A C-BASED SIMULATION FRAMEWORK FOR AUTOMATED GUIDED VEHICLE SYSTEMS

by

Jeffrey K. Wilson

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
Master of Science
in
Industrial and Systems Engineering

APPROVED:

Dr. M. P. Deisenroth, Chairman

Dr. O. K. Eyada

Dr. R. J. Reasor

June 1992

Blacksburg, Virginia

A C-BASED SIMULATION FRAMEWORK FOR AUTOMATED GUIDED VEHICLE SYSTEMS

by

Jeffrey K. Wilson

Committee Chairman: Dr. M. P. Deisenroth
Industrial and Systems Engineering

(ABSTRACT)

The purpose of this research was to develop and validate a simulation framework for automated guided vehicle systems (AGVSs). The framework that was developed, AGVSF, uses the discrete, next-event simulation method and the C programming language. AGVSF consists of an organizational structure that provides for control of the execution of the simulation and a set of modular C functions used to model the AGVS.

The structure of AGVSF allows the user to organize the simulation logic in a consistent manner. The modularity and flexibility of the code result from clearly defining the interdependencies of the functions that make up the various events and operations of the simulation. This enables the user to substitute functions where needed to represent new operational methods which are not directly provided in the original framework code set. A set of functions is provided within AGVSF for modeling basic AGVSs and AGVS layouts. The framework concept has been validated by simulating an AGVS under different operating conditions and control algorithms.

ACKNOWLEDGEMENTS

I would like to thank Dr. Mike Deisenroth for the support he has given me as my advisor, committee chairman and instructor. I can honestly say that we got more done before 9:00 in the morning than some people do all day.

I would also like to thank the United States Army for making my studies possible through the Advanced Civil Schooling Program. Thanks also to the hundreds of soldiers and officers who, through their dedication and hard work, made me look good enough to be selected for this program.

I would like to thank my wife, Estella, for catering to my every whim and for being at my beck and call during the past year. I would also like to thank my parents, Bob and Ruth Wilson, for setting such a fine example of integrity and hard work for me to follow and for instilling in me the values which I now hold dear.

Most of all, I want to thank God for giving me the ability and the stamina to accomplish this goal. I know that it was through His power that I was able to complete this work. Thanks also to our many friends at Grace Covenant Presbyterian Church, whose prayers kept me going when the obstacles seemed too high and the path too long.

TABLE OF CONTENTS

List of Figures

1. INTRODUCTION

1.1 Problem Statement

Automated guided vehicles (AGVs) are proving to be an excellent way to provide flexible material handling for modern manufacturing. Unfortunately, the complexity of automated guided vehicle systems (AGVSs), especially the large number of operational variables, makes them impossible to study using current analytical methods. This leaves simulation as the best way to analyze AGVS operations. Current simulation packages used to simulate AGVSs have several disadvantages, including their requiring the use of assumptions that limit the accuracy of the results and their use of high level simulation languages. The purpose of this research was to develop and evaluate AGVSF (Automated Guided Vehicle Simulation Framework), a flexible AGVS simulation tool using the C programming language. AGVSF gives the user a framework for accurately simulating a wide range of AGVSs in a language that is fast, powerful and widely used.

1.2 Background Information on AGVs

In order to create a useful AGVS simulation tool, a good understanding of the basic concepts of AGVs is required,

including; why AGVs are used, what they are, how they operate, how AGVs are controlled, what the various operating parameters are, why simulation is used to study AGVSs and what output is needed for system analysis. This knowledge was needed to ensure that the simulation framework that was developed is truly useful and anticipates the needs of the user. This chapter provided background information on the basic concepts of AGV operations.

1.2.1 History of Use of AGVs

AGVs are driverless vehicles used to transport material. The first use of an AGVS was by a manufacturing firm in the United States in 1954 [9]. Since that time, great advancements have been made in all areas of AGV technology, making them popular around the world in both manufacturing and service industries. AGVs are routinely used today as an alternative for manned fork lifts and tractors and for unmanned material handling systems such as roller conveyors [27]. Examples of their use include: transporting raw materials and palletized loads in warehousing and manufacturing, moving workpieces and fixtures between workstations in flexible manufacturing systems (FMSs) [27], delivering mail in office buildings, carrying food and

linens in hospitals and transporting automobiles on the assembly line.

1.2.2 Advantages of AGVs

The main advantages of AGVSs over manned material handling systems are the reduction in labor costs and the ease with which an AGVS is integrated into an automated manufacturing system [27].

AGVSs are much more flexible than fixed systems, such as conveyors. AGV layouts take up less space, are less obtrusive and can be changed more easily and inexpensively [27].

1.2.3 Components of an AGVS

It is generally accepted that an AGVS is made up of the following hardware and software components [32]:

1) AGVs.
2) Guidance and information transfer system.
3) Traffic control system.
4) Pickup and deposit (P/D) stations and load transfer equipment.

The efficiency of the AGVS is determined by how well these parts are suited for the application and how well they interact. Each component will be discussed in the following sections.

1.2.4 Types of AGVs

The size and carrying capacity of AGVs vary according to the size and weight of the loads they are intended to carry. Small unit load AGVs used in electronics manufacturing may be as small as 2 feet by 3 feet and have a carrying capacity of less than 100 pounds. At the other end of the spectrum, some assembly AGVs are 4.5 feet wide and 28 feet long and carry up to 95,000 pounds [7]. Because the characteristics of an AGV depend on their use, AGVs are usually categorized by function. The most common types are listed below.

A tow tractor or tugger vehicle is an AGV that pulls a train of unpowered trailer vehicles, like those used at airports to transport baggage between aircraft and the terminal [12]. The load is transported on the trailers. The number of trailers pulled by each tractor depends on the load carried by each. The more trailers that are towed, the larger the turning radius required. Tow tractor AGVs are

typically used for long distance hauling (more than 200 feet) of bulk loads.

Pallet truck AGVs are guided pallet jacks. They are normally configured with forks to the rear that can carry up to two loads at a time. This type of AGV is limited to picking up and putting down loads on the guide path, requiring another method for removing the load from the path. These AGVs are typically used for long distance warehouse delivery systems without automated receiving docks [12].

A unit load AGV is the most common and most versatile of all AGV configurations. It is identified by its ability to carry the load on a deck and transfer the load automatically to P/D points (control points where P/D stations are located). They are generally symmetrical, able to travel either forward or backward without any loss of capability. These AGVs are typically used when distances are short but the throughput volume is high, such as servicing an automated storage and retrieval system (AS/RS) and servicing flexible manufacturing systems (FMSs) [12].

A fork truck AGV is essentially a guided fork lift. A fork truck AGV differs from the pallet truck type AGV

described above in its ability to load and unload palletized loads from predesignated storage locations like a manually operated fork lift. The forks may be mounted to the rear or to the side of the AGV. These vehicles are typically used when palletized loads are precisely located at various heights [12].

Assembly vehicle AGVs are essentially guided mobile workstations designed to carry a work piece through the stages of its assembly. They are used for a variety of products from small assemblies to entire automobiles [12].

1.2.5 AGV Operations

AGVs have a variety of wheel configurations for steering. Generally, powered wheels steer and unpowered wheels or castors provide stabilization. Common configurations are to have one, two or four powered wheels that turn for steering [27]. The use of differential steering, with two unsteerable wheels at the center of the AGV and castors mounted at the front and rear is becoming very popular [32]. The AGV turns when one wheel rotates more than the other, which is how tracked vehicles turn. If the wheels are rotated in opposite directions, the AGV will

pivot about its center, giving the AGV a very small turning radius.

Unit load AGVs often have powered conveyors mounted on their decks for transferring loads to and from P/D stations [27,32]. These automated P/D stations have a conveyor or some other device that mates with the conveyor on the AGV. Typically, when the AGV senses that it is properly aligned next to the P/D station, the on-board controller activates the conveyor to receive or deliver the load [10]. Other AGVs have non-powered conveyors and depend on the P/D station to provide the mechanism for moving the load [32].

Safety is a major consideration in the use of AGVs. The prospect of having large unmanned vehicles moving around an area occupied by people and valuable equipment would be frightening without proper safety systems. The following precautions are typical of those used in AGVSs. Speed is normally limited to a maximum of 1 meter/second and slower at intersections and in heavy traffic areas to ensure that vehicles can stop quickly and that loads will not shift during braking and turning. Pressure-sensitive bumpers are used to sense the contact of the AGV with an object and initiate an emergency stop. Most have a collapsing design such that the AGV will be stopped before the bumper is fully

collapsed [28] Non-contact sensors such as infrared and ultrasonic can sense the presence of objects in the AGV's path that are a few meters away [27]. They are often used to signal the control unit to slow an AGV down as it approaches obstacles or other AGVs. These systems may be adversely affected by environmental noise often found in manufacturing areas. Additionally, their range is limited by the need for AGVs to maneuver in narrow paths. Other standard safety features include manual emergency stop buttons and audio or visual signals such as horns or beepers, bright colors, flashing lights and well marked paths [28].

AGVs are powered by on-board batteries of two types. Traction batteries are high-energy batteries designed to give a finite number of charges. Consequently, they are normally completely discharged before they are recharged. A discharge monitor notifies the AGV controller when the batteries are 80% discharged and the AGV is sent to the charging area. These batteries are used for large AGVs and two sets of batteries are required for each AGV to be operational 24 hours per day [18]. Automotive batteries are typically used in light-duty AGVs because they have lower energy capacities. They can be recharged at any point, and are suitable for "opportunity charging", where the AGV

connects to a charging system at every stop. This is common in FMS applications and in assembly operations [32].

1.2.6 **Path Layouts**

The AGV path layout is one of the most important factors in the operational efficiency of an AGVS [32]. A poor layout design can create bottlenecks and excessive travel which can make the AGVS uneconomical. There are several layout options and configurations which are commonly used.

The AGVS designer can choose to use uni-directional or bi-directional flow of AGVs. Each section of path in the network is designated as permitting AGV travel in either one direction or both. Bi-directional flow adds flexibility to the AGVS but requires a more powerful traffic control system to determine the routing and to resolve vehicle blocking conflicts. Uni-directional flow simplifies the traffic control task, but may result in longer travel distances and space and path requirements.

Tandem configurations isolate each AGV in its own loop without connection to other loops. The result is that AGV routing is very simple because there is no chance for conflict or blocking to occur because there are no

intersections. Transfer mechanisms must be used to move loads between loops, however, resulting in delays if transfers are frequently required. These systems are also susceptible to delays from vehicle breakdown, since no AGV can perform another's tasks [2,17].

Spurs and sidings are used to connect the load transfer equipment at P/D points with the main track, keeping it clear for through traffic during loading and unloading. Using spurs requires that the AGVs be able to backup, while sidings increase space and path requirements.

1.2.7 Navigation Techniques

1.2.7.1 Physical Guidepaths

Most AGVs used in manufacturing follow an inductive guidewire path defined by a wire buried a few centimeters deep in the floor. The wire is a conductor that carries an AC signal at a known frequency. This signal creates a magnetic field that induces a voltage in a pair of coils mounted on the bottom of the AGVs. By measuring the difference between the voltages in the two coils, the AGV controller can keep the AGV centered over the wire [27]. The AGV either follows a single frequency (and the system

controller shuts off all but the proper path at intersections) or a multiple frequency system is used and the AGV chooses the path designated by the correct frequency. Guidewire paths are durable and reliable, even in industrial environments.

An alternate guidance method is an optical guidepath, which uses a line painted on the floor in a contrasting color (such as white on a black floor). A lamp on the AGV illuminates the stripe and photocells detect the reflected light. A variation on this method is to use fluorescent paint. The AGV illuminates the stripe with an ultraviolet light, causing it to give off light at a frequency not found in the environment [10]. Optical guidepaths are easier and less expensive to install and modify than inductive guidepaths, but they are best suited for clean environments such as electronics manufacturing and offices.

A more durable option to optical guidepaths is the use of magnetic guidepaths. The path is defined by a metal stripe that can be covered by paint, paper and some debris and still be detected by the AGV's sensors [25].

One of the first guidance systems used with AGVs consists of a slot along the AGV pathway which is engaged by

a post which protrudes down from the AGV. This system is expensive to install and modify and limits the flexibility inherent in AGVSs and so is rarely used today.

1.2.7.2 **Virtual Guidepaths**

Many new methods have been developed that allow AGVs to move without a fixed guidepath. One of the advantages of virtual guidepaths is that changing the network requires only a software change, minimizing system downtime and increasing routing flexibility. Some of the terms that have been developed for such AGVs are self guided vehicles (SGVs), free-ranging vehicles and autonomous vehicles. Some of the navigation methods used are listed below.

Dead reckoning was one of the first navigation systems developed for free-ranging AGVs [27]. The AGV paths are defined in the memory of the AGV controller and the position of the AGV is updated using encoders to tell the controller how far each wheel has travelled. Wheel slippage can cause a buildup of positioning error [7], so the AGV's position is periodically re-initialized using a secondary method.

One example of a secondary navigation method is a beacon system used by Caterpillar Industrial, Incorporated's SGV.

It uses a LASER scanner to triangulate from any three of the many bar-coded targets mounted in the work space. An on-board computer calculates the AGV's position to within 1/4" [24]. By using a dead-reckoning system as its primary navigational system and this beacon system as a secondary system, the SGV can operate with the same accuracy available from guidewire systems. One drawback is that the beacon system must have line-of-sight access to three targets simultaneously to work [7].

