NASA Contractor Report 191545

Simulator for Heterogeneous Dataflow Architectures

Mahyar R. Malekpour

Lockheed Engineering & Sciences Company
Hampton, Virginia

Contract NAS1-19000

September 1993

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-0001

Table of Contents

Acronyms

ADM	Advanced Development Model
AIE	ATAMM Integrated Environment
AMG	Algorithm Marked Graph
AMOS	ATAMM Multicomputer Operating System
ATAMM	Algorithm To Architecture Mapping Model
CMG	Computational Marked Graph
ENS	Ensemble
FDT	Fire/Data/Time
FU	Functional Units
GRF	Graph
GVSC	Generic VHSIC Spaceborne Computer
NMG	Node Marked Graph
OOP	Object-Oriented Programs
PI-Bus	Parallel Interprocessor Bus
SGP	Single Graph Play
TBI	Time Between successive Inputs
TBIO	Time Between Input and corresponding Output
TBO	Time Between Outputs
TCE	Total Computing Effort
TGP	Total Graph Play
VHSIC	Very High Speed Integrated Circuit

1. Introduction

The Algorithm To Architecture Mapping Model (ATAMM) is a Petri net based model capable of describing the periodic execution of large-grained, data-independent algorithm graphs on multiprocessor architectures. ATAMM provides a description of the data flow and control flow necessary to provide for the predictable execution of an algorithm in real-time.

The objective of this research is to develop a software simulator capable of simulating the execution of a graph on a given system under the ATAMM rules. The purpose of the simulator is to enable a study of the behavior and performance of both heterogeneous and homogeneous multicomputer dataflow systems prior to the availability of hardware prototypes. This simulator is able to assist with the development of ATAMM-based architectures and the investigation of theories concerning the ATAMM model. This simulator is user-friendly and flexible to permit examining different attributes of a generic system. The simulator also provides the means to identify an architecture by specifying different parameters of the system in order to evaluate the periodic execution of an algorithm on a given hardware system. Evaluation of the simulator is conducted through several case studies.

Section 2 of this report is an overview of ATAMM. Performance measures are also defined in Section 2. The implementation issues of this new simulator, which will hereafter be referred to as the Heterogeneous ATAMM Simulator or simply as the Simulator, are discussed in Section 3. The design and development of the Simulator are presented in Section 4. Case studies and simulation results of example algorithm graphs are presented in Section 5. This report concludes, Section 6, with a discussion of ongoing and future research to expand the model to a broader class of multiprocessor architectures.

The use of brand names is for completeness and does not imply NASA endorsement.

2. Overview of ATAMM

2.1 Model Components

ATAMM is designed to model the control, scheduling, and communication issues for computational algorithms accepting periodic input data and generating periodic output data [1]. ATAMM models data-driven real-time algorithms which may be represented by data-independent directed graphs. The nodes of the graph are assumed to be of sufficient computational complexity to warrant parallel execution. The target hardware system has previously consisted of a set of homogeneous processors. This Simulator, however, is intended to support the extension of ATAMM to heterogeneous processors.

The model consists of a set of Petri net marked graphs [2, 3, 4] which combine the functions of an algorithm with the necessary computing activities. The Algorithm Marked Graph (AMG), the Node Marked Graph (NMG), and the Computational Marked Graph (CMG) constitute the three components of the ATAMM. The Algorithm Marked Graph (AMG) represents a specific decomposition of the functional computation requirements. The AMG, as illustrated by the example in Figure 1, uses nodes (circles) to represent blocks of code or processes which are to be executed and edges (directed line segments) to represent data dependencies between the nodes. Each AMG node is executed to completion before another node may be scheduled on the same processor. A token (solid dot) on an edge represents the presence of a single data packet. All edges may have a pool of buffers and can accommodate more than one token at a time. A node consumes one token from each of its input edges when it fires (begins execution) and deposits one token on each of its output edges when it completes execution. Source and sink transitions for input and output signals are represented as rectangles.

The Node Marked Graph (NMG), illustrated in Figure 2, is a representation of the execution of an AMG node by a processor. Three primary activities associated with execution of an AMG node, reading of input data (R), processing of input data to generate output data (P),

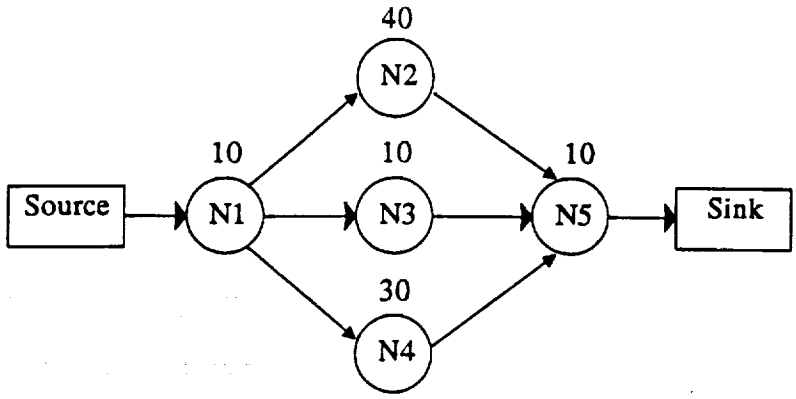

Figure 1. An example Algorithm Marked Graph.

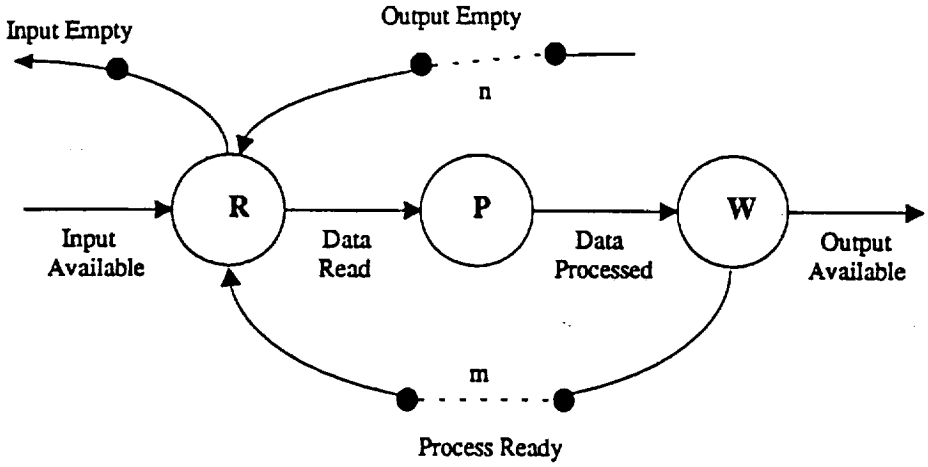

Figure 2. An example Node Marked Graph.

and writing of output data (W), are incorporated in the NMG. A recent enhancement of the model [4, 5] allows m tokens on the *Process Ready* edge, which permits m simultaneous instantiations of the node to be executed in parallel on different processors with different data packets. The n tokens on the *Output Empty* edge indicate that the predecessor AMG node can be instantiated up to n times before an output is consumed by the successor node. The value of n is always greater than or equal to m. The values of n and m are determined by a graph analysis

procedure and are typically different for each AMG node. Tokens on the *Output Available* edge indicate the presence of data on the edge.

The Computational Marked Graph (CMG), illustrated in Figure 3 (for the AMG of Figure 1 and for the simple case of m = 1 for all nodes) is constructed by replacing each AMG node with its NMG and replacing each AMG edge with an edge pair, consisting of a forward directed edge representing dataflow and a backward directed edge representing control flow. As both a graphical and mathematical model, the CMG is useful for determining the performance bounds as well as the data and control flow required for a hardware implementation.

Two types of concurrency are possible when executing an algorithm decomposition as specified by the CMG. First, several nodes of the dataflow graph without data interdependency may be simultaneously performed on the same data packet. This is referred to as parallel concurrency because it is the result of inherent parallelism in the graph [6]. The amount of parallel concurrency depends on the number of parallel paths in the algorithm decomposition as well as the number of available resources. Second, several nodes of the dataflow graph may be simultaneously performed on different data packets. This happens when new data packets are accepted for execution before the completion of computation of previous data packets. This simultaneous processing of different data packets is referred to as pipeline concurrency [6]. This type of concurrency has a direct effect on throughput. The amount of pipeline concurrency depends on the number of available resources as well as the structure of the AMG.

2.2 Performance Measures and Bounds

The two primary performance measures for a graph are the steady-state Time Between Outputs (TBO) and the Time Between Input and corresponding Output (TBIO). TBO is the elapsed computing time between successive algorithm outputs. Therefore, the inverse of the steady-state value of TBO is a measure of throughput in data packets per unit time. The TBO lower bound, TBO_{lb}, and hence the upper bound on throughput, is determined by the algorithm

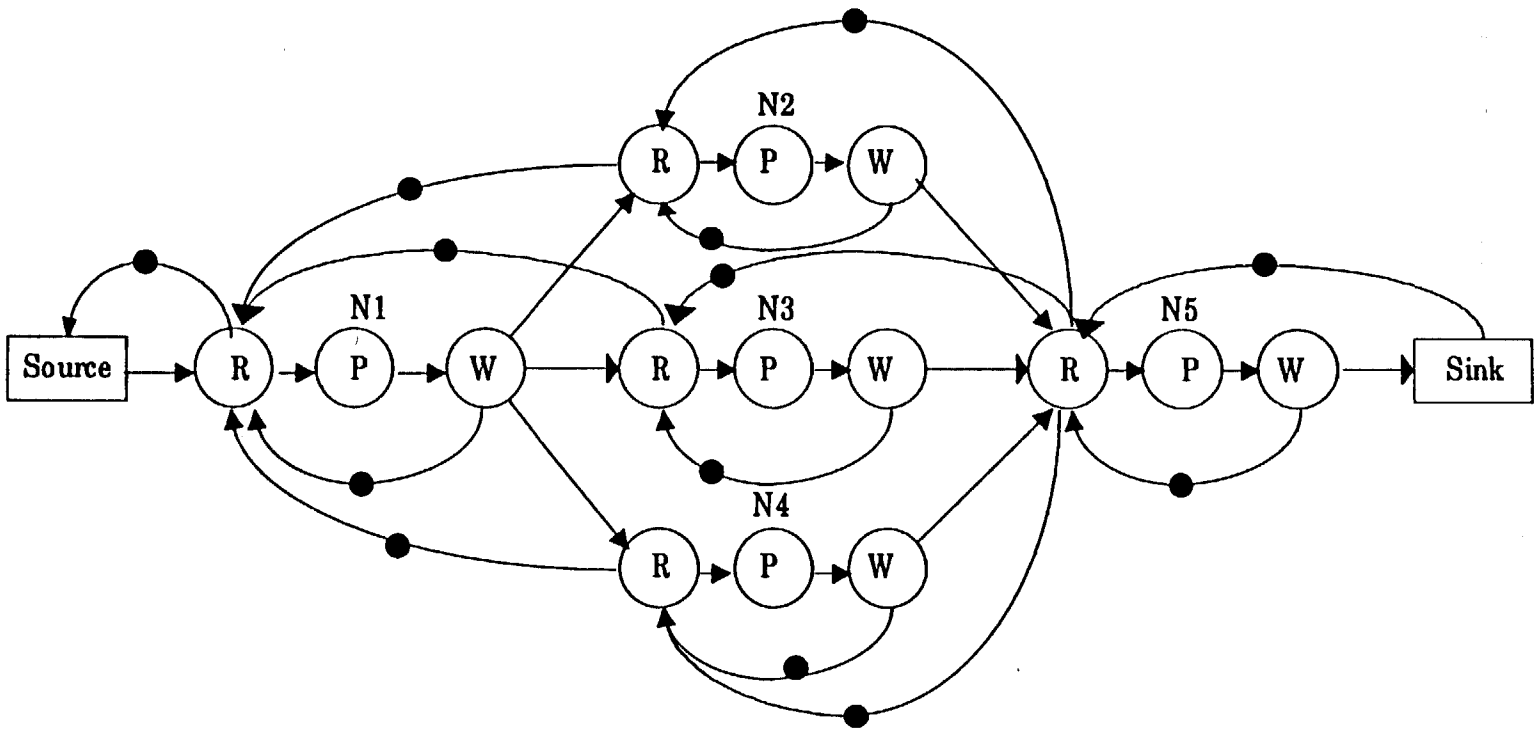

Figure 3. An example Computational Marked Graph.

graph and the number of available resources. The algorithm imposed TBO_{lb} is determined by the largest time per token of all directed circuits in the CMG [6]. In graphs with recurrent circuits, TBO_{lb} is determined by the time per token of the largest recurrent circuit in the CMG. The second bound on TBO is imposed by the availability of resources [6] and is given by the ratio of TCE over R where TCE (Total Computing Effort) is the summation of all the node latencies of a CMG and is the time required for all graph nodes to execute a single data packet. R is the number of resources. For instance, the TBO of the AMG of Figure 1, which has no recurrent circuit, is limited only by the number of available resources. TBIO is defined as graph latency, which is the time for a single data packet to progress from source to sink. The algorithm-imposed lower bound, $TBIO_{lb}$, is determined by the critical path from source to sink. However, the TBIO is a function of TBO and is determined by analyzing the algorithm graph and considering the number of resources.

