1-1-1990

Alternative Implementations Of Knowledge Representation And Acquisition Methods In Distributed Interactive Simulation: Investigations And Findings

Victor Coleman

Find similar works at: https://stars.library.ucf.edu/istlibrary
University of Central Florida Libraries http://library.ucf.edu

This Research Report is brought to you for free and open access by the Digital Collections at STARS. It has been accepted for inclusion in Institute for Simulation and Training by an authorized administrator of STARS. For more information, please contact STARS@ucf.edu.

Recommended Citation
https://stars.library.ucf.edu/istlibrary/10
Alternative Implementations of Knowledge Representation and Acquisition Methods in Distributed Interactive Simulation: Investigations and Findings
Alternative Implementations of
Knowledge Representation and Acquisition Methods in
Distributed Interactive Simulation: Investigations and Findings

Institute for Simulation and Training
12424 Research Parkway, Suite 300
Orlando FL 32826

University of Central Florida
Division of Sponsored Research

Intelligent Simulated Forces Laboratory
ALTERNATIVE IMPLEMENTATIONS OF
KNOWLEDGE REPRESENTATION AND ACQUISITION METHODS IN
DISTRIBUTED INTERACTIVE SIMULATION:
INVESTIGATIONS AND FINDINGS

VICTOR COLEMAN
GILBERT GONZALEZ
MIKEL PETTY
SCOTT SMITH
AMY VANZANT-HODGE
JON WATKINS
DOUGLAS WOOD

INTELLIGENT SIMULATED FORCES LABORATORY

The Institute for Simulation and Training
12424 Research Parkway, Orlando, Florida 32826

University of Central Florida
# TABLE OF CONTENTS

1. INTRODUCTION
   1.1 PURPOSE ........................................... 1
   1.2 DISTRIBUTED INTERACTIVE SIMULATION ............... 1
   1.3 COMPONENTS OF A COMPUTER GENERATED FORCE ......... 2
   1.4 CGF EXPANDABILITY .................................. 3
   1.5 GATHERING KNOWLEDGE FOR THE SYSTEM ............... 5

2. KNOWLEDGE MODELING ........................................ 6
   2.1 ALGORITHMIC SOLUTIONS ................................ 6
   2.2 FINITE STATE MACHINE SOLUTIONS ................... 6
   2.3 PRODUCTION SYSTEMS .................................. 6
   2.4 INFERENCE ENGINES ................................... 7
   2.5 LISTS AND TABLES ..................................... 7

3. KNOWLEDGE ACQUISITION FOR BEHAVIOR MODELING METHODS ........ 7
   3.1 ALGORITHMIC SOLUTIONS ................................ 7
   3.2 FINITE STATE MACHINES ................................ 8
      3.2.1 TESTBED IMPLEMENTATION OF FSM ................. 11
      3.2.2 FSM MODELING BY TABLES ......................... 12
      3.2.3 FSM MODELING BY LADDER DIAGRAMS ............... 13
      3.2.4 PETRI NETS AS A MODELING TOOL ................. 13
      3.2.4.1 AKATS (AUTOMATED KNOWLEDGE ACQUISITION TOOL) 14
   3.3 PRODUCTION RULE LISTS ................................ 15
      3.3.1 PRODUCTION RULE GENERATION .................... 16
      3.3.2 RULEBASE LANGUAGE (RBL) PREPROCESSOR ........ 17
      3.3.3 PRODUCTION RULE INTERPRETER .................... 20
      3.3.4 USE OF THE RULEBASE TOOLS ...................... 22
   3.4 INFERENCE ENGINES AND EXPERT SYSTEMS .............. 22
      3.4.1 AUTOMATING KNOWLEDGE ACQUISITION ............. 23
         3.4.1.1 COMMERCIAL KNOWLEDGE ACQUISITION SYSTEMS 23
         3.4.1.2 REPERTORY GRIDS AS A KNOWLEDGE ACQUISITION TECHNIQUE 24
   3.5 LISTS AND TABLES ..................................... 25
   3.6 THE KROME KNOWLEDGE ACQUISITION SYSTEM ............ 26