There are many other secondary methods in use by other AGV manufacturers. Inertial navigation systems [10] use a gyroscope to detect the acceleration created when the AGV turns, speeds up and slows down. An on-board computer integrates the acceleration twice to calculate the displacement from the last known location. LORAN navigation systems can be used to locate AGVs within one foot [24]. Another common method is to lay down a pattern of painted or metallic lines to form a grid that the AGV can sense as described above (physical guidepaths). A copy of the grid layout is stored in the memory of the AGV controller so that the location of the AGV can be plotted as grid lines are crossed. Radio frequency identification (RFID) tags can also be used by positioning several of these devices at various known points around the work area to act as

transponders for the AGVs. When they are interrogated by a microwave beam from the AGV, they respond with a unique RF response that a microprocessor on board the AGV interprets and identifies [25].

Two other methods have been researched and show promise as navigation systems of the future. Direct imaging systems using machine vision techniques show excellent potential, but current technology is expensive and limits applications to stable environments [10]. Another method uses "passive encoded floor tiles" by covering the floor of the workspace with tiles that have been encoded with a unique pattern of circular metal dots. Each tile's pattern can be read by an AGV's linear sensor array, regardless of tile orientation [1].

1.2.8 Communication With AGVs

The system controller and the AGVs must be able to communicate instructions and status information to each other. The mobility of the AGV makes a hard wire connection between them impossible, so alternate methods are used. There are two types of communication -- continuous and discrete.

Continuous communication means that the AGV and controller can communicate at any time, regardless of position. This is an ideal situation, but not always a feasible one, due to limitations in communications technology. The most common method is radio frequency communication. Radio transmissions can be disrupted by noise produced in many industrial environments, including power generation and spot welding. Another method is to bury additional cables along with the guidepath wire to provide an inductive communication system [10].

Most AGVSs use discrete communication, where communication can only occur at certain points along the path. The most popular method is to use inductive wire loops along the side of the path. When the AGV passes over the loop, an on-board antenna transmits and receives control signals to and from the controller. An alternative is an optical system using light-emitting diodes to send and receive signals in digital or analog form [10].

1.2.9 **AGV Control**

There are two methods for controlling the travel of AGVs in an AGVS -- centralized and decentralized [16].

Centralized control uses a central computer to allocate and route AGVs and keep track of their position on a map of the layout kept in memory. Centralized systems normally use continuous communications to ensure accuracy [32].

Decentralized control uses substation computers to control the flow of AGVs in each subsection of the AGV route layout. The larger the system layout, the more substations are required. Decentralized control generally uses discrete communications to direct the AGVs along the route.

There are advantages and disadvantages to using both of these methods [16]. Decentralized control systems use a more modular approach which requires less powerful computers and is usually easier to maintain and expand than a centralized system. Decentralized control is generally better suited for AGVSs that are large, have complex layouts or have high materials flow. A centralized system works well for simpler AGVSs with fewer AGVs and less flow of materials [16].

1.3 Background Information on Simulation of AGVs

1.3.1 Methods for Analyzing AGVSs

There are many ways to analyze AGVSs, including analytical tools, spread-sheet analysis with queueing theory, heuristic approaches and simulation. Unfortunately, most AGVSs are much too complex to effectively analyze with any method other than simulation. In order to use the other approaches, many simplifications and assumptions must be made so that the complications arising from the random nature of the systems are not captured [30].

With simulation, however, even complex AGVSs can be modeled and the results of changing several operational parameters at once can be seen [11]. Computer simulation consists of designing a computer model of a system and using the computer to gather results of the model's operation. By changing the model to reflect various possible operating conditions, simulation can be used to evaluate the effect of changes to very complex systems [21].

1.3.2 Operational Parameters Used in Simulating AGVSs

Following is a list of the operational parameters most frequently used in the simulation of AGVSs [4,29,32]:

1) AGV velocity.
2) Number of AGVs.
3) AGVS layout.
4) Direction of travel in layout.
5) Location of P/D points.
6) Acceleration and deceleration of AGVs.
7) AGV dispatching rules.
8) Priorities at traffic intersections.
9) Size of buffers.
10) Load and unload time.

1.3.3 Assumptions Made in Simulation of AGVSs

Assumptions are sometimes made in simulation studies to simplify the modeling process. Sometimes these assumptions are valid, but sometimes they result in an inaccurate model that fails to reflect the true nature of the system. The following assumptions are sometimes made in simulation studies of AGVSs [29]:

1) AGVs do not pass each other.
2) Acceleration and deceleration are ignored.
3) Empty AGV travel is not accounted for.
4) The number of AGVs in the system is fixed.
5) Blocking time is assumed to be zero.
6) Load splitting is not permitted.
7) Travel times are based on shortest-route distances.
8) Track layout is fixed.
9) Guidepath direction is fixed.
10) P/D point locations are fixed.

1.3.4 Objectives of AGVS Simulation Studies

Most simulation studies are conducted to determine the values for the set of operational variables that will result in the most efficient operation of the system. Since analytical methods are unable to provide a true "optimal" solution, the studies are performed by simulating the AGVS under a variety of operating conditions and searching for the combination of values that is best for that system. Depending on the manufacturing process and the relative value of the system's resources, what defines the best combination will vary among AGVSs. Some of the most common objectives of AGVS simulation studies are [29]:

1) Find the minimum number of AGVs required.
2) Minimize total travel of vehicles.
3) Determine the best layout.
4) Determine the best AGV routing schedule.
5) Maximize the number of loads delivered over time.
6) Determine the effects of blocking.
7) Evaluate vehicle dispatching rules.

1.3.5 Use of Random Variates in AGVS Simulation

In order to properly model the random nature of many of the functions in an AGVS, a simulation tool must provide the modeler with a choice of standard probability distributions [14]. Following is a list of standard probability distributions used in AGVS simulation [13]:

1) Beta
2) Binomial
3) Erlang
4) Exponential
5) Gamma
6) Lognormal
7) Normal
8) Poisson
9) Triangular
10) Uniform

2. LITERATURE REVIEW

Because of the growing popularity of using AGVs for material handling in manufacturing operations, there has been a great deal of research done in the area of computer simulation of AGVSs. There are many different tools available to the prospective AGVS simulator and many techniques have been developed to simplify the process. Many systems have been successfully modeled and simulated to aid in their development and modification. The following sections describe the research that has been done.

2.1 AGVS Simulation Packages

The different types of packages for simulating AGVSs can be broken into four categories [11]:

1) Manufacturing simulators allow the user to simulate an AGVS with little or no programming [13].
2) Some general purpose simulation languages are general in nature but have special features to simplify the modeling of AGVSs [13].
3) Some packages contain a set of specialized simulation functions written in a general purpose programming language. These packages are a cross

between type 2 above and type 4 below. The user can call the simulation functions to carry out any simulation tasks and still retain the power of the general purpose programming language [11].

4) General purpose programming languages require that the user do all the programming required to perform a simulation [11].

Any of these 4 types of packages could be used to simulate AGVSs, but there are tradeoffs that make some more desirable than others. Manufacturing simulators are the easiest to use and yield the fastest results, but they are least flexible and unable to model complex systems [11]. Using a general purpose simulation language with an AGV extension package, like SLAM II by Pritsker [21], gives the user the ability to model most AGVSs. Using a general purpose programming language gives the user the full power of the language without restriction but in return the user must provide all of the software required for the simulation functions and the AGVS modeling [11].

Because of their utility and power, several type 3 packages have been developed. Khan [11] gives an exhaustive list of 19 such packages based on various general purpose

programming languages, including CBST, a package that he developed that is based on the C programming language.

CBST is a group of simulation support functions that supports discrete, continuous and combined simulation. CBST is designed to offer the advantages of both a general purpose simulation toolkit and the C general purpose programming language. Although the scope of CBST is broader than the objectives of this research, many of the programming concepts and executive simulation routines presented in CBST are used to form the basis of discrete next-event simulation for this framework. CBST is thoroughly described in Chapter 3.

In addition to CBST, DISC and CSIM are type 3 packages that are C-based. DISC, developed by Selvaraj, et al. [26], is a "library of C functions" which supports discrete event simulation using a next event approach. CSIM, developed by Schwetman [22,23] has C functions that enable the user to perform simulation in C using the process world view [11].

Dutt [6] developed a package called GVSim as "a tool for evaluating various guided vehicle systems". It is a library of C subroutines that enables the user to input some of the operating parameters, simulates the AGVS and provides output

reports. GVSim uses a new route selection scheme called the quickest path concept. This concept takes the shortest path and checks it against scheduled AGV traffic to compute the travel time after resolving conflicts at intersections. Alternate routes are checked to find the quickest path.

GVSim offers the user some flexibility in changing the AGVS's operating parameters, but it falls short of providing a truly flexible tool kit that can be used to simulate the range of complex systems and operating rules in use today. Because Dutt also used a discrete, next-event simulation approach, some of the concepts used in GVSim were incorporated into this framework. In particular, his method of entering AGVS layouts was used as a starting point for this framework's data entry routines.

De Meter [5] developed GIBSS (Generalized Interaction Based Simulation Specification), a simulation framework that supports simulation of systems with differing levels of modeling detail between system elements. GIBSS achieves modularity by classifying the model in terms of three classes of sub-models. Each sub-model can be simulated independently or assembled as appropriate with other sub-models.

Law and Haider [13] conducted a survey of two simulation languages (AutoMod and SLAM II) and six manufacturing simulators (FACTOR, MAST, PROMOD, SIMFACTORY, WITNESS and XCELL+) that have special accommodations for the modeling of AGVSs. They also offer a list of six groups of desirable features for simulation of manufacturing systems, including modeling flexibility, ease of model development and availability of debugging aids.

2.2 AGVS Simulation Techniques

2.2.1 AGVS Development

Ulgen, Onur and Kedia [29] present a hierarchical taxonomy for use in developing an AGVS and then present the results of a simulation study of a cellular assembly system that was done using SLAMSYSTEM software. They conclude that the decisions made during the design process of an AGVS are key to preventing schedule-related problems. In fact, they found that a good design will perform well under a variety of scheduling rules.

Goetschalckx and McGinnis [8] present the concept of an engineering work station (EWS) that could be used "to analyze and simulate the AGVS design directly from the

layout drawing." The tool they propose uses a six step interactive process to design the system and would use simulation to define such operational parameters as the number of AGVs and dispatching rules.

Bozer and Srinivasan give the advantages and disadvantages of using tandem loop configurations in AGVSs. They found that tandem configurations have the following benefits; they use simple vehicle dispatching rules, there is no blocking, the same control system can be used in each loop, they are easily expanded, and they take advantage of the flexibility of bi-directional AGVs in the loop. The limitations of tandem loop configurations are; a load may have to be handled by several AGVs to reach its destination, more space is required, they are less tolerant to AGV breakdown and such configurations require balanced workloads among loops to prevent bottlenecks. Although Bozer and Srinivasan presented a numerical analysis to support their claims, they state that further research will be required using simulation to compare tandem layouts with conventional uni-directional and bi-directional layouts [2].

2.2.2 AGVS Modeling

Davis [4] presents a thorough discussion of the issues involved in modeling AGVSs. She gives methods for modeling difficult system characteristics using general purpose simulation languages, such as SIMAN, without significantly degrading the model's performance.

Vosniakos, Davies and Mamalis [30,31,32] categorize quantities which can be monitored in the simulation of an FMS as either AGV-centered (such as AGV utilization, idle time, blocking time at intersections and track utilization) or FMS-centered (such as FMS output, work station utilization and buffer levels). Vosniakos and Davies present the results of a simulation study of an FMS programmed in ECSL, a simulation language based on activity coding. They examine the effects of different scheduling policies for various uni-directional and bi-directional layouts with one, two and three AGVs in the system. They conclude that simulation is vital in determining the outcome of changing AGVS operating parameters.

2.3 Simulation Studies of AGVSs

Because the complexity of AGV systems make simulation the only analysis tool feasible for their study, there are many such studies available in the literature. Some have been performed to see the results of changing operating parameters in existing AGVSs, while others compare the relative merits of more general approaches such as new conflict resolution algorithms or path layouts.

Eade [7] explains how simulation is used in a Lycoming gas-turbine engine plant "to test the operational viability of possible paths, operating loads, and interference events." The plant uses seven AGVs and is part of the Army's effort to modernize defense related industries.

Lee used SIMAN to create a discrete event simulation of an AGVS to study the effects of varying the number of AGVs, AGV speed and inter-arrival time of loads entering the system [15]. Other operational parameters, such as layout and dispatching rules were held constant. The results of this study were that the AGVS was most sensitive to changes in load inter-arrival time.

Prasad and Rangaswami [20] conducted a simulation study using AutoMod on two very different control logic methods for an AGVS supporting an integrated circuit manufacturing operation. They compared the results of using a global control system against those obtained from a local control system ("taxicab mode") while the other operational parameters are held constant. These results were used to choose the control strategy for that operation.

Petkovska, et al. [19], present a simulation study of a printed circuit board manufacturing facility supported by AGVs. The simulation was written in GPSS-F and was used to determine "optimal" parameters for input and output buffer size, the number of AGVs and the AGV routing scheme.