To achieve a desired TBO for a given algorithm graph, ATAMM requires that the input data to the algorithm graph be supplied at the steady-state TBO rate. Therefore, the injection rate, defined as the Time Between successive Inputs (TBI), and TBO are synonymous at the steady-state and are used interchangeably.

Other performance measures are speedup and resource utilization. Speedup for a homogeneous processor system is defined as the ratio of TCE over TBO. Resource utilization for a homogeneous processor system [6], U, is defined by

$$U = \frac{TCE}{TBO * R} ,$$

where R is the number of available resources, and

$$TBO \geq TCE / R, \qquad \text{for } 0 \leq U \leq 1.$$

The speedup and resource utilization may similarly be defined for the heterogeneous processor configurations.

2.3 Control Edges

A control edge is an AMG edge which imposes an artificial data dependency between two AMG nodes [6]. The control edges are used to either alter node schedules to eliminate needless concurrency or to improve resource utilization.

3. Simulator Implementation Issues

3.1 Target Hardware Architecture

The generic heterogeneous architecture considered is displayed in Figure 4. This generic heterogeneous architecture consists of a number of processor groups that in turn are composed of a number of resources or functional units (FU), which are the actual processing units, and a number of local networks. Although the functional units and the local networks within each processor group are assumed to be homogeneous, the different processor groups are not required to have similar characteristics. In other words, a heterogeneous system is realized by groups of processors with different characteristics that communicate with each other over the global network.

The Advanced Development Model (ADM) [7, 8] and the Generic VHSIC Spaceborne Computer (GVSC) [8] are typical architectures which have been the primary targets of ATAMM implementations. These systems consist of four identical MIL-STD-1750A functional units that communicate over a Parallel Interprocessor bus (PI-bus) , as shown in Figure 5, and a MIL-STD-1553B communication module that is also connected to the PI-bus and serves as the front-end of the system. The 1553B is essentially a 1750A with less memory and the 1553B interface. These are examples of heterogeneous systems with two groups of processors where one group has four resources (four 1750As) while the other has only one resource (one 1553B) and they communicate over a global network, the PI-bus. However, previous ATAMM implementations on these hardware systems modeled only the behavior of the homogeneous set [9] of 1750A processors. The new simulator described herein could support the modeling of the more general heterogeneous architecture.

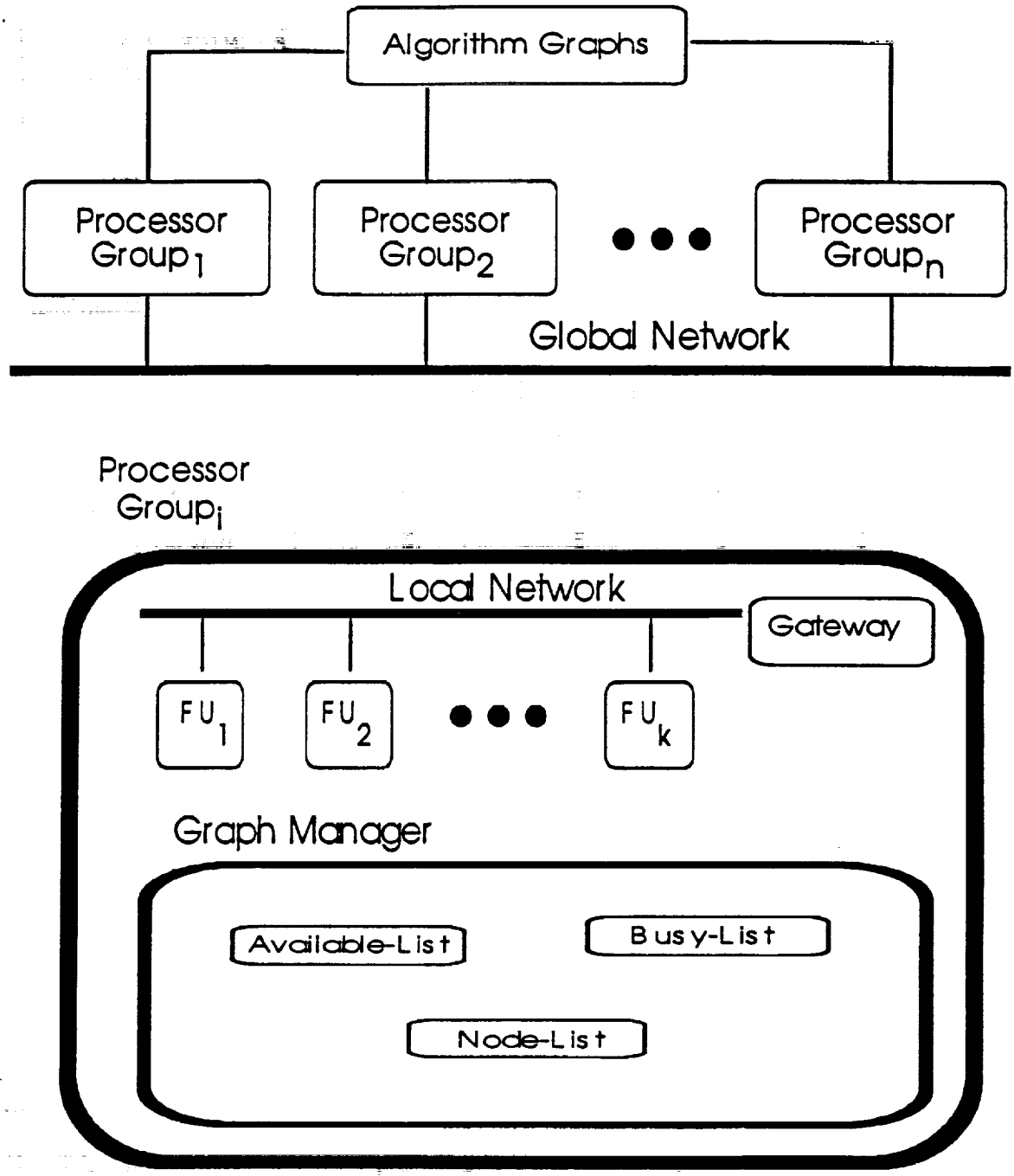

Figure 4. Architecture modeled by the Heterogeneous Simulator.

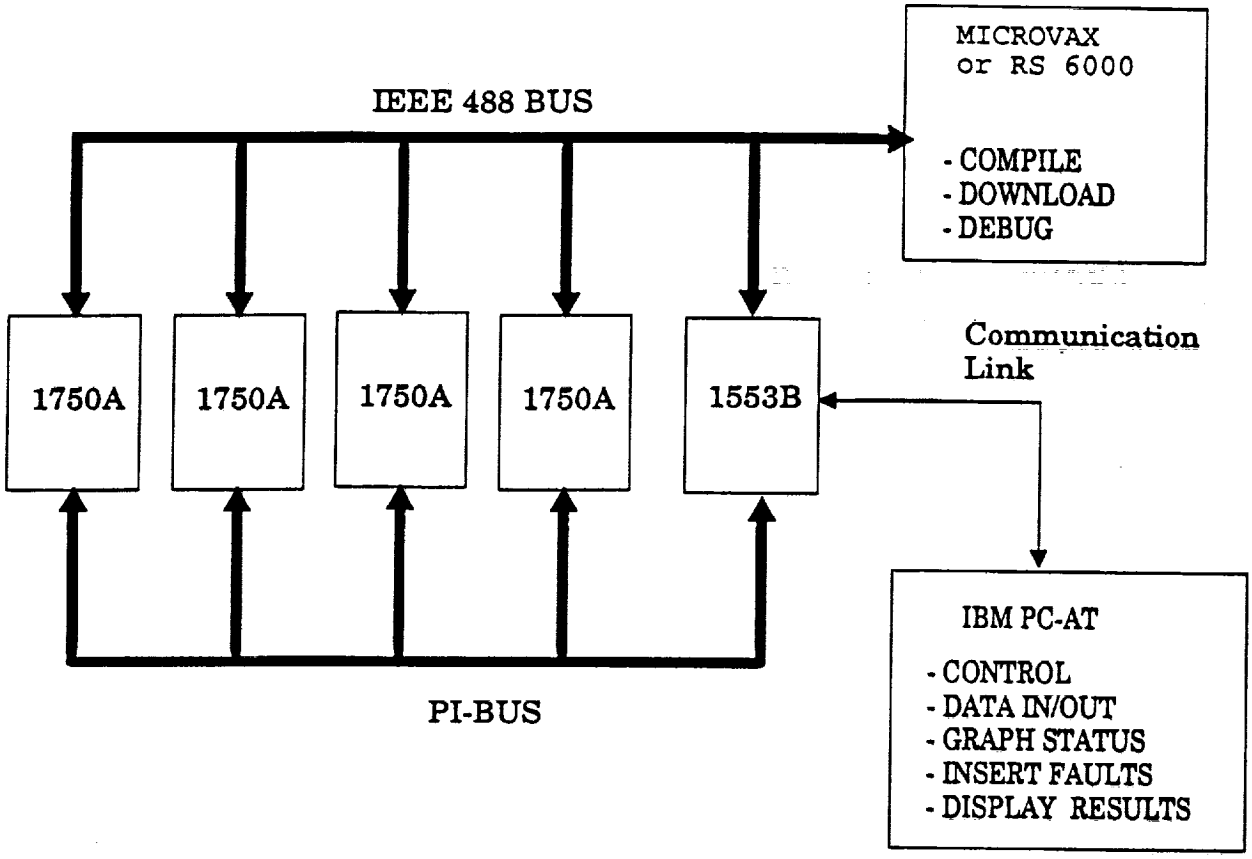

Figure 5. Layout of the ADM and GVSC systems.

3.2 Implementing ATAMM

Systems implementing the ATAMM consist of four logical components: the graph manager, the global memory, a set of functional units, and the communication bus [9]. The graph manager is responsible for ensuring that the overall system operates according to the ATAMM rules. The functional unit is the logical component that executes all three node marked graph (NMG) transitions of each algorithm operation. When a read transition of the CMG graph is enabled, the graph manager assigns a functional unit from the list of available functional units to execute the corresponding algorithm node. If there are additional enabled nodes, the graph manager assigns them, according to priority, to the subsequent resources in the available list. The

graph manager updates the marking of the CMG using status information reported by the functional units. The input and output data corresponding to each AMG node are stored in the global memory. In the context of ATAMM, the memory is considered to be logically global to all functional units. However, in a real system, the global memory may be either centralized or distributed. The functional unit communicates with the graph manager to update the status of the CMG, and with the global memory to read and write data. The communications between the graph manager, the global memory, and functional units are asynchronous and are carried out by means of a communication bus. To synchronize movement of tokens in the CMG and to arbitrate among different functional units, it is assumed that only one functional unit communicates with the graph manager at any one time. This is accomplished by the means of a semaphore. Therefore, the functional unit that possesses the semaphore has control of the communication bus and can communicate with the graph manager and update the status of the CMG. In this regard, the communication bus and semaphore are often used interchangeably.

Thus far, ATAMM implementations have only considered systems with a single semaphore and a single communication bus. One of the purposes of this Simulator is to explore systems with multiple semaphores. In order to ensure that all functional units have an identical copy of the graph data structure, a functional unit grabs the semaphore before changing the graph data structure. In a distributed system, the updated graph data structure is transmitted to all functional units by a broadcast, and only then does the functional unit release the semaphore for other communications.

The graph manager and global memory may be distributed among all the functional units. This distribution of activities has the advantage of increasing the number of functional units in the system and at the same time improving the potential for achieving a higher degree of fault tolerance to processor failure. Also, a distributed global memory eliminates the need for shared memory among functional units.

The integration of the graph manager with the operating system constitutes the ATAMM Multicomputer Operating System (AMOS). The resource list, global memory, and the algorithm

marked graph provide the necessary support to AMOS. An AMOS controlled architecture consisting of personal computers has been developed and tested to validate the ATAMM rules [10, 11]. In this testbed, a centralized graph manager and centralized global memory are utilized. Other testbeds with increased functionality, the ADM and the GVSC, utilize a distributed graph manager and distributed global memory.

3.3 Generic State Diagram of the AMOS

The generic state diagram of the AMOS is shown in Figure 6. The AMOS is composed of six states: Idle, Reading, Processing, Writing, Grab-Semaphore, and Graph-Manager. Other implementations of ATAMM have included other states such as Testing [7, 8]. Initially, all functional units start in the Idle State. A functional unit remains in this state until either its identification number (ID) appears at the top of the resource list, which is a First-In-First-Out list of available functional units, or it receives a message indicating that a node has been assigned to it by another functional unit acting as graph manager. When idle with its ID at the top of the resource list, the functional unit monitors the status of the CMG until a read transition of an algorithm node becomes enabled. Once an enabled read node is identified, the functional unit attempts to acquire the semaphore which makes it the active graph manager of the system. It then assigns a node to itself, consumes one token from each input edge of the algorithm node, updates the CMG marking, and removes itself from the available list.

