

**GENESIS: A Graphics Environment
for the Modeling and Simulation
of MultiAccess Networks:
Modeling Language Considerations
and Simulator Design**

**Chia Shen
James F. Kurose**

**COINS Technical Report 85-33
November 1985**

Contents

1. Introduction	3
2. Background	6
2.1 Overview of Past Work	6
2.2 An Overview of Multiple Access Communication	7
3. GENESIS: Modeling Language Considerations, Channel Modeling Mechanisms, and Simulator Design	9
3.1 Modeling Language Considerations: Modeling Primitives for Protocol Operations . .	9
3.1.1 Jobs	10
3.1.2 Modeling Elements	10
3.1.3 Routing	12
3.2 The Channel Model	13
3.3 Simulator Design	16
3.4 Overall GENESIS Architecture and the Modeling Environment	20
4. Examples	23
4.1 non-persistent CSMA/CD	23
4.2 Token Passing Ring	27
5. Conclusion	30
A. Attributes of the nonpersistent CSMA model	31
B. Modeling Elements Specification	34
C. Data Structures supporting the Simulation Engine	37
D. Design Specifications of GENESIS Simulation Engine	45

Abstract

The development of computer simulation programs for modeling and evaluating computer communication networks represents a significant software development effort and can be very costly in terms of the effort required to build, verify, modify and maintain these programs. Our work described in this report is part of the on-going research to reduce this effort in the system development process. Here we discuss our current efforts on the design, development and implementation of GENESIS, an interactive, high-level, graphics-oriented environment for constructing and evaluating simulation performance models of multiple access computer communication networks and protocols. We will focus on modeling language considerations, channel modeling mechanisms and the simulator design aspects of GENESIS in this report.

1. Introduction

Evaluating the performance of computer communication networks is a critical, yet complex, step in the design, planning and evaluation process for such systems. Traditionally, there have been two approaches to modeling these systems: "analytic" models (often numerically solved) [14] and discrete event simulation models (solved by simulating the model in software) [25].

Analytic models must be mathematically tractable (in that at least a numerical approximation must be possible) and as a result, either only small parts of a complex system may be modeled or few details of the entire system can be considered. Simulation models are not subject to such constraints and thus can more accurately reflect the actual system being modeled. For this reason, the performance evaluation of almost all "real-world" systems is accomplished through simulation. However, the development of computer simulation programs represents a *significant software development effort* and can be very costly in terms of the effort required to build, verify, modify and maintain these programs. Moreover, considerable expertise in the area of statistical analysis may be required to correctly interpret the simulation results. As a result of these problems, the performance evaluation of the various design alternatives in a system's design space can be a severe bottleneck in the system development process. [2].

The goal of the on-going research effort described in this report is the development of an interactive, high-level, graphics-oriented environment, known as GENESIS, for constructing and evaluating simulation performance models of a class of computer communication networks known as multiple access networks [32] [19]. This class of networks includes such diverse types of networks as single and multi-hop packet radio, satellite, and local area networks. We additionally note that the tools and methods developed for modeling this class of networks will superset those required for wide-area networks as well. Indeed, the work reported in here finds much of its inspiration in the related work reported in [26] [27] [28]. GENESIS is currently being developed on a Digital Equipment Corporation MicroVax II workstation (VMS) and is written in C.

In the GENESIS modeling environment, the performance analyst constructs a simulation model simply by composing various high-level modeling primitives. The primitives themselves are provided and defined by the environment and are designed to directly reflect the objects (network components and actions) in the modeling domain. This high level of abstraction and close mapping between the modeling primitives and the network being modeled greatly facilitates the modeling process. The modeling environment also provides graphical support for interactive model construction, revision and maintenance. At the heart of GENESIS is a discrete-event-based "simulation engine" which performs the actual simulation of the performance model and is structured in a manner similar to

that described in [25].

We stress that our goal in building such an environment GENESIS (as opposed to a system such as PLANS [24] for example) is not simply to build a simulator for a specific protocol or set of protocols. Rather, our more general aim is to provide the modeler/analyst with a rich set of modeling primitives (and an environment in which these primitives can be easily manipulated) for creating and simulating a performance model of *any* arbitrary, modeler-defined protocol. Thus, the analyst would be able to rapidly construct and evaluate protocols ranging from contention-based access protocols [19] to token passing schemes [30] to higher level network protocols and algorithms such as routing [29] and flow control [15].

We believe that our work is of considerable interest and importance for several reasons. First, multiple access networks represent a broad class of computer communication networks including packet radio, satellite and local area network systems. They are thus of considerable practical and theoretical interest to the academic, industrial and military communities and have been the subject of intense research and development activities [32] [19] [30]. As discussed earlier, these efforts must generally rely on simulation when performance analysis is required. We believe that modeling environment such a GENESIS will greatly enhance the efficiency and productivity of the modeler by providing for rapid model development, debugging and maintenance through the use of graphical programming and animation techniques, and by providing structuring techniques which encourage the development of well-structured, logically correct simulation models.

In a broader sense, our efforts represent an important effort in the areas of programming environments and modeling and performance evaluation. Only recently have results from software engineering and program development environments begun to be applied in the area of simulation software development [2] [7] [22]. There are, additionally, many important differences between the specific problem of developing modeling software for multiple access communication networks and the general software development problem addressed by software engineers. For example, most high level network modeling languages [28], including the language to be developed within this research, are *declarative* rather than procedural in nature. We also believe that software development in a restricted problem domain has provided the opportunity for the development of modeling constructs, structuring techniques and modeler aids designed specifically for this subclass of software development problems. Indeed, a primary purpose of this report is a presentation of some of these problem-specific capabilities.

The contents of this report are as following. In section 2, previous work in the design of languages for the modeling and simulation of computer communication networks is surveyed and an brief overview of multiple access communication is given. Then in section 3, modeling language

considerations, channel modeling mechanisms and simulator design of GENESIS are presented. Specifically, we discuss the set of modeling elements, which constitute the primitives of the modeling language for protocol operations provided by GENESIS, we have considered so far in section 3.1; the mechanisms for modeling the channel in section 3.2 and the design of the simulator in section 3.3. Section 3.4 gives a brief overview of the system architecture of GENESIS. To illustrate the language facilities provided by GENESIS and the ease with which simulation models of multiaccess protocols can be constructed and evaluated in GENESIS, examples of model construction of two multiaccess protocols, non-persistent CSMA/CD and token ring, are given in section 4. Finally in section 5, we consider some open questions in the future development of GENESIS and summarize this report.

2. Background

2.1 Overview of Past Work

Software simulation has long been the primary tool for analyzing the performance of computer communication networks. Four distinct "generations" of simulation programming languages can be readily identified.

General purpose applications programming languages such as FORTRAN, PL/I, PASCAL, etc., can be considered as "zeroth" generation simulation languages. Since these languages provide no programming constructs which directly support simulation programming, we do not consider them to be true simulation languages at all. Rather, they are simply general purpose languages in which a simulation may be programmed. The modeler/analyst using a zeroth generation modeling language is faced with the myriad, low-level details of programming the simulation "from scratch", and must thus contend with problems such as random number generation, event list maintenance, statistics gathering, etc..

The "first generation" modeling languages such as GPSS [11], SIMULA [5], and SIMSCRIPT [3] were developed to overcome such difficulties and thus free the modeler from the necessity of contending with low-level, implementation-dependent simulation details. These languages are based on queuing model approaches towards simulation [13] in which systems resources are modeled as queues and customers (jobs) move from queue to queue. These languages thus provide the modeler with an important level of abstraction: rather than specifying the simulation model in terms of data structures, (e.g., event lists) which are to be manipulated as the simulation proceeds, the analyst specifies the simulation in terms of jobs, queues and the interconnection of queues.

These first generation languages, however, were designed as *general purpose* simulation languages and were not meant to model any specific class of systems. The second generation network modeling languages, such as RESQ [28], [26] [27], PAWS [4] and STEP-1 [33], were designed specifically for modeling and analyzing computer systems and *wide-area* (long-haul) computer communication networks. These languages provide the analyst with a rich set of high-level modeling elements (constructs) which can be combined in a *declarative language* to create, simulate and evaluate high-level models of computer networks. That is, the language constructs directly reflect the objects (network components and actions) in the modeling domain and the analyst simply *declares* the characteristics of these elements as opposed to programming the detailed simulation algorithms. The high-level language elements are then translated ("compiled") into a low-level event-based simulation.

At this level of abstraction, the modeling elements can be easily mapped onto the network components. As a result, simulation models can be easily and rapidly constructed, modified, verified and evaluated and the modeler's effectiveness and productivity increases dramatically. A performance analyst can now produce useful results *in days* for problems which would have otherwise required *weeks* of effort [28]. This increased productivity, in turn, permits the analyst to more thoroughly explore the design space and answer questions that would have otherwise been left unanswered.

Recently, a third generation of modeling languages for computer communication networks has begun to emerge. The development of these third generation languages is based on the observation that, even when presented with high-level modeling constructs and a sophisticated declarative language and editor for manipulating these constructs, users of second generation languages will typically first develop a pictorial or graphical specification of a model and only then translate this specification into a second generation language. Indeed, there has always been a strong pictorially-oriented flavor in even the description of the second generation languages themselves [28], [4].

The focus of the emerging third generation of modeling languages is thus the integration of high-level constructs within a graphics-oriented *modeling environment* in which the analyst may easily manipulate these constructs and build, modify, evaluate and debug high-level models of computer communication networks. These third generation modeling environments will undoubtedly result in yet further increases in modeler productivity.

Current work on the third generation modeling languages has just begun. One effort recently reported in the literature is [2]. In this work, the authors discuss extending the second generation language PAWS [4] into a graphics-oriented modeling environment. Although various aspects of the graphical user interface were discussed, many issues remain to be resolved as the system is still currently under development. A similar system, known as GIST [7], has also been recently developed. This environment, however, provides only a subset of the RESQ or PAWS modeling constructs and provides no facilities for the construction of hierarchically structured performance models. Finally, we note that several efforts have also recently been reported on developing graphical interfaces and support environments for first generation (general purpose) modeling languages [10] [35] [34] [22]. The work reported in [10] [22] is of particular interest due to the capability of animating the simulation within the graphical environment.

2.2 An Overview of Multiple Access Communication

During the past ten years, the use of a shared multiple access communication channel has found widespread acceptance as an economical means of interconnecting a set of distributed computing

resources. The characteristics of these multiple access channels differ greatly from those of traditional point-to-point communication channels. In particular, the geographically distributed nature of the stations and the fact that the channel can (to a first approximation) only carry a single successful transmission at one time requires that stations coordinate their message transmission by following some prescribed *channel access protocol*. Numerous research efforts have been directed towards developing such *multiple access protocols*; a survey of this work can be found in [32] [19]. Local area networks such as Ethernet [23], the ALOHA satellite network [1] and the PRnet packet radio network of the Defense Advanced Research Projects Agency [12] are all examples of networks employing multiaccess protocols.

In the simplest model of a multiple access network, stationary stations communicate over an error-free broadcast communications channel. If a single station transmits a message (according to some specified access protocol), this message is received correctly by all the network stations; when two or more stations transmit messages simultaneously, the messages *collide* and no message is received correctly at any network station. A rigorous mathematical analysis of the performance of multiple access protocols in even such an idealized network environment has proven to be extremely difficult (see, for example, [31]).

In "real-world" multiple access networks, problems such as channel noise, signal fading, and capture may occur. The stations may also be mobile, as in packet radio networks. Furthermore, the spatial distribution of network stations may result in the formation of *multi-hop* multiple access networks in which numerous multiple access channels are interconnected (partially overlap) to form a wide-area multiple access network; in this case issues such as routing, addressing and flow control must also be considered. These problems considerably complicate the problem of designing efficient and robust protocols and make a mathematical analysis of protocol performance essentially intractable. In this case, the network analyst must rely on simulation to evaluate the performance of the network protocols.

3. GENESIS: Modeling Language Considerations, Channel Modeling Mechanisms, and Simulator Design

The widespread acceptance of multiple access communication networks, in which a set of distributed computing resources interconnects via a shared multiple access communication channel, has spawned the need to model and evaluate the performance of such systems in order to explore the design space. As discussed in the previous section, current existing second and third generation network modeling languages were designed for wide-area computer communication networks and therefore provide no direct support for the modeling, simulation and analysis of multiple access networks. The characteristics that are unique to multi-hop multiple access communication networks, such as channel contention and station mobility, require the development of new modeling and simulation techniques and methodology that supersedes those for wide-area networks. Thus, the focus of our work on GENESIS has been on the issues involving the specification, design and development of a high-level, graphics-oriented environment for the modeling and simulation of multiple access networks.