Cheng [3] wrote a simulation in PASCAL to study the relative merits of five different AGV dispatching rules in an AGVS that supports an FMS. The five dispatching rules used are:

1) First available AGV (FAFS).
2) AGV with most idle time (MIT).
3) AGV with least idle time (LIT).
4) Closest AGV (SRD).

5) Longest distance (LRD). (This rule is not used in industry, but it is used here as a benchmark.)

He found that the dispatching rules based on logical and distance-based rules (FAFS, and SRD) consistently out-performed AGV-based rules (MIT, LIT and LRD). Surprisingly, LRD sometimes gave better results than both MIT and LIT.

3. REVIEW OF CBST

The discrete, next-event simulation functions used in AGVSF, the framework developed in this research, are based on the discrete simulation portion of Khan's CBST. These are a group of functions designed to offer the advantages of both a general purpose simulation toolkit and the C general purpose programming language [11]. Although Khan describes the operation of the functions, structures and variables that make up CBST in great detail, a brief review of the discrete simulation portion of CBST and the CBST tools that are used in AGVSF is presented here for convenience. Where possible, CBST designations for structures, variables and arrays have been maintained for clarity and ease of reference between AGVSF and CBST.

3.1 Purpose

The purpose of the discrete, next-event simulation portion of CBST is to provide the simulation tools that the modeler needs to simulate the operation of any given model. The functions that make up CBST form a modular toolkit that the modeler can combine with his/her own functions to develop a flexible model which retains the flexibility and power of the C programming language.

3.2 Concept of Operation

CBST provides the basic functions, structures and variables required for discrete, next-event simulation, including; executive control, variable initialization, list management, entity management, resource management, random variate generation, data collection and reporting. The simulation involves entities that are defined by attributes and processed by resources according to time-events that are defined by the modeler. Entities are created as required and they capture and release resources as they undergo processing. Lists represent queues for resources or other groupings of entities and events are scheduled on the event list. Once the simulation variables are initialized, a four step cycle is entered that carries out the simulation of the model until the stopping conditions are met.

The first step of the executive control loop is to check the stopping conditions that have been specified by the user. Because these conditions are user-specified, they can represent any state of the simulation model, including simulation time, number of entities processed, the value of a given variable or the length of a queue.

If the stopping conditions are not met, the next event is removed from the event list and the simulation time is advanced to the time of that event's occurrence. The entity involved in the event is passed to the correct event routine based on the event type. The event routine carries out the logic for processing the entities according to the logic that the user builds into the model. The event routines can call any of the functions in the CBST toolkit to accomplish this, as well as any user-written functions that describe the operation of the simulated system.

Once the event routine has been completed, the cycle begins with checking the stopping conditions. Once they are met, control is returned to the calling function, **main()**. This is a simple but powerful engine that runs the simulation and it gives the user a great deal of flexibility in modeling a system.

3.3 Structures and Variables

CBST defines many structures and variables that it uses to manage the entities, resources and events in the simulation and to provide interface with the executive control routines described above. Several of these

structures and variables are used extensively in this framework's functions to bridge the gap between modeling an AGVS and using the tools available in CBST.

Attributes of resources, regardless of type, that are required for resource management are kept as elements of a CBST defined data structure of type **resnote**. These structures are kept in an array of structures designated **rn[NRES]**, where **NRES** is the number of resources in the simulation model as defined by the user. The elements of the **resnote** data structure include a pointer to the resource's name, the number of units of capacity that the resource has, and several elements that indicate the present status of the resource and the history of the resource's utilization. These elements are used by other CBST functions to generate utilization statistics for the resource. In order to integrate the **resnote** structure into this framework, elements identifying the resource type and a pointer to the data structure that holds the individual attributes for the specific link, AGV, or buffer that this **resnote** structure represents have been added.

The information that CBST needs to manage a list is kept in a data structure of type **listnote**. The elements of the **listnote** data structure include a pointer to the list's

name, variables holding the maximum and current lengths of the list and structures that hold both time persistent and observation-based statistics for the list. These elements are used by other CBST functions to generate utilization statistics for the list. All of these data structures, except the one that is used for the event list, are kept in an array of structures designated **lb[NLISTS]**. The dimension of the array, **NLISTS**, is the number of lists in the simulation model as defined by the user. The **listnote** data structure for the event list is named **evb** and is maintained separately from the other lists.

A separate data structure is available to hold the data required to gather statistics for time persistent and observation based variables -- **tpvar** and **obsvar**. Because the values of time persistent variables must be tracked continuously over the time of the simulation, **tpvar** has elements that hold data on the area accumulated over time and the time of the last changes in value. The elements of **obsvar** include the mean value of the observation-based variable, the number of observations, and the sum of the observations.

Event notices are kept on the event list in data structures of type **element**. These structures are created

when an event is scheduled and the memory for them is freed during the event routine. The elements of **element** are those required to order the events properly in the doubly linked event list. These include a pointer to the entity involved in the event, the time of the event and pointers to the **element** structures that precede and follow the event.

It is up to the modeler to define the attributes that are required to describe the entities in the system through the use of the data structure **entity**. CBST's only requirement is that an array of parameters that can be used to rank the entities in lists be included as an element of the **entity** structure. This array is called **r[]** and its dimension is specified by the user.

The time for the simulation time clock is kept as the value of the global variable **tnow**. It is initialized by CBST's initialization routines and its value is advanced as events are removed from the event list.

3.4 Functions

The CBST toolkit has scores of functions available to carry out discrete simulation. Those high-level functions

that have been used extensively in AGVSF are described below.

3.4.1 Executive Control Functions

The executive control cycle that was described in Section 3.2 is carried out by calling the following CBST functions. **discrete()** is called from **main()** to begin the simulation and, in turn, calls **discpinit()**, **uinit()**, **stopcheck()**, **nextevent()** and **tevents()**. **discpinit()** and **uinit()** are used to initialize CBST and user variables, respectively. **stopcheck()** is written by the user to describe the stopping conditions for the simulation. **nextevent()** removes the next event notice from the event list and **tevents()**, another user-written function, is called to determine the proper event routine to send the event's **element** structure to carry out the logic of the model.

3.4.2 List Management Functions

CBST provides several functions that are very useful in managing lists of entities. Entities can be filed on a list in the proper order by calling one of four filing functions; **filefifo()**, **filelifo()**, **filelvf()** or **filehvf()**. These functions file the entities in first-in/first-out, first-

in/last-out, lowest-value-first and highest-value-first order, respectively. Entities are removed from a list by using either **removelst()**, **removenum()** or **removespec()** to remove the first entity, the entity with a given rank number or the entity with a given value of **r[0]**, respectively.

The only CBST function that is used directly for management of the event list in AGVSF is **schedule()**, which calls a series of lower level CBST functions to place an event notice on the event list.

3.4.3 Entity Management Functions

The function **makeentity()** allocates space for a data structure of type **entity** in the heap. **makeentity()** is called when an entity is created (enters the system). When the entity is destroyed (leaves the system), the C function **free()** is called to release that memory for other uses.

3.4.4 Resource Management Functions

The following CBST functions are used by AGVSF to manage the capturing and releasing of units of resource by entities during the simulation. **rescapture()** seizes a specified number of units of a resource for an entity and **resrelease()**

is used to release the units of resource once processing is complete. **resavail()** can be used in decision-making branches to determine whether or not any units of a resource are currently available. **resincr()** and **resdecr()** can be used to increment and decrement the number of units of capacity of a resource.

3.4.5 Random Variate Generation Functions

CBST has functions which generate random numbers and return samples from 13 different distribution functions. The high-level functions that are used in AGVSF are **expon()**, **uniform()**, **erlang()**, **weibull()**, **beta()**, **gamma()**, **normal()**, **gaussian()**, **lognormal()**, **triang()**, **bernoulli()**, **binomial()** and **poisson()**.

3.4.6 Data Collection and Reporting Functions

There are several CBST functions for data collection, computation and reporting that are very useful to the modeler for analyzing the results of the simulation of the system being studied. **notevalue()** is used to sample the value of an observation-based variable. **accumarea()** is used to collect the value of a time persistent variable over simulated time. **clrov()** and **clrtp()** are used to clear

observation-based and time persistent variable values, respectively. Similarly, **clrlist()** and **clrres()** clear the statistics for lists and resources. **rlist()**, and **rresource()** print current statistics for specified lists and resources, respectively, and **rlhead()** and **rrhead()** can be used to print headings for list and resource statistics. CBST supplies functions to print reports for resources and lists, **repres()** and **replist()**. These reports can be used by the modeler or modified to report statistics of special interest in analyzing the simulation model.

4. REVIEW OF AGVSF

4.1 Purpose

The purpose of AGVSF, the automated guided vehicle simulation framework created in this research is to provide the user with a flexible simulation tool for modeling and analyzing AGVSs. It gives the user an environment that simplifies the task of representing the attributes and parameters of an AGVS without forsaking the modularity, power and speed inherent in the C programming language.

4.2 AGVSF Concept

AGVSF provides all of the tools required to perform discrete, next-event simulation of an AGVS model, including; executive control, initialization, list and entity management, random variate generation, data collection and reporting, AGVS input and event routines [11]. The basic tools for discrete, next-event simulation are based on CBST, the C-Based Simulation Toolkit developed by Khan. Because CBST was designed to support discrete, continuous and combined simulation, it was modified significantly to extract the portions necessary for discrete simulation.

The philosophy used in designing AGVSF was to attempt to anticipate every possible need that a user might have in modeling and simulating an AGVS, from the simplest models to the most complex, and provide for them in the framework. The advantage of this approach comes from the flexibility it allows the user in deciding on the level of complexity to be used in the simulation model and to use only those tools that are necessary to achieve that level. Because a modular approach was used in designing AGVSF, the user can assemble the tools necessary for the job and replace framework-supplied functions and routines with those designed by the user when necessary.

The 81 functions comprising approximately 8000 lines of source code that make up AGVSF can be divided into three categories, based on the user's need to make changes to simulate an AGVS. The first category consists of functions that will rarely or never be modified by the user. These are such functions as executive control and random variate generation. These functions remain the same, regardless of the AGVS being modeled, and should not need to be modified.

The second category of functions are those that may be modified by the user if necessary to accurately model and analyze the operation of a specific AGVS. Examples would be

the report generation functions and the functions used to perform AGV routing and allocation. The functions provided by AGVSF for these tasks are general in design, but it is anticipated that they may be routinely modified by the user.

The third category of functions are those that will be supplied by the user and are not included in the simulation framework. An example of such a function is one that models the operation of individual workstations supported by the AGVS. While the purpose of AGVSF is to model the movement of AGVs and loads between various points in the AGVS, the user is free to add the necessary code to model the processing of the loads at individual workstations.

In order for a user to be able to make such changes and additions, AGVS has been developed using well-defined, modular functions. By designing functions to perform a specific purpose and by clearly defining the parameters and variables involved, the user is able to make changes to the model by substituting in functions and algorithms without making major changes to the framework.

Another concept that is fundamental to the design of AGVSF is that there are different levels of simulation modeling complexity that may or may not require the use of

all of the tools available in the framework. A basic AGVS model may require only event routines that model the movement of loads and AGVs in the system. A more complex model may include battery discharging and recharging for the AGVs, as well as AGV breakdown and repair. The next level of complexity may include the breakdown and repair of individual workstations or the modeling of other material handling systems that support the workstations. These different levels of simulation complexity can be easily accommodated by using the modular approach described above.

4.3 AGVSF Structure

4.3.1 Data Structures

AGVSF is designed to simulate the flow of loads and AGVs in the AGVS. It does this by storing the attributes of the AGVS in data structures and then using event, list and resource management functions inside event routines to simulate the operation of the AGVS.

Loads are represented by entities, which are created, modified and destroyed as called for by the event routines that model the AGVS. The attributes of the entities are stored as elements of data structures of type **entity**. These

attributes include the load number, load type, arrival time into the AGVS, location, processing status and the identity of an AGV that has been allocated to it, if any.

Since all loads of the same type have some common attributes, these shared attributes are stored as elements of a separate data structure of type **load_type**. These attributes include load interarrival distribution, load arrival location, load departure location, processing sequence and whether or not the load is restricted to certain AGVs. **load_type** structures are stored in an array of dimension defined by the user.

The AGV path is represented by a series of links which are separated by control points. Link attributes are stored in data structures of type **link**. Elements of **link** structures include the identity of the points that make up the link's endpoints, the links that can be travelled to from each endpoint, the link length, whether the link can support unidirectional or bidirectional AGV flow, and how many AGVs can occupy the link at once. A speed reduction factor is present to allow for slower AGV speeds if the link is curved or congested. **link** structures are stored in an array of dimension defined by the user.

Control point attributes are stored in data structures of type **point**, which have elements that include the type of control point, (intersection, pickup and drop-off, parking, recharge or communication) the links that connect to that point, the recharge capability of the point and the identity of any buffers at that point. **point** structures are stored in an array of dimension defined by the user.