Before progressing to the next state, Reading, the functional unit examines the algorithm graph and assigns other enabled nodes to the subsequent functional units in the available list. It notifies other functional units via fire-messages, updates the CMG accordingly, broadcasts the updated graph data structure, and then releases the semaphore. This broadcast is termed a "Fire" broadcast. Assigning other enabled nodes to idle functional units while holding the semaphore, is an enhancement to the GVSC AMOS that reduces the communication overhead.

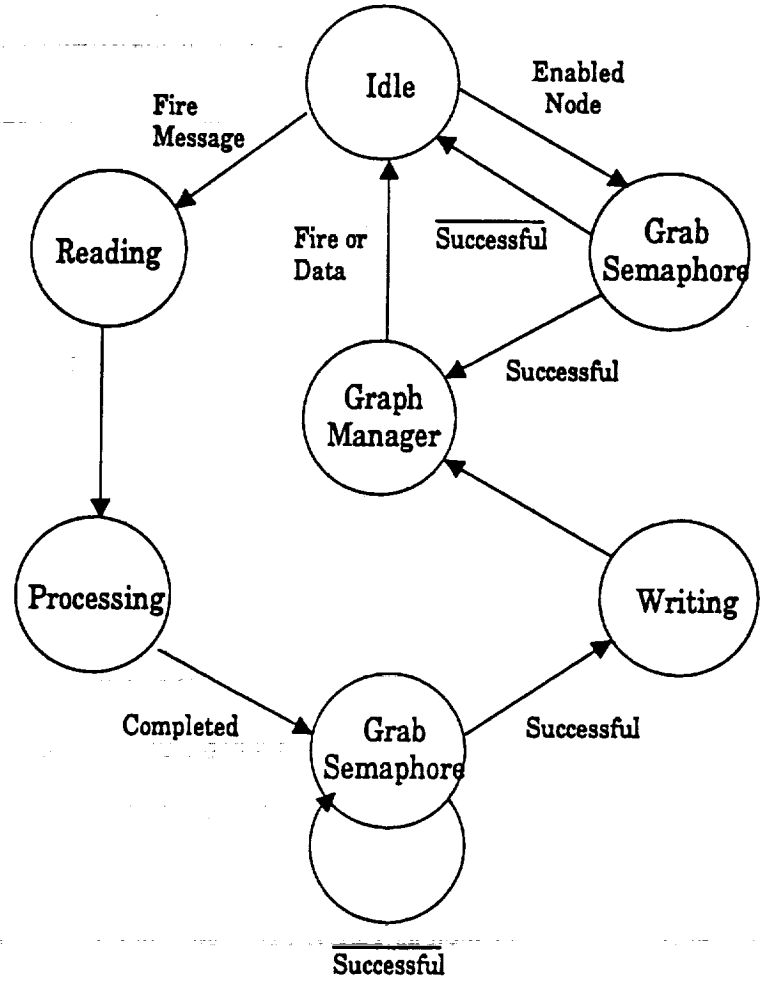

Figure 6. AMOS state diagram.

The "Fire" broadcast contains the updated version of the CMG, the updated resource list, and the ID of the functional units processing the AMG nodes. This broadcast, as well as the other broadcast discussed next, provide the status information necessary for the graph manager to maintain the status of the CMG. When the graph manager is distributed, this communication is especially important to ensure that all individual graph managers contain the same CMG marking.

Upon detecting a fire message in the Idle state, the functional unit transits to the Reading State where it reads the input data in preparation for node execution. The functional unit then migrates to the Processing State where it performs the task represented by the algorithm node. The functional unit remains in the Processing State until the node operation is complete. Then,

the functional unit attempts to undergo another state transition to the Writing State by grabbing the semaphore. In the Writing State it updates the CMG, writes the output data, and broadcasts the updated information to other functional units. This broadcast, termed a "Data" broadcast, provides the updated CMG and the output data of the node to the other functional units. The functional unit then goes to the Graph-Manager State. Now that the functional unit holds the semaphore and is the active graph manager, it attempts to fire as many nodes as possible prior to releasing the semaphore. Since the operation of the system is asynchronous, the graph manager must generally be interrupt driven.

The CMG and resource list in the global memory of a functional unit can be updated while in any state by "Fire" or "Data" broadcasts from other functional units. The "Fire" and "Data" broadcasts not only provide the communication necessary for the integrity of overall system operation, but also the means to analyze the system performance. By labeling, time tagging, and storing information about each broadcast, such as the event (Fire and Data), the node number, and functional unit ID, the token movement within the CMG, as well as functional unit activity can be reconstructed. Other measurements such as TBIO, TBO, and functional unit utilization and concurrency may also be extracted.

3.4 Event-Driven

The previous ATAMM simulator [9] was clock-driven in the sense that the system-clock of the simulator was incremented by one tick at a time. Simulation of algorithm graphs proved to be slow and time consuming. To speed up the simulation process, the system-clock of the Simulator must be incremented by more than one tick without violating the timing constraints of the system. Since the Simulator has the full knowledge of the overall system, it can determine the exact time of occurrence of the next event and thus increment the system-clock accordingly. In this regard, the Simulator defined herein is event-driven. Since, in general, the next event will take place in the time interval of greater than or equal to one system-clock tick, the event-driven Simulator is expected to be considerably faster.

3.5 Simulation of Graphs with Variable Node Latencies

The previous ATAMM simulator [9] simulated graphs with fixed node latencies. Since algorithm graphs representing real applications may not have fixed latencies, it is desirable to be able to simulate graphs with variable node latencies. This is accomplished by representing the timing latency of AMG nodes by statistical functions. The Simulator then determines the actual latency of an AMG node during the simulation process, for every input data packet, by executing the appropriate statistical function representing the AMG node. When the AMG nodes have variable latencies and upon multiple instantiations of nodes, it is possible that the data packets produced by the nodes may arrive out of order. To enforce firing of AMG nodes at the proper time with the appropriate data packet, the data packets are tagged to guarantee correctness of the CMG marking.

Specific statistical functions are included in the Simulator and additional functions may be inserted. The *Delta* function represents the fixed node latency and is assumed to be a positive value. Using the Delta function, the Simulator defaults to the fixed node latency case. The *Uniform Distribution* function requires a lower bound and an upper bound. The *Gaussian* function requires a mean and a standard deviation. The *Discrete* function requires an input file where the discrete values for each input data packet are stored. The *Exponential* function requires a mean value.

3.6 Simulation of Graphs with Static Node to Processor Assignments

The previous implementations of the ATAMM targeted homogeneous architectures [3, 7, 8] where all nodes of the algorithm graphs are mapped to and executed on all identical functional units of a system. However, it may not be practical or necessary to always have a fully redundant system. In some real systems, memory constraint is a limiting factor. In other systems, functional units may have different characteristics from one processor group to another. By partitioning the algorithm graph into groups of nodes and assigning each group to a different processor group, the

same performance as the fully redundant system (a single processor group) may be achieved. A proper partitioning of the graph can minimize interprocessor communication overhead and increase throughput.

Analysis of the AMG reveals that it is often possible to group some of the nodes into separate sets and statically preassign each set to different processor groups to get equivalent performance. In the static assignment of nodes to processor groups, execution of the sets are assumed to be confined to the functional units to which they are assigned. However, in a fully redundant system where all nodes are assigned to all functional units of a single processor group, these sets may appear as patterns that migrate from processor to processor.

To accommodate for the static assignment of nodes to processor groups, this Simulator is designed so that each processor group is independent of other groups. The assigned nodes are encapsulated within each processor group and are internally managed by the group.

3.7 Simulation of Multiple Graphs

While simulating multiple independent graphs, it is often necessary to phase the graphs with respect to one another and to simulate them in a predefined sequence. The phasing and sequencing of algorithm graphs requires certain dependencies among them. These dependencies are imposed by the introduction of control edges that connect the sources of different graphs together. However, due to the nature of the phasing and sequencing problems, these control edges must be dealt with separately in the Simulator. To handle these control edges, the Simulator starts the phasing process of a source as soon as an input control edge becomes active. This corresponds to performing an OR operation on the control edges. The Simulator then fires the source after the specified delay interval.

3.8 Graph Entry, Simulator, Analysis, and AIE Tools

The relationship of the Simulator with the other ATAMM tools is shown in Figure 7. As shown in the figure, the input to the Simulator is a graph (GRF) file; graph files have ".grf"

extensions. The Simulator output is a Fire/Data/Time (FDT) file; FDT files have ".fdt" extensions. The GRF file contains the algorithm marked graph and the setup information about the Simulator, e.g., the number of groups of processors and number of functional units in each group type. The FDT file is a collection of time-tagged events which provide a means of evaluating the results of algorithm graph execution. Basic information in the FDT file include the time of occurrence of each event, name of the event, node identifier, node color, and functional unit identifier. The format of the GRF and FDT files are discussed in Section 4.10. The GRF file is the output of the Graph Entry software tool developed to draw a graph and define attributes of the nodes and the edges. The FDT file serves as the input to the Analysis Tool [12] which graphically displays algorithm and resource activities and provides automatic and user-interactive performance assessment. To smooth out the transition of the Graph Entry output into the Simulator and the Simulator output into the Analysis Tool, an ATAMM Integrated Environment (AIE) was proposed to integrate these ATAMM tools.

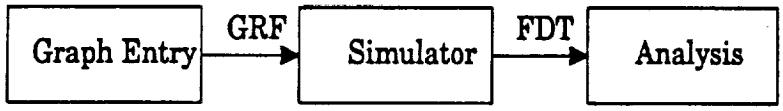

Figure 7. Flow of information among the ATAMM tools.

4. Simulator Design and Development

The development of the Simulator is presented in this section. This Simulator allows the study of the behavior of algorithms in heterogeneous dataflow architectures operating in real-time based on ATAMM. The Simulator permits an architecture-independent study of behavior and performance of a system prior to the availability of a hardware prototype.

4.1 Object-Oriented Programming

Object-oriented programming lends itself to modeling different parts of a complex entity and the relationship among its parts. The objects can be defined and developed separately to ensure privacy of data, reusability, and readability. This also makes maintenance and debugging more manageable and systematic." Further discussions of OOP are provided in Appendix F and in Reference [9].

4.2 Programming Environment and Language

The implementation of the Simulator requires a powerful programming language and software environment. The Simulator is written in the C++ programming language. The main reasons are: 1) it is an object-oriented language with multiple inheritance and thus is a good system programming language; 2) it provides good data structures, control flow primitives, and a rich set of operators; and 3) it is compatible with Microsoft Windows[1]. The Simulator is developed in the Microsoft Windows environment because of its object-oriented programming capabilities including message passing and a vast library of graphics routines, especially the windowing capabilities. Other Microsoft Windows environment features include the capability to run more than one application in parallel, permitting the user to run more than one instance of the Simulator at the same time. This provides a means to simulate and compare two or more

1 Microsoft Windows is a trade mark of Microsoft Corporation.

simulations simultaneously. As another example, the Simulator, the Graph Entry, and the Analysis tools can be running concurrently allowing an easier transition between them.

The objects are defined and developed separately to ensure privacy of data, reusability, and readability. This makes maintenance and debugging more manageable and systematic [9]. Every object that directly interacts with the user has its own independent window which allows the display of different windows to be viewed at the same time.

4.3 Objects and their Relationships

The main logical components or objects of the Simulator are, in part, a result of the ATAMM. Since the ATAMM is a set of rules by which an algorithm graph can be mapped to an architecture, the three main classes of objects are Graph-Manager, Graph, and Processor-Group. The Processor-Group object consists of a set of functional units and, hence, the FU-List object and the Functional Unit object (within the FU-List object) are introduced. Any system has some means of communication among its components; thus the Network object evolved. A management mechanism for arbitration among these objects is provided by the Simulator-Kernel object [9]. Interconnection among these and other entities is portrayed in Figure 8.

4.4 Simulator-Kernel

The Simulator-Kernel provides, manages, and simulates the multitasking environment where the functional units can operate without conflict. This object is the operating system for the Simulator and the heart of this software. The arbitration among different objects is enforced in a non-preemptive manner, where every object is given enough time to accomplish its task. This is easily realized by employing object-oriented programming methodology [9].