4. CONTROLLING THE DEVELOPMENT PROCESS ...................... 27
   4.1 KNOWLEDGE ACQUISITION COMPONENTS ................... 29
      4.1.1 EDITORS ......................................... 29
      4.1.2 BEHAVIORAL DATABASE MANAGEMENT ............... 30
   4.2 USE OF BEHAVIOR MODELS AND SCENARIO GENERATION .... 30
   4.3 SIMULATION INTERACTION ................................ 32
   4.4 THE COMPLETE SYSTEM ................................ 33
   4.5 GENERAL DESIGN CONSIDERATIONS ....................... 34
      4.5.1 INTERFACES ....................................... 34
      4.5.2 HELP ............................................. 34
      4.5.3 ERROR HANDLING .................................. 34

BIBLIOGRAPHY ................................................. 36
Appendices

A  RULE BASE LANGUAGE PREPROCESSOR TASK DOCUMENTATION........A-1
B  SAFOR SYSTEM FUNCTIONAL OVERVIEW...............................B-1
C  SAFOR GLOBAL LIST MANAGER........................................C-1
1. INTRODUCTION

1.1 PURPOSE

This document is the interim technical report required as a deliverable under Task 4, "Hardware/Software Testbed Development" of DARPA contract N61339-89-C-0044, INTELLIGENT SIMULATED FORCES: EVALUATION AND EXPLORATION OF COMPUTATIONAL AND HARDWARE STRATEGIES. It describes investigations into alternative approaches to Knowledge Representation, Knowledge Acquisition and the use of that knowledge for Behavioral Modeling in the area of Distributed Interactive Simulation [1].

1.2 DISTRIBUTED INTERACTIVE SIMULATION

Distributed Interactive Simulation (DIS) is an exercise in which a number of simulation devices are interconnected and the individual entities generated thereon are able to interact within a shared virtual environment. The simulators may be interconnected at one geographic location by a Local Area Network (LAN), or they may be physically dispersed but linked via a Long Haul Network.

Certain aspects of the training applications of DIS may require the generation of large numbers of entities in an arena to train a relatively small number of individuals. This is a valid use of a large scale system. However, using a fully manned simulator to generate each entity in a large force would require a large investment both in equipment and human resources.

In some cases the training may be directed toward commanders and there may be no requirement to train the individual crews and squads of manned vehicle simulators. In other cases an opposing force may be required and there may be few, if any, individuals familiar with enemy doctrine and procedures to provide the crews for manned enemy vehicle simulators. Indeed, there will probably be no manned simulators designed specifically to represent enemy vehicles.

In such situations, the need exists for automated players. These are entities whose behavior, including that of their simulated crews, to observe, maneuver, communicate, and produce and carry out plans, is generated by software. The term Computer Generated Force (CGF) is used in this discussion to refer to the combined hardware and software system used to generate a set of automated players. It is also used to refer to the collection of players as a cooperating military unit.

Depending on the number of entities simulated and their organizational structure, a CGF may be tasked with certain missions or goals. These goals may be specified at the beginning of an exercise by an individual acting as an exercise controller or they may be provided during the
exercise by an individual acting as the commander of the automated force. At some level, though, there must be at least one instance of human mission assignment or control.

An interaction or interleaving of human control with software generated behavior is possible at levels below the highest level simulated. When the option is provided for replacement by direct human intervention of functions available through software, a Semi-Automated Force exists.

1.3 COMPONENTS OF A COMPUTER GENERATED FORCE

Within DIS, interaction between simulators is accomplished through the exchange of messages or Protocol Data Units (PDUs). For the purposes of this discussion, a PDU will be equated to a message transmitted via a network.

All players on a DIS generate PDUs describing themselves, and all players will receive PDUs generated by other players [19]. Like a manned simulator, a CGF is a source of PDUs. The difference is that the behavior of the vehicles and other entities provided by the CGF is generated by software and not by direct human control.

To be useful and to provide a credible simulation, the behavior expressed by the PDUs generated by a CGF should be indistinguishable from that generated by manned simulators. Therefore, what is needed is a mechanism to generate the required PDUs in real-time; this construction will give the PDUs the appearance on the network of manned simulators. They must move realistically, appear to make reasonable decisions, respond appropriately to changes in the environment and otherwise appear to interact properly with the other elements of the DIS.

In order to generate cues describing the physical behavior and appearance of the entities (acceleration, roll, pitch, yaw, dust clouds, sound, smoke and flames, etc.) a CGF must simulate the behavior that results in these changes in appearance. This often means modeling the human decision-making that would be controlling these entities were they produced by manned simulators.