Three of our current major efforts have been on modeling language considerations, channel modeling mechanisms and the design of the simulation engine. In the following subsections, we describe these aspects of GENESIS in detail.

3.1 Modeling Language Considerations: Modeling Primitives for Protocol Operations

A critical step in the design of a third generation modeling environment for multiple access networks is the selection of a set of high-level modeling elements which are of sufficient semantic richness to specify simulation models of arbitrary (i.e., user-defined) multiple access networks and communication protocols. (We again stress that this contrasts sharply with the standard approach of using a zeroth or first generation modeling language to model the performance of a *specific* multiple access protocol, as in [24], for example.)

At the highest level, any simulation model consists of a number of *jobs* which flow between *modeling elements* and/or the *multiaccess communication channel*, according to specified *routing rules*. In fact, the role of the GENESIS simulation engine is simply to perform this movement of jobs. In the following subsections we thus examine each of these key elements of GENESIS (*jobs, modeling elements, and routing*) in more detail and the *multiaccess communication channel* will be examined in section 3.2.

3.1.1 Jobs

In practice, jobs divide broadly into two classes. (Although GENESIS makes no such distinction among jobs, we draw this distinction for pedagogical purposes, since we have found such differences are almost always implicitly defined by an analyst's use of jobs). One class of jobs, which we refer to as *data-oriented jobs*, correspond directly to objects (messages, acknowledgments, polling requests, token messages, programs, etc.) in the modeling domain. The second class of jobs, known as *control-oriented jobs*, cause some action to be taken with respect to the data-oriented jobs and can be thought of as modeling the *flow of execution* or action taken by a network protocol. For example, a data-oriented job may represent a message which will be transmitted only after a control-oriented job sets the value of a boolean flag in the simulation to TRUE.

Either type of job may also have a number of associated *job variables*. Values may be assigned to a job's job variables when it passes through a *set node*. Each set node has one or more associated assignment statements, which are exactly analogous to assignment statements in traditional programming language. When a job passes through a set node, the assignment statement is executed and the value of the expression on the right hand side of the assignment statement is assigned to the variable (typically a job variable or a global variable) on the left hand side of the assignment statement. This ability to associate additional information with a job through the use of job variables is extremely useful in modeling features such as the semantic contents of a message (e.g., a message type or size) or for dynamically determining the next modeling element to be visited by a job.

3.1.2 Modeling Elements

The GENESIS modeling elements are the objects between which jobs flow. When a job arrives at an element, some action may be taken upon that job by the simulation, an action may be taken upon another job, or the state of the simulation may be changed. Each modeling element has three important properties:

- an *action* to be taken by the simulation
- one or more *attributes* which further qualify the action to be taken.
- a graphical *icon* representing the object

Figure 1 lists three of these elements and their associated actions, attributes and icons and Appendix B lists all the modeling elements we have developed so far. The list is not meant to be

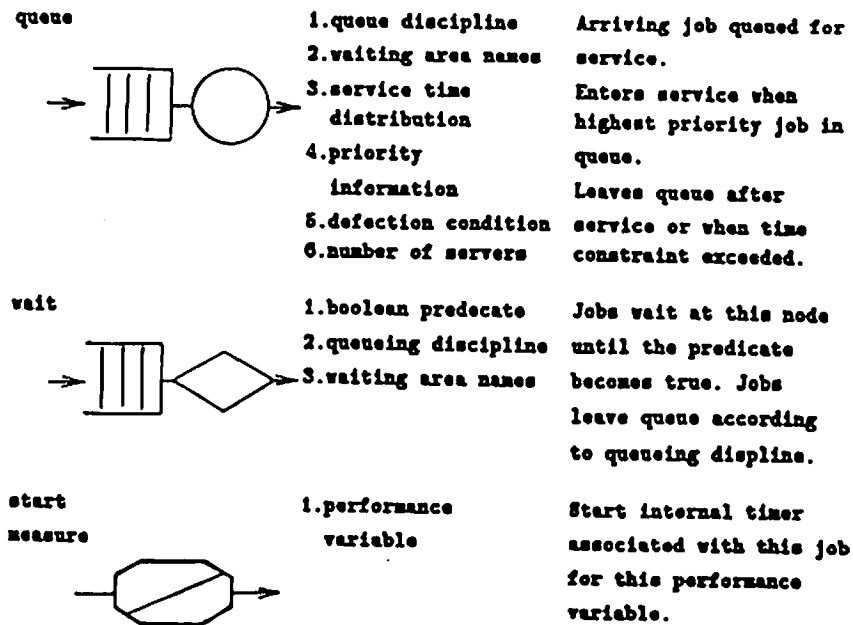


Figure 1: Several representative modeling elements

exhaustive (and we expect to add/delete from any such exhaustive list as we continue to gain modeling experience using GENESIS) but rather to provide a feel for the types of modeling objects which are provided by the environment.

The first example in Figure 1 is a *queue modeling construct* which might be used to model a point-to-point communication link, a central processing unit, or some other network resource. The action taken when a job arrives at a queue is that the job is queued for service; this job is eventually selected for service at the queue according to the specified queueing discipline. As shown in Figure 1, a queue's attributes include its service discipline (e.g., FCFS, LCFS, priority, etc.), the names of the one or more *waiting areas* associated with the queue, the distribution of job service time, a queue defection condition, and the number of servers at the facility.

A second example of a modeling element is the *wait node*, which is provided to permit synchronization among jobs and may also be used to model simultaneous resource possession by a single job. Due to the modeling power and flexibility which can be achieved by the use of this single element type, we have chosen to introduce wait nodes into GENESIS rather than incorporating the notion of passive queues used in [28] [4] to model synchronization and simultaneous resource possession. Each wait node has an associated predicate which specifies the conditions under which a job is allowed to pass through the wait node; a queueing discipline also specifies the order in which jobs leave the wait node should a number of jobs become eligible to leave simultaneously. When a job arrives at a wait node, the predicate value is checked. If the predicate value is TRUE, the job passes through the wait node instantaneously. If the predicate value is false, the job remains

at the wait node until the predicate becomes true and the job is selected to exit the wait queue. In our nonpersistent CSMA/CD model in section 4, a wait node is used to insure that a station never attempts to transmit more than one message at a time when there are multiple messages queued for transmission.

A final example of a modeling element is the *startmeasure* node. This node is useful for obtaining performance statistics concerning the time between two events in the simulation. In our non-persistent CSMA/CD model, for example, a *startmeasure* node is used to determine the statistics of the channel access delay. A symbolic name, known as a *performance variable*, must also be associated with each *startmeasure* node. When a job flows through a *startmeasure* node, a timer is associated with the job and is started (internally, by the simulation). When the job flows through a *stopmeasure* node with the same associated performance variable as the *startmeasure* node, the timer is stopped. This timer value is then taken as a sample of time needed by a job to pass from a *startmeasure* node to a related *stopmeasure* node.

We have additionally found that many of the modeling constructs previously developed for modeling wide-area networks [26] [27] [28], are also useful in modeling multiple access networks. For example, global variables, set nodes, source, sink and split nodes are useful modeling constructs for modeling high-level protocol functions such as routing and flow control in multi-hop multiple access networks.

A *submodel* facility is also provided by the GENESIS modeling environment. As in RESQ [28], a submodel is essentially an analyst-defined parameterized template of an interconnected set of modeling elements which together represent a single functional unit in the system being modeled. This template can be *invoked* to create several *instances* of the submodel, in much the same way that a procedure can be invoked (called) multiple times in traditional programming languages; we refer to each such created instance of a submodel as an *invocation*. In the following example, a submodel is created to model the operation of a generic network station and is invoked once for each actual station in the network. The use of submodels can greatly aid in the development of hierarchical, well-structured, and logically correct models.

3.1.3 Routing

Job routing specifies the manner in which jobs move between the modeling elements contained in the simulation model. The movement of a job from one modeling element to another may be fixed or probabilistic, or may depend on the current state of the simulation. Thus, when a job leaves a given element in the model, its destination may be determined:

- **deterministically.** In this case, the job's destination is always the same when leaving a given element. In our model in the next section, for example, a job always visits a startmeasure node immediately after leaving a source node.
- **probabilistically.** In this case, the job's destination is independently determined from a fixed set of analyst-specified probabilities each time it leaves the given element.
- **according to the current simulation state.** In this case, the job's destination may depend on the current status of the simulation. For example, in our nonpersistent CSMA/CD model, a job leaving the SET_SEND set node either proceeds to a delay node or enters the communication channel, depending on whether the multiaccess channel is sensed busy or idle. A job's routing may also depend on other simulation dependent values such as the number of jobs at a given queue, the value of a job variable, and/or the current simulated time.

3.2 The Channel Model

The unique characteristics of multiple access communication channels makes modeling the channel one of the most challenging tasks in the development of our performance evaluation environment. In order to model and simulate the multiple access aspects of the channel, the communication channel model itself must have the capability of accounting for zero, one, or multiple simultaneous transmissions by any of the network stations. Problems such as signal fading and capture, propagation delay, changing network topology, or a changing number of network stations must also be taken into consideration. Moreover, to accomplish our goal of providing the modeler/analyst with an environment for modeling and simulating any *arbitrary*, user-defined protocol and, at the same time, being able to provide the modeler/analyst with sufficient domain-specific tools, our channel model must be conceptually high-level, flexible and general, while at the same time internally efficient and capable of simulating a large class of channel characteristics.

At the highest level, the channel consists of a black box with a number of *channel ports*. A port can be thought of as a "plug" into which a set of modeling constructs (typically a submodel representing a network station) can be connected; there is thus typically one port for each station in the network model. Each such port has three associated *port variables* and two types of *port connections*.

Port Variables

The port variables are used to define the "state" of a given port. For each port, i , $i = 1 \dots N$, where N is the user-defined number of ports, the port variables for station i are:

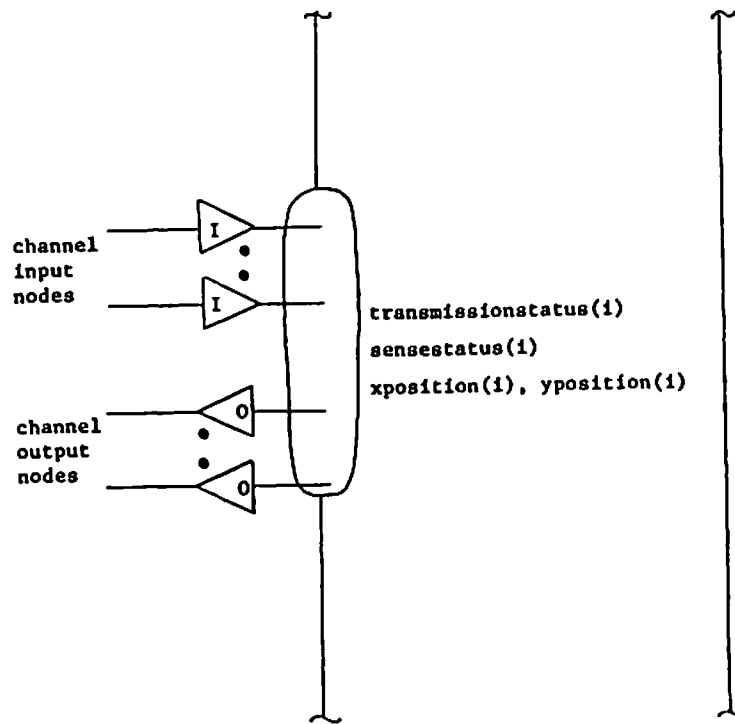


Figure 2: Channel port variables and port connections at port i

- $transmissionstatus(i)$, which has the value 1 while a message is being sent into port i , and has a value 0 otherwise.
- $sensestatus(i)$, which indicates the number of transmissions initiated at *other* ports which are currently available at port i . Note that the value of this variable may depend on port i 's position in the network as well as the channel's fading and capture characteristics.
- $xposition(i)$ and $yposition(i)$ define port i 's x and y position on the channel

The values of each of these port variables can be used (queried) directly within the simulation itself. Such values might occur, for example, in an expression at a set node, a predicate expression associated with a wait node, or in a job's routing decision. In our model of nonpersistent CSMA/CD in the following section, the value of $sensestatus(i)$ is used at a wait node to detect a message collision within the period of time during which station i is vulnerable to message transmissions by other stations.