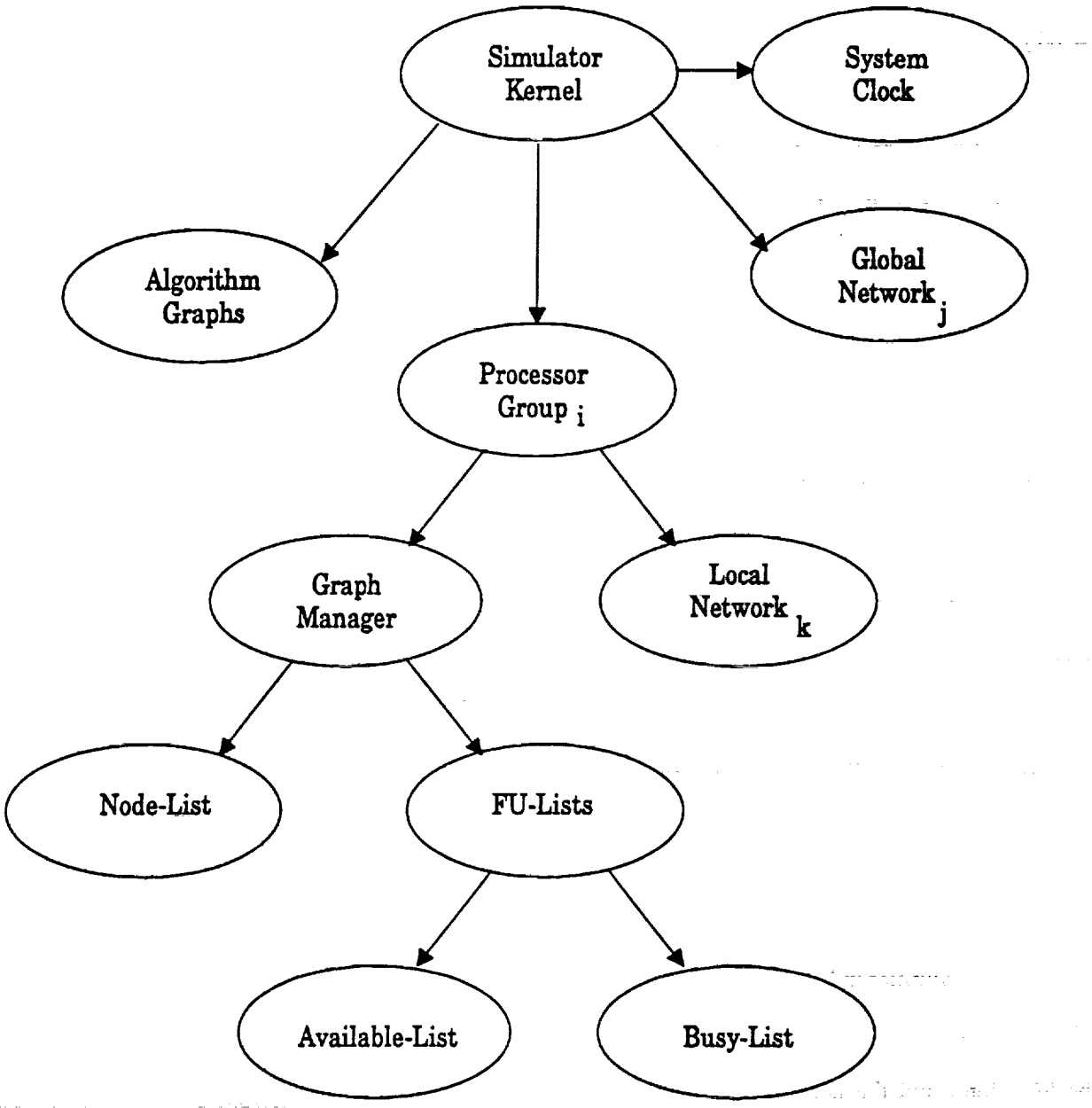

Figure 8. Interconnection of objects.

The Simulator-Kernel object has a number of child objects including Processor-Group, Network, Algorithm Graph, and a System-Clock object. The Simulator-Kernel passes full control over the system to a constituent, specifically to a Processor or to a Network object, and by doing so, suspends itself. Upon completion of its task, the target object returns control back to the Simulator-Kernel along with the anticipated time of the next event in that object. Transfer of control is accomplished through the message passing capability of object-oriented programming. Upon execution of all objects, the Simulator-Kernel updates the System-Clock appropriately to indicate the time of occurrence of the next event in the entire system. Since the Simulator has the full knowledge of the system, it is aware of the timing and nature of the next event. If, however, the time of occurrence of the next event is beyond the upper bound of all events, the Simulator stops the simulation process and provides an error message with indications of the probable causes. This process continues for all objects, in an orderly fashion, until simulation of the graph is complete.

The order in which the objects are invoked is as follows. First, the Processor-Group object, described in the following section, is invoked. It then passes control to the Graph Manager and, subsequently, to the Functional Units via the FU-Lists object. Second, the Network object is invoked to carry out its communication task. The Network object, described in Section 4.7, in turn, passes control to its child objects. The Processor-Group and the Network objects have the same behavior as the Simulator-Kernel toward their constituents. Finally, the System-Clock is appropriately updated. The hierarchy of passing control to the lowest level objects, child objects, is also portrayed in Figure 8.

Thus far, the functionality of the Simulator-Kernel from an internal information viewpoint was described. Another functional aspect of this object is its central role with respect to user interactions. The Simulator-Kernel object and all other objects that require user interactions have their own independent windows through which information may be passed and displayed. For these objects, the terms object and window are used interchangeably.

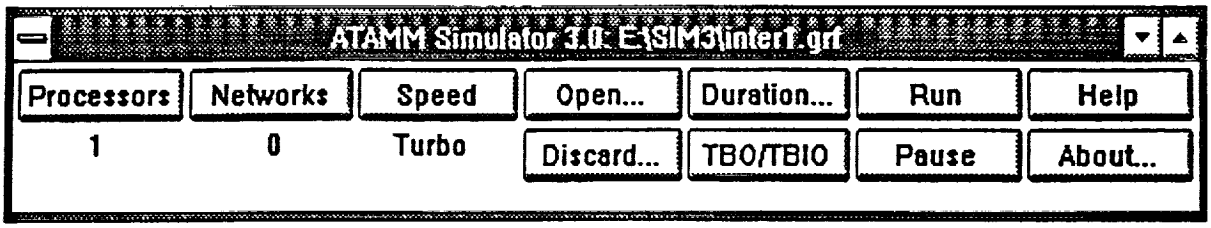

Figure 9. Simulator-Kernel.

For user interactions, the Simulator-Kernel provides a set of push buttons in its window, Figure 9. Some of these push buttons contain a sublayer of selections. The top level selections are for informative purposes while the sublayer selections perform an operation. For example, the second layer of the "Processors" and "Networks" push buttons are the "+" and "-" push buttons that allow the user to increase and decrease the number of these objects, respectively. The speed of the simulation can be adjusted through the "Speed" button to turbo, fast, medium, or slow. The "Open..." button allows the user to open a GRF file and to load the algorithm marked graphs for simulation. The "Discard..." button lets the user specify the number of initial data packets that are to be discarded. The number of discarded data packets corresponds to the data packets prior to reaching the steady state. This number is important in calculating the TBO, TBIO, ensemble TBO, and ensemble TBIO points where the ensemble values are defined as the average values. The duration of the simulation process, the "Duration..." button, can be defined by specifying the number of data packets. The "TBO/TBIO" and "Ensemble" toggle key let the user set up the Simulator for calculating the TBO and TBIO points or the ensemble TBO and ensemble TBIO points. The "Run" and "Stop" toggle buttons allow the user to initiate and terminate the simulation process. When calculating "TBO/TBIO" points, the Simulator prompts for an output FDT file name. When calculating "Ensemble" TBO and TBIO points, the Simulator prompts for the number of ensemble points desired. The "Pause" and "Resume" toggle buttons pause and resume the simulation process, respectively. All windows have a help option where the "Help" buttons invoke the appropriate help files for specific guidance concerning window functions. The

"About" button invokes the signature and displays the general information about the Simulator. This Simulator only operates in the simplex mode.

The Simulator keeps track of clock ticks, number of events, and number of data packets into and out of the graph. It also reports the current status of these activities for user's information upon receiving control of the system via the "System" window. The speed of simulation may be adjusted to turbo, fast, medium, or slow at any time. This provision is provided for animation purposes where the simulation of the graph is carried out at the desired pace. Since this window is the heart of this software, existence of other windows depend on its existence, i.e., closing this window results in termination of the Simulator.

4.5 Algorithm Graphs

The Algorithm Graph object of the Simulator is a set of objects that are connected together by a set of linked lists. The objects that constitute the Algorithm Graph objects are the nodes and the edges. The node object has three variations and represent the nodes, the sources, or the sinks of the algorithm graphs. The edge object has two variations and represent the data or the control edges of the algorithm graphs. These objects and their interrelations represent the algorithm marked graphs. The input algorithm marked graph files provided by the Graph Entry tool, discussed in Section 4.10, conveys the necessary information about these objects.

When loading an input file, the Algorithm Graph object scans the input file and upon detecting a node or an edge, creates a new instance of the appropriate object and sends a message to the object to read its own data and initialize itself. The Algorithm Graph object then inserts the object into the proper linked lists. The linked lists that represent the algorithm graph are a linked list of node and a linked list of edge objects. Each node object has, in turn, two linked lists of edge objects, one for the input edges and the other for the output edges. Each edge object has two linked lists of edge objects, one for the output edges of its initial node and the other for the input edges of its terminal node. The source object has two additional linked lists of edges, one for the input control edges and the other for the output control edges of the source. These

control edges that connect the sources of the algorithm graphs together are for phasing and sequencing purposes and require special treatment by the Simulator. The data structure of the algorithm graphs, as portrayed by the Algorithm Graph object, is depicted in Figure 10.

4.6 Processor-Group

To model and simulate a heterogeneous architecture, the Processor-Group object is designed to represent a generic system where different attributes of the system can be tailored to match a particular architecture. Since every Processor-Group object represents a homogeneous system, two or more of these objects characterize a heterogeneous system. In a heterogeneous system, different Processor-Groups may have different characteristics, e.g., number of functional units, test time, and speeds; but all functional units within a Processor-Group object share similar characteristics. The Functional Unit object is designed so that it can undertake any or all tasks represented by the input AMG. In this regard, the sources, the sinks, and the nodes of the AMG are treated equally. In this Simulator, the number of Processor-Groups, Functional Units, and Networks are not limited by any upper bound, but by the availability of memory.

The objects that constitute the Processor-Group object and their relationships are portrayed in Figure 8. The Processor-Group object treats its constituents in the same manner as its parent object, the Simulator-Kernel. The Processor-Group passes control to the Graph-Manager object which, in turn, passes control to the Functional Units (within FU lists) to carry out the execution of the AMG nodes assigned to this Processor-Group.

Through Processor-Group's window, the number of Functional Units can be specified to match a particular architecture such as that shown in Figure 11. The submenu of the "FU" menu selection increases or decreases the number of Funtional Units by selecting "+" or "-", respectively. The upper bound on the number of Functional Units within a Processor-Group object can be specified via the "FU Limit" push button. The upper bound of the number of Functional Units is the maximum number of resources during the simulation process. If the total

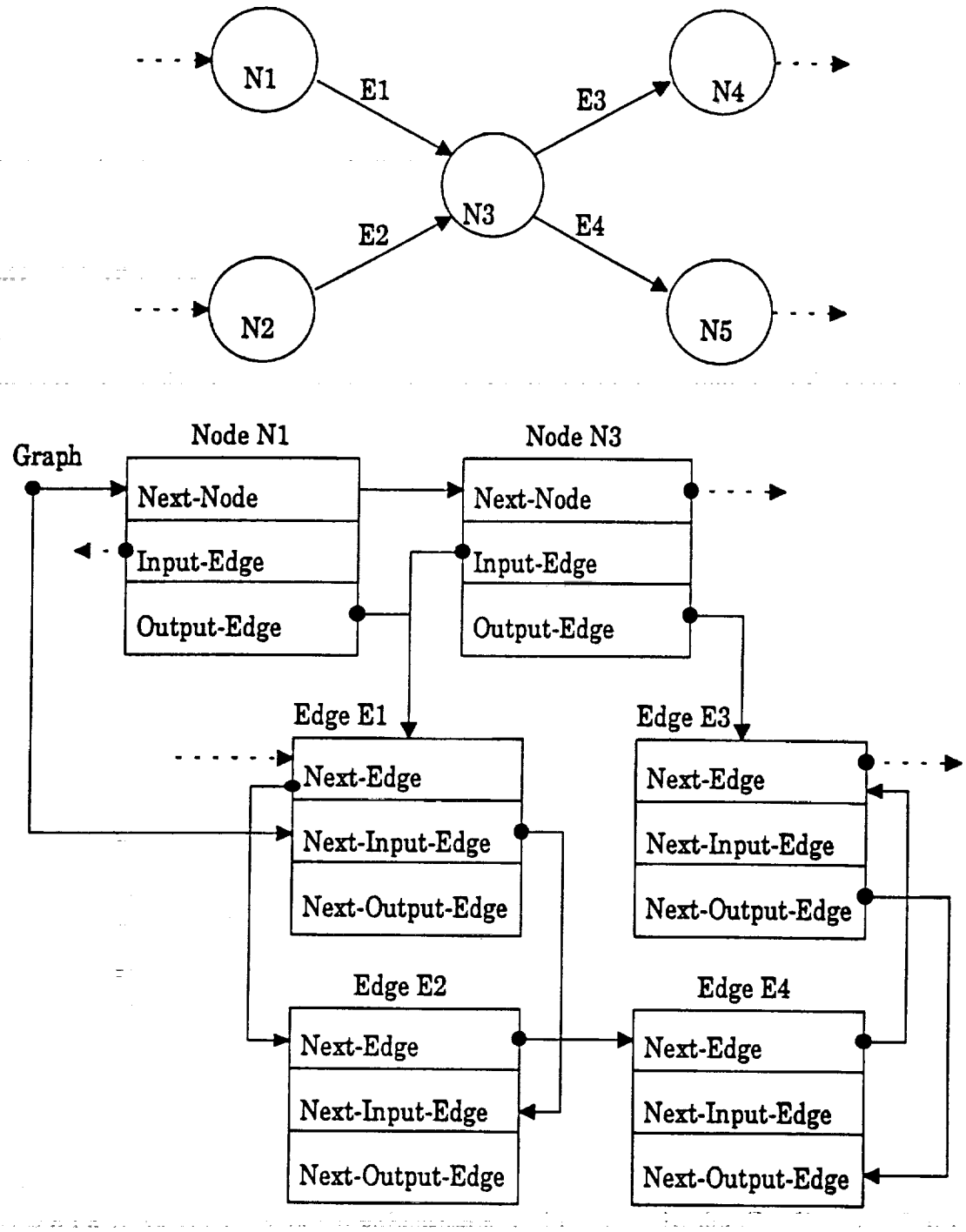

Figure 10. Portion of an example graph and its data structure.

number of Functional Units in a Processor-Group object is less than the upper bound, the Simulator creates, during the run time, as many Functional Units as necessary to carry out its operation without violating the upper bound restriction. The relative speed of a Processor-Group object compared to other Processor-Groups can be specified by the "Speed" submenu. The relative speed of a Processor-Group object can be decreased by "+" and increased by "-". The "Help" button invokes the appropriate help file where specific guidance for the Processor-Group window is provided. A push button is provided for a future selectable "Test Time" to simulate self-testing by the Functional Units. However, the specific use of self-test is not yet implemented.