The aspects of the individual that must be considered here include environmental sensing, reactions to stimuli, the generation and following of short and long term plans, communication and cooperation with other entities, and vulnerability to events.

Collection of data describing the environment does not, by itself, generate behavior. The use of such data in the formation of sequences of actions directed toward goals is by far the most complicated and difficult portion of the task.
Simulation of the thought processes performed by one or more players is a complex problem requiring the coordination of multiple cooperating (and sometimes conflicting) efforts. Human decision-making can be broken down into subcomponents. When this is done, it may appear that a number of different kinds of behavior are involved and these different kinds of tasks may require unique models for description and simulation.

Some aspects of behavior, such as control of a vehicle by a driver, may be most easily described as a series of states with definite transitions caused by certain events or conditions. Plans generated to carry out a mission might be expressed as combinations of lists of tasks to be accomplished serially and lists of tasks to be performed simultaneously. Rules for making decisions about the relative values of threats or choices of actions to take may sometimes be expressed as collections of conditional statements.

In any attempt to categorize and understand decision making requirements, there will be a requirement for various Behavioral Modeling facilities which will require information in some form to guide their operation. The static information used by a Behavioral Modeling component will be referred to as a Behavioral Database to differentiate it from the dynamic information derived from Environmental Monitoring and Terrain Reasoning.

A CGF must interact with other simulation elements through a communications medium. Whether the simulated medium channels visual information, radio traffic, auditory cues, or information describing physical contact, some mechanism is required to coordinate transmission and reception of PDUs which encode the information. This is referred to as the Network Management facility because computer network facilities are the prime mechanism at present.

Finally, the CGF must be built on some sort of software structure that provides the computational resources, access to behavioral databases, sequencing, and other tools required to perform a simulation. This may be performed by a computer's operating system or by a special purpose Executive or Monitor program.

1.4 CGF EXPANDABILITY

A desirable characteristic of a CGF would be the notion of expandability. This concept of the potential for increased functionality after initial development may be viewed both horizontally and vertically.

Development of behavior for a CGF is primarily a process of Knowledge Engineering. Extraction of rules and descriptions of behavior from Domain Experts is necessary in order to gain some confidence that the behavior to be generated will be
correct. If it were possible to reuse behavioral subfunctions that were developed for, say, an armored vehicle, to develop the capability to simulate an unarmored wheeled scout vehicle, the potential to save immense amounts of work would be realized. For this reason, an **Object-Oriented** design is likely to be a wise approach to the overall design philosophy. Whether this would be best implemented using an Object-Oriented language remains to be seen. However, the capability to expand horizontally by adding new types of entities and new functions to preexisting entities is necessary and entirely possible using good software engineering techniques.

In the vertical dimension, however, expandability means grouping lower level organizational units into higher levels and being able to construct new higher order behavior on top of robust and complete lower level capabilities. For example, for a squad to be simulated faithfully, it may be necessary to simulate each individual or team within the squad to some level of detail and with some level of autonomy.

While it may be possible to impose some organizational control on each individual element of a group via an artificial external control module, such mechanisms may break down when complicated interactions are required. It is more likely that coordinated behavior would be successfully generated when individual instantiations of the components execute autonomous behavior with cooperative goals included in their individual goals and cooperative functions included in their behavioral databases.

This concept may be best illustrated by examples. First, consider the case of maneuvering in formation. For a vehicle to maneuver to its assigned station in a moving formation, it must determine a course and speed to generate the necessary relative movement within some period of time. This may be done independently by each individually simulated entity or it could be performed by a coordination algorithm that directs all vehicles within a formation. The first method would require a vehicle to make decisions based on its own observations of the behavior of surrounding entities. The second method would require a rather omniscient point of view. Both cases are complex and may be very difficult to solve in a manner that could avoid collisions. It is likely, however, that solutions based on the limited local knowledge of each entity would be more likely to generate realistic human behavior than solutions which had the benefit of complete knowledge of all parties' intentions.

A second example involves organizational affiliation and control. If every entity maintains a concept of its chain of command and its own position within it, it may be more likely to model attrition realistically, including the confusion that may arise when a link is removed from the chain. Ideally,
control should be transferred in a predefined manner when a leader is eliminated and an entity should still be able to carry out its standing orders in the event it becomes leaderless.