Buffers represent the queues where loads arrive into (input buffers) and depart from (output buffers) the AGVS. At these buffers, the AGVS interfaces with the systems that it supports. Loads arrive at an input buffer when they enter the AGVS and depart the AGVS at output buffers for processing at workstations. When the processing is complete and ready for transportation by an AGV, the load reenters the AGVS at another input buffer and the process continues until the load departs the AGVS when all processes have been completed. Buffer attributes are stored as elements of data structures of type **buffer**, and include the name of the buffer, the type of buffer (input or output), the buffer capacity and the point associated with the buffer. **buffer** structures are stored in an array of dimension defined by the user.

AGV parameters are stored in data structures of type **agv_indiv**, and as was done with common load attributes, common attributes of an AGV type are stored in data structures of type **agv_type**. Elements of **agv_indiv** include the AGV name and type, the AGV's location, the next and final destination of the AGV, the action to be taken when the AGV reaches its final destination, the loads that the AGV is allocated to transport, the charge level of the AGV and the status of the AGV (empty, assigned, operational, etc.). Elements of **agv_type** include empty and loaded speeds, acceleration and deceleration profiles, AGV length, separation distance required between AGVs, the number of loads an AGV can carry, loading and unloading times, the battery discharge and recharge profiles and the probability distributions for failure rate and repair. **agv_indiv** and **agv_type** structures are stored in arrays of dimensions defined by the user.

Many AGVS control schemes use information about the status of the supported workstation to make decisions about the transportation of loads in the AGVS. For this reason, AGVSF includes a data structure of type **station** to store the attributes of the workstation that are required for resource and management. These attributes include the station name, capacity and index numbers of the associated input and

output buffer. Clearly, these attributes would not be sufficient to model the operation of a complex FMS cell that is made up of several discrete machines and material handling devices. If the user wishes to simulate the operation of station components in detail, he or she can add the data structures and use the tools supplied by AGVSF to do so. **station** structures are stored in an array of dimension defined by the user.

4.3.2 Resources

As with many other simulation tools [21], assets with limited capacity that are used by the entities in the simulation model are represented as resources in AGVSF. Units of resource must be seized before they can be used by entities or other resources. When the resource is no longer needed, the units of resource are released. In this framework, links, AGVs, buffers and stations are represented by resources. It is also possible for the user to represent individual workstation components as resources if such a level of modeling complexity is required.

Attributes of resources, regardless of type, that are required for resource management are kept as elements of data structures of type **resnote**. These elements include the

resource name, capacity, utilization statistics, resource type and a pointer to the structure that holds the attributes for the link, AGV, buffer or station that this **resnote** structure represents. **resnote** structures are stored in an array of dimension defined by the user.

AGVSF uses a collection of resource management functions from CBST. These functions are called from event routines and during initialization, data collection and reporting. They enable the user to check the availability of a resource, capture units of a resource, release units of a resource, increase or decrease the number of units of a resource and collect and report time persistent statistics on resource utilization.

4.3.3 Event Scheduling

In discrete, next-event simulation, an event is defined as an instantaneous occurrence that may change the state of the system [13]. AGVSF uses event management routines from CBST to schedule events on the event list, which is a doubly linked list that keeps events in order of occurrence [11]. A data structure of type **element** is used to store the attributes necessary for event management, including pointers to the **elements** before and after it on the event

list, the event type, the time of event occurrence, the time the event was placed on the event list and a pointer to the entity involved in the event.

The information needed to manage the event list is kept in a data structure of type **listnote** [11]. The elements of this structure include the name of the list and both time persistent and observation-based statistics for the list. The event list structure **listnote** is maintained separately from the other lists, which are discussed below.

4.3.4 Lists

There are other lists used in AGVSF besides the event list. Such lists represent queues, such as loads awaiting transportation, AGVs waiting to seize a unit of link resource, or a load waiting in an output buffer for a station to become available. Unlike the event list, which orders the elements by time of occurrence, the user can specify the priority to be used in placing a load or an AGV on a list, such as first-in/first-out (FIFO), last-in/first-out (LIFO), lowest value first (LVF) or highest value first (HVF). This ordering priority is specified when the user calls the list management function from an event routine or initialization function.

As with the event list, the information needed to manage all other lists is kept in a data structure of type **listnote**. Unlike the **listnote** structure for the event list, data structures for other lists are kept in an array of **listnote** structures of dimension defined by the user, one for each list used in the model. The **listnote** structures are referenced by a pointer to the structure in the array, allowing the list management functions to add and remove items from the list and assemble statistics on the list.

4.4 Event Routines

4.4.1 Basic Event Routines

AGVSF includes five event routines that are basic to even the simplest AGVS simulation models. These event routines are described below.

Event 1, Load Enters the AGVS - the structure of type **entity** is initialized; the next entry event of a load of this type is scheduled; event two, ("load arrives at an input buffer") is scheduled for the current time.

Event 2, Load Arrives at Input Buffer - the status of

the input buffer and the load are updated; the AGV allocation function is called; if an AGV is assigned, the route request function is called; the dispatch function is called; the next link in the path is seized or the AGV waits in queue for it; if the link is seized, the event "AGV arrives at a control point" is scheduled for this AGV.

Event 3, AGV Arrives at a Control Point - the status of the load (if any) and the AGV are updated; if this is not the AGV's final destination, the routing request and dispatch functions are called; if it is, the appropriate action is taken as determined when the AGV was allocated. The action taken will result in another event being scheduled or in the AGV being placed on a list waiting for a unit of resource to become available.

Event 4, AGV Receives Load - the status of the load, the AGV and the input buffer are updated and a function is called to take action based on the changed status of the input buffer; the routing request and dispatch functions are called for disposition of the AGV.

Event 5, Load Arrives at Output Buffer - the status of the load, the AGV and the output buffer are updated and a function is called to take action based on the changed status of the output buffer; an AGV-initiated assignment request function is called to determine the disposition of the AGV that finished off-loading.

4.4.2 Complex Event Routines

AGVSF includes four event routines that would be used in more complex AGVS simulation models:

Event 6, Low Battery - the status of the AGV is updated and a function is called to determine the disposition of the AGV, such as dropping off any current load at its final destination and proceeding to the closest recharging point.

Event 7, Battery Charging Complete - the status of the AGV is updated and an AGV-initiated assignment request function is called to determine the disposition of the AGV.

Event 8, AGV Breakdown - the status of the AGV is updated, a function is called to determine the disposition of any loads on the AGV and an "AGV repaired" event is scheduled based on the repair profile for this type of AGV.

Event 9, AGV Repaired - the status of the AGV is updated to reflect its operational status and the next "AGV breakdown" event for this AGV is scheduled; an AGV-initiated assignment request function is called to determine the disposition of the AGV.

4.4.3 User-Provided Event Routines

There are other event routines that are provided by the user of the framework as required. An example would be an event routine which clears statistics for lists, resources and other time persistent and observation-based variables. The user would schedule this event as necessary to collect data at specific times or conditions for analysis of the AGVS. Other routines might be included for modeling other systems that interface with the AGVS, such as workstations that process loads or other material handling systems that transport loads at these workstations. Specifically, the

user may wish to model the breakdown and repair of either whole workstations or parts of workstations in a manner similar to event routines 8 and 9 above. The modular nature of AGVSF simplifies the addition of these event routines.

4.5 AGVSF Use

In order to use AGVSF to model an AGVS and to conduct a simulation study of its operation, the user must be prepared to modify the framework code where necessary and to add functions to increase the level of modeling complexity if desired. It is assumed that the user has a working knowledge of the C programming knowledge.

4.5.1 User-Written Code

As discussed in Section 4.3, the user may want to write additional event routines in order to generate additional output statistics or to expand the simulation to include other systems that interface with the AGVS. Additionally, the user must write a **stopcheck()** function to define the stopping conditions for the simulation. AGVSF calls this function before each event is removed from the event list to determine whether or not the stopping conditions, such as the elapsed simulation time or the number of loads that have

entered the AGVS, have been reached. Simple examples of this function are provided in AGVSF.

4.5.2 Framework Modifications

Before running the simulation, the user must modify the framework code to reflect the operation of the AGVS. The most common changes will be those made to the event routines and to the algorithms and functions that are used in the event routines that carry out the control logic of the AGVS model. Examples of such functions are the AGV allocation function, the routing request function, the dispatch function, and functions used to calculate the discharging and charging of AGV batteries. The functions provided with the framework are intended to serve as general examples of those routines that are in use in AGVSs. The user must also ensure that the initialization function is written to schedule the first event of each type at the proper time. The user can also modify the function **main()** to make repeated simulation runs.

Additionally, it is important that the user review the define statements used in AGVSF to ensure that the values given are large enough to accommodate the AGVS to be modeled but not so large that they take up needless computer memory.

This is especially true of the constants that are used to dimension arrays of structures for storing AGVS data, since many of these structures are large.

4.5.3 AGVS Initialization

Before the user runs the framework code, the attributes of the points, links and buffers that make up the AGVS layout must be carefully determined. The same process must then be done for the load types and AGVs that are in the AGVS. These attributes will be needed in the initialization portion of the framework to load the data structures that describe the AGVS's layout, resources and entities. Each point, link, buffer, load type, AGV type, individual AGV and station should be assigned an index number that can be referenced when describing the interrelation of AGVS components. The random variate distributions used to model interarrival times, AGV breakdown and AGV repair should also be selected.

The initialization function in AGVSF asks the user if the AGVS data will be entered manually or from existing files. The first time an AGVS is modeled, the user selects the manual mode and the framework prompts the user to enter the attributes of each item that comprises the AGVS. After

the AGVS data has been entered once, the contents of the data structures that describe the AGVS are stored in files with the file extension **bin** -- one file for each structure or array of structures. If more than one AGVS will be modeled, the **bin** files are stored in separate directories.

After AGVS initialization, the first events are scheduled on the event list, using the event list management functions. Global variables and other structures used in the simulation are initialized to proper values.

4.5.4 Simulation

Once the initialization process is complete, AGVSF will proceed to perform the simulation of the AGVS until the stopping conditions are met. This is accomplished by using the same discrete event simulation process used in CBST [11]. The framework sets up an indefinite loop that first calls **stopcheck()** to see if the stopping conditions have been met. If they have not been met, the event management function **nextevent()** removes the next event from the event list and the function **tevents()** calls the proper event routine based on the event type. When the stopping conditions have been met, AGVSF returns to **main()** to end the simulation or to make repeated runs if the user has provided

for that option. Output reports are generated as called for in the event routines or in **main()**.

4.6 AGVSF Users

There are three categories of users envisioned for AGVSF. AGV designers would use it to evaluate new design parameters under a variety of simulated conditions. AGVS designers would use AGVSF to evaluate the relative merit of AGVS designs using all applicable combinations of operational parameters. Companies using AGVs in daily operations would use it to evaluate the results of changing the operational parameters of an existing AGVS to "optimize" or expand the system. In all cases, the flexibility of AGVSF and the power and portability of the C programming language would be very valuable.

5. AGVSF FUNCTIONS

AGVSF contains many functions which serve as tools for the modeler to use in simulating the operation of an AGVS. These functions fall into one of three categories:

1) Functions that are not modified by the user.
2) Functions that may be routinely modified by the user.
3) Functions that are supplied by the user, based on guidance supplied by the framework.

In the sections that follow, a brief description of the use and operation of the functions that make up the framework is presented.

5.1 Unmodified Functions

The largest group of functions that will not need to be modified by the user are the CBST functions that support discrete, next-event simulation that were described in Chapter 3. These functions form the basis of the framework's simulation capability and they do not need to be changed to accommodate changes in the AGVS being modeled.

The next major group of functions that do not need modification is the set of AGVSF functions that are used to

initialize the data structures that represent the AGVS. Because the data structures that hold this information were designed to be as robust as possible and to anticipate the need for all possible parameters that describe the AGVS, these functions should be able to model any AGVS without modification. If a modeler found that additional structures or parameters were required, they could be easily added by using the existing functions as examples.

There are five sets of AGVS initialization functions:

1) Manual initialization functions such as **init_point(), init_link(), and init_load_type(),** take keyboard entry from the modeler for the parameter values of every AGVS component and store the data in the structures that describe the AGVS. These functions prompt the user for each parameter value and retrieve the value from the keyboard.
2) Other functions read the AGVS data from binary files. These functions, **read_init_point(), read_init_link(), read_init_load_type(),** etc., read the data that describes the AGVS from unformatted binary files and transfer this data to structures. The user selects this initialization option from a prompt once the

AGVS data has been correctly entered manually once.

3) Another group of functions print the AGVS initialization data to the screen. These functions, **points_output()**, **links_output()**, **load_type_output()**, etc., take the initialization data that has been manually entered and output it to the screen one data structure at a time so that the user can immediately verify the accuracy of the data. Once this data has been verified, the user can choose not to call these functions.

4) AGVSF also provides functions that write AGVS initialization data to formatted files that can be examined by the user at a later time. As with the output functions described above, these functions, **fpoints_output()**, **flink_output()**, **fload_type_output()**, etc., can be bypassed once this data has been verified.

5) The last set of functions write the AGVS initialization data to binary files to be read by **read_init_()** functions. Once the AGVS data has been stored in binary files through the manual initialization process, the AGVS can be repeatedly simulated under any variety of

operating conditions. The binary files remain valid as long as the AGVS components and their parameters are unchanged.

Two AGVSF functions of this type provide support to event routines. **get_rv()** receives a pointer to a structure of type **distrib**, calls one of the twelve random variate distribution functions and returns the random value. It is used in event routines to supply values for entity interarrival times and AGV breakdown/repair times. **pause()** can be used anywhere in the simulation by the modeler to stop the simulation until the enter key is pressed.