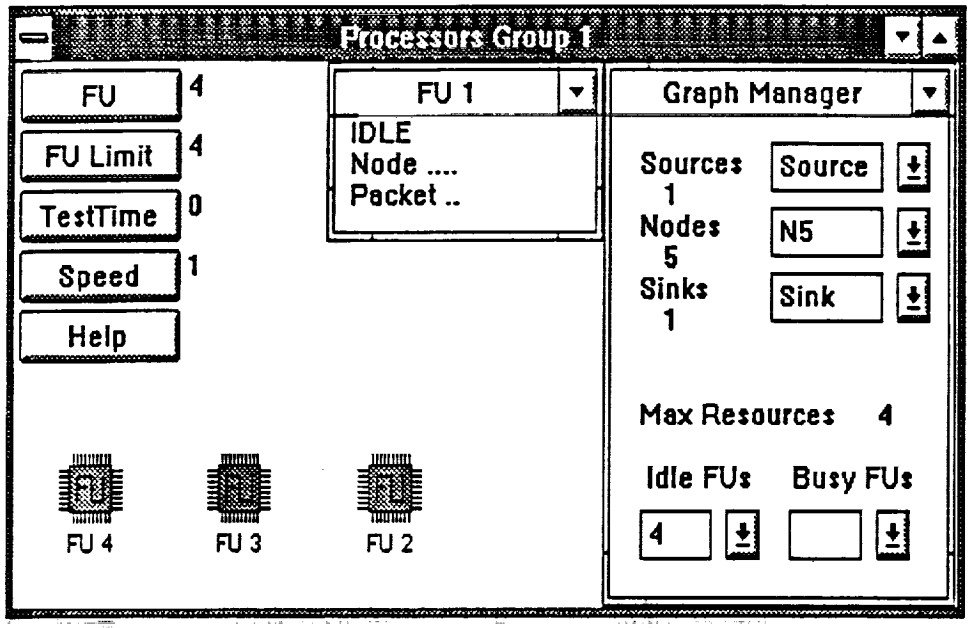

Figure 11. Processor Group.

4.6.1 Graph-Manager

The graph manager is responsible for ensuring that the overall system operates according to the ATAMM rules. The Graph-Manager object, representing the graph manager of ATAMM, updates and monitors the status of the CMG. When a read transition of this graph is enabled, the Graph-Manager assigns a Functional Unit from the list of available Functional Units to perform the corresponding algorithm node according to priority if more than one node is enabled. If there are additional enabled nodes, the Graph-Manager assigns them to the subsequent Functional Units

in the available list. The Graph-Manager updates the marking of the CMG using status information reported by the Functional Units.

Since the Graph-Manager object is part of the Processor-Group object, it only keeps track of the AMG nodes that are assigned to the Processor-Group object by a linked list of source, node, and sink objects. Although the source and the sink objects have a lot in common with the node objects, they also have some differences. For instance, the source objects must deal with the special source control edges and the sink objects must keep track of the output data packets. Therefore, the source and the sink objects are stored in separate linked lists from the node objects to keep their operations separate and to speed up the simulation process. The data structure of the algorithm graphs, as portrayed by the Graph-Manager object, is depicted in Figure 12.

Upon updating the CMG, if necessary, the Graph-Manager broadcasts the updated information to other Graph-Managers. The necessity of broadcasting part or all of the updated CMG depends on the partitioning of the nodes of the AMG. If dependencies exist among the AMG nodes of the Graph-Managers or if an AMG node is assigned to multiple Graph-Managers, then whenever one Graph-Manager is updated, part or all of the updated information ought to be shared with other Graph-Managers. Since the Graph-Manager object has knowledge of the system, it is also responsible for creating Functional Units at run time, based on need without violating the upper bound limitation of the Processor-Group object.

The Graph-Manager object displays information about the graph and the status of the Functional Units in the Processor-Group object. This information mainly consists of the count and names of the sources, nodes, and sinks that are assigned to the Processor-Group object, and the content of the idle and busy Functional Unit lists of the Processor-Group object.

4.6.2 Functional Units

The Functional Unit object is designed to carry out the tasks represented by the AMG nodes. The Functional Unit object, therefore, does not distinguish between the sources, the sinks,

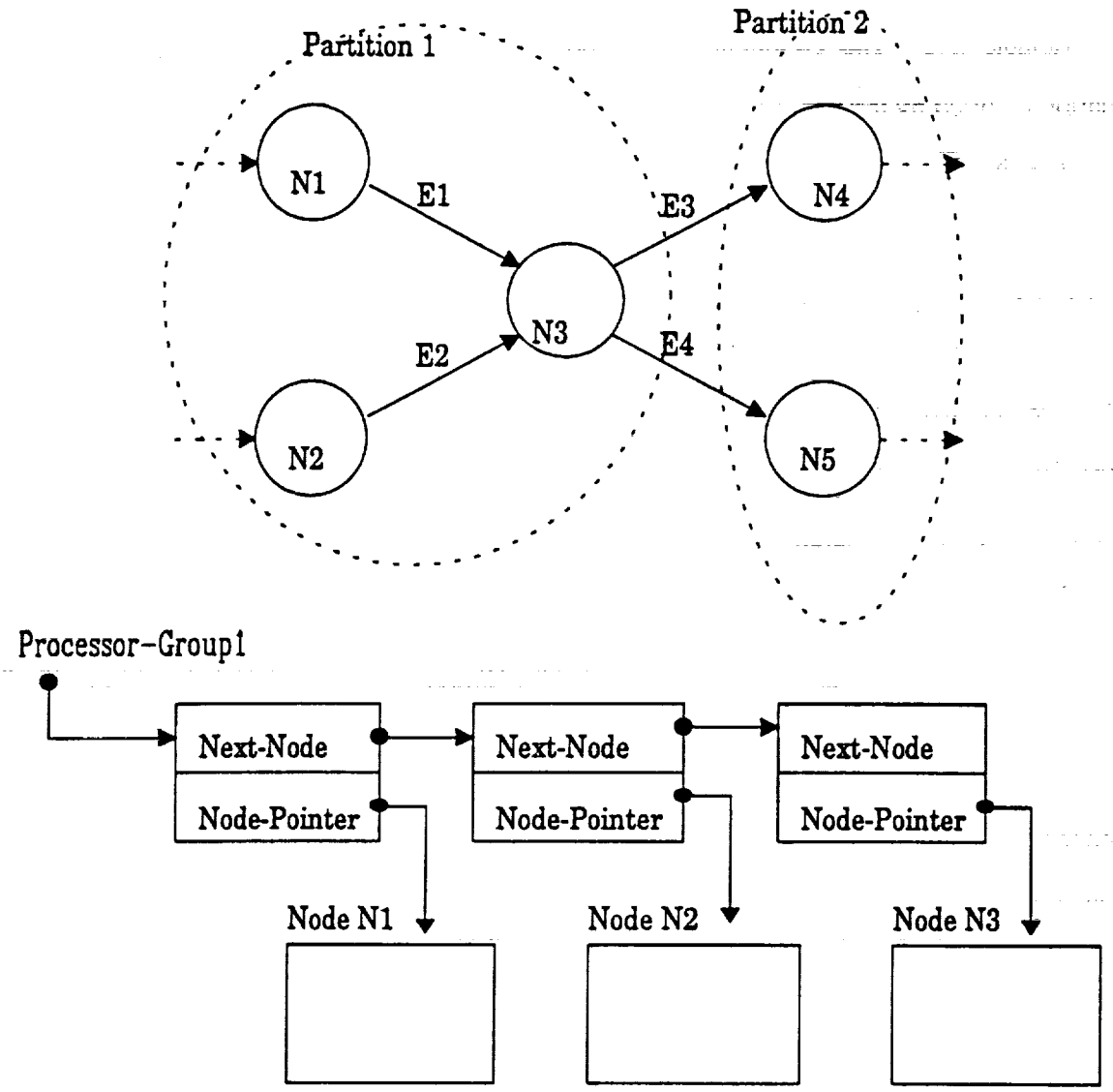

Figure 12. An example graph and the node data structure for one group of nodes.

and the nodes of the AMG. To carry out execution of an AMG node of any kind, the Functional Unit must be assigned a node to execute. The assignment of an AMG node to the Functional Unit is accomplished by the Functional Unit that currently holds the semaphore and is the active graph manager of the system. The active graph manager, a Functional Unit, can assign an AMG node to itself or another Functional Unit. A Functional Unit becomes the active graph manager when it is

in the Writing State or when it is both in the Idle State and at the top of the list of available Functional Units. The active graph manager possesses the semaphore and is the only Functional Unit that can talk over the Network while other Functional Units listen. To grab the semaphore, the Functional Unit may have to compete with others. The semaphore is granted based on the specified protocol of the defined architecture. Sections 4.6.5 and 4.7 discuss the communication network protocols.

To complete execution of the AMG node, the attached Functional Unit goes through a sequence of states as depicted in Figure 6 for the AMOS. These states define the operating system characteristics of the ATAMM Multicomputer Operating System (AMOS) and, thus, the state diagram of the Functional Units. This state diagram is described in the next Section. Through its window, the Functional Unit object displays information about its current status such as current state, the name of the assigned AMG node, and the number of the current data packet.

4.6.3 Functional Unit State Diagram Description

Idle

When idle, the Functional Unit awaits a fire-message indicating an AMG node is assigned to it for execution. It also continuously scans the Idle-List of available Functional Units to determine whether it is at the top of the list. When it finds itself at the top of the list and still idle, it scans the CMG for enabled read nodes. A CMG read node is enabled when every one of its input edges have a token with the appropriate tag and all of its output edges have an empty buffer. If there are enabled CMG read nodes, it attempts to grab the semaphore to become the active graph manager. Upon receiving a fire-message, the Functional Unit migrates to the *Reading* State.

Grab Semaphore 1

In this state the Functional Unit attempts to establish a communication link with other Functional Units. After establishing a communication link and grabbing a semaphore, the

Functional Unit becomes the active graph manager of the system and moves to the *Graph Manager State.* Otherwise, it goes to the *Idle State.*

Graph Manager

Being the active graph manager, the Functional Unit assigns the CMG read nodes to the idle Functional Units in the Idle-List. It sends fire-messages to the appropriate Functional Units, possibly including itself; moves the assigned Functional Units from the Idle-List to the Busy-List of Functional Units; updates the CMG and broadcasts the updated information to others. After the "Fire" broadcast, it releases the semaphore. The Functional Unit then migrates to the *Idle State.*

Reading

The *Reading State* represents the activity of reading the input data. The reading of input data is accomplished by consuming one token from each input edge with the appropriate token tag. After reading the node's input data, the Functional Unit progresses to the *Processing State.*

Processing

In this state, the Functional Unit executes the task represented by the node. The duration of this state is represented by the process time of the node. However, when simulating graphs with variable node times, the duration of this state is computed on the fly by calling the appropriate statistical function that represents the node. Upon completion, it progresses to the *Grab Semaphore State.*

Grab Semaphore 2

To write the generated output data on the output edges, the Functional Unit must grab the semaphore and become the active graph manager of the system. It remains in this state and competes for the semaphore until it is granted.

Writing

After becoming the active graph manager, the Functional Unit migrates from the Busy-List to the Idle-List of Functional Units. It then writes the output data on the output edges of the nodes and updates the CMG accordingly. The writing of output data is accomplished by inserting one token on each output edge registering the tag associated with it. The updated information is broadcast to other Functional Units via the "Data" broadcast. Before releasing the semaphore, it goes to the *Graph Manager State*.