The manner in which a value is assigned to a port variable varies from one port variable to another. The value $transmissionstatus(i)$ may be set either implicitly or explicitly. When a message passes through the channel input connection (to be discussed shortly) of port i , the

value of *transmissionstatus(i)* is implicitly set to 1 (transmitting). Similarly, the value of *transmissionstatus(i)* is implicitly set to 0 (idle) upon normal termination of a message transmission. A message transmission at port *i* may be prematurely aborted (e.g., as in a multiaccess protocol with collision detection capabilities) by explicitly setting the value of *transmissionstatus(i)* to 0 at a set node, as is done in our nonpersistent CSMA/CD model in the following section. Values may also be directly assigned to the port variables, *xposition(i)* and *yposition(i)*, at a set node; this permits the modeling and simulation of mobile nodes in multiple access networks.

Finally, the value of *sensestatus(i)* is never directly set within the analyst's model. Rather, this value is dynamically computed by the simulation. As discussed earlier, this value may depend on the station's position in the network as well as channel characteristics such as fading and capture and the connectivity (hearing graph) of the network being modeled. A default method for computing *sensestatus(i)* is provided by GENESIS and assumes that all ports are within hearing distance of each other and that no signal fading and capture occurs. The analyst may model more complicated channel properties by supplying a *user-written* C routine which when passed a port number, *i*, and the identities and locations of the ports which are currently transmitting, returns the value of *sensestatus(i)*.

Port Connections

Each port also supports two kinds of connections for sending jobs (messages) into the channel and for receiving jobs (messages) out of the channel. These connections are represented by nodal icons shown in figure 2. Each connection to a port is *named* (just as all other modeling elements are given symbolic names by the modeler) and an arbitrary number of each of these two types of connections can be associated with a single channel port.

The first of the two types of port connections is the channel input connection. As discussed above, when a job passes through a channel input connection, the simulator *internally sets* the state of the port to which this connection is bound to 1 (transmitting); the transmission starting time is also internally recorded. Each channel input connection has a single associated attribute: a *message-length distribution*, which specifies the distribution of time required to send a message into the channel through this port connection (note this is equivalent to specifying a channel bit rate (capacity) and the distribution of the number of bits in a message).

The second type of connection is the channel output connection, which models the ability of a station to read messages from the channel. When this connection is included for port *i*, a copy of *every* successfully transmitted message (i.e., a message which passes port *i*, uncorrupted and

in its entirety) is emitted from the connection whenever the trailing end of a transmitted message propagates past the attached port.

Internal Channel Model

The internal implementation of our channel model is passive and uses techniques from relational databases to minimize the amount of simulation overhead required. There are two types of channel attributes: static and dynamic. Static characteristics such as the channel's fading and capture characteristics, and the number of attached stations are directly specified by the modeler in the same manner as other modeling element attributes. The dynamic characteristics of the channel are stored as tuples, with one tuple for each attached station. Each tuple records the station name, the starting and ending times of its last transmission, and its current position in the network.

The channel model is passive in the sense that procedures are (indirectly) called to compute or update the state of the channel when this information is required (e.g., when the value of a port variable is changed) or when the channel state changes (e.g., when a job enters the channel through a channel input connection.) Given the relational model of the dynamic channel attributes, computing the state of the channel translates to a simple query on these dynamic attributes; changing these attributes simply updates the tuples. Several well-known algorithms can be used to implement these database operations [6]. In effect, the state of the channel is procedurally determined from the past transmission starting and stopping times whenever this information is needed. Thus, *the actual transmission and propagation of messages along the channel need never be explicitly simulated*. This results in considerable reduction in the simulation overhead and a concomitant decrease in the amount of time required to simulate a multiaccess network and its protocols.

3.3 Simulator Design

As mentioned earlier, the development of simulation programs for modeling and evaluating communication networks represents a major software effort and our goal in building GENESIS is to reduce this effort such that the analyst can build arbitrary communication network simulation models using high-level constructs without worrying about the implementation details of the simulation itself. The actual simulation of network models constructed using the modeling language and the channel model described in the previous two sections is performed by a discrete-event-based "simulation engine" in GENESIS.

The simulation engine essentially moves jobs (see section 3.1 for discussion of jobs) from node to node in the user-defined simulation model. Routing of jobs within a simulation model is entirely

defined by the semantics of the modeling constructs used in the model and the occurrence of some event (see Appendix D for specifications). As described in section 3.1 and specified in Appendix B, each modeling construct in GENESIS has three important properties, an action to be taken, one or more attributes and a graphical icon. The first two of these three properties constitute the semantics of each modeling construct. Our simulation engine is so designed that the semantics of each modeling construct are mapped directly into one or more simulation routines which carry out the action and manipulate the attributes. Events are happenings, such as a JOB-ARRIVAL or a JOB-COMPLETION event, during a simulation that cause jobs to traverse from node to node in a simulation model. Meanwhile, when a job passes through a node, some future event(s) will also be scheduled as defined in the specification of a node (see Appendix D). Therefore, the simulation is driven by this causal relationship of discrete event occurrence and event scheduling.

The two most important data structures employed in the simulation engine to support the event-driven simulation are an event list and a job list (see Appendix C). The event list is a list of future events being scheduled. Each element in the event list contains information for one event and this information includes the event type (e.g. JOB-ARRIVAL, JOB-COMPLETION), the time when the event should take place, a pointer to a job element (described below) of the job being affected by this event and a port id which specifies the port at which the event is scheduled. Similarly, the job list is a list of jobs currently exist in the network model. When a JOB-ARRIVAL event occurs, a job element will be added to the job list and conversely as a JOB-COMPLETION event takes place, the job element corresponding to the JOB-COMPLETION event will be removed from the job list. The information kept in each job element in the job list includes a job id, the name of the current node (i.e. modeling construct) this job is at, the arrival time and the service time (generated from user-defined distribution) of the job, parent or child job(s) associated with this job and a set of user-defined job variables (as described in section 3.1).

Besides the event list and the job list, additional data structures are necessary in supporting the simulation of the channel and other aspects of the network model. The static and dynamic attributes of the channel model are represented by two table form data structures, called ChannelProperties and ChannelRelation respectively. Furthermore, a Node Table, a Symbol Table, a Routing Table, a Queue Table and an Expression Table are the other major data structures employed in the simulation engine. The specifications of all the above data structures can be found in Appendix C.

At the highest level in the simulation engine, the simulation of any network model proceeds as follows:

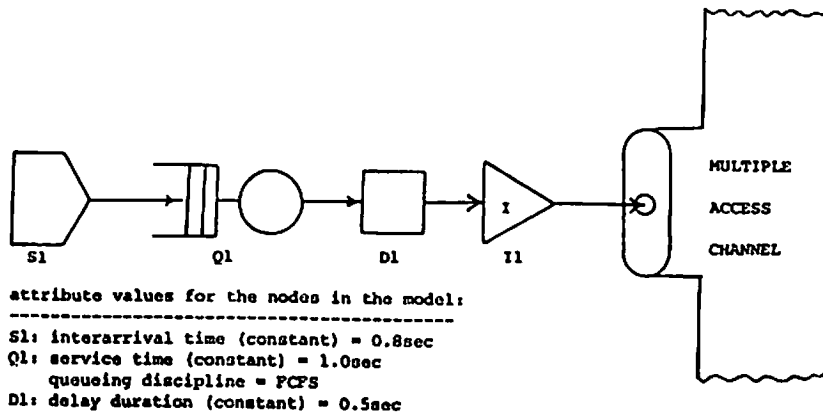


Figure 3: Example of a simple four node network model

Initially, a JOB-ARRIVAL event is scheduled for all source node(s) in the model and the internal channel model is initialized with all the attributes specified by the modeler. Then the simulation repetitively executes the following loop until the simulation time is up or some other stopping condition becomes true:

Loop

- Get an event off the front of the event list;
- Set current simulated time to event time;
- Process the event;
- Check waiting conditions;
- End

(The wait node as one of the modeling constructs in GENESIS allows jobs to wait (i.e. to be blocked) on some conditions (e.g. to wait for a certain value of some global variable). The last step in the above loop is thus to check if any of the waiting conditions have become true after processing the event. If so, wake up the job(s) waiting on the conditions.)

To further illustrate how the simulation is actually carried out, we present a simple four-node network model below. (See figure 3). (note: The purpose of this example is NOT to demonstrate the facilities GENESIS provides to a modeler for constructing any arbitrary user-defined models (see section 4 for such examples). It is only to show the operations involved in the simulation engine.)

Suppose at source node S1, the arrival rate is deterministic with interarrival time equals to 0.8 seconds. The queuing discipline at Q1 is first-come-first-serve and the service rate is also deter-

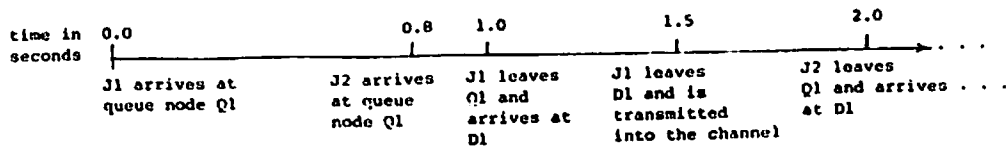


Figure 4: The occurrences of events along the time axis in the four-node simulation example

ministic with constant service time 1.0 second for all jobs generated and there is a constant delay, say 0.5 second, at delay box D1. (Again, since we are not demonstrating the kind of distributions GENESIS supports here, we are using all deterministic distributions to simplify the example.)

Now let's see how this simple four-node model is simulated in the high level simulation frame (please refer to figure 4 and Appendix D when reading the rest of this section). Initially, a JOB-ARRIVAL event will be scheduled for the source node S1 with event time equal to 0, and the channel model will be initialized according to some modeler specified attributes. Then we enter the beginning of the loop. This JOB-ARRIVAL event will be taken off the event list and the simulated time is set to the event time, i.e. 0. To process this event, as specified in the JOB-ARRIVAL event in Appendix C, simulation routine ARRIVAL will be called. It would create a job record for the arrival job with job id, say, J1 and put it on the job list; schedule next arrival for S1, i.e. a JOB-ARRIVAL event record will be put on the event list for next arrival job J2 with event time equal to 0.8 (since the interarrival time is 0.8 seconds) and move the current job J1 to its destination queue node Q1, i.e. the simulation routine Arrival-at-Active-Q will be invoked. Arrival-at-Active-Q generates the service time according to the modeler-specified distribution. In this case, the service time is always 1.0 second. If the queue is empty (and it is the case here), a JOB-COMPLETION event will be scheduled for job J1 at time 1.0 (i.e. after being serviced at the queue).

Now we are at the last step of the loop and since there is no wait node in our simple model, we are led back to the start of the loop again. The event with the smallest (i.e. nearest future) event time will be taken off the event list next. So the JOB-ARRIVAL event for J2 is processed here by setting the current simulation time to the event time 0.8 and invoking Arrival-at-Active-Q again. J2 is then put on the queue in Q1 and a JOB-ARRIVAL event is scheduled for the next job J3 with event time equal to 1.6.

We are back to the beginning of the loop now. This time, the JOB-COMPLETION event for J1 will be processed and the simulated time will be set to 1.0. In processing the JOB-COMPLETION event, the simulation routine *Depart-from-Active-Q* is invoked. This routine first checks to see if there is more than one job in this queue and if there is, a JOB-COMPLETION event will be scheduled for next job in the queue based on the service time. In our case, J2 is on the queue and thus a JOB-COMPLETION event is scheduled for it with event time equal to 2.0 (current simulation time + service time at this queue). Then J1 is routed the next node in the model, i.e. the delay box D1, by calling routine *Arrival-at-Delay* which simply schedules a JOB-COMPLETION for J1 at time 1.5 (since D1 has a constant delay of 0.5 second and current simulated time is 1.0).