5.2 User-Modified Functions

The largest group of AGVSF functions that will routinely be modified by the user are the basic and complex event functions described in Chapter 4. These functions represent the logic used to process entities and manage resources within the AGVS. The event routines that are provided in AGVSF are designed to represent the most general AGVS. The user will modify the branching code in the event routines to represent the logic used to model the AGVS.

Most of the AGVSF functions that support the event routines are of this type. **init_r()** initializes the elements in the array **r[]** for entities that are entering the AGVS. **arr_buffer_full()** is called from event routine #1 and takes the action required when the arrival buffer is full, such as not scheduling the arrival of the next load or exiting the simulation in the case of an input buffer with infinite capacity.

find_output_buffer() returns the index number of the output buffer where an entity receives its next process step. This is determined by logic in the function (such as a **switch** or other branching method) that determines the next output buffer based on the state of the AGVS and the load to be processed. Similarly, **dispose_load()** determines the disposition of a load that has arrived at an output buffer, such as departing the AGVS or scheduling event #2 for the load, **loadinput()**. **agv_job_req()** is used to determine the disposition of an AGV that is idle, based on the state of the AGVS and the queue of loads waiting for transportation, if any.

The functions **agv_release()**, **buffer_release()**, **link_release()**, and **station_release()** all interface between the AGVSF data structures and the CBST resource management

functions to perform the steps required to release units of resource that had been seized by entities. One of the primary duties of these functions is to take the action required based on the changed state of the AGVS caused by the release of these resources. For example, a load may be waiting in queue for an output buffer. **buffer_release()** removes the first entity from the queue for this buffer and calls **rescapture()** to seize the buffer for this new load. These functions also update the status of the entity and the AGV, link, buffer or station involved in the event.

AGV allocation, route requesting and dispatching are accomplished by the AGVSF functions **agv_alloc()**, **route_req()** and **agv_dispatch()**. These functions carry out the logic of the allocation, routing and dispatching algorithms used in the AGVS and therefore will be modified for each different AGVS. Each function updates the status of the AGV involved and calls the appropriate CBST resource management function. If an AGV is available for allocation, **route_req()** and **agv_dispatch()** are either called immediately or as soon as the AGV reaches the next control point. If no AGV is available for allocation, the load is put into queue for transportation. **find_travel_time()** is called from **agv_dispatch()** to calculate the time required for the AGV to travel to its next control point, based on the speed

parameters for the AGV and the link, as well as the AGV's acceleration and deceleration profiles, if they are used. This time is added to **tnow** to schedule the time of occurrence of event #3 for this AGV, **agvarrcp()**.

list_add() and **list_remove()** perform the necessary steps to add and remove loads and AGVs from queues and implement the CBST list management functions. These functions receive information that specifies the resource involved and its type (AGV, link, buffer or station) and make the change based on the ordering scheme used for that resource's queue. The user must modify the logic used in these functions to reflect that used for each resource in the AGVS that has a queue.

5.3 User-Supplied Functions

The user must supply any functions required to model the performance of systems supported by the AGVS, such as event routines for breakdown and repair of the stations or station components. These events may effect the processing of loads at the stations and hence the input and buffers that the loads go to for processing and transportation. Routines that model the processing of the loads at the stations may be required, such as **finprocess()**, which is used in the example models described in Chapter 6. Another option would

be to create an event routine called **load_arr_station()** which would take action to process the load once it arrives at a given workstation. The user will write **report()** and routines to clear statistics for system variables as required to get meaningful data to analyze the AGVS.

6. AGVSF EXAMPLES

Three different AGVSs were modeled using AGVSF. The purpose of these models is to validate AGVSF as a simulation tool and to demonstrate the flexibility of AGVSF in modifying one model to develop another. The three models described below are related and the AGVS in each becomes more complex with each version. Model 2 uses the same AGVS which Pritsker presented and simulated using SLAM II [21]. By comparing the values of key indicators of AGVS performance that result using AGVSF with those presented by Pritsker, the simulation accuracy of AGVSF can be verified.

6.1 Model 1

Model 1 is a simple AGVS that might be used as a first step by an AGVSF user in developing a working model for simulating a larger FMS. By starting with this simple AGVS, the modeler will have a base model with which to determine the system components and control logic that might be required for a larger model. Once this model has been simulated correctly with AGVSF, the modeler is ready to add those components and introduce the new control logic in a modular fashion.

6.1.1 Model 1 AGVS Layout

The AGVS layout for Model 1 is shown in Figure 6.1. It is comprised of two workstations connected by a one-way loop for AGV traffic. A two-way spur connects one cell to the loop and a siding is present to provide access to an AGV parking area at point CP4. There are a total of seven links and six points that make up this pathway. Points CP1 and CP6 are P/D points, where loads are transferred between the AGVs and the buffers. Buffers 1 and 3 are combined input/output buffers and buffer 2 is an input buffer.

6.1.2 Model 1 AGVs and Loads

The AGVS in Model 1 uses two identical AGVs to transport loads between the fixturing station and the machining station. They are unit load carriers with a capacity of one load, a loaded speed of 4 feet per second and an unloaded speed of 4.5 feet per second. Loading and unloading times for these AGVs are 45 seconds at all buffers. Acceleration, deceleration, battery charging, battery discharging and mechanical breakdown of the AGVs is not simulated in this model. Both AGVs start the simulation with an initial position at the parking area at CP1 on Link 4.

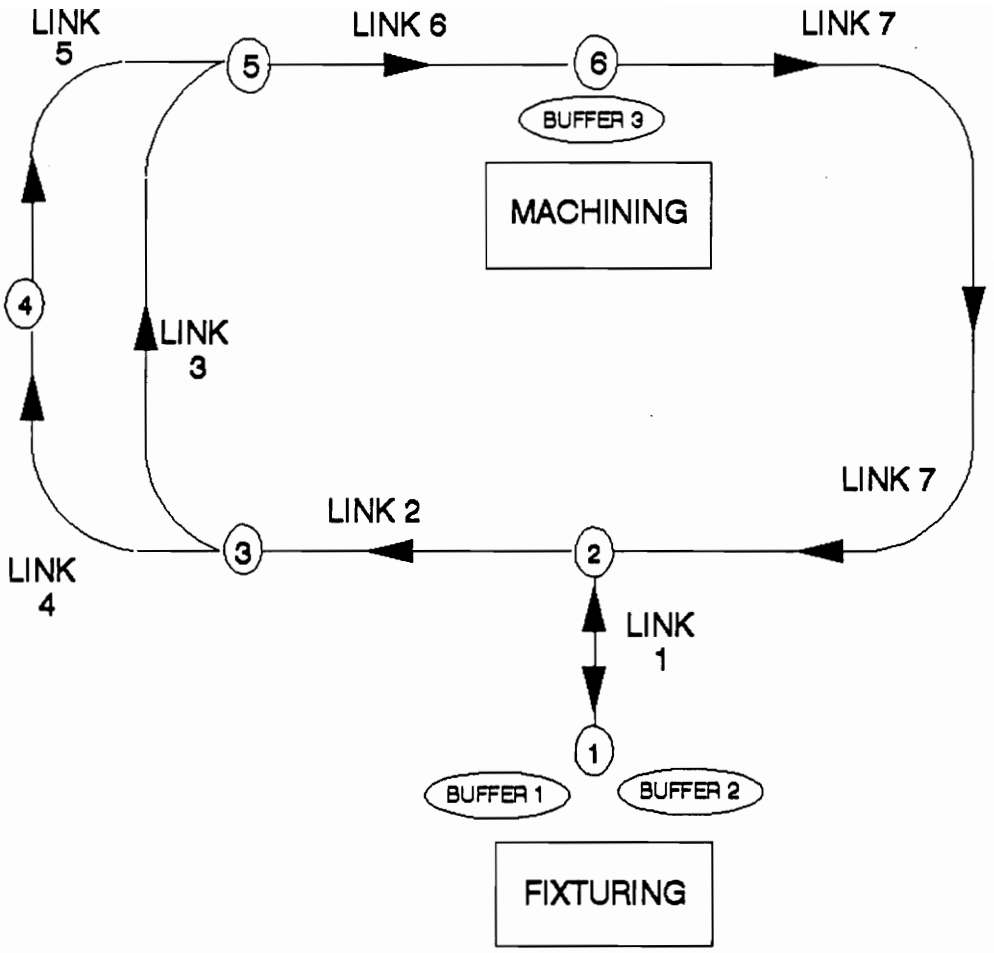

Figure 6.1, Model 1 AGVS Layout

The loads processed in Model 1 are all of the same type and arrive at Buffer 1 according to an inter-arrival distribution with a mean value of 436 seconds. Fixturing the loads takes 50 seconds and unfixturing requires 30 seconds. The machining time required for the loads is modeled by a triangular distribution with a minimum value of 300 seconds, a maximum value of 500 seconds and a modal value of 400 seconds.

6.1.3 Model 1 Operation and Control Logic

The loads to be processed enter the AGVS at Buffer 1, which has unlimited capacity. The fixturing station removes the loads from Buffer 1 and mounts them on a standard workholding pallet for shipment and processing at the machining cell. The palletized loads are stored at Buffer 2 until an AGV arrives at CP1 to pick up the load. Buffer 2 has a capacity of five palletized loads. Once loaded on the AGV, the load is transported from CP1 to CP6 by way of links 1, 2, 3 and 6.

AGVs are able to pass each other on the main loop of the AGVS pathway and on the siding for the AGV parking area. If one AGV occupies a position on Link 1, the other AGV must

wait at CP2 on Link 7 before it can travel on the spur to the fixturing station.

When the AGV carrying the palletized load arrives at CP6, it transfers the load to Buffer 3. Once the load is at the buffer, the AGV is released. The machining station then machines the load and sends it to the input portion of Buffer 3 to await transportation. When an AGV arrives at CP6 and the load is transferred from Buffer 3, the machining cell becomes available for another load.

The AGV transports the load by way of Link 7 and Link 1 to CP1, where the load is transferred to Buffer 1. The load waits in Buffer 1 until the fixturing station is available to remove the load from the fixture. Loads waiting to be unfixtured have priority over loads that are waiting for fixturing in Buffer 1. When the load has been removed from the fixture, it leaves the AGVS.

Several details of the AGVS control logic had to be reflected in the event routines and functions of the AGVSF model. The relationship between Buffer 2, the fixturing station and Buffer 1 is such that if Buffer 2 fills with loads waiting to seize the machining cell, a load that finishes fixturing will not be able to leave the station to

make it available to process another load -- the station will be blocked. Loads waiting in queue for the machining cell (Buffer 3 and the machining station) to become available cannot request an AGV until the cell is available. Loads that have returned to Buffer 1 to have the workholding pallet removed have priority over new loads waiting for the fixturing station, so loads should be filed on that list using a lowest value first (LVF) rule based on the time the load entered the system. If a faster AGV approaches a slower moving or stopped AGV along the main loop or siding of the AGVS pathway, it must be allowed to pass the slower AGV.

6.1.4 Modeling Model 1 With AGVSF Tools

6.1.4.1 Data Structures, Resources and Lists

The first step in simulating the operation of Model 1 was to identify the data structures, resources and lists that would be used to represent the AGVS components and operating logic. The parameters and inter-relationships of the points, links, buffers, stations, AGVs and loads were identified and listed to simplify the AGVS initialization procedure.

All links, buffers, AGVs and stations were identified as resources. The resource capacities of Links 2 through 7 were set at two and the capacity of Link 1 was set at one. This allows AGVs to pass on the main loop and the parking siding, but requires an AGV to wait at CP2 if Link 1 has been seized by the other AGV. The buffer capacities were set at 9999 (infinite), five and one for Buffers 1, 2 and 3, respectively. All stations and AGVs have capacity of one.

Five queues were identified that were represented as lists in the AGVSF model:

1) Loads waiting for fixturing.
2) Loads waiting to be transferred from the fixturing station to Buffer 2.
3) Loads waiting in Buffer 2 for the machining cell to become available.
4) AGVs waiting for Link 1 to become available.
5) Idle AGVs.

6.1.4.2 AGVSF Function Modifications

The user-supplied function **stopcheck()** was added to provide a means for stopping the simulation run after 89,400 seconds. The event routine **loadoutput()** calls the function **dispose_load()** to determine the proper disposition for a

load that arrives at an output buffer. Specifically, this function schedules a user-provided event called **finprocess()** based on the process to be performed and the time required to perform each of the three processing steps for the loads. The **finprocess()** event takes the appropriate action required when a load finishes processing at either of the stations. A function called **dispatch_rule()** was also added to determine if the load's next station requires that the cell be available before an AGV can be requested. (This function was probably not required for this simple model, but as more process steps and stations are added, it will be a useful tool.)

Several of AGVSF's functions had to be modified to reflect the control logic described in Section 6.1.3. The function **agv_job_req()** was modified to dispose of idle AGVs by sending them to the parking point at CP1. The function **route_req()** holds the information on how an AGV travels in the AGVS, so it had to be modified to reflect the paths between the control points. The method used was to create three two-dimensional arrays to store the index numbers of the next link, next point and final link for the AGV based on the value of the starting point and the final point to travel to. For this simple model, this approach was easier than developing a branching system using **if** statements or

other methods. The **agv_alloc()** function was updated with the index numbers of the **listnote** structures in the CBST array **lb[]**, which were assigned in the function **uinit()**. The arrival of the first load was scheduled in **uinit()**.