For a CGF to generate the type of behavior described above, an entity must be simulated and observed so that corrections to the behavior can be implemented for more realism. The notion of expandability should allow for this type of dynamic behavior modification. The difficult task is gathering the knowledge in such a form that integration of the knowledge into the new behavior becomes transparent.

1.5 GATHERING KNOWLEDGE FOR THE SYSTEM

As mentioned earlier, simulated entities will generate their behavior using a variety of mechanisms which may include:

- Algorithmic solutions
- Finite State Machines
- Production Systems
- Inference Engines
- Lists and Tables.

Information from Domain Experts is usually required to construct behavioral models which exhibit reasonable levels of credibility. However, experts in tactical areas are rarely trained in programming; therefore, the traditional means of extracting information has been through use of a Knowledge Engineer with intimate knowledge of the details of the simulation process. The Knowledge Engineer interviews the Domain Expert and generates databases to drive the simulation. This has proved to be time consuming and difficult, and is one of the worst bottlenecks in the knowledge transfer process. This project has investigated several ideas for the automation of part or all of the tasks performed by a Knowledge Engineer.

Ideally, a system should be developed that would prompt a Domain Expert to provide the appropriate information for the behavior to be simulated and would convert that information to a form that could be utilized efficiently by the simulation. This does not appear to be practical for the areas best suited to efficient, algorithmic solutions; however, several approaches to the other areas are discussed below.

One of the most important goals of this project, as set forth in the contract, is to speed up the process of putting a Domain Expert's knowledge into the system so that a CGF's behavior can be modeled accurately within a short amount of time. An example of this would be a commander entering battle tactics for platoon behavior and then being able to watch a computer generated platoon demonstrate the tactical behavior. As stated above, no single practical solution exists for accomplishing this goal. IST has been developing a software testbed of low level functions that will be used as the basis
for building the behavior models and knowledge acquisition methods discussed in this document. (The testbed is discussed in detail in [1].) To utilize the testbed, IST is also working on an overall system design which will allow the user to create behavior models, assign them to entities, create or modify scenarios for simulation (specifying entities, their characteristics and positions), select a scenario and invoke its simulation, and return to modify behavior models if the entities' actions are not true.

Section 2 describes some methods of modeling behavior investigated by IST. Section 3 then discusses knowledge acquisition techniques in relation to those methods for modeling behavior. Section 4 describes a possible off-line, interactive system for knowledge acquisition, scenario generation, and simulation building.

2. KNOWLEDGE MODELING

Before any description of knowledge extraction methods can be discussed, it is necessary to know how the behavior will be modeled so that we know what form the knowledge has to take. The following methods are currently being investigated as means for generating behavior:

2.1 ALGORITHMIC SOLUTIONS

Algorithmic solutions are being considered for procedures that must be done repetitively and often.

Example: Periodic computation of vehicle speed and location.

Example: Relative motion problems such as station-keeping or interception trajectories may be solved using Maneuvering Board methods (vector manipulation techniques) [10].

2.2 FINITE STATE MACHINE SOLUTIONS

Finite State Machine solutions are being considered for procedures best modeled as sequences of discrete states.

Example: Control of vehicle turning and acceleration parameters to come to a specified heading, brake to a stop, or pass through a given location.

2.3 PRODUCTION SYSTEMS

Production Systems are being considered for solutions best modeled as IF-THEN rules.

Example: IF GREATLY_OUTNUMBERED and FUEL_RANGE > 10_MILES THEN BEGIN RETREAT ELSE BEGIN HASTY_ATTACK.
2.4 INFEREN CE ENGINES

Inference Engines are being considered for solutions to problems involving partial data, levels of uncertainty, and requiring educated guesses concerning the environment.

Example: ENEMY REPORTED IN REGION AND UNIDENTIFIED CONTACT SIGHTED AT EXTREME RANGE IMPLIES CONTACT IS ENEMY WITH CONFIDENCE FACTOR OF 75%.

2.5 LISTS AND TABLES

Lists and Tables are being considered to specify the composition of missions or other complex tasks.

Example: A Reconnaissance Mission might be compiled into a structure consisting of several lists: A list of intermediate destinations to be reached sequentially, a list of activities to be performed continuously during the mission, and a list of activities to be triggered by events.