Once again, we are led back to the beginning of the loop and the JOB-COMPLETION event for J1 at D1 will be the next event to be processed since the JOB-COMPLETION event for J2 is not due to take place until simulated time equals 2.0. This time, the JOB-COMPLETION event invokes *Depart-from-Delay* which immediately routes J1 to the start transmission node I1 by calling *Arrival-at-Channel*. *Arrival-at-Channel* 'sends' the job into the channel by enter the start transmitting time and set the state of the port to TRANSMITTING in the internal channel model. Also, *Arrival-at-Channel* schedules POSSIBLE-SUCCESS-TRANSMISSION events for arrival of the end of message transmission for all ports which have a channel-message-read node. Then we come back to the beginning of the loop and simulation continues by taking the next event (in time) off the event list (i.e. the JOB-ARRIVAL for J3) and proceeds in the similar fashion as described above.

Before closing up this section, it is important to notice that besides the time elapsed in the protocol operations modeled, e.g. service time, delay time, etc., the simulation processing in all the other nodes (e.g. start transmission node) is instantaneous, i.e. processing in these nodes does not affect the simulated time. It is the nature of discrete-event driven simulation that the simulation does not have to run 'continuously' along the time axis and, as figure 4 shows, it only has to simulate at discrete time intervals as events take place.

3.4 Overall GENESIS Architecture and the Modeling Environment

A sketch of the overall GENESIS architecture is shown in Figure 5. To specify a simulation model in GENESIS, the user uses the high-level modeling language provided by the environment as described in the previous sections. The /it graphical model construction environment and the /it graphical simulation display and control are the two front-end graphics units interfacing with the user. The *graphical model construction environment* supports the user in building the simulation model graphically by providing the user a graphical description of the simulation model. The basic

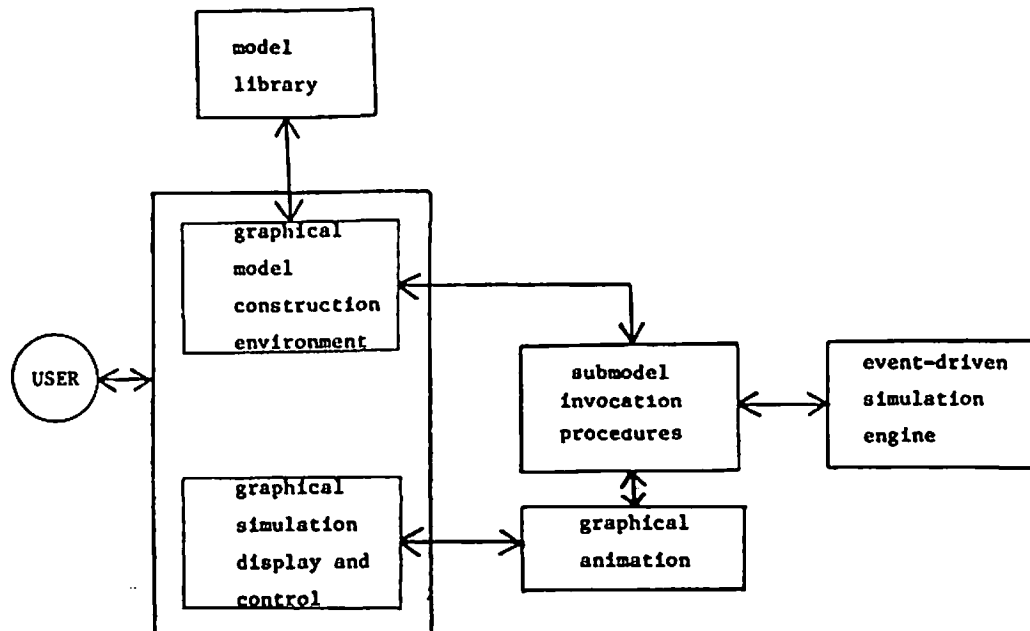


Figure 5: Block diagram of GENESIS architecture

programming elements in the environment are icons, symbols which carry both contextual and semantic information. Icons, which are stored in the model library, are picked up from palettes located at the side of the display and placed on the modeling field portion of the display. The attributes associated with the modeling element icon are then specified using pop-up windows on the screen. Objects can be connected either manually (under user control) or by the environment's line drawing algorithms. The *graphical simulation display and control* together with the *graphical animation* module will further support the graphical simulation by displaying the actions and effects of the various modeling objects as the simulation proceeds, i.e. animating the simulation execution. This feature will be particularly helpful in debugging simulation models and in understanding the general behavior of the modeled system and the interactions among the modeled system components.

As previously discussed, the submodel facility is provided in order to support and encourage the development of hierarchical, well-structured models. Graphical support for the use of submodels includes the ability to replace the original graphical description of a subnetwork with a single block representation of the submodel subnetwork and the use of zoom in/out features to focus the modeler's attention of the portion of the simulation model currently under consideration. The

role of the *submodel invocation procedures* is to create a complete internal copy of *each* invoked submodel definition. This includes mapping the parameters in the submodel invocation to those in the submodel definition, creating (for the simulation) instances of the modeling elements defined in the invoked submodel, and connecting these elements into the job routing specification.

At the furthest end in the modeling environment from the user is the *event-driven simulation engine* as described in section 3.3. This is where the user-defined high-level simulation model specification is mapped into the low-level internal environment data structures and internal procedures which carry out the details of the simulation.

Since GENESIS is still in the state of development, modification and addition to its overall architecture should be expected.

4. Examples

In this section, we present two examples to demonstrate how various modeling language elements and the channel model discussed in the previous few sections can be combined to create high-level models of multiple access protocols. To construct such simulation models, the modeler simply selects appropriate modeling constructs and associate necessary semantic contents (i.e. attributes) with each modeling construct to accomplish the required protocol operations being modeled. Our goal in presenting these two examples is *not* to examine the operation of the specific multiple access protocols. Rather, our aim is to show the natural and easy manner in which simulation models can be constructed using such a high-level approach and therefore to demonstrate the power and flexibility of our approach.

4.1 non-persistent CSMA/CD

Figure 6 shows the graphical representation of the highest level of our model of a 4 station multiaccess channel. At this level, the model simply consists of the channel and four invocations of a single parameterized station submodel. Figure 7 shows the pictorial representation of the submodel definition for a station using a non-persistent CSMA/CD protocol [32]. The attributes of the channel and of each of the elements shown in figures 6 and 7 are given in the textual portion of the model description contained in appendix A. This textual specification of the declared attributes of a model is obtained by default after the model has been constructed graphically. Note that in this particular model, the actual connection between a station invocation and the channel is specified in the submodel definition, while the channel port characteristics are specified in the main model.

Let us now briefly discuss the submodel's operation. As indicated in the textual portion of the model in appendix A, two parameter values are contained in the submodel definition. The first parameter, *stationid*, is used in the submodel to uniquely identify each invocation of the station submodel. The second parameter, INPUTPORT, provides the name of the channel input connection into which the station will send its messages.

Jobs (representing messages to be transmitted) are generated at source node SOURCE1 according to a user-specified random process (in this case, with an exponentially distributed interarrival time of .01 seconds). Once a job leaves the source node, it first visits the startmeasure node, STARTACCESS, in order to obtain performance statistics regarding the time between its arrival at a sending station and its successful transmission into the channel. The job then waits at the wait node WAITSEND until the condition (SENDING==FALSE) becomes true; this condition is required to insure that a station never simultaneously transmits two of its own messages.

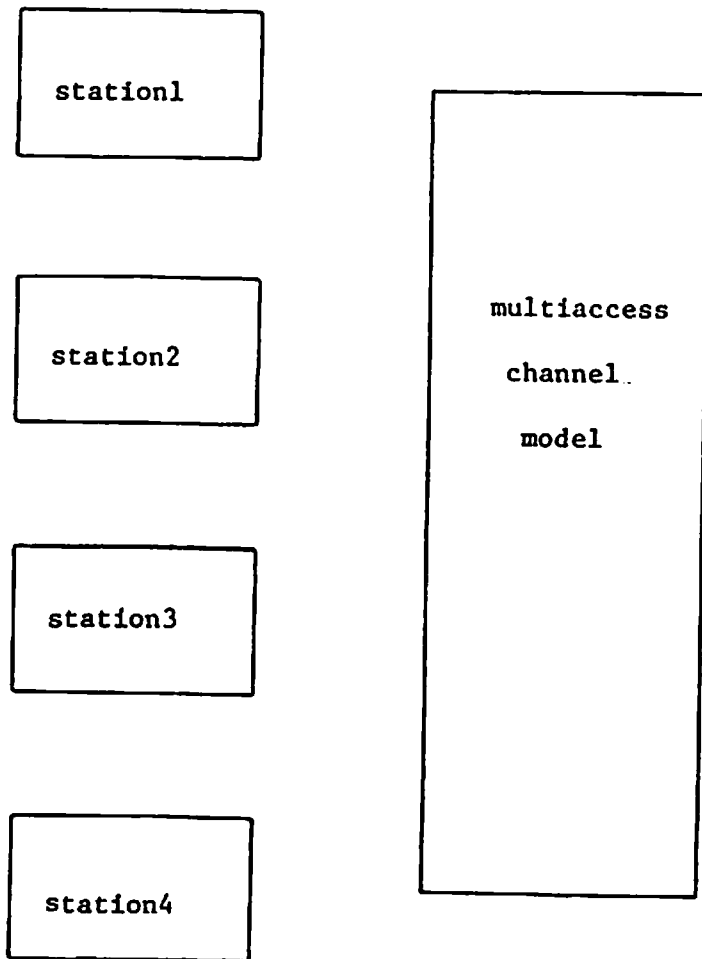


Figure 6: The main model of a 4 station CSMA/CD network

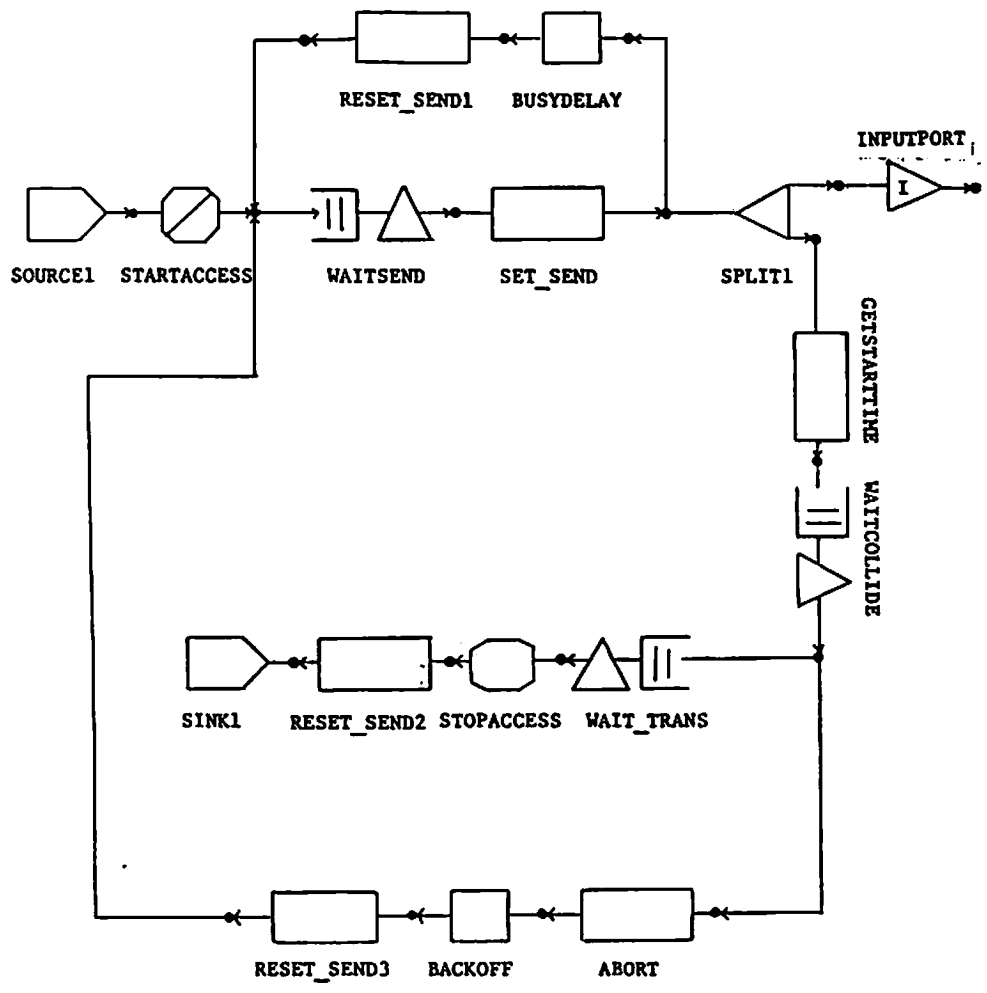


Figure 7: A station submodel for non-persistent CSMA

When a job exits WAITSEND, it first sets the value of SENDING to TRUE at the set node SET_SEND to insure that all other arriving jobs will wait at WAITSEND. The job's destination after leaving the set node then depends on the current value of *sensestatus(stationid)*. If *sensestatus(stationid) ≠ 0*, then the channel is busy and the message is delayed a random amount of time (in this case, we chose a random amount of time, arbitrarily between 0 and 0.5 seconds). The job then sets the value of SENDING to FALSE to permit messages to again be transmitted by the station, and then rejoins the WAITSEND wait queue.