4.6.4 FU Lists

The FU-Lists object manages the Functional Units and the Idle-List and Busy-List of Functional Units within a Processor-Group object. It creates and destroys Functional Units and moves them between the Idle-List and Busy-List upon receiving appropriate messages from the Graph-Manager object. It also keeps track of the number of Functional Units in the Processor-Group object. This object was created to facilitate the management of the Functional Units objects.

4.6.5 Local-Networks

The Local-Network object is envisioned to manage the arbitration of local semaphores among the Functional Units and to provide a means of establishing communication with the Global-Network object. Although all implementations of ATAMM have considered only a single semaphore, the Local-Network and Global-Network objects are intended to explore systems with multiple semaphores and a hierarchy of semaphores. In this regard, the Local-Network is a child of the Global-Network.

Nonetheless, this Simulator assists in the development of theories regarding the ATAMM under ideal conditions. Networks do not exist under ideal conditions. Due to lack of time, the Local-Network object is not yet implemented. In this regard, the communication latency is zero

and the simulation is performed under the ideal condition. However, the system is still limited to a single semaphore to ensure the integrity of the CMG markings.

4.7 Global-Networks

The Global-Network object is envisioned to manage the arbitration of global semaphores among the different Processor-Group objects and to provide a means of establishing communication with the Local-Network objects. For the reasons stated in Section 4.6.5, the Global-Network object is not yet implemented. The communication latency among the Processor-Group objects are also zero and the simulation is performed under the ideal condition. The single semaphore mentioned earlier is global throughout the system and ensures the integrity of the CMG markings.

4.8 TBO/TBIO and Ensemble TBO/TBIO

The Ensemble object is designed to calculate the TBO, TBIO, ensemble TBO, and ensemble TBIO points. During the simulation process, the time when a data packet is injected into an algorithm graph and the time when the same data packet exits the algorithm graph are recorded. This information is then used to calculate the TBO and TBIO points for all data packets. Through the Ensemble's window, the TBO and the TBIO points are plotted as shown in Figure 13. This process continues until the TBO and TBIO points of all data packets are determined. However, if ensemble TBO and ensemble TBIO points are desired after each simulation of the algorithm graphs, the TBO and the TBIO points are averaged for each simulation to calculate the ensemble (or average) TBO and ensemble (or average) TBIO points, respectively, and only these averages are recorded and plotted. This process continues until all ensemble points are determined.

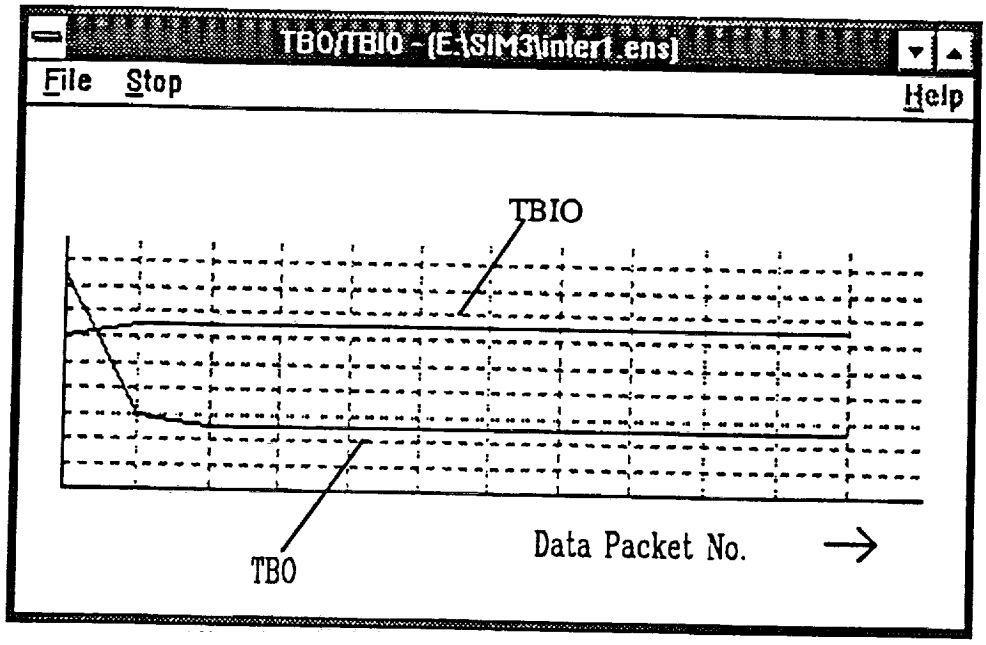

Figure 13. The TBO and TBIO plots.

Through the menu options of the Ensemble window, the calculated TBO and TBIO points along with their averages can be stored in an ensemble (ENS) file for future references by the "Save..." option. Ensemble files have ".ens" extensions and are described in Section 4.10. It is also possible to print the plotted points as depicted by this window. The "Average" option gives the averages of the points and the "Grid" option draws a grid along the x-axis and the y-axis for better visualization of the plotted diagrams. The "Scale Down" option allows resizing of the plotted diagram to the desired scale.

4.9 System

The System object is created to display the status of the system. While the simulation is in progress, the System-Clock and name of the output FDT file are displayed. Continuous display of the System-Clock gives an indication of the speed and duration of the simulation process.

4.10 The Input and Output File Formats

The input algorithm marked graph files provided by the Graph Entry tool are a set of node and edge objects and the information about the relationships among them. The format of the input graph (GRF) file generated by the Graph Entry tool is given Appendix A.

The format of the output FDT file as generated by the Simulator is defined in Appendix B. For details on the meaning and significance of each element, please refer to documents provided with the ATAMM Analysis tool [12]. An example of an FDT is provided in Appendix C.

The computed TBO, TBI, ensemble TBO, ensemble TBIO points, and their averages are stored in the output ensemble (ENS) files. Two examples of the ENS files are provided in Appendices D and E. Appendix D represents the TBO and TBIO points for a single simulation and Appendix E lists the ensemble TBO and ensemble TBIO points for each of 12 simulations and the ensemble (average) for all simulations.

4.11 How to Use the Simulator

To simulate an algorithm graph, the algorithm graph must first be generated by use of the Graph Entry tool. The graph must be drawn and its attributes such as read, process, and write times of the nodes; node function (for variable node latencies); node assignment to groups of processors; buffer sizes and initial tokens of the edges; and injection time and sequencing of the sources must be defined. The algorithm graphs can then be loaded into the Simulator. The Simulator extracts the necessary information from the GRF file and sets up the system accordingly. It is also possible to specify the system attributes through the Simulator's objects. The procedure to simulate an algorithm graph is shown in Table 1 as well as in the help files provided by the Simulator software.

1. "Open..." an existing graph,
2. create as many "Processor-Group" objects as necessary and Design these objects to fit your specifications (this information could also be provided by a GRF file),
3. "Discard..." as many data packets as necessary,
4. select "TBO/TBIO" or "Ensemble",
5. specify "Duration..." of the simulation process (this information is also provided by a GRF file and as a sink attribute),
6. set the "Speed" of the simulation process, and
7. "Run" the Simulator. When finished, the Simulator will prompt accordingly.
8. To exit the Simulator, either double click on the system menu button of the Simulator's window or choose the exit option in its system menu.

Table 1. Simulator Execution Procedures.

5. Case Studies and Experimental Results

In this section, two case studies are presented as a demonstration of the application capabilities of the Simulator in studying the behavior of algorithm graphs under the ATAMM rules. These case studies are conducted and presented in a manner that typically would take the user of the Simulator through the procedural steps for creating algorithm graphs and evaluating the desired system. An example graph referred to as Intermediate 1 (*Inter1.grf*) and depicted in Figure 14 is considered for all case studies. The first case study is a homogeneous simulation of the *Inter1.grf* graph. The second case study is a heterogeneous simulation of the *Inter1.grf* graph that demonstrates capabilities and features of the Simulator in static assignment of nodes to different groups of processors.

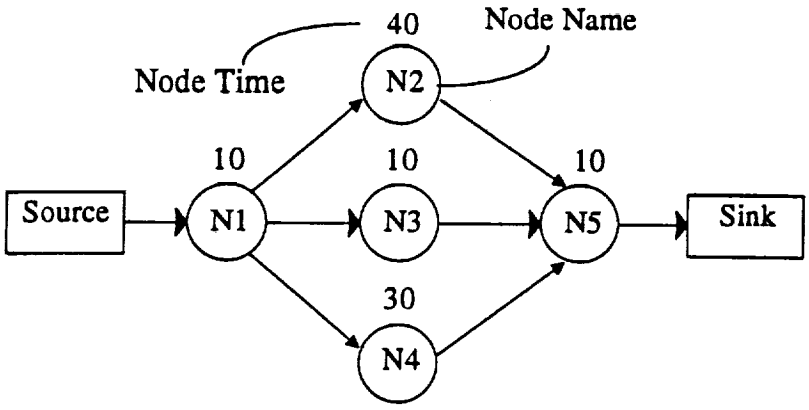

Figure 14. The *Inter1.grf* graph.

5.1 Case Study 1

This case study is primarily conducted for validating the results of the simulation with the theoretical predictions and compliance with a previous simulator [9]. All nodes execute on a single Processor-Group. The timing latencies of the nodes in Inter1.grf are shown in Figure 14. For this case study the read time and write time of the nodes are assumed to be zero time units for

the ideal simulation of the graph. The Single Graph Play (SGP) and the Total Graph Play (TGP), [5, 6], for four resources of this graph are shown in Figure 15. The TGP of Figure 15 is the modified TGP of the graph after adding a control edge from node N3 to node N4.

After loading the *Inter1.grf* file, the Simulator-Kernel window's caption bar is updated and reflects the name of the file loaded, as shown in Figure 9. The Processor-Group windows are also updated to reflect the specified system, Figure 11. Results of the simulation of the graph are then analyzed by the Analysis Tool [12] and are shown in Figure 16. Analysis of the results of the simulation of the graph reveal compliance with the theoretical prediction where TBO equals 25, as depicted in Figure 15.

5.2 Case Study 2

The static assignment of nodes and heterogeneous capabilities of the Simulator are studied here. In this case study, the nodes N1 and N2 and the Source are assigned to one Processor-Group with two functional units. Nodes N3, N4, N5, and the Sink are assigned to another Processor-Group with two functional units. This partition of nodes among Processor-Groups is consistent with the modified TGP of Figure 15 and should result in the same TBO and TBIO performance as for Case Study 1. Analysis of the results reveal that the same performance as the previous case study are achieved. Figure 17 is the task and resource activity display and the cursors mark a time interval corresponding to the TGP of the graph.

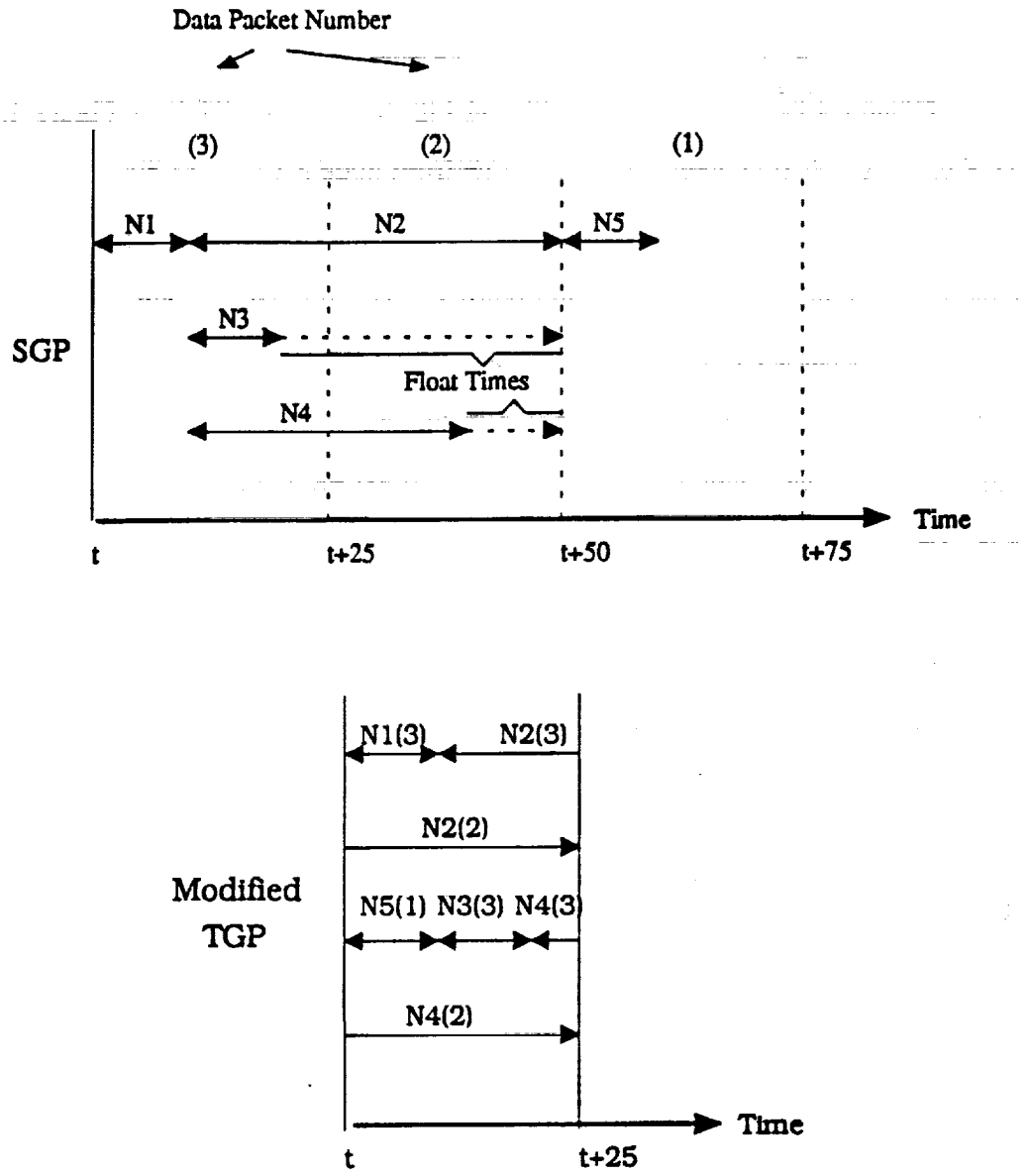

Figure 15. SGP and Modified TGP.