The testbed software is structured to permit integration of a wide variety of tools such as those listed above. With the potential for so many tools, knowledge acquisition methods must also be flexible, modular, and easily integrated into the system. The following section will describe some knowledge acquisition techniques as they relate to the above described methods.

3. KNOWLEDGE ACQUISITION FOR BEHAVIOR MODELING METHODS

3.1 ALGORITHMIC SOLUTIONS

Algorithms may be considered the lowest level in a vertical hierarchy of behavior generation techniques. They should be implemented as small, efficient pieces of code that are easily integrated with the testbed. Given the algorithm in verbal or written form, a programmer can convert it to code. It is then a matter of testing and refinement until the algorithm works to the satisfaction of the designer and/or programmer.

Knowledge acquisition for algorithmic solutions may pose difficult problems for the Domain Expert. His knowledge must be expressed in such a way that an algorithm can be deduced from it. Training a Domain Expert to think in algorithms may be extremely difficult. Even a Knowledge Engineer with programming experience may have difficulty converting the Domain Expert's knowledge to algorithmic form. This derives from the fact that most algorithms are based on very low level concepts, such as the physical properties of the entities that can be programmed or on approximate solutions that mimic real behavior [9,14,22,23]. Both types of algorithms require
supplementing the conscious knowledge of the Domain Expert with information concerning its structure that may not be apparent to him.

The desire for expandability creates the need for a means for creating and modifying algorithms to meet future capabilities within the testbed. An interactive, automated module to allow the creation or modification of software is one possible solution. Section 4 contains a discussion of various means to accommodate this interactive environment.

3.2 FINITE STATE MACHINES

A Finite State Machine (FSM), in the context of this investigation, refers to a concept that states that any single aspect of an entity can be in only one of a finite number of unique states at any one time and that transitions between states are determined by conditions such as a change in physical characteristics, a change in the entity's relation with the environment, or a change in the entity's relation with other entities.

An FSM may be represented by a series of statements, a multidimensional table, or by a graphical means such as a diagram that uses:

- Named circles (bubbles) to represent states
- Directed lines (arrows) between states to indicate permissible transitions
- Rectangular blocks on the transition lines holding text to describe the criteria for transition
- Text within the named circles to describe the action(s) to be taken while in that state.

A graphical approach appears to be most suitable for non-programmers. An interface could be developed which would allow a domain expert to define the overall structure of the state diagram with a graphics editor using a menu and a pointing device such as a mouse. This might include placing the state bubbles and drawing the transition lines. Information describing the transition criteria and the state specific actions could be selected from a menu or entered as text.

The usefulness of an FSM is best illustrated in the modeling of simple functions that are executed repeatedly. One such function may be to tell an entity to go to a location. To create an FSM for the command "goto x,y", the behavior of going to a location must be broken down into a series of possible states that the entity (e.g. a vehicle) could be in.
For example:

1) Prepare to go
2) Initial Braking
3) Turning
4) Accelerate (Start Moving)
5) Cruising
6) Decelerate (Braking)
7) Idle.

Next, the transitions and relationships between states must be defined. The following is one possible state diagram for the above states where boxes represent states, lines represent relations between states, and words beside lines describe the transition conditions. In this example, the actions to be performed in each state have not been listed. They might include such actions as "increase throttle", "apply brakes", "turn right maximum", "ste turn to zero", etc., depending on the type of vehicle.
The breakdown of an action into a sequence of states, as shown from the diagram above, is not always clear cut. Unfortunately, it may be difficult for Domain Experts to break what appears to be continuous behavior into discrete states because they do not have the necessary training. Even for the experienced designer, some implementation issues warrant extra care in the design of the FSM. Such issues include initializations, cycles, and nondeterministic outcomes.

Another area of concern for implementation is keeping data values for each entity separate from those of other entities.
An object oriented design would solve this problem by encapsulation of data with each instantiation of an entity (object). As an entity proceeds sequentially through a series of states, each state may require the calculation and storage of intermediate values, for example speed or the distance from the entity to a destination. These values may be used in multiple states and may be stored in some global structure so that each state has access to the value or they may be isolated to a state by being stored in some structure associated with the state. This can be accomplished by data encapsulation inherent in object oriented design.

FSMs were successfully used for modeling predatory fish behavior in the independent works of [13] and [18].