If *sensestatus(stationid) == 0*, then the channel is currently idle and a copy of the message is made at the split node, SPLIT1. This job copy (now a data-oriented job, representing the message being sent into the channel) proceeds to the node parameter INPUTPORT, which is the channel input connection for this station; at this point in the simulation, a message transmission by station *stationid* begins. The message transmission time will be determined either by the connection's message transmission time distribution or by the premature abortion of the transmission, as discussed below.

A second job (now a control-oriented job representing the flow of execution of the protocol) exits from the lower half of the split node and records the starting time of the message transmission at the set node, GETSTARTTIME. The job then waits at the node, WAITCOLLIDE until either a message collision is detected or the message has had sufficient time to propagate to all other stations in the network and back, whichever comes first.

If the job leaves the wait node and *sensestatus(stationid) == 0*, then the CSMA/CD protocol insures that the message will be successfully transmitted in its entirety. Thus, the control-oriented job waits until the data-oriented job has terminated transmission and then visits the stopmeasure node STOPACCESS, where the value of SENDING is set to FALSE to insure that additional messages may now be transmitted by this station. The job then leaves the set node and is removed from the simulation when it arrives at the sink node SINK1.

If the job leaves the wait node and *sensestatus(stationid) ≠ 0*, then a collision has occurred (i.e., another station has also attempted a simultaneous message transmission). In this case, the job immediately aborts message transmission by passing through the set node, ABORT, and sets the value of the port variable, *transmissionstatus(stationid)* to 0. The control-oriented job is then delayed (again, as specified by the non-persistent CSMA protocol) a random amount of time. After leaving the delay node, the job then sets the value of TRANSMITTING to FALSE (to permit the station to again attempt message transmissions) and then returns to the wait node.

Finally, we note that since we are only interested in determining the average access time for a message, we have not included a channel output connection to a port in the station submodel.

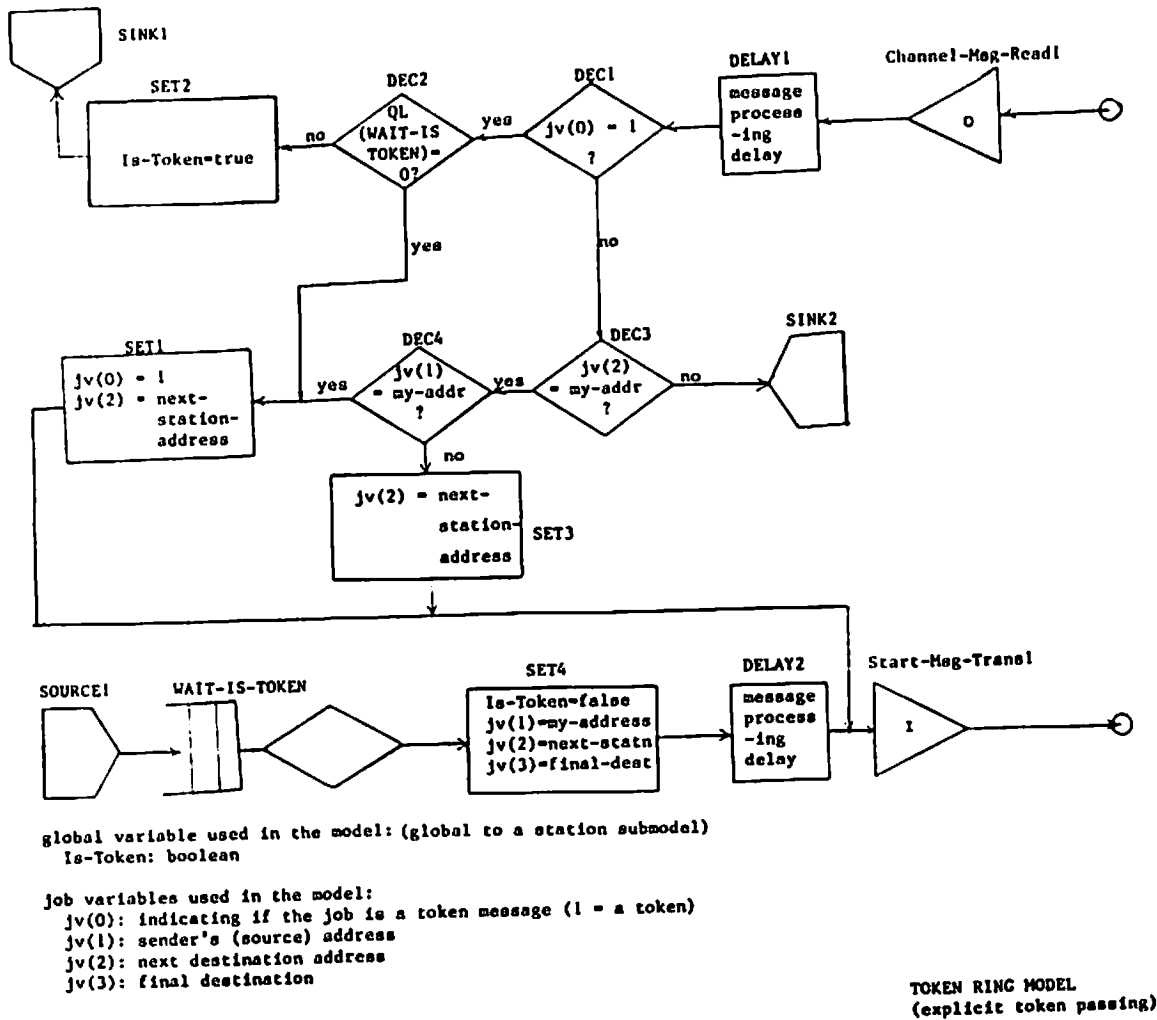


Figure 8: A token ring protocol submodel

Thus, a message sent into the channel will never appear out of the channel at another station. If we were also interested in the actions taken by a station when a message was received (e.g., if we wished to model the effects of transmission errors), then we need only include the channel output connection and specify (through the appropriate use of modeling elements) the actions to be taken upon the receipt of a message at a station.

4.2 Token Passing Ring

Here, we present an example of the token passing ring protocol submodel in Figure 8 to illustrate that the modeling language in GENESIS is not only limited to model multiple access protocols for one kind of communication channel, i.e. broadcast channel. Its generality provides the modeler the ability to model multiple access protocols for /it any arbitrary kind of communication channels, such as the unidirectional communication channel being modeled in this example. Again, as in the

CSMA/CD protocol submodel example given above, the token passing ring submodel would be invoked once for each station in the network system being modeled.

In the token ring protocol with explicit token passing [19], the transmission of messages into the channel is *controlled* by the protocol. Thus no collision among message transmissions would occur. The main challenge of modeling this protocol is then the token passing mechanism. As one can see after we trace through the submodel later that the *control* aspect of the protocol is the major part of the submodel.

In this example, we assign meanings to four job variables for any job that comes in or generated by the station protocol submodel (see Figure 8 for explanation of the four job variables). The semantics of these job variables directly support the modeling of the unidirectional channel. A global variable, *Is-Token*, is used in the submodel to indicate that the current station submodel has the token and therefore the permission to transmit if it has any messages to send. Now, let's trace through the submodel to see the various protocol operations it does.

At the source node SOURCE1, jobs are generated based on some interarrival time distribution specified at this node and the jobs will be queued at the wait node WAIT-IS-TOKEN to wait for the condition *Is-Token == true* to become true.

Meanwhile, if there is any job on the channel passing by this station, the channel message read node Channel-Msg-Read1 will take the job off the channel and route it to DELAY1. After some message processing delay (e.g. check the checksum value), the semantic contents of the job will be examined by the protocol model. If *jv(0)* equals 1 at decision node DEC1, indicating that this job is a *token* message, a function QL will be invoked at decision node DEC2 to check the queue length of WAIT-IS-TOKEN. If the queue length is not 0 there (i.e. there are jobs waiting to be sent at the station), then the global variable *Is-Token* will be set to true and the job enters SINK1. Now, since the *Is-Token == true* condition is true, the next job waiting to be sent at WAIT-IS-TOKEN can be transmitted by assigning values to its job variables at the set node SET4, experiencing some message processing delay at DELAY2 and finally going through Start-Msg-Trans1 into the channel. On the other hand, if the queue is empty at WAIT-IS-TOKEN (i.e. the queue length == 0), the token message will be assigned the next station address at SET1 and passed on to the next station from the Start-Msg-Trans1.

If the job read in from the channel is *not* a token message, i.e. *jv(0)* is not equal to 1, it has to be decided if this job should be passed onto the next station in the network to preserve the unidirectional channel property. *jv(2)* will be checked at decision node DEC3 first to see if the job has the current station's address as its next destination. If it doesn't, then the job is discarded by entering the sink node SINK2. If it is destined for the current station, *jv(1)* will be examined

to see if the job is originally sent by the current station. If it is originated here, it means that the message has gone through the entire network and therefore it can be taken off the channel by the sender. The sender (i.e. the current station) will re-issue the token message to give the next station permission for transmission by setting $jv(0)$ to 1 and $jv(2)$ to the address of next station at SET1 and send the token out through Start-Msg-Trans1. However, if the job is not originated by the current station (but has the current station as its next destination) as decided by the decision node DEC4, it will simply be passed on to the next station by setting the next destination address to the address of next station at SET3 and transmitting out through Start-Msg-Trans1.

5. Conclusion

In this report, we have described a *third generation*, graphics-oriented, modeling and performance evaluation environment, GENESIS, for simulating performance models of multiple access networks and protocols. The high-level approach used in GENESIS will greatly enhance the productivity of the network modeler since the performance models will be significantly easier to generate, debug, controlled and understood within such an environment. GENESIS is right now under development and some issues still need to be considered further. How to animate the simulation process such that the modeler can actually have control over the simulated system interactively is one of such issues. Also, the set of modeling primitives we have designed so far are by no means exhaustive and as we gain more experience during the development process, more modeling constructs may be developed and existing ones may be modified to make the environment more effective. Finally, to provide more flexibility and freedom to the modeler, GENESIS will support user-defined modeling elements when the system-defined modeling primitives are not sufficient and this is another area needed further development.

ACKNOWLEDGMENTS

Many individuals have contributed their ideas, time, and effort in the development of GENESIS. We are particularly grateful to Kurtiss Gordon, Saraswathi Krithivasan, and David Jacobs for their contributions.