Most of the AGVSF functions that deal with list management had to be updated with the index numbers of the **listnote** structures as well. These include the functions **list_add()** and **list_remove()**, which call the CBST list management functions. The **list_add()** function specifies that loads be added to the queue for the fixturing station using a LVF priority based on entry time into the system and that all lists use FIFO priority. Additionally, code was added to perform the logical branching required to determine the list to be modified based on the resource type and resource index number passed to **list_add()**.

The functions **buffer_release()** and **station_release()** had to be modified because of the requirement for the entire machining cell to be available before a load can seize the machining station and request transportation to CP6. Loads are not removed from the machining station's queue until Buffer 3 is released and the cell is available. If Buffer 2 is released, **buffer_release()** calls **list_remove()** to remove

the first load in queue for it. Buffer 1 has unlimited capacity and so no queue is required for it.

The AGVSF functions **find_station()** and **find_output_buffer()** were modified by adding the code required to determine the identity of the station and output buffer where the next processing step will performed. Because there are only three processing steps and only one station that can perform each step, this was done with a simple **switch** statement.

The five basic AGVSF event routines -- **loadenters()**, **loadinput()**, **agvarrcp()**, **agvrecload()**, and **loadoutput()**, were all modified to reflect the operation of this AGVS. Most of these changes involved calling the AGVSF functions that carry out lower level branching and decision making. The **report()** event routine was scheduled to occur at 89,400 seconds and written to report the number of loads that were created and finished processing, as well as to provide standard output data on resource and list utilization.

6.2 Model 2

Model 2 is a group of machining cells supported by an AGVS that was presented and simulated by Pritsker using SLAM

II [21]. It expands on Model 1 but remains a fairly simple example of a possible application of AGVs as material handling tools. The differences between the two models and the AGVSF tools that were used to make these changes are discussed below.

6.2.1 Model 2 AGVS Layout

The AGVS layout for Model 2 is shown in Figure 6.2. It builds upon the basic layout of Model 1 by adding five more machining cells and a second loop to divide the AGV traffic among two sets of three cells. This expanded layout has a total of 15 links and ten control points. The only two-way traffic is still along the spur that connects the fixturing station with CP2. The AGV parking area remains at CP4 on Link 4. Each machining station has a combined input/output buffer associated with it located at the P/D point for that machining cell.

6.2.2 Model 2 AGVs and Loads

The two AGVs used in model 2 have the same parameters and operating characteristics as those used in Model 1. The loads that enter the system are of the same type and have the same inter-arrival distribution as in Model 1. The time

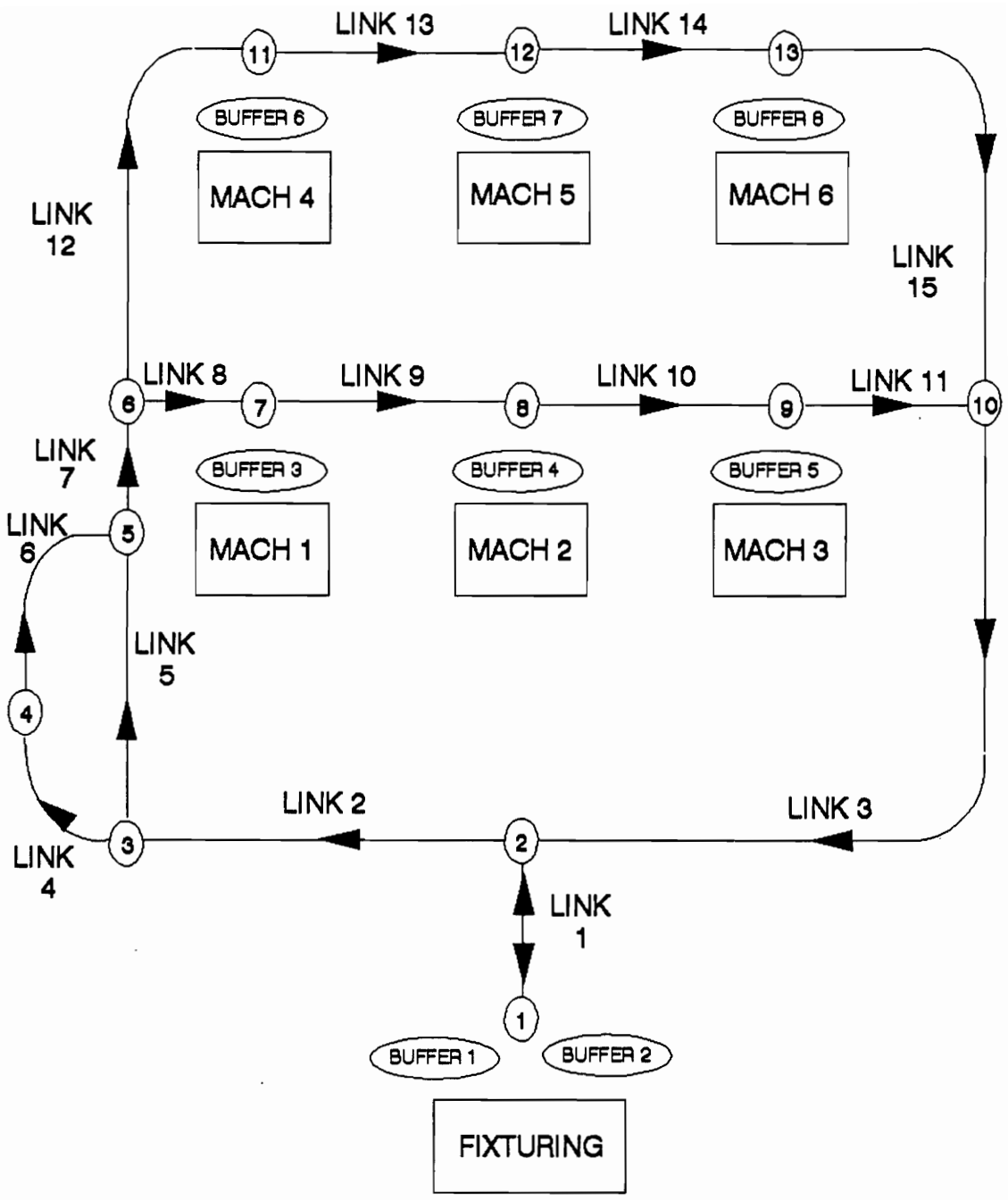

Figure 6.2, Model 2 and Model 3 AGVS Layouts

required to fixture a load is 220 seconds and the time for fixture removal is 180 seconds. The machining process is represented by a triangular distribution with a minimum value of 960 seconds, a maximum value of 2720 seconds and a modal value of 1860 seconds.

6.2.3 Model 2 Operation and Control Logic

The difference in operation between Model 2 and Model 1 is caused by the five additional machining cells. There is still only one machining process step to be performed and it can be performed at any of the six machining stations, MACH 1 through MACH 6. The policy that a machining cell must have both its buffer and machining station available before it can be seized by a load still applies. The priority of use in seizing an available cell goes to the cell with the smallest station index number. For example, if the cells containing MACH 2 and MACH 4 are both available, a load arriving in Buffer 2 will seize MACH 2 and then request an AGV for transportation to CP8.

These changes mean that one queue had to be modified and another added. The queue for the machining cell was modified to hold loads that are waiting in Buffer 2 for the next machining cell to become available, regardless of which

cell that is. If all cells are busy, up to five loads could be waiting in this queue. The new queue is required to hold loads that are waiting for transportation between CP1 and one of the machining cells. This queue could be occupied by up to five loads in one of two conditions. Loads could be waiting for an available AGV for transportation from CP1 to the P/D point for the machining station that the load has already seized. Loads could also be waiting for an AGV to become available for transportation from a machining cell to the fixturing cell. When an AGV becomes available, the first load in this queue is removed and seizes the AGV.

6.2.4 Modeling Model 2 With AGVSF Tools

6.2.4.1 Data Structures, Resources and Lists

Data structures were created for each additional point, link, station and buffer in Model 2. Inter-relationships between the various AGVS components were identified and written down for initialization of the AGVS data structure files. A total of 17 new **resnote** data structures were added to the CBST array **rn[]** during the initialization process; five stations, five buffers and seven links. The AGV queue was added as a **listnote** structure to the CBST array **lb[]**.

6.2.4.2 AGVSF Function Modifications

Several of the functions used in Model 1 were modified to develop Model 2. The most common changes were due to the addition of a list for the AGV queue. Code was added to the event **loadinput()** to add loads to this queue if none are available. The functions **uinit()**, **list_add()** and **list_remove()** were all updated with the index numbers of the listnote structures for this list. The **dispose_load()** function was changed to reflect the new times required for fixturing, fixture removal and machining of the loads. New cases were added to the function **dispatch_rule()** to identify the new machining cells as requiring the buffer and the station to both be available before the station can be seized. The function **agv_job_req()** was modified to check the new AGV queue before being dispatched to the AGV parking area. New two-dimensional arrays were added to **route_req()** to represent the routing information for AGVs in Model 2.

A new user-written function called **cell_avail()** was added to check the availability of each machining cell, starting at MACH 1 and proceeding to MACH 6. It identifies the index number of the station in the first available cell or notifies the calling function if no cells are available.

This function is called in the event routine **loadinput()** to identify the location where a load will be machined.

6.3 Model 3

The final example of AGVSF modeling introduces more changes to the operating characteristics and control logic of the AGVS simulated in Model 2. The differences between the two models and the AGVSF tools that were used to make these changes are discussed below.

6.3.1 Model 3 AGVS Layout

The AGVS layout for Model 3 is the same layout shown in Figure 6.2 that was used for Model 2. There are differences in the parameters of the AGVS components, however. Each machining cell buffer now has a capacity of five palletized loads instead of one. The fixturing station can now add or remove a workholding fixture from two loads at a time. Station MACH 6 has been changed so that it performs a rough grinding process that is different from the machining processes performed at the other machining stations.

6.3.2 Model 3 AGVs and Loads

The AGVs in Model 3 are the same as those used in Models 1 and 2 except that this model simulates the breakdown and repair of the AGVs. The time between AGV failures is modeled by an exponential distribution with a mean of 40,000 seconds. The repair of a broken-down AGV is modeled by a normal distribution with a mean of 300 seconds and a standard deviation of 10 seconds. This distribution represents the time required for maintenance personnel to replace or repair the AGV.

Model 3 has two types of loads that require transportation in the AGVS. The first load type is the same as that used in Models 1 and 2. The second load type has an interarrival time modeled by an exponential distribution with a mean value of 2000 seconds. It enters the AGVS at Buffer 1 and undergoes fixturing at the fixturing station just like the other load type, but it requires an additional machining step at MACH 6 before it can receive its final machining at one of the other machining stations. Load type one can be machined at MACH 1 through MACH 5.

6.3.3 Model 3 Operation and Control Logic

The operational changes introduced in Model 3 cause many changes in the control logic of the AGVSF model. The addition of a second load type and the distinction between MACH 6 and the other machining stations must be considered when routing loads to the next station for processing.

Increasing the buffer capacities of the machining cells means that loads can now request transportation to a machining cell when there is space in the input/output buffer. Queues are now required for each machining station for loads waiting in the cell's buffer for machining. One queue is used for loads waiting to seize a machining cell buffer if Buffers 3 through 7 are not available. Another queue is required for loads waiting to seize a unit of Buffer 8.

The logic for machining cell selection is now based on the number of loads in the cell's buffer, not the index number of the station. New restrictions have been placed on which machining stations can perform the machining steps.

The disposition of broken-down AGVs and the loads they carry must be considered, as well as the scheduling of the breakdown and repair events.

6.3.4 Modeling Model 3 With AGVSF Tools

6.3.4.1 Data Structures, Resources and Lists

The changes to the capacities of the fixturing station and Buffers 3 through 8 were made in the AGVSF model by reinitializing the data structures with the new capacity values. The AGVSF initialization functions automatically update the **resnote** structures with the new capacities for the resource associated with each station and buffer. Several **listnote** structures were added to the array **lb[]** to represent the additional queues required in Model 3.

6.3.4.2 AGVSF Function Modifications

Two event routines were used in Model 3 that were not used in the previous model -- **agvdown()** and **agvup()**. These event routines take the appropriate action when the event signifying an AGV breakdown or repair occurs. The user-written function **agv_break_down()** is called to perform the steps required to update the status of the AGV and its load

(if present) and to schedule the repair event for the AGV. The disposition of a travelling AGV is that it arrives at its next control point and link at the time of the **agvup()** event. The load stays on the AGV. A stationary AGV stays at the point where it broke down. The first failure for each AGV is scheduled in **uinit()** according to the given failure distribution. Subsequent failures for that AGV are scheduled in the **agvup()** routine.

The AGVSF functions that perform the branching required to allocate a machining cell for loads were all changed to reflect the processing required by the two load types. These functions include; **loadinput()**, **cell_avail()**, **station_release()**, **buffer_release()** and **dispose_load()**.