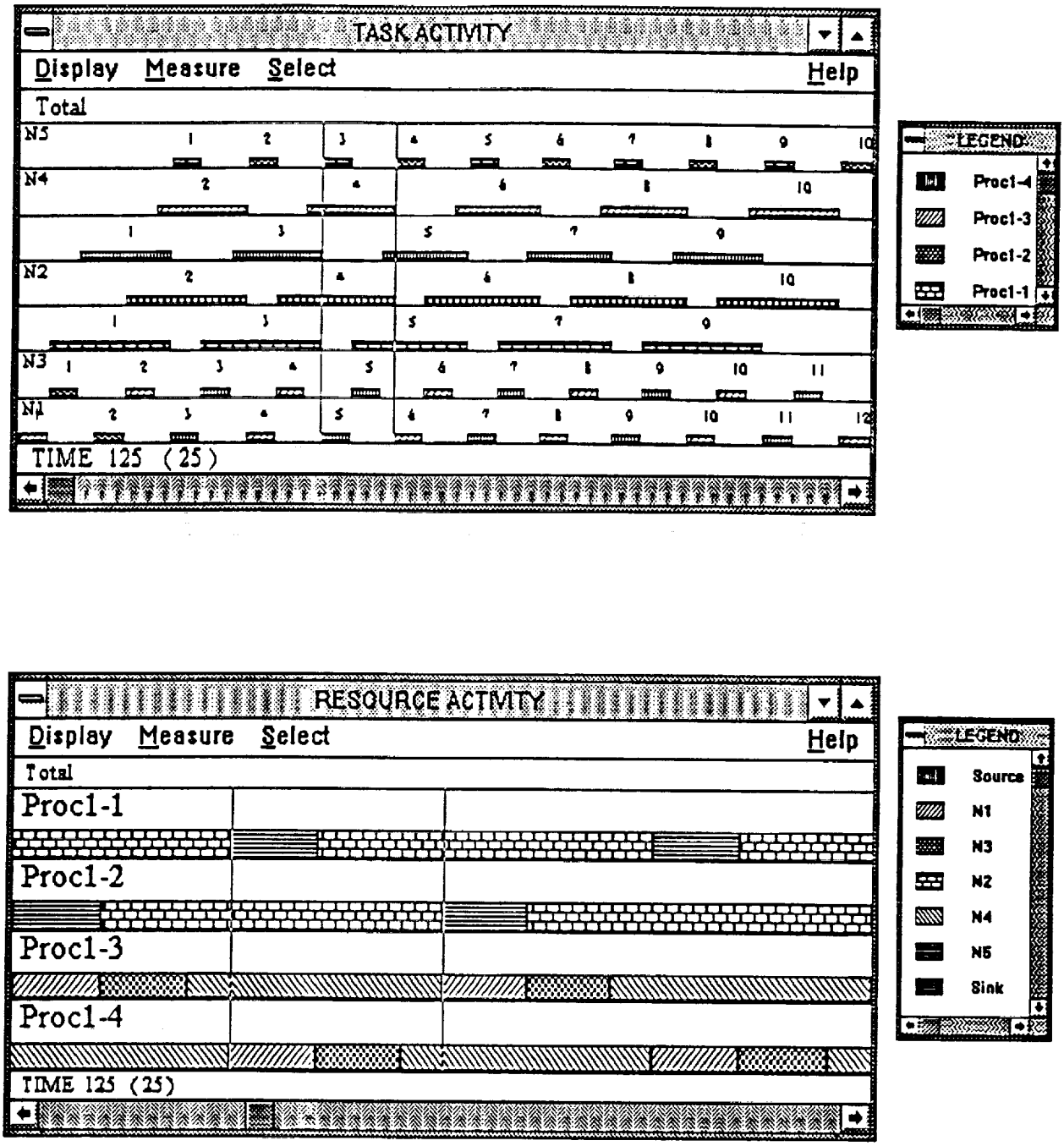

Figure 16. The task and resource activities for Case Study 1. The spacing between the vertical cursors is 25 time units.

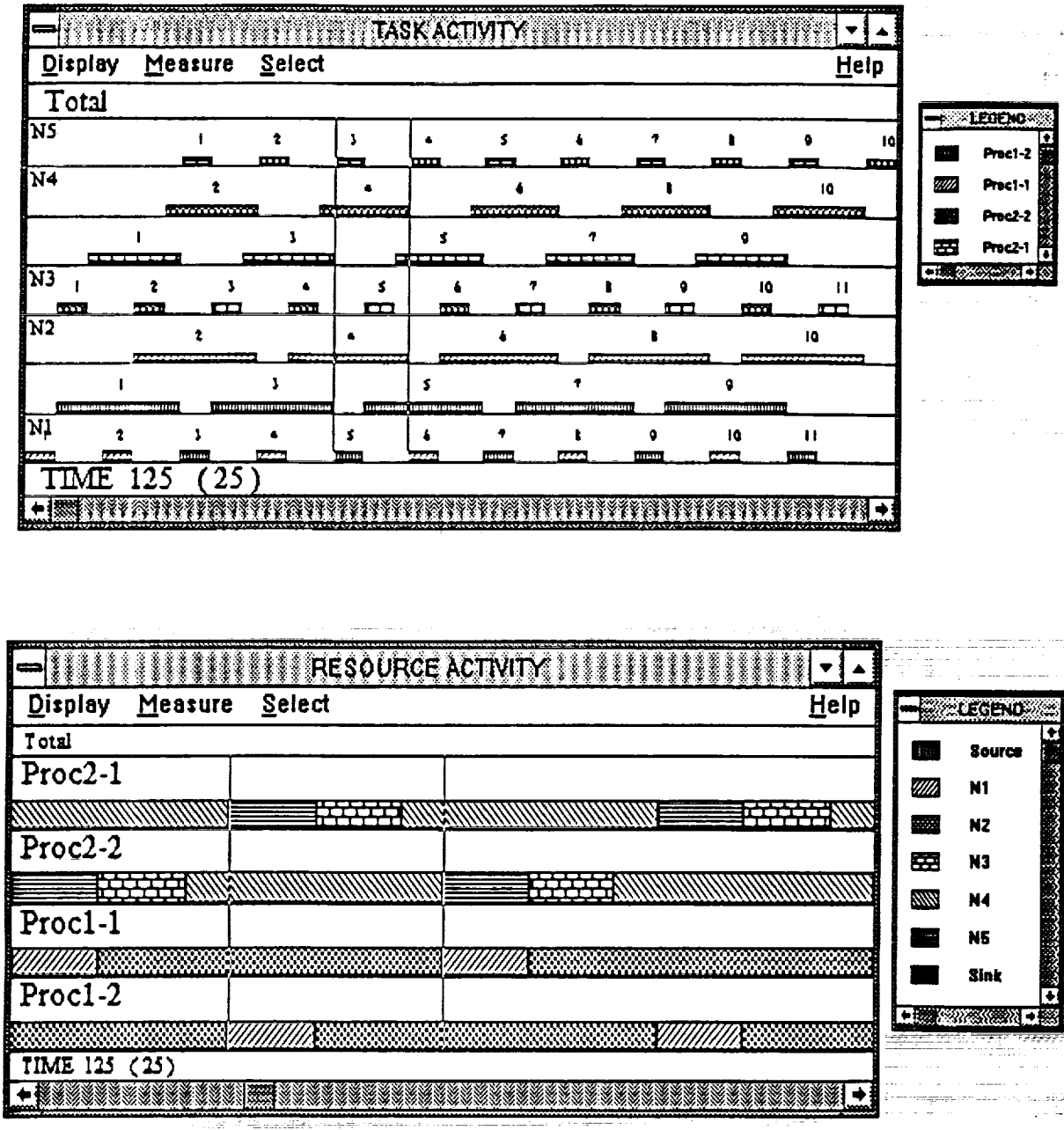

Figure 17. The task and resource activities for Case Study 2. The spacing between the vertical cursors is 25 time units.

6. SUMMARY

A Simulator is developed to simulate the execution of algorithm marked graphs in accordance with the ATAMM rules. Whereas previous ATAMM simulators assume that all algorithm graph nodes are executed on a homogeneous set of functional units, this new Simulator enables groups of graph nodes to execute on different processor groups, where each processor group may represent a different type of functional unit. Thus, a heterogeneous architecture may be simulated. The Simulator is based on object oriented programming and is event-driven to accelerate simulation speed. It provides the simulation functions in an ATAMM Integrated Environment, which also includes a Graph Entry tool for describing graphs for simulation, a Design Tool for analyzing and altering a graph to obtain desired performance, and an Analysis Tool for playing back the results of a simulation. Test cases show that the simulator accurately executes the ATAMM rules for both a heterogeneous architecture and a homogeneous architecture, which is a special case for only one processor group.

References

[1] John W. Stoughton and Roland R. Mielke, "The ATAMM Procedure Model for Concurrent Processing of Large Grained Control and Signal Processing Algorithms," Proceedings of NAECON 88, 121, May 1988.

[2] R. R. Mielke, John W. Stoughton, and S. Som, "Modeling and Performance Bounds for Concurrent Processing," NASA CR 4167, Grant NAG1-683, August 1988.

[3] Asa M. Andrews, Robert L. Jones, Paul J. Hayes, and Harry F. Benz, "Simulator for Enhanced ATAMM Multiprocessing," AIAA Computing in Aerospace 8: A Collection of Technical Papers, Vol. 2, 542, October 21-24, 1991.

[4] R. L. Jones, P. J. Hayes, A.M. Andrews, S. Som, John W. Stoughton, and R. R. Mielke, "Enhanced ATAMM for Increased Throughput Performance of Multicomputer Data Flow Architectures," Proceedings of the IEEE NAECON 91, Vol. 1, 238, May 1991.

[5] Sukhamoy Som, R. Mielke, R. Obando, J. Stoughton, P. J. Hayes, and R. L. Jones, "Throughput Enhancement by Multiple Concurrent Instantiations in the ATAMM Data Flow Architecture," Proceedings of the ISMM International Symposium on Computer Applications in Design, Simulation, and Analysis, Las Vegas, NV, 71, March 1991.

[6] S. Som, J. W. Stoughton, and R. R. Mielke, "Strategies for Concurrent Processing of Complex Algorithms in Data Driven Architectures," NASA CR 187450, Final Report, Grant NAG1-683, October 1990.

[7] P. J. Hayes, R. L. Jones, H. F. Benz, A. M. Andrews, J. W. Stoughton, R. R. Mielke, M. R. Malekpour, and P. R. Appleget, "VHSIC Multiprocessor Implementation of the ATAMM Strategy," GOMAC91/1991 Digest of Papers, 521, November 1991.

[8] R. Mielke, J. Stoughton, S. Som, R. Obando, M. Malekpour, and B. Mandala, "Algorithm to Architecture Mapping Model (ATAMM) Multicomputer Operating System Functional Specification, " NASA CR 4339, Cooperative Agreement NCC1-136, November 1990.

[9] Mahyar R. Malekpour, John W. Stoughton, and Roland R. Mielke, "Simulator for Concurrent Processing Data Flow Architectures," NASA CR 189604, Cooperative Agreement NCC1-136, March 1992.

[10] S. Som, J. W. Stoughton, and R. R. Mielke, "Performance Prediction, Simulation, and Measurement for Real-Time Computing in a Class of Data Flow Architectures," Proceedings of the ISMM International Conference on Computer Applications in Design, Simulation, and Analysis, ACTA Press, pp. 64-68, New Orleans, March 5-7, 1990.

[11] W. R. Tymchyshyn, "ATAMM Multicomputer System Design," Master's Thesis, Old Dominion University, Norfolk, Virginia, August 1988.