Appendix

A. Attributes of the nonpersistent CSMA model

/* This first file is the textual portion of the specification of the main (highest-level) model of a four station non-persistent CSMA/CD network. Note that the actual CSMA protocol (i.e., the behavior of each station) is specified in a submodel */

MODEL NAME: fourstation

```
%include station          /* include the station submodel */

INVOCATION: station1      /* create an instance of a station */
  SUBMODEL NAME: station  /* by providing the submodel name */
  STATIONID: 1           /* and a value for the parameter */
INVOCATION: station2
  SUBMODEL NAME: station
  STATIONID: 2
INVOCATION: station3
  SUBMODEL NAME: station
  STATIONID: 3
INVOCATION: station4
  SUBMODEL NAME: station
  STATIONID: 4

MULTIACCESS CHANNEL MODEL /* Now declare channel characteristics */
  CHANNEL SENSE:          /* use default connectivity, fading, capt */
  NUMBER OF PORTS: 4     /* 4 stations, one per channel port */
  PORT 1 -
    INITIAL POSITION: (0.0,0.0)
    CHANNEL INPUT CONNECTIONS: input1
      MESSAGE LENGTH DISTRIBUTION: fixed(0.0001) /* 1K pkts, 10MB chanl */
    CHANNEL OUTPUT CONNECTIONS:
  PORT 2 -
    INITIAL POSITION: (100.0,0.0)
    CHANNEL INPUT CONNECTIONS: input2
      MESSAGE LENGTH DISTRIBUTION: fixed(0.0001)
    CHANNEL OUTPUT CONNECTIONS:
```



```

PORT 3 -
  INITIAL POSITION: (300.0,0.0)
  CHANNEL INPUT CONNECTIONS: input3
  MESSAGE LENGTH DISTRIBUTION: fixed(0.0001)
  CHANNEL OUTPUT CONNECTIONS:
PORT 4 -
  INITIAL POSITION: (400.0,0.0)
  CHANNEL INPUT CONNECTIONS: input4
  MESSAGE LENGTH DISTRIBUTION: fixed(0.0001)
  CHANNEL OUTPUT CONNECTIONS:
ROUTING:                               /* no routing needed in main model */
SIMULATION RUN INFORMATION -
  QUEUES FOR QUEUEING TIME DISTRIBUTION:
  PERFORMANCE VARIABLES DISTRIBUTIONS: station1.accesstime
  SIMULATED TIME: 10000
  NUMBER OF SIMULATED EVENTS:
END

/* this submodel contains the model of an individual station */
SUBMODEL: station                       /* submodel for a generic station */

NUMERIC PARAMETERS: stationid          /* number of this station */
NODE PARAMETERS: INPUTPORT             /* channel input port for this station */

GLOBAL VARIABLES: sending              /* whether this station may send */
  TYPE: boolean
  INITIAL VALUE: FALSE
PERFORMANCE VARIABLES: accesstime      /* time from arrival to success */

DELAY NODES: busydelay, backoff        /* delay times chosen arbitrarily */
  DELAY AMOUNTS: uniform(0.0,0.5), uniform(0.0,0.5)

SET NODES: set_send
  ASSIGNMENT: sending=TRUE
SET NODES: reset_send1, reset_send2, reset_send3
  ASSIGNMENT: sending=FALSE
SET NODES: getstarttime                /* get start time for this msg */
  ASSIGNMENT: jv(0)=time

```

SINK NODES: sink1

SOURCE NODES: source1

INTERARRIVAL TIME: exp(.01) /* exponential with mean of .01 secs */

SPLIT NODES: split1

STARTMEASURE NODES: startaccess

PERFORMANCE VARIABLE: accesstime

STOPMEASURE NODES: stopaccess

PERFORMANCE VARIABLE: accesstime

WAIT NODES: waitsend

WAITING AREAS: waitsend1

CONDITION: sending==FALSE

WAIT NODES: waitcollide

WAITING AREAS: waitcollide1

CONDITION: (sensestatus(stationid)>0) OR (time-jv(0)>.00005)

/* .00005 is worst case end-to-end delay taken from the */

/* Ethernet Specification (see Comp. Comm. Rev, July 1981) */

WAIT NODES: wait_trans

WAITING AREAS: wait_trans1

CONDITION: transmissionstatus(stationid)==0

JOB ROUTING DEFINITION:

source1 -> startaccess -> waitsend -> set_send

set_send -> busydelay split1; if(sensestatus(stationid)>0)

if(sensestatus(stationid)==0)

busydelay -> reset_send1 -> waitsend1

split1 -> inputport getstarttime

getstarttime -> waitcollide1

waitcollide1 -> wait_trans1 abort; if (sensestatus(stationid)==0)

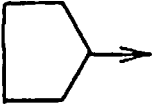

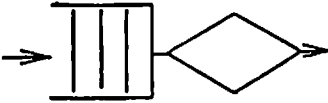


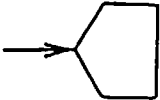
if (sensestatus(stationid)!=0)

wait_trans1 -> stopaccess -> reset_send2 -> sink1

abort -> backoff -> reset_send3 -> waitsend1

END

B. Modeling Elements Specification

NODE NAME -----	ICON ----	ATTRIBUTES -----	ACTIONS -----
source		1.interarrival time distribution	Generate job arrivals according to specified distribution.
queue		1.queue discipline 2.waiting area names 3.service time distribution 4.priority information 5.defection condition 6.number of servers	Arriving job queued for service. Enters service when highest priority job in queue. Leaves queue after service or when time constraint exceeded.
wait		1.boolean predecate 2.queueing discipline 3.waiting area names	Jobs wait at this node until the predicate becomes true. Jobs leave queue according to queueing disipline.
start measure		1.performance variable	Start internal timer associated with this job for this performance variable.
stop measure		1.performance variable	Stop internal timer associated with this job for this performance variable. Use the time value as a sample value of performance variable.
sink		none	Job arriving at a sink node is removed from the simulation.

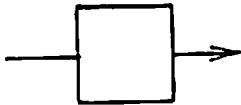
set



1.assignment
expression

Job flowing through set nodes causes value of expression to be assigned to a specified job or simulation variable.

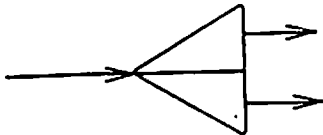
delaybox



1.delay distribution

Job arriving at delay box leaves box after specified amount of time.

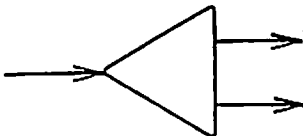
split



none

Job passing through split node is duplicated. A duplicate exits via upper path, original via lower path.

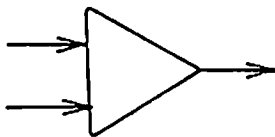
fission



1.fields for the
dependent job

When a job goes through a fission-node, a dependent (child) job will be created with given attributes.

fussion

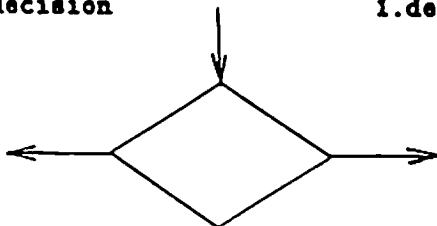


none

The fussion node is where all the dependent jobs created by some fission node(s) come together.

A job that arrives at a fussion node first must wait for the other dependent job.

decision

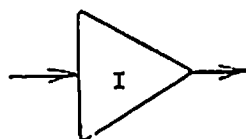


1.decision expression

A decision node has two outgoing routes. A job arrived at a decision node will be routed out at either outgoing route based on the evaluation of

the decision expression specified for the decision node.

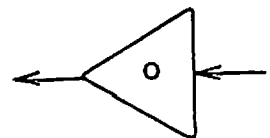
channel input



1. message transmission time distribution

A job entering this node causes the value of transmissionstatus(i) to be set to 1(transmitting). This value will be set to 0(idle) after an amount of time drawn from the specified message transmission time distribution.

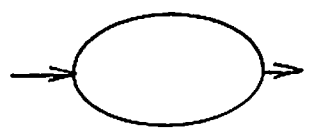
channel output



none

A job which is sent into the multiaccess channel at some port, j, is emitted from this node if it is transmitted in its entirety (without) collisions) and port j is within hearing distance of port i.

op-node



1. objects to be operated on
2. user-defined operations

This node is to support operations on user-defined objects. When a job goes through an operation-node, operations specified at this node will be performed upon some user-defined object(s).

C. Data Structures supporting the Simulation Engine

```
/* This file contains all the declarations for all the definitions
   and global data structures associated with the event list. */

/* the following macro definitions are used to define the various
   event codes. */

#define LEAVESOURCE 1
#define LEAVEQUEUE 2
#define LEAVEDELAY 3

/*****
 * EventRecord contains information of an event to
 * happen at some time in the future. EventRecords
 * are linked into a linked list with the next_event
 * field and an EventRecord will be removed from the
 * list upon completion.
 *****/

struct EventRecord
{
    int      event_id;      /* an integer code for event */
    double   event_time;    /* time at which event occurs */
    struct JobRecord *job; /* ptr to job affected by event */
    struct StationRecord *station;
    struct EventRecord *previous_event,
                          *next_event;
};

extern struct EventRecord *EventList; /* ptr to front of event list */
```

```

/*****
 * JobRecord represents a job with the information      *
 * associated with that job. All jobs are linked into  *
 * a linked list with the field next_job. Although    *
 * conceptually a job moves from node to node in the  *
 * network model during the simulation, the actual job *
 * record stays in the job list until the completion of *
 * the job. The attributes of a job will be modified  *
 * whenever necessary during the simulaton.          *
 *****/

```

```

struct JobRecord
{
    int    Job_id;
    int    current_node; /*an integer code for a node */
    int    arrival_time;
    int    service_time;
    struct JobRecord *next_job;
    struct JobRecord *parent;
    struct JobRecord *child;
    int    JobVars[10]; /* job variables, semantics are defined */
                    /* by the user */

    /* These fields are to be considered later in the project. */
    /* struct JobRecord *link_pointer-1, */
    /* *link_pointer_2; */
    /* wait_type wait_cond; */
};

extern struct JobRecord *JobList; /* pointer to JobRecord */

```

```

/* This file contains all the external definitions for the node
   table.                                                                 */

#define NNODES 100
extern struct NodeEntry {
    int    n_name;          /* index into symbol table def for this node */
    int    n_type;         /* code for the node type. Codes are:
        1 - class          2 - split
        3 - fission       4 - fusion
        5 - set           6 - delay
        7 - source        8 - sink
    */
    double n_value;        /* unused */
    int    n_expr;         /* index into the expression table for an
        expression associated with this node */
    double n_waitime;      /* perf. value - total wait time of all jobs */
    int    n_maxlen;       /* perf. value - max length of a class */
    int    n_ndepart;      /* perf. value - # departures from this node */
    struct distel *n_distptr; /* perf value - pointer to a linked list */
        /* of values for which queueing time */
        /* distributions are to be measured */
} NodeTable[NNODES];

```



```

/*****
 * This file contains the definition for the symbol table.  *
 * For now, this is only a simplified version of the symbol *
 * table to hold the names for nodes and queues.          *
 *****/

#define SYNTABSIZE 100

struct SymTabEl
{
    char *name;
    double value;
    int typecode;           /*e.g. global or job variables */
    int inited;            /* value initialized or not */
};

extern struct SymTabEl SymTab[SYNTABSIZE];

```

```

/* This file contains all the external definitions for the queue
   table.
*/

#define NQUEUE 100

struct classel { /* this structure used in each queue table entry */
    int classname; /* index into node table for this class */
    struct classel *next; /* ptr to next class element for this Q */
};

extern struct QueueEntry { /* QUEUE Table - info about all queues */
    int q_name; /* index into symbol table def for this Q */
    int q_type; /* code for the node type. Codes are:
                1 - active          2 - passive */
    int q_nservers; /* number of servers */
    int q_discipline; /* queuing discipline. codes are:
                    1 - FCFS          2 - LCFS
                    3 - Random
                    4 - nonpreemptive Pri
                    5 - preemptive resume priority
                    6 - preemptive restart priority
                */
    struct classel *classlist; /* ptr to list of classes associated w/ Q */
    double q_value; /* unused */
    int q_expr; /* unused */
    double q_time_busy; /* perf. value - total time queue is busy */
    double q_waittime; /* perf. value - total wait time of all jobs */
    int q_maxlen; /* perf. value - max length of a class */
    int q_ndepart; /* perf. value - # departures from this node */
    struct distel *q_distptr; /* perf value - pointer to a linked list
                               of values for which queuing time
                               distributions are to be measured */
} QueueTable[NQUEUE];

```

```

#define EXPRISIZE 50

extern struct exprel
{
    int type;
    double value;
} exprtab[EXPRISIZE];

extern struct exprel *position;

/*****
*   for the TYPE field of exprel   *
*       1 = number                 *
*       2 = +                      *
*       3 = - (binary)             *
*       4 = *                      *
*       5 = /                      *
*       6 = - (unary)              *
*       7 = symbol                  *
*       8 = exponentiation         *
*       9 = ln                      *
*      10 = sqrt                    *
*      11 = absolute                *
*      12 = mod                     *
*      13 = div                     *
*      14 = max                     *
*      15 = min                     *
*      16 = uniform distribution    *
*      17 = exponential distribution *
*      18 = geometric distribution *
*      19 = poisson distribution    *
*      20 = bernouli distribution   *
*      21 = erlang distribution     *
*      22 = normal distribution     *
*****/