3.2.1 TESTBED IMPLEMENTATION OF FSM

FSMs can be used for modeling actions such as acceleration, turning, turret movement, gun aiming, sequencing through gunning (loading, aiming, and firing), coordinating group vehicle movement, etc. For these actions, it may be difficult to distinguish between the FSM and the algorithm. This is just one example of the overlapping implementation that exists between behavior models.

The testbed currently uses an FSM to perform the control functions to cause a vehicle to "goto x,y", as stated above. Each state is represented by a function that is executed when that entity is updated. The state routines are:

```plaintext
prepare_to_go();
turning();
initial_braking();
start_moving();
cruising();
braking();
```

A location in a control block for each vehicle is used to hold the address of the routine to be executed in the current state. Periodically this location is examined and when it is not NULL, the address therein is executed as a subroutine. A state subroutine may perform some action required in that state but it will always test the conditions necessary to cause a transition to some other state. When conditions require a transition, the routine will write the address of a different state routine into that location. In the next period, the new state routine will be executed.

In the function `prepare_to_go()`, the condition

```
DISTANCE_TO_DESTINATION < 5.0
```

is tested. When it has been satisfied, the state routine
location is set to a null address to cease action for the command "goto". If the condition

\[
\text{DISTANCE\_TO\_DESTINATION} < 20.0
\]

is met, then the state routine location is set to the address of the routine \text{initial\_braking()}. If the distance to destination is > 20.0, it is set to the address of the routine \text{turning()}. The pseudocode below illustrates other actions which must happen during this state, such as calculation of course error, computation of the radius of a turn, setting the rate of acceleration, and calculation of requested speed.

```c
prepare_to_go()
{
    compute distance_to_destination and course_error (error between current heading and bearing of destination)

    if (current distance to destination < 5.0)
        Set turn to straight ahead
        set state routine address to NULL
        set requested acceleration rate to Normal
        set requested speed to zero
        return;
    else
        if course_error not zero
            compute and use minimum turn_radius allowable at current speed

    if (distance to destination < 20.0)
        set state routine address to initial_braking()
        set requested speed to zero
    else
        set state routine address to turning()
}
```

3.2.2 FSM MODELING BY TABLES

A graphical representation of a state diagram would indicate possible state changes as directed paths between nodes. An alternative method for modeling an FSM is the use of tables.

A multi-dimensional array may be used to hold information representing (1) transitions between states, (2) events, such as timeouts, interrupts, or messages that trigger the
transitions, and (3) actions to be taken when event/state combinations occur. Typically, when an event is detected, the table is entered using the current state and the event type. Table data will specify any actions to be taken, including state changes. Multiple levels of indirection may be incorporated when appropriate; however, too many levels may be unnecessarily difficult to comprehend or maintain. A spreadsheet-like interface might be used to build and display these tables.

3.2.3 FSM MODELING BY LADDER DIAGRAMS

Ladder diagrams are used to describe the control sequences used to direct numerically controlled machines. These may provide a means of modeling behavior flow but IST has not yet investigated this area.

3.2.4 PETRI NETS AS A MODELING TOOL

Petri nets [17] are a tool used for modeling and studying systems. Petri nets can model parallel and discrete events and are worthy of investigation as a means of modeling a Finite State Machine. (See [15] for one implementation.)

Formally, a Petri net consists of a set of places \( P \), a set of transitions \( T \), an input function \( I \) and an output function \( O \). The input and output functions relate transitions to places. The graphical representation of a Petri net consists of a circle to represent a place, a bar to represent a transition, and directed arcs to represent the interconnections.

![Figure 2 Representation of a Petri net](image)

The diagram above shows a simple Petri net where \( p_1 \) and \( p_2 \) are places and \( t_1 \) is a transition. The arc from \( p_1 \) to \( t_1 \) represents an input to \( t_1 \) while the arc leaving \( t_1 \) represents an output from \( t_1 \). Tokens reside within places and are used as a means of representing movement and execution of the Petri net. The assignment of tokens to places of a Petri net is called marking.

The execution of a Petri net is controlled by the tokens that reside in the places of the net. A Petri net executes by firing transitions. A transition fires by removing tokens from its input places and creating new tokens which are
distributed to its output places. A transition may fire only if it is enabled, i.e. each of its input places has at least as many tokens in it as arcs from the place to the transition. Multiple tokens are needed for multiple input arcs.