All functions that deal with list management were updated to reflect the additional lists in the AGVSF model. This included initializing the list names in **uinit()** and modifying the branching logic used in **list_add()** and **list_remove()**.

7. RESULTS AND CONCLUSIONS

7.1 Results From Example Models

7.1.1 Model 1

Model 1's operation was simulated ten times using ten different random number streams. The results show that the machining process for the loads takes too long to keep pace with the load type's interarrival time. This was to be expected, since the mean of the arrival distribution is 436 seconds and the mode of the machining distribution alone is 400 seconds. When the time for fixturing, transportation and fixture removal are added, it is obvious that the system will quickly fall behind in machining loads. The result is that Buffer 2 fills with fixtured loads, blocking the fixturing station.

After 89,400 seconds for each of the ten runs, an average of 203 loads had been created, but only 131 loads had completed processing. An average of 66 loads were left unfixtured in Buffer 1, waiting for the fixturing station to become available. The average utilization rates for the fixturing station and the machining station were 0.927 and 0.909, respectively.

The two AGVs were not as heavily used, with an average utilization of 0.322 for AGV 1 and 0.205 for AGV 2. The difference in the rates for the two AGVs is because there are only two AGVs and only two points from which a load can request an AGV. AGV 1 always transports loads from CP1 to CP6 for machining and AGV 2 always transports the load from the machining cell back to CP1. When the next load requests an AGV, AGV 1 is first on the list of idle AGVs, so this usage pattern is maintained for the entire simulation. Thus, AGV 1 always makes the longer trip around the main loop of the pathway to pick up a load and must make another full trip around it before it begins to travel back to the AGV parking area at CP4. AGV 2's journey is shorter, since it picks up the loads at CP6 and returns to CP4 after only one trip around the main loop.

7.1.2 **Model 2**

Model 2 was also simulated ten times using ten different random number streams. The results are consistent with the results presented by Pritsker after simulating the same AGVS using SLAM II [21]. Unfortunately, Pritsker only presents the results of one simulation run and the performance of this AGVS is very dependent on the number of loads that enter the system. Because of the exponential interarrival

distribution, there was significant variation in the number of loads entering the AGVS during different runs of the simulation.

Pritsker reports that 191 loads were created and 188 of them completed processing at the end of the 89,400 second simulation time. The same interarrival distribution produced an average of 205 loads created with a standard deviation of 11.18 and 195.7 loads processed with a standard deviation of 10.83 using the AGVSF model. In fact, only two of the ten runs had fewer than 191 loads created -- they had 187 and 189 loads enter the system, respectively.

Consequently, Pritsker's results show less congestion than the results of the AGVSF model. For example, Pritsker reports a mean time in system of 3744 seconds per load, compared to 3986 seconds for AGVSF. It is logical to expect higher utilization rates for the AGVs, stations and buffers as well. Pritsker's results show an average utilization rate of 0.817 for the AGV fleet. The AGVSF simulations yield an average of 0.877 for the two AGVs.

Pritsker identifies a potential bottleneck in the system at the fixturing station, due to an average queue length for this station of 2.09 loads and a standard deviation of 2.66.

AGVSF reports an average queue length of 2.81 and a standard deviation of 1.165. These results reinforce Pritsker's findings that a lack of availability at the fixturing station reduces the throughput of the system.

7.1.3 Model 3

Data collected from ten simulations of Model 3's operation show that increasing the capacity of the fixturing station helped reduce the bottleneck that was identified in Model 2. There are 22% more loads entering the AGVS due to the addition of a second load type and the number of machining cells available to process type one loads has been reduced from six to five. The resulting increase in the mean time in system for type one loads was approximately 20%, to a mean of 4796 seconds.

Increasing the capacities of the machining cell buffers seems to have decreased the time that machining stations had to wait for loads to arrive for machining. Mean utilization of machining stations increased from a mean of 0.773 in Model 2 to 0.945 in Model 3. The mean number of loads processed in Model 3 was 186 for type one loads and 41 for type two loads. The mean number of loads created was 200 and 44, respectively.

The effect of AGV breakdown in Model 3 appears to have been small, with an average number of non-operational AGVs at 0.0160. AGV utilization increased to an average of 1.158 AGVs in use during the simulations, due in part to the extra transportation requirements of type two loads.

7.2 Problems Encountered

While conducting this research, many obstacles were encountered having to do with the programming environments used, the CBST functions used and a lack of programming experience. In an effort to save future researchers from having to solve the same problems, a discussion of each problem is presented.

7.2.1 Programming in Unix C and Microsoft QuickC

The code for AGVSF was developed using two different programming environments; Unix C in the X-Windows environment and Microsoft QuickC version 2.5. The QuickC environment is very user-friendly and offers excellent editing and debugging tools. Running a working program on the RISC 6000 using Unix C is fast, but the environment is not as friendly for developing code. Consequently, the code was developed in QuickC and then run on Unix to gather data.

Because both of these C compilers comply with the ANSII standard, there were very few compatibility problems between them. Unix was more restrictive and sometimes caught errors that were not found by QuickC. One problem that persisted, however, is the inability to use the AGVS initialization files generated in one environment to initialize the data structures of the AGVS model in the other environment. This is due to differences in how Unix and DOS systems read and write unformatted files. Consequently, the **bin** files that are read to initialize the data structures in multiple AGVSF runs must be created and used in the same environment.

Although the QuickC environment is very easy to work in, it is restricted by the memory constraints forced on it by DOS. If the AGVSF code is compiled as one program in QuickC, the generated executable code exceeds the 64K limit set by the compiler. This is even true when the large memory model is used. The result was that the AGVSF code had to be divided into two separate programs to run in QuickC. This is not a serious problem, because the initialization process is naturally separated from the simulation and analysis processes. Depending on the size of the AGVS to be modeled and the amount of random access memory (RAM) available to the compiler, AGVSF may even be run using the small or medium memory models in QuickC if the

program is divided in such a way. It was found that the use of device drivers and the version of DOS being used determined the amount of RAM available to the QuickC compiler.

7.2.2 CBST Functions

A few problems were encountered with the C functions provided in CBST [11]. Some were simple typographical errors that were easily resolved, but others were difficult to locate and correct. The CBST function **removespec()** is designed to remove a specified entity pointer from a list. In its original form in CBST, **removespec()** only works properly if there is at least one entity pointer on the list after the pointer to be removed. The CBST function **gamma()** does not compile in Unix C on the RISC 6000 because gamma is a reserved word for that compiler. The function worked properly when the name was changed to **gamm()**. The vast majority of the CBST functions worked flawlessly, though, and the code seems to be very well-written and readable.

One capability that the CBST functions don't provide is the ability to remove a single specified event from the event list. The event list management functions that are provided are **cancel1()** and **cancelall()**. **cancel1()** takes off

the first event of a specific type from the event list. **cancelall()** takes off all events of a specified type. If a system is being modeled in which a resource can be removed from an operational status, e.g. AGV breakdown, it would be convenient to be able to remove the next event for that resource, in this example an arrival event for that AGV at the next control point. Without this ability, the AGVSF functions had to be modified to account for this condition.

7.2.3 Programming Techniques

Numerous problems arose during the development of AGVSF due to inexperience in C programming techniques. The existence of a 0th element in C arrays was a constant source of errors, including the overwriting of the structures which hold the AGVS initialization data. As Khan points out, the user of this framework should be very careful to remember that the first element in every array has index number 0 [11].

Another problem arose from the use of pointers in the data structures that describe the AGVS components. In an effort to take full advantage of the power and flexibility of pointers, they were included as data structure members whenever possible. After a great deal of AGVS code had

already been written, it was found that an additional process was required to convert these pointer values to array index numbers or other values in order to store meaningful data in the **bin** files that allow the user to store the initialized AGVS data structures. The problem is that the same addresses are generally not used to store the same data in subsequent program runs. This was a hard-earned lesson.

7.3 Areas for Future Research

The simulation framework developed by this research could be improved upon to make it easier to use and to expand its application to other types of systems. The operator interface functions of AGVSF could be modified to allow the user to make changes to an existing AGVS model. A menu could be used to let the user select the component parameters to be changed. Currently, AGVSF users must enter the entire set of AGVS data if component parameters are changed. Ideally, a graphical interface system could be implemented for both data input and for graphical display of simulation results and output.

An event list management function could be added that would remove a single specified event from the event list,

as discussed in Section 7.2.2. This would give the user an excellent tool for simulating dynamic availability of system resources.

Functions could be created to model other systems, such as automated storage and retrieval systems, conveyors and other FMS components to allow detailed simulation of integrated manufacturing systems. The present AGVS framework, though limited in scope to simulate AGVSs, was designed with such additions in mind.

7.4 Conclusions

The purpose of this research was to develop and evaluate a flexible AGVS simulation tool using the C programming language. AGVSF is such a tool. AGVSF defines the parameters that are necessary for simulating AGVS operation at all levels of complexity and provides the data structures required to store this information in a consistent and accessible format. AGVSF's functions act as an interface between the AGVS model and the discrete, next-event simulation tools provided by CBST. The result is a flexible environment from which the user can select the appropriate tools needed to model the AGVS to be simulated.

The results of the example model simulations presented in Section 7.1 demonstrate that AGVSF is an effective tool for simulating the operation of AGVSs. The example models also demonstrate how AGVSF can be used to modify an existing model to develop new models with different configurations and operating conditions. The combination of the C programming language and a modular framework of well-defined functions results in fast execution speed without loss of flexibility.

8. REFERENCES

1. Bouguechal, N., D. A. Bradley and R. V. Chaplin, "A Navigation System for Automated Guided Vehicles Using Encoded Tiles," Institution of Electronic and Radio Engineers International Conference on Factory 2000, September 1988, pp. 233-240.

2. Bozer, Y. A. and M. M. Srinivasan, "Tandem Configurations for AGV Systems Offer Simplicity and Flexibility," Industrial Engineering, February 1989, pp. 23-31.

3. Cheng, T. C. E., "A Simulation Study of Automated Guided Vehicle Dispatching," Robotics and Computer-Integrated Manufacturing, Vol. 3, No. 3, 1987, pp. 335-338.

4. Davis, D. A., "Modeling AGV Systems," Proceedings of the 1986 Winter Simulation Conference, 1986, pp. 568-574.

5. De Meter, E. C., GIBSS: A Framework for the Multi-Level Simulation of Manufacturing Systems, Ph. D. Thesis, Virginia Polytechnic Institute and State University, 1989.

6. Dutt, S., Guided Vehicle Systems -- A Simulation Analysis, Report for M.E., Virginia Polytechnic Institute and State University, Blacksburg, VA, 1991.

7. Eade, R., "AGVs Make Their Move," Manufacturing Engineering, September 1989, pp. 53-55.

8. Goetschalckx, M. and L. McGinnis, "Engineering Work Station is Design Tool for CAE of Material Flow Systems," Industrial Engineering, June 1989, pp. 34-38.

9. Hammond, G. C., "Evolutionary AGVs -- From Concept to Present Reality," Chap. 1.1, in: Automated Guided Vehicle Systems, Hollier, R. H., IFS Publications, Ltd., London, 1987, pp. 3-9.

10. Hammond, G., AGVs at Work, IFS Publications, Ltd., London, 1986.

11. Khan, F. U., Development of a C-Based Simulation Toolkit Supporting Discrete, Continuous, and Combined Simulation, M.S. Thesis, Virginia Polytechnic Institute and State University, 1991.

12. Lasecki, R. R., "AGVs: The Latest in Material Handling Technology," CIM Technology, Winter 1986, pp. 90-94.

13. Law, A. M. and W. Haider, "Selecting Simulation Software for Manufacturing Applications: Practical Guidelines and Software Survey," Industrial Engineering, May 1989, pp. 33-46.

14. Law, A. M. and M. G. McComas, "Pitfalls to Avoid in the Simulation of Manufacturing Systems," Industrial Engineering, May 1989, pp. 28-31.

15. Lee, J., R. H. Choi and M. Khaksar, "Evaluation of Automated Guided Vehicle Systems by Simulation," Computers in Industrial Engineering, Vol. 19, 1990, pp. 318-321.

16. Lindgren, H., "Centralised and Decentralised Control of AGVs," Chap. 2.3, in: Automated Guided Vehicle Systems, Hollier, R. H., IFS Publications, Ltd., London, 1987, pp. 79-86.

17. Mahadevan, B. and T. T. Narendran, "Design of an Automated Guided Vehicle-Based Material Handling System for a Flexible Manufacturing System," International Journal of Production Research, Vol. 28, No. 9, 1990, pp. 1611-1622.

18. McEllin, P., "AGV Designs," Chap. 2.1, in: Automated Guided Vehicle Systems, Hollier, R. H., IFS Publications, Ltd., London, 1987, pp. 53-64.

19. Petkovska, G., V. Nedeljkovic and M. Hovanec, "Simulating a Factory Production Process with Automated Guided Vehicles," IFAC Information Control Problems in Manufacturing Technology, 1989, pp. 455-459.

20. Prasad, K. and M. Rangaswami, "Analysis of Different AGV Control Systems in an Integrated IC Manufacturing Facility, Using Computer Simulation," Proceedings of the 1988 Winter Simulation Conference, 1988, pp. 568-574.