```

```

/* This file contains the definitions of the external data structures
   for the routing table.  The ith element in the routing table is
   the *from* node for a routing definition.  **All* the "to" nodes
   (directly reachable from this "from" node) are contained in a linked
   list pointed to by the entry in the routing table          */

#define NROUTENTRY 100

extern struct RoutEntry          /* the "from" nodes, one entry per node */
{
    struct RoutElm *ptr;        /* pointer to list of elements,
                                one element per destination */
} RouteTable[NROUTENTRY];

struct RoutElm                  /* the "to" nodes */
{
    int destination;           /* node table index of a destination */
    float rout_type;          /* in [0,1] => probabilistic routing */
                                /* = 1      => deterministic routing */
                                /* = -1     => boolean expression */
    int exptr;                 /* index into the expression table */
                                /* for the expression to be evaluated */
    struct RoutElm *next;     /* pointer to the next destination of */
                                /* the same "from" node */
};

```

```
/****** data structures for the channel *****/
```

```
/******  
 * ChannelProperties is a table containing the static *  
 * properties of a channel given by the user before *  
 * simulation. *  
 *****/
```

```
extern struct ChannelProperties
```

```
{  
    int number_of_ports;    /*total number of ports in the channel*/  
    float fading;          /*fading rate*/  
    boolean capture;       /*indicate if capture effect is  
                            going to be considered in the  
                            model*/  
};
```

```
/******  
 * All the dynamic channel attributes during a simulation *  
 * are contained in a relation. Each ChannelRelationRecord *  
 * is a tuple in the relation containing information of a *  
 * port's activities in the channel. *  
 *****/
```

```
extern struct ChannelRelationRecord
```

```
{  
    int port_id;  
    int port_position;  
    char state;    /* either 'i' or 't' for IDLE or TRANSMITTING */  
    int trans_start_time;  
};
```

D. Design Specifications of GENESIS Simulation Engine

Detailed Design Specification for GENESIS SIMULATION ENGINE

```
/*-----*/
*   This file contains the simulation routine specifications   *
*   for GENESIS.                                             *
*-----*/
***** high level simulation description *****

begin /* simulation */
-----

    Init_simulation      /* together with other things,
                          schedule a Job_Arrival event for all sources */

    Init_Channel(ChannelProperties, ChannelRelation)

    Loop
    ----

        get an event off the front of the event list

        set current simulated time to event time

        process the event

        call Check_Wait_Conds

    end /* loop */
    ---

end /* simulation */
---
```

```

/*****
*  Init_simulation                               *
*  -----                                       *
*  Purpose: To initialize the simulation.       *
*  input parameters:                           *
*      EventList                               *
*  return parameters:                          *
*      EventList -- with newly scheduled arrivals. *
*****/

```

```

Init_simulation(EventList)
-----

```

```

*schedule a Job_Arrival event for all sources:
  for all sources do
    --create an even record
    --fill in the event record fields and the event time
    --insert the event into the event list

```

```

end /* Init_simulation */

```

```

/*****
*  Arrival                                       *
*  -----                                       *
*  Purpose: To handle an arrival from a source node. *
*  input parameters:                             *
*      job_id, msg_size, control, current_node,   *
*      user_def_vars, arrival_time, service_time *
*      -- variables with information of the arrival *
*      job.                                       *
*  return parameters:                            *
*      JobRecord -- For the arrived job.         *
*      EventRecord -- Next arrival scheduled.    *
*****/

```

```

Arrival (job_id, msg_size, control, current_node, user_def_vars,
----- JobRecord, EventRecord)

```

```

* Create a job record
  -- fill in all possible fields

```

```

* Schedule next arrival from this source node
  -- generate RN according to RV distribution
  -- create an event record
  -- fill in the event record fields and event_time = t0 + RV

* Call Get_destination(current_node, jobrecord, route_var, destination)

* Call Move_to_destination(destination, jobrecord)

end /* Arrival */

```

```

/*****
*   Get_Destination                               *
*   -----                                       *
*   Purpose: To find the destination node for a given *
*             job at a given current node and return *
*             the destination.                     *
*   input parameters:                             *
*       current_node -- the node the job is at now. *
*       jobrecord    -- the job to be routed.      *
*       route_var    -- the value of route_var     *
*                       indicates the outgoing     *
*                       direction of a job at a   *
*                       multi-branching node, e.g. *
*                       SensorI.                   *
*   return parameters:                             *
*       destination -- the destination node for the *
*                       job.                         *
*                                                     *
*****/

```

```

Get_Destination (current_node, jobrecord, route_var, destination)
-----

```

```

* Find the next node the job should go to according to the current node
  and the model specification

* Return the destination

```



```
end /* Get_Destination */
```

```
/******  
* Move_to_Destination *  
* ----- *  
* Purpose: To move a job to the given destination node.*  
* input parameters: *  
* destination -- the destination node of a job. *  
* JobRecord -- the job to be routed to the *  
* destination node. *  
* return parameters: *  
* none. *  
*****/
```

```
Move_to_Destination (destination, jobrecord)
```

```
-----
```

```
case of destination
```

```
active_queue : arrival_at_active_q(...)  
port(i).msg_in: arrival_at_channel(...)  
port(i).msg_out:Channel_Msg_Read(...)  
decision_node : arrival_at_decision(...)  
delay_node : arrival_at_delay(...)  
fission_node : arrival_at_fission(...)  
fussion_node : arrival_at_fussion(...)  
set_node : arrival-at_set(...)  
sink_node : arrival_at_sink(...)  
split_node : arrival_at_split(...)  
transmission_terminate: arrival_at_transmission_terminate(...)  
wait_node : arrival_at_wait(...)
```

```
end /* case */
```

```
end /* Move_to-Destination */
```

```
/******  
* Arrival_at_Delay *  
* ----- *  
* Purpose: To handle a job arrived at a delay box. *  
*****/
```

```

*   input parameters:                               *
*       JobRecord -- the job arrived at delay box. *
*   return parameters:                               *
*       EventRecord -- event scheduled.             *
*****/

```

```

Arrival_at_delay ( JobRecord, EventRecord )
-----

```

- * Get the delay expression to calculate the delay duration from the user define model structure
- * According to the delay duration, calculate the time when the job should be leaving delay box, i.e. $t = t(\text{current}) + D$
- * Schedule an event job_completion based on t

```

end /* Arrival_at_delay */

```

```

/*****
*   depart_from_delay                               *
*   -----                                         *
*   Purpose: When a job completes its delay at the *
*           delay box, this routine will be called *
*           by the job_completion event to route the *
*           job out to the next node in the net.   *
*   input parameters:                               *
*           JobRecord -- the job completed delay.  *
*   return parameters:                               *
*           none.                                    *
*****/

```

```

depart_from_delay ( JobRecord )
-----

```

- * Call Get_Destination (current_node, JobRecord, route_var, destination)
- * Call Move_to_Destination (destination, jobrecord)

```

end /* depart_from_delay */

```

```

/*****
* Arrival_at_Active_Q
* -----
* Purpose: To handle an arrival at an active queue.
* input parameters:
*     JobRecord -- the job arrived at this node.
* return parameters:
*     EventRecord -- scheduled job completion
*                   event.
*****/

```

Arrival_at_Active_Q (JobRecord, EventRecord)

- * Fill in current_node in jobrecord
- * Generate service time by calling, for example, get_exp-rv(mean)
- * If queue empty

schedule job completion (i.e. add to eventlist at time
t0 + job_service_time)

else

link jobrecord into linked list of jobs waiting for service at
this queue

end /* Arrival_at_Active_Q */

```

/*****
* Depart_from_Active_Q
* -----
* Purpose: To handle a departure from an active
*         queue. This routine will be called
*         directly as the result of a JobRecord
*         being taken off from the front of event
*         list with event being job_completion.
* input parameters:
*     JobRecord -- the job to depart.
*****/

```

```

*   return parameters:
*       EventRecord -- job completion event
*                   scheduled.
*****/

Depart_from_Active_Q ( JobRecord, EventRecord )
-----

* If ( >1 jobs in present queue)

    schedule job_completion for next job in queue waiting for service
    /* assuming FIFO */

/* then route the job just completed at this queue */
* Call Get_Destination(current_node, jobrecord, route_var, destination)

* Call Move_to_Destination(destination, jobrecord)

end /* Depart_from_Active_Q */

/*****
*   Arrival_at_Sink
*   -----
*   Purpose: To handle a job arrived at a sink node.
*   input parameters:
*       JobRecord -- the job arrived at Sink.
*       Job_list  -- list of all jobs in the
*                   system.
*   return parameters:
*       Job_list -- the list of all jobs with
*                   the job at the sink node
*                   being removed.
*****

Arrival_at_Sink ( JobRecord, Job_list )
-----

* Remove the job record from job list

```

* Do garbage collection

end /* arrival_at_sink */

```
/* *****  
 * Arrival_at_Wait *  
 * ----- *  
 * Purpose: Handles a job arrived at a wait node with *  
 * some waiting condition. *  
 * input parameters: *  
 * jobrecord -- the job arrived at the wait node. *  
 * wait_cond -- the condition for the job to wait *  
 * on. *  
 * return parameter: *  
 * jobrecord -- with the wait_cond updated and *  
 * linked to the waiting list for *  
 * that condition. *  
 * *****
```

Arrival_at_Wait (JobRecord, wait_cond)

* Put wait pointer to condition into jobrecord.wait_cond

* Fill in jobrecord.current_node

* According to the wait condition, put the job record on that wait queue

end /* Arrival_at_Wait */

```
/* *****  
 * Arrival_at_Set *  
 * ----- *  
 * Purpose: To set a variable to some value as a *  
 * given job passing through the set node. *  
 * input parameters: *  
 * set_var -- the name of the variable. *  
 * set_val -- the value the variable to be set *  
 * to. *
```

```

*           JobRecord -- the job going through the set *
*                   node. *
*   return parameters: *
*           set_var -- return the variable with value *
*                   set_val. *
*****/

```

```

Arrival_at_Set ( set_var, set_val, JobRecord )
-----

```

- * Set the variable (e.g. job variable, global variable, etc.) to the value (arbitrary expression)
- * Call Get_Destination (current_node, jobrecord, route_var, destination)
- * Call Move_to_Destination(destination, jobrecord)

```

end /* Arrival_at_Set */

```

```

/*****
*   Arrival_at_Decision *
*   ----- *
*   Purpose: To handle an arrival at a decision node. *
*   input parameters: *
*           JobRecord -- the job arrived at this node. *
*   return parameters: *
*           none. *
*****/

```

```

Arrival_at_Decision ( JobRecord )
-----

```

- * Get the control expression to be evaluated of this decision node
 - look at the curret node in the user defined model structure
 - retrieve the expression to be evaluated with current value(s) for the variables in the expression
- * Evaluate branching condition
- * According to the branching condition result, set route_var

```

* Call Get_Destination(current_node, jobrecord, route_var, destination)

* Call Move_to_Destination(destination, jobrecord)

end /* Arrival_at_Decision */

/*****
* Arrival_at_Fission *
* ----- *
* Purpose: To handle a job arrived at a fission node.*
* input parameters: *
* JobRecord -- the job arrived at this node. *
* Job_list -- the list with all jobs. *
* return parameter: *
* Job_list -- with the new duplicated job *
* added to the list. *
*****/

```

```

Arrival_at_Fission ( JobRecord, Job_list )
-----

```

```

* Create a job record, jobrecord1, with duplicated field values
  except parent/child fields and establish parent_child relationship

* Call Get_Destination(current_node, jobrecord, route-var, destination)

* Call Move_to_Destination(destination, jobrecord)

* Call Get_Destination(current_node, jobrecord1, route_var, destination)

* Call Move_to_Destination(destination, jobrecord1)

end /* Arrival_at_Fission */

```

```

/*****
* Arrival_at_Fussion *
* ----- *
* Purpose: To handle a job arrived at a fussion node. *
* input parameters: *
* JobRecord -- the job arrived at this node. *
*****/