[12] Robert Jones, John Stoughton, and Roland Mielke, "ATAMM Analysis Tool," NASA CR 187625, Cooperative Agreement NCC1-136, October 1991.

Appendix A

Format of Graph Description in Graph-Entry Output File

Note 1. *ITALICs* underline are for information
Note 2. Only *ITALIC* is for a choice or decision
Note 3. All Times are Positive Long Integer Values
Note 4. All Locations 'X Y' are range 1..100

(HEADER)
Version 2.0.13
System_Max_CPUs -- Number of CPUs allowed in the system range 1..32
Current_Number_CPUs -- Initial number of CPUs range 1..System_Max_CPUs
Max_Index -- Max Indexes for the Operating Point Table range 1..10
Current_Index -- Initial Index for the run range 1..Max_Index
Max_Number_Groups -- Number of Heterogeneous Groups
Max_Nodes -- Max Number of Nodes in all Graphs
Max_Arcs -- Max Number of Arcs in all Graphs.
Max_Sources -- Max Number of Sources.
Max_Sinks -- Max Number of Sinks.
Self_Test_Time
Display_CPU_Number -- Used to Display a certain configuration of a Graph
Display_Index_Number -- Display which index configuration
Selected_Group
Show_All_Objects -- Display for Enabled or Disabled Control Arcs
Origin.X -- Used to size of the Graph Window
Origin.Y -- Used to size of the Graph Window
Right_Bottom.X -- Used to size of the Graph Window
Right_Bottom.Y -- Used to size of the Graph Window
Grid_Status -- Grid Display ON or OFF
Heterogeneous -- True / False Flag for Heterogeneous System Simulations
Number_CPUS_Group -- (Array[Max_Number_groups..1] of Integers)
Object_Type -- (NODE, SOURCE, SINK, ARC)
LOOP
if Object_Type = NODE then
Node_Graph_Number
Block_Index -- unique for all blocks in all graphs
Node_Number
Node_Name
Node_Mode -- (SIMPLEX, DUPLEX, TMR)
Node_User_File_Name
Node_Priority
Node_Instantiations(1..System_Max_CPUs,1..Max_Index)
Node_Read_Time

```
    Node_Process_Time                   -- (Mean Value of Process Time)
    Node_Write_Time
    Node_Color
    Node_Number_Inputs
    Node_Number_Outputs
    Node_Random_Function                -- (A,B,C, etc.)
    Node_LowerProcessTimeBound          -- (Smallest Possible Bound on Process Time)
    Node_UpperProcessTimeBound          -- (Largest Possible Bound on Process Time)
    Node_ProcessType    -- (Boolean array[1..Max_Number_Groups] Heterogeneous)
    Node_SubGraph_File_Name             -- If node has a subgraph.
    Node_Location                       -- (X  Y)  Coordinates
end if
If Object_Type  = SOURCE then
    Source_Graph_Number
    Block_Index
    Source_Number
    Source_Name
    Source_Mode
    Source_Priority                     -- Graph Priority (?)
    Source_TBI(1..System_Max_CPUs,1..Max_Index)
                                        -- Time Between Inputs (TBI)
    Source_Number_DataPackets           -- Number of Data Packets for each Source Edge
    Source_Write_Time
    Source_ProcessType  -- (Boolean array[1..Max_Number_Groups] Heterogeneous)
    Source_Location                     -- (X Y) Location of the Source
end if
if Object_Type  = SINK then
    Sink_Graph_Number
    Block_Index
    Sink_Number
    Sink_Name
    Sink_Mode
    Sink_Read_Time
    Sink_Number_DataPackets             -- Number of Data Packets Received at Sink
    Sink_ProcessType    -- (Boolean array[1..Max_Number_Groups] Heterogeneous)
    Sink_Location                       -- (X Y) Location of the Source
end if
if Object_Type = ARC then
    Edge_Number
    Edge_Type                           -- (CONTROL, DATA)
    Edge_Initial_Type                   -- (SOURCE_TYPE, NODE_TYPE, SINK_TYPE)
    Edge_Initial                        -- Number of Initial
    Edge_Initial_String                 -- Name of the Node, Source, Sink
    Edge_Initial_Block_Index            -- Block Index of the Initial Block
    Edge_Initial_Parm_Number            -- Position in procedure call (0 if CONTROL)
```

```
      Edge_Terminal_Type                -- (SOURCE_TYPE, NODE_TYPE, SINK_TYPE)
      Edge_Terminal
      Edge_Terminal_String              -- Name of the Node, Source, Sink
      Edge_Terminal_Block_Index         -- Block Index of the Terminal Block
      Edge_Terminal_Parm_Number         -- Position in procedure call (0 if CONTROL)
      if Edge_Type = DATA then
        Edge_Data_Type                  -- TBD Either a File_Name or Data_Type Name
        Edge_Size                       -- TBD  Whether or not to include.
        Edge_Tagging_Rule               -- Data Packet Distance
      else
        Edge_Tagging_Rule(1..System_Max_CPUs,1..Max_Index)
        if Edge_Terminal_Type = SOURCE_TYPE then
          Edge_Delay(1..System_Max_CPUs,1..Max_Index)
                                        -- Firing Delay for Terminal
          Edge_Selector(1..System_Max_CPUs,1..Max_Index)
                                        -- Output edge selection for token
        end if
      end if
      Edge_Inital_Tokens(1..System_Max_CPUs,1..Max_Index)
                                        -- Seperated by <CR>
      Edge_Tokens_Limit(1..System_Max_CPUs,1..Max_Index)
                                        -- Arc not enabled if size = 0
      Edge_Max_Buffers
      Edge_Number_Joints
      Edge_Joint (1..Max_Number_Joints)  -- X Y coordinates
  end if
REPEAT UNTIL <EOF>
```

Appendix B

Format of the FDT File Generated by the Simulator

```
//The FIELDS to be read from an FDT event
Fields = 5
TIME "%lu "
EVENT "%s "
TASK "%s "
COLOR "%d "
RESOURCE "%s"

//The possible EVENTS that can be found in the FDT file
//{FIRE, DATA, RUN, HALT, EVENT}
Events = 10
NodeRead      >FIRE
NodeProcess
NodeWrite
NodeIdle      >DATA
FU_Test       >FIRE
FU_EndTest    >DATA
SourceWrite   >FIRE
SourceIdle    >DATA
SinkRead      >FIRE
SinkIdle      >DATA

//The possible ACTIVITIES that can be found in the FDT file
Activities = 4
Process       >NodeProcess
ReadWrite     >SourceWrite,SinkRead,NodeRead,NodeWrite
Test          >FU_Test
Idle          >NodeIdle,SourceIdle,SinkIdle,FU_EndTest

//The clock resolution of time tags in clock ticks per second
//Clock = 1000000
```

Appendix C

FDT File Example

```
// Simulator Version 3.0, Output FDT File
// Graph file name: E:\SIM3\inter1.grf
25 SourceWrite Source 1 Proc1-4
25 SourceIdle Source 1 Proc1-4
25 NodeRead N1 1 Proc1-3
25 NodeProcess N1 1 Proc1-3
35 NodeWrite N1 1 Proc1-3
35 NodeIdle N1 1 Proc1-3
35 NodeRead N4 1 Proc1-2
35 NodeProcess N4 1 Proc1-2
35 NodeRead N3 1 Proc1-1
35 NodeProcess N3 1 Proc1-1
35 NodeRead N2 1 Proc1-4
35 NodeProcess N2 1 Proc1-4
45 NodeWrite N3 1 Proc1-1
45 NodeIdle N3 1 Proc1-1
50 SourceWrite Source 1 Proc1-3
50 SourceIdle Source 1 Proc1-3
50 NodeRead N1 1 Proc1-1
50 NodeProcess N1 1 Proc1-1
60 NodeWrite N1 1 Proc1-1
60 NodeIdle N1 1 Proc1-1
60 NodeRead N4 1 Proc1-3
60 NodeProcess N4 1 Proc1-3
60 NodeRead N3 1 Proc1-1
60 NodeProcess N3 1 Proc1-1
65 NodeWrite N4 1 Proc1-2
65 NodeIdle N4 1 Proc1-2
65 NodeRead N2 1 Proc1-2
65 NodeProcess N2 1 Proc1-2
70 NodeWrite N3 1 Proc1-1
70 NodeIdle N3 1 Proc1-1
75 NodeWrite N2 1 Proc1-4
75 NodeIdle N2 1 Proc1-4
75 NodeRead N5 1 Proc1-1
75 NodeProcess N5 1 Proc1-1
75 SourceWrite Source 1 Proc1-4
75 SourceIdle Source 1 Proc1-4
75 NodeRead N1 1 Proc1-4
75 NodeProcess N1 1 Proc1-4
85 NodeWrite N5 1 Proc1-1
85 NodeIdle N5 1 Proc1-1
```

Appendix D

ENS File Example for TBO & TBIO for 12 Data Packets of a Single Simulation

// Simulator Version 3.0, TBO/TBIO Points
// Graph file name: E:\SIM3\inter1.grf

Number of TBO/TBIO points at Sink: 12

TBO	TBIO
85.00	60.00
30.00	65.00
25.00	65.00
25.00	65.00
25.00	65.00
25.00	65.00
25.00	65.00
25.00	65.00
25.00	65.00
25.00	65.00
25.00	65.00
25.00	65.00

TBO/TBIO Averages:

30.42	64.58

Appendix E

ENS File Example of Ensemble TBO and Ensemble TBIO Points for 12 Simulations

// Simulator Version 3.0, TBO/TBIO Ensemble Points
// Graph file name: E:\SIM3\inter1.grf

Number of TBO/TBIO Ensembles at Sink: 12

TBO	TBIO
30.00	64.00
30.00	64.00
30.00	64.00
30.00	64.00
30.00	64.00
30.00	64.00
30.00	64.00
30.00	64.00
30.00	64.00
30.00	64.00
30.00	64.00
30.00	64.00

TBO/TBIO Ensemble Averages:

30.00	64.00

Appendix F

Object-Oriented Programming

The following is quoted from [9] because of its importance in the development of this Simulator.

"Structured programming flourished because it was efficient in terms of human resources. Building and testing programs in discrete pieces enabled large applications to be developed in less time with fewer bugs than their non-structured counterparts. In addition, the run-time impact of structuring becomes less evident as a program grows in size. Object-oriented programming extends structured programming by encapsulating both data and their associated functions.

In traditional procedural languages like C or Pascal, the programmer defines data structures and writes functions and procedures to operate on the data. Although normally a correspondence exists between which functions operate on which types of data, most procedural languages offer no formal support for this correspondence; it is entirely the programmer's responsibility to manage such an abstraction.

In an object-oriented approach, *both data and operations that work with that data* are combined into a single logical unit known as an object. Dividing a program into objects encompassing both data and operations makes the program more closely represent the logical design that is being implemented. As a result, object-oriented programs are generally easier to understand and maintain than procedural programs.

Object-oriented programming is merely the art of breaking a program down and organizing it. In the case of structured programs, the primary concern is what the program is doing. A structured program is based on operations. When writing object-oriented programs, the program is organized around data types and their associated operations. It is a significant change in perspective; instead of functional hierarchies, there are data hierarchies. Programming in an object-oriented language involves creating objects and sending them commands or messages to do things.

Object-oriented programs are based on four concepts: classes, objects, methods, and inheritance. A *class* is similar to a Pascal RECORD. It describes an overall structure for any number of types based upon it. The main difference between a class and a record is that a class combines data fields (called *instance variables*) and procedures and functions (called *methods*) that act upon the data.

An *object* is a variable of a class. All objects derived from a class are considered members of that class and share similar characteristics of that class.

Methods are procedures and functions encapsulated in a class or object. Calling a method is referred to as *passing a message to an object.* Object-oriented programs do most of their works by sending messages to objects.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1993	Contractor Report (6/1/91-8/31/92)

4. TITLE AND SUBTITLE
Simulator for Heterogenous Dataflow Architectures

5. FUNDING NUMBERS
C NAS1-19000
WU 586-03-11-31

6. AUTHOR(S)
Mahyar R. Malekpour

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Lockheed Engineering & Sciences Company
144 Research Drive
Hampton, VA 23666

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING MONITORING AGENCY REPORT NUMBER
NASA CR-191545

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Paul J. Hayes

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 33

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A new simulator is developed to simulate the execution of an algorithm graph in accordance with the Algorithm to Architecture Mapping Model (ATAMM) rules. ATAMM is a Petri Net model which describes the periodic execution of large-grained, data-independent dataflow graphs and which provides predictable steady state time-optimized performance. This simulator extends the ATAMM simulation capability from a heterogenous set of resources, or functional units, to a more general heterogenous architecture. Simulation test cases show that the simulator accurately executes the ATAMM rules for both a heterogenous architecture and a homogenous architecture, which is the special case for only one processor type. The simulator forms one tool in an ATAMM Integrated Environment which contains other tools for graph entry, graph modification for performance optimization, and playback of simulations for analysis.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Simulation software, dataflow architecture, Petri Nets, concurrent processing, multiprocessing	55
	16. PRICE CODE A04

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL