is extremely useful in implementing customer/queues and submodels. All icons are children or grandchildren of the icon Global. Global is the only icon that can produce actions visible on the screen. All other icons implement their visible actions via Global.

If an icon $I_1$ is a peer of icon $I_2$ then $I_1$ will be notified for any change occurring to $I_2$. Suppose that $I_1$ is a line connected to a block $I_2$. If $I_2$ moves to a new position then line $I_1$ will be notified to reshape itself accordingly so that it remains connected with $I_2$. This relation is useful in implementing the nodes/submodels that are related to each other in someways.

With the design of simulation objects and icon objects in Eva described above, it is fairly straightforward how one can develop a visual simulation tool with hierarchial modeling capability and programmable icons. In the next section we illustrate such an example arising from an Earth Observing System (EOS) application.

3.4 An Example Eva Application

To gain some experience with Eva, we have implemented a tool for the study of EOS. The application involves satellite based observation systems that consist of a number of satellites equipped with instruments that are capable of obtaining data from the earth. The data collected may be transmitted to ground stations (possibly via other relay satellites) and subsequently processed. To implement a visual simulation tool for analyzing the performance of such a system, a ground computer and communication network model was developed and animated in Eva objects which implements a subset of features found in Q+. In addition, new objects such as a satellite and some environmental objects (e.g., rain storms) are created using the simulation and icon objects described above. The satellite objects move along trajectories that are defined by analytic equations. Each satellite has a set of targets from which it may collect data. Algorithms are implemented in the satellite object that enable the receiving of data as certain criteria are met, e.g., distance and state of the instruments. The net result is that we have a visual simulation of a communication network which dynamically reconfigures itself as the simulation progresses. Figure 2 depicts a snapshot of the state of this simulator. Each satellite and ground station is in fact a submodel in the sense of a Q+ model. However the object-oriented approach allows the objects to be refined and to carry functions and data such as analytic equations and conditions without perturbing the visual simulation paradigm established in Q+.

4. CONCLUSION

Our approach to simulation has become one of developing ready-to-use tools, such as Q+, and tool environments for constructing customized tools, such as Eva. This approach is allowing us to maintain both immediate capabilities and unlimited flexibility to respond to a variety of simulation challenges. Further experiences with our approach will be reported at the conference.

REFERENCES

Routing rules list:

- like, discrete, min_total, max_total, min_block, max_block, min_busy, max_busy, min_idle, max_idle, \{, Expression

FIGURE: 1

Distributed Computing Environment Using Q+ Expressions and Subnetworks
FIGURE: 2

EOS Application In EVA Environment