```

```

*           Job_list  -- list of all jobs in the net.      *
*   return parameters:                                     *
*           Job_list  -- with all the related jobs        *
*                   but one removed from the list.      *
*****/

```

```

Arrival_at_Fussion ( JobRecord, Job_list )
-----

```

```

* Check for previous arrival of all related jobs

```

```

* If not all arrived

```

```

    add this job to fussion queue

```

```

else

```

```

    remove all related jobs from fussion queue except one

```

```

    call Get_Destination(current_node, jobrecord, route_var, destination)

```

```

    call Move_to_Destination(destination, jobrecord)

```

```

end /* Arrival_at_Fussion */

```

```

/*****
*   Arrival_at_Split                                     *
*   -----                                             *
*   Purpose: To handle a job arrived at a split node. *
*   input parameters:                                   *
*       JobRecord -- the job arrived at Split.        *
*       Job_list  -- the list of all jobs.            *
*   return parameters:                                  *
*       Job_list  -- with the job just created        *
*                   at this split node added to      *
*                   the list.                          *
*****/

```

```

Arrival_at_Split ( JobRecord, Job_list )
-----

```

```

* Create a job record, jobrecord1, with appropriate field values

```



```

* Insert this job into the job list

* Call Get_Destination(current_node, jobrecord, route_var, destination)

* Call Move_to_Destination(destination, jobrecord)

* Call Get_Destination(current_node, jobrecord1, route-var, destination)

* Call Move_to_Destination(destination, jobrecord1)

end /* Arrival_at_Split */

```

```

/*****
* Transmission_terminate *
* ----- *
* Purpose: To terminate the transmission of a message *
* for a given port. *
* input parameters: *
* JobRecord -- the job to be aborted. *
* port_id -- the port the job belongs to. *
* return parameters: *
* JobRecord -- with the new trans_end_time *
* updated. *
*****/

```

```

Transmission_Terminate(jobrecord, port_id)
-----

```

```

* Set the state of the port to IDLE

* Update trans_end_time of the sending port in ChannelRelation

* Get_Destination(current_node, jobrecord, route_var, destination)

* Move_to_Destination(destination, jobrecord)

end /*Transmission-Terminate */

```

```

/*****
* Check_Wait_Conds *
* ----- *
* This routine would be executed after processing *

```

```
*   each event to wake up any jobs which are waiting *  
*   on some condition(s) that have become true now.   *  
*****/
```

Check_Wait_Conds

For every waiting queue

 If ≥ 1 job on the queue and the job's waiting condition is true

 Get_Destination(current_node, JobRecord, route_var, destination)

 Move_to_Destination(destination, JobRecord)

 end /* for loop */

end /* Check_Wait_Conds */

```
/* Channel simulation routines */
```

```
/*  
 * Init_Channel *  
 * ----- *  
 * Purpose: To initialize the ChannelProperties table *  
 * and the channel relation. *  
 * input parameters: *  
 * none. *  
 * return parameters: *  
 * ChannelProperties -- initialized. *  
 * ChannelRelation -- initialized. *  
 */
```

```
Init_Channel (ChannelProperties, ChannelRelation )
```

```
-----  
/* Run automatically before the simulation, after all the data are  
collected from the user about the channel */
```

```
* Read in all field values for ChannelProperties
```

```
* Initialize the ChannelRelation with initial port data (i.e. port  
position and port-to-port delay, etc.)
```

```
end /* Init-Channel */
```

```
/*  
 * Arrival_at_Channel *  
 * ----- *  
 * Purpose: To handle a job entering the channel. *  
 * Note, this routine is to be mapped to the *  
 * Begin Transmission node, i.e. the channel *  
 * input port. *  
 * input parameters: *  
 * JobRecord -- the job arrived at the channel. *  
 * ChannelRelation -- to be updated. *  
 * return parameters: *  
 * ChannelRelation -- updated. *  
 * Event_list -- with events scheduled at this *  
 */
```

```

*                               node added to the list.           *
*****/

```

```

Arrival_at_Channel
-----

```

- * Update ChannelRelation for the port which sent the message into the channel
 - enter new trans_start_time and trans_end_time
 - set the state of the port to TRANSMITTING
- * Update all port locations according to their previous locations and their mobile_speeds if they are mobile ports
- * Schedule possible_succ_transmission events for arrival of end of message transmission at all ports which have a channel_msg_read node or are waiting for a change in the channel status
- * Schedule event for arrival of start of message transmission arrival only at ports which have a job waiting for change of channel status

```

end /* Arrival_at_Channel */

```

```

/*****
* Channel_Status                                           *
* -----                                                 *
* Purpose: Given the port id, returns the current        *
*           Channel status for that port.                 *
* input parameters:                                       *
*   port_id -- the port at which the                      *
*               channel status should be                 *
*               computed.                                 *
* return parameters:                                       *
*   status -- the channel status at that port            *
*             at current time.                            *
*****/

```

```

Channel_Status(port_id, status)
-----

```

```

* Number_of_active_port = 0

* For ( i = 1 to number of ports of this channel) do
  if ( trans_end_time + delay(i,port_id) < current_time)
  then nothing /* no message sent by i currently at port_id */
  else {
    number_of_active_port = +1
    record active port ids
    if capture
      -- check all "active ports"
      -- determine if there exists one dominating port
      then number_of_active_port = 1 /* success */
    }

* If number_of_active_port = 0 then status = idle
  else if number_of_active_port = 1 then status = success
    else status = collision

end /* Channel_status */

```

```

/*****
* Channel_Msg_Read *
* ----- *
* purpose: Reads a message off the channel. *
* Note, this is to be mapped to the message *
* out port, i.e. the channel output port. *
* input parameters: *
* port_id -- the port to read the message. *
* return parameters: *
* message -- the message to be read. *
*****/

```

```

Channel_Msg_Read ( port_id, message )
-----

```

```

* Channel_Status( port_id, status )

```

```
* If ( status = success )  
    read the message off the channel  
    Call Get_Destination (current_node, JobRecord, route_var, destination)  
    Call Move_to_Destination (destination, JobRecord)  
Else /* unsuccess transmission */  
    do nothing  
  
end /* Channel_Msg_Read */
```

REFERENCES

- [1] "The ALOHA System", Proc. FJCC, AFIPS, Houston, TX, Nov. 1970, pp. 281-285.
- [2] J. Browne et al., "A Graphical Programming Language for Computer Network Simulation", *Annual Simulation Conference*, pp. 96-128., 1985
- [3] Consolidated Analysis Centers INC., SIMSCRIPT II.5 Reference Handbook, Santa Monica, CA., 1971.
- [4] K.M. Chandy et al., "Simulation Tools in Performance Evaluation", *Proc. Computer Performance Evaluation Users Group*, San Antonio TX, Nov. 1981.
- [5] O. Dahl and K. Nygaard, "SIMULA - an ALGOL Based Simulation Language", *Comm. ACM*, Vol. 9, No. 9, pp. 671-678.
- [6] C. Date, *An Introduction to Database Systems*, Addison Wesley, Reading, MA, 1980.
- [7] K. Doshi, S. Madala, J. Sinclair, "GIST: A Tool for Specifying Queueing Network Models", Tech. Report TR8511, Department of Electrical Engineering, Rice Univ., May 1985.
- [8] E. Glinert, S. Tanimoto, "PICT: An Interactive Graphical Programming Environment", *IEEE Computer*, Vol. 17, No. 11, pp 7-29.
- [9] F. Hopgood *et al.*, *Introduction to the Graphics Kernel System (GKS)*, Academic Press (NY,NY), 1983.
- [10] IEEE, "Application Briefs", *IEEE Computer Graphics and Applications*, June 1985, pp. 15-17.
- [11] "General Purpose Simulation System V: Operations Manual", SH20-0867-3, 3rd edition, IBM Corporation, Data Processing Division, White Plains, New York, 1971.
- [12] R. Kahn *et al.*, "Advances in Packet Radio Technology", *Proc. IEEE*, Vol. 66, Nov. 1978, pp. 1468-1496.
- [13] H. Kobayashi, *Modeling and Analysis*, Addison-Wesley Publishing, North Reading, MA, 1978.
- [14] L. Kleinrock, *Queueing Systems*, Wiley Interscience, 1976.
- [15] L. Kleinrock and M. Gerla, "Flow Control, a Comparative Survey", *IEEE Trans. on Commun.*, Vol. COM-28, No. 4 (April 1980), pp. 553-575.
- [16] "J. Kurose and S. Salza, "Structure and Internal Operation of the RESQ Language Translator", IBM Proprietary Technical Report, July 1983.
- [17] J.F. Kurose, M. Schwartz and Y. Yemini, "Controlling Time Window Protocols for Time-Constrained Communication in a Multiple Access Environment", *8th Int. Data Communication Symposium*, October 1983. submitted to *IEEE Trans. on Commun.*
- [18] J.F. Kurose and M. Schwartz, "A Family of Window Protocols for Time Constrained Communication in a Multiple Access Environment", *IEEE INFOCOM*, April 1983. submitted to *IEEE Trans. on Commun.*

- [19] J. Kurose, M. Schwartz, Y. Yemini, "Multiple Access Protocols and Time-Constrained Communication", *Computing Surveys*, March 1984.
- [20] J.F. Kurose and C. Shen, "GENESIS: A Graphical Environment for the Modeling and Performance Analysis of Protocols in Multiple Access Networks", submitted to *IEEE International Communication Conference*, June 1986.
- [21] J.F. Kurose and C. Shen, "GENESIS: A Performance Evaluation Environment for Modeling Multiple Access Networks", submitted to *IEEE Journal on Selected Areas in Communications*.
- [22] B. Melamed and R. Morris, "Visual Simulation: the Performance Analysis Workstation", *IEEE Computer*, Vol. 16, No. 8, pp. 87.
- [23] R. Metcalfe and D. Boggs, "Ethernet: Distributed Packet-switching for Local Computer Networks", *Commun. of the ACM*, Vol 19, July 1976, pp. 495-403.
- [24] T. Nishida et al., "PLANS: Modeling and Simulation System for LANs", *Int. Conf. on Modeling Techniques and Tools for Performance Analysis*, (Paris, May 1984), INRIA.
- [25] C. Sauer and K.M. Chandy "Computer System Performance Modeling," Prentice Hall, Englewood Cliffs, NJ, 1981.
- [26] C. Sauer, E.A. MacNair, J.F. Kurose, "The Research Queueing Package version 2: CMS Users Guide", *IBM Res. Rep. RA-139*, Yorktown Heights, NY, April 1982.
- [27] C. Sauer, E.A. MacNair, J.F. Kurose, "The Research Queueing Package version 2: Introduction and Examples", *IBM Res. Rep. RA-138*, Yorktown Heights, NY, April 1982.
- [28] C. Sauer, E. MacNair, J. Kurose, "Queueing Network Simulation of Computer Communication", *IEEE J. on Selected Areas in Communications*, Vol. SAC-2, No. 1 (Jan. 1984), pp. 203-220.
- [29] M. Schwartz and T. Stern, "Routing Techniques Used in Computer Communication Networks", *IEEE Trans. on Communications*, Vol. COM-28, No. 4 (April, 1980), pp. 539-553.
- [30] W. Stallings, "Local Networks", *ACM Computing Surveys*, Vol. 16, No 1. (March 1984), pp 3-42.
- [31] H. Takagi, "Analysis of Throughput and Delay for Single- and Multi-Hop Packet Radio Networks", PhD Thesis, Department of Computer Science, *UCLA Report CSD-830523*, May 1983.
- [32] F. Tobagi, "Multiaccess Protocols in Packet Communication Systems", *IEEE Tans. Communications*, Vol COM-28, pp. 468-488, April 1980.
- [33] S. Tripathi et al., "STEP-1: A User Friendly Performance Analysis Tool", *Proc. Int. Conference on Modeling Techniques and Tools for Perf. Analysis*, INRIA, Paris 1984.
- [34] R. Willis, W. Austell, "GMSS: Graphical Modeling and Simulation System", *Proc. 16th Simulation Symp.*, (Tampa, Fla., Mar. 1983), pp. 137-160.
- [35] Y. Yamamoto, M. Lenngren, "Graphical Model Building System", *Proc. 16th Simulation Symp.*, (Tampa, Fla., Mar. 1983), pp. 161-175.