21. Pritsker, A. A. B., Introduction to Simulation and SLAM II, John Wiley and Sons, New York, 1986.

22. Schwetman, H., "CSIM: A C-Based Process-Oriented Simulation Language," Proceedings of the 1986 Winter Simulation Conference, 1986, pp. 387-396.

23. Schwetman, H., "Using CSIM to Model Complex Systems," Proceedings of the 1988 Winter Simulation Conference, 1988, pp. 246-253.

24. Schwind, G. F., "AGVS Deliver More Flexibility, Easier Programming," Material Handling Engineering, October 1987, pp. 57-64.

25. Schwind, G. F., "AGV's: Creative Solutions Go Looking for Problems," Material Handling Engineering, September 1988, pp. 44-47.

26. Selvaraj, S., E. L. Blair, M. L. Smith and W. M. Marcy, "Discrete Event Simulation in C with DISC," Computers in Industrial Engineering, Vol. 18, No. 3, 1990, pp. 263-274.

27. Todd, D. J., "Automated Guided Vehicles," Chap. 10, in: Fundamentals of Robot Technology, John Wiley and Sons, New York, 1986, pp. 205-212.

28. Tracey, P. M., "AGV System Safety -- New Developments to Meet Changing Needs," Chap. 1.6, in: Automated Guided Vehicle Systems, Hollier, R. H., IFS Publications, Ltd., London, 1987, pp. 43-50.

29. Ulgen, O. M. and P. Kedia, "Using Simulation in Design of a Cellular Assembly Plant with Automatic Guided Vehicles," Proceedings of the 1990 Winter Simulation Conference, 1990, pp. 683-691.

30. Vosniakos, G. and B. Davies, "Simulation Sutdy of an AGV System in an FMS Environment," International Journal for Advanced Manufacturing Technology, Vol. 3, No. 4, August 1988, pp. 33-46.

31. Vosniakos, G. and B. Davies, "On the Path Layout and Operation of an AGV System Serving an FMS," International Journal for Advanced Manufacturing Technology, Vol. 4, No. 3, August 1989, pp. 243-262.

32. Vosniakos, G. C. and A. G. Mamalis, "Automated Guided Vehicle System Design for FMS Applications," International Journal of Machine Tools Manufacturing, Vol. 30, No. 1, 1990, pp. 85-97.

APPENDICES

Appendix A

Listing of AGVSF Data Structures

```
/**************/
/* num_struct, holds number of each data structure type. */
/**************/

typedef struct{
    int points;
    int links;
    int load_types;
    int agv_types;
    int agv_indivs;
    int buffers;
    int stations;
    int resnotes;
    int distribs;
    int res_index;
    }num_struct;
num_struct data_struct;



/**************/
/* tpvar, CBST structure for time persistent variables. */
/**************/

typedef struct{
    double ttot;
    double tlast;
    double tclr;
    double mean;
    double areah;
    double hharea;
    }tpvar;



/***************/
/* point, defines structure point, for point attributes. */
/***************/

typedef struct{
    int pt_type;
    int pt_charge;
    int pt_restrict;
    int pt_links[(MAXLINKS +1)];
    int pt_buffers[MAXBUFFERS+1];
    } point;
```

```
/***********/
/* link, defines structure link, for link attributes. */
/***********/

struct link{
    char link_name[20];
    int link_res;
    point *link_ends[3];
    int num_link_next[3];
    struct link *link_next[3][MAXLINKS];
    float link_length;
    int link_direct;
    float link_speed;
    int link_restrict;
    int link_cap;
    };
    typedef struct link link;
link linkarray[NLINKS+1];



/***********/
/* distrib, a structure to store parameters for random
** probability distribution functions. */
/***********/

typedef struct{
    int dist_type;
    int dist_n;
    int dist_2;
    double dist_3;
    double dist_4;
    double dist_5;
    double dist_6;
    }distrib;
distrib distribarray[NDIST+1];



/**************/
/*  listnote, CBST structure for lists.*/
/**************/

typedef struct{
     char *name;
     int lmax;
     int lcur;
     tpvar length;
     obsvar wait;
     }listnote;
```

```
/***********/
/* agv_type, defines structure of agv_type attributes. */
/***********/

typedef struct{
    float agv_speed_empty;
    float agv_speed_loaded;
    int agv_accel;
    int agv_decel;
    float agv_length;
    float agv_sep;
    int agv_discharge;
    int agv_recharge;
    int agv_num_loads;
    float agv_loading_time;
    float agv_unloading_time;
    distrib *failrate;
    distrib *repairtime;
    }agv_type;
agv_type agv_typearray[NAGVTYPES+1];



/***********/
/* agv_location, defines structure of type agv_location. */
/***********/

typedef struct{
    point *agv_point;
    link *agv_link;
    }agv_location;



/***********/
/* agv_indiv, defines AGV individual attributes. */
/***********/

typedef struct{
    char agv_name[20];
    int agv_res;
    agv_type *type;
    agv_location agv_present;
    agv_location agv_next;
    agv_location agv_final;
    int agv_action;
    entity *agv_load[NAGVLOADS+1];
    float agv_charge;
    int agv_status[9];
    }agv_indiv;
agv_indiv agv_indivarray[NAGVS+1];
```

```
/***********/
/* entity, defines individual load attributes. */
/***********/

typedef struct{

    double r[NR];
    int load_number;
    load_type *type;
    double load_arr_time;
    load_location *location;
    load_location *next_step_loc;
    int proc_status;
    int load_status[6];
    int res_seized[4];
    }entity;



/***********/
/* load_type, defines structure for load type attributes. */
/***********/

typedef struct{
    char load_name[20];
    int load_cnt;
    int proc_seq[NSTEPS+1];
    distrib *load_arr_distrib;
    buffer *load_arr_buffer;
    int load_restrict;
    }load_type;
load_type load_typearray[NLOADTYPES+1];



/***********/
/* load_location, defines load_location type structure. */
/***********/

typedef struct{
    point *load_pt;
    link *load_link;
    int load_agv;
    buffer *load_buffer;
    int load_station;
    }load_location;
```

```
/***********/
/* buffer, defines structure for buffer attributes. */
/***********/

typedef struct{
    char buffer_name[20];
    int buffer_res;
    int buffer_type;
    int buffer_cap;
    }buffer;
buffer bufferarray[NBUFFERS+1];



/***********/
/* station, defines structure for workstation attributes. */
/***********/

typedef struct{
    char station_name[20];
    int station_res;
    int station_input;
    int station_output;
    int station_cap;
    }station;
station stationarray[NSTATIONS+1];



/***********/
/* resnote, used in CBST for resource management.*/
/***********/

typedef struct{
     char *name;
     int cap;
     int utilcur;
     tpvar util;
     double busy;
     int bflag;
     double idle;
     double idltimmax;
     double bsytimmax;
     long int entcount;
     int res_type;
     agv_indiv *agv_struct;
     link *link_struct;
     buffer *buffer_struct;
     station *station_struct;
     }resnote;
resnote rn[NRES];
```

```
/**************/
/*  element, CBST structure for events.  */
/**************/

typedef struct element element;
struct element{
     element *after;
     element *before;
     int evtype;
     double evtime;
     double tentry;
     entity *pent;
     };



/**************/
/*  obsvar,  structure for observation-based variables. */
/**************/

typedef struct{
        double num;
        double mean;
        double sumx;
        double sumxx;
        }obsvar;
```

Appendix B

List of AGVSF Functions.

```
void main(void);
void structinit(void);


/* Manual initialization function declarations. */

void init_pt(void);
void init_link(void);
void init_load_type(void);
void init_agv_type(void);
void init_agv_indiv(void);
void init_buffer(void);
void init_station(void);
void init_dist(distrib *m);


/* Functions to init data structures from .bin files. */

void read_init_data_struct(void);
void read_init_point(void);
void read_init_link(void);
void read_init_load_type(void);
void read_init_agv_type(void);
void read_init_agv_indiv(void);
void read_init_buffer(void);
void read_init_station(void);
void read_init_distrib(void);
void read_init_resnote(void);


/* Functions to output initialization data. */

void data_output(void);


/* Functions to output init. structure data to screen. */

void points_output(void);
void links_output(void);
void load_types_output(void);
void agv_types_output(void);
void agv_indivs_output(void);
void buffers_output(void);
void stations_output(void);
void resnotes_output(void);
void distribs_output(void);
```

```
/* Functions to output init structure data to .dat files */

void fpoints_output(void);
void flinks_output(void);
void fload_types_output(void);
void fagv_types_output(void);
void fagv_indivs_output(void);
void fbuffers_output(void);
void fstations_output(void);
void fresnotes_output(void);
void fdistribs_output(void);


/* Functions to output init structure data to .bin files */

void data_struct_bin_output(void);
void point_bin_output(void);
void link_bin_output(void);
void load_type_bin_output(void);
void agv_type_bin_output(void);
void agv_indiv_bin_output(void);
void buffer_bin_output(void);
void station_bin_output(void);
void distrib_bin_output(void);
void resnote_bin_output(void);


/* Event routines */

void loadenters(element *b);
void loadinput(element *b);
void agvarrcp(element *b);
void agvrecload(element *b);
void loadoutput(element *b);
void batterylow(element *b);
void batteryhigh(element *b);
void agvdown(element *b);
void agvup(element *b);
void stationdown(element *b);
void stationup(element *b);
void report(void);


/* Functions used in event routines. */

void uinit(void);
void tevents(element *a);
int stopcheck(void);
double get_rv(distrib *dist);
void init_r(entity *pent);
void arr_buffer_full(entity *pent);
```

```
int find_output_buffer(entity *pent);
int find_station(entity *pent);
int dispose_load(entity *pent);
int list_add(entity *pent, int res_type, int res_index_num);
entity *list_remove(int res_type, int res_index_num);
void agv_release(entity *pent);
void buffer_release(entity *pent);
void station_release(entity *pent);
void link_release(entity *pent);
int agv_job_req(agv_indiv *pagv);
int dispatch_rule(int buffer_index);
int agv_alloc(entity *pent);
int route_req(entity *pent);
int agv_dispatch(entity *pent);
double find_travel_time(entity *pent);
double recharge(entity *pent);
int discharge(entity *pent);
void pause(void);
```

Appendix C

Listing of CBST Functions.

```
/*  Executive control.  */

void discrete(void);

/*  initialization  */

void discpinit(void);
void initlist(listnote *k);


/*  list and entity management  */

entity *makeentity(void);
void dumplist(element *s);
element *insert(element *s, int n, double x, double y,
                                          entity *ptrent);
void insprior(int n, double x, entity *q, int n2, char *d);
void filelvf(int n, entity *c, int n2);
void filehvf(int n, entity *c, int n2);
element *makeelement(void);
element *nextevent(void);
int cancel1(int type);
int cancelall(int type);
void fixevlist(element *p);
void fixlist(int n, element *p);
void schedule(int a, double b, entity *pen);
entity *remove1st(int n);
entity *removenum(int n, int rank);
entity *removespec(int n, double x);
void insfifo(int n, double x, entity *p);
void inslifo(int n, double x, entity *p);
void filefifo(int n, entity *c);
void filelifo(int n, entity *c);


/*  resource management     */

void resincr(int num, int units);
void resdecr(int num, int units);
void rescapture(int num, int units);
void resrelease(int num, int units);
int resavail(int num);
```

```
/*  random deviate generation   */

double ran3(int n);
double normal(int n, double mean, double stdev);
double erlang(int n, double beta, int m);
double weibull(int n, double alpha, double beta);
double gaussian(int num);
double gam1(int n, double alpha);
double gam2(int n, double alpha);
double gamm(int n, double alpha, double beta);
double expon(int n, double beta);
double beta(int n, double a, double b, double a1,double a2);
double beta1(int n, double alpha1, double alpha2);
double uniform(int n, double lo, double hi);
double triang(int n, double lo, double hi, double mid);
double triang1(int n, double c);
double lognormal(int n, double mean, double stdev);
int bernoulli(int n, double p);
double binomial(int n, int t, double p);
int poisson(int n, int mean);


/*  Data collection, statistical computation, reporting. */

void pfilestat(int n);
void mfilestat(int n, double enttime);
void notevalue(double xobs, obsvar *var);
void accumarea(double height, tpvar *var, double updtime);
void clrlist(void);
void clrres(void);
void clrtp(tpvar *var);
void clrov(obsvar *var);
void rlist(listnote *a, int n);
void rresource(int num);
void replist(void);
void rlhead(void);
void rrhead(void);
void bldarr(double *pyb, double time);
void cplot(int cols);
double **getarray(void);
void gettpstat(double *pvec, tpvar *ptp, double value);
void getovstat(double *pvec, obsvar *pov);
```

VITA

Jeffrey Karl Wilson was born in Plattsburg, New York on 10 October 1960. He received his Bachelor's degree in Electrical Engineering at the University of Wyoming in December 1982. He was commissioned in the Unied States Army in December 1982 and has served in various positions, including Ammunition Supply Platoon Leader and Explosive Ordnance Disposal Detachment Commander. He continues to serve as a Captain in the Army's newly established Acquisition Corps.

After receiving his Master's degree, he will be assigned to the Army's Science and Technology Center, Europe, in Frankfurt, Germany. He is currently a member of: the Society of Manufacturing Engineers (SME), ALPHA PI MU, PHI KAPPA PHI, the Ordnance Corps Association and the International Association of Bomb Technicians and Investigators (IABTI).