A Finite State Machine can be easily modeled by a Petri net. If each place represents a state, then the transitions would be the actions that occur when changing states and the tokens would represent the conditions to make the changes. Many modifications to Petri nets have been explored and some of these may be better suited for modeling Finite State Machines. Investigation should continue in this area because Petri nets provide a graphical means to enter information about changing states. Using Petri nets for knowledge acquisition may also be applied to other knowledge models.

3.2.4.1 AKATS (AUTOMATED KNOWLEDGE ACQUISITION TOOL)

A system developed by Harris Corporation [20], called AKATS (Automated Knowledge Acquisition Tool), is being investigated because of its user-friendly graphical interface (written in the Smalltalk language) and its capability to represent knowledge through the use of Petri-Nets. AKATS provides a format and a structured process to be used by a Subject Matter (Domain) Expert (SME) to build knowledge bases for training applications.

Knowledge is modeled in two ways in the AKATS system, as Procedural knowledge or as Declarative knowledge. Procedural knowledge is represented by Modified Petri Nets (MPN), which are constructed by the Subject Matter Expert through the use of a graphical, mouse-driven interface. A MPN consists of nodes connected by arcs. A node can be a step representing an action or an event which in turn represents the result or condition of a step. Declarative knowledge consists of any supporting information or knowledge that can be attached to any step of the Modified Petri Net. The five forms of declarative knowledge include rules, facts, procedures, concepts and other.

The AKATS system provides a set of knowledge engineering tool sets that will aid an SME in the knowledge building process. These tools include an on-line help system, a tutorial system, and an automated Task Hierarchy creation/revision process.
On-line help comes in two forms: user-requested and system-initiated. User-requested help is represented in two menus: **Quick Help**, which presents a list of objects and actions that answer the question "Explain how to <action> <object>", and **Natural Language Menu**, which provides a structured form of natural language that allows the SME to ask many different questions or the same question in different forms. The on-line help also provides three levels of help: beginner, intermediate, and advanced. Each level varies in the amount of explanation detail.

The tutorial system contains twelve tutorials which provide instruction in the three major functions of knowledge acquisition - naming, describing, and organizing. The **Effective Net Building (ENB)** tutorials teach the SME how to build valid MPNs. These tutorials include:

- "Walk-Through Demonstration" - This tutorial "walks" the SME through the creation of a sample MPN (procedural knowledge)
- "Declarative Knowledge Introduction" - This tutorial provides instruction on how to identify and describe facts, rules, procedures, and concepts
- "Declarative ENB Modules" - This tutorial presents a set of modules which test the SME's ability to differentiate between the different types of declarative knowledge

Both Procedural and Declarative knowledge representations are created using graphics editors. With these tools, a Subject Matter Expert can build a knowledge domain without the assistance of a Knowledge Engineer. The graphics interface for creating MPNs consists of a screen with a column of icons to the left of the screen. When the user selects an icon (such as a square to represent a place or a small black box to represent a transition) the system prompts the user for information associated with that icon and the user uses a mouse to position the icon on the screen. The user can also select from the icon list a connection between places and transitions.

The editor used to create the MPN is a feature of this product which may be useful in creating the Finite State Machines described above. Unfortunately, IST researchers have not been able to find out how the information displayed in procedural and declarative knowledge is translated to be used by a knowledge execution mechanism, such as an inference engine.

### 3.3 PRODUCTION RULE LISTS

Through investigation at IST, original efforts to model behavior resulted in lists of *If...Then..* type statements, typically called **Production Rules**. These rules are generated
by a Domain Expert using a text editor. IST has developed a tool to convert these English-like rules into executable code. These lists are submitted to a Preprocessor which generates a file consisting of a series of "C" language subroutines and data structures relating them. The preprocessor's output format is compatible with an IST developed Production Rule Interpreter which efficiently executes these compiled lists of rules at run-time.

3.3.1 PRODUCTION RULE GENERATION

The format for a production rule is as follows:

\[
\text{if } ( \text{condition1 AND/OR condition2 AND/OR ... conditionN}) \text{ then } ( \text{action1 action2 ... actionM})
\]

A rulebase is a database consisting of production rules. A rather contrived example of a rulebase file following this format follows:

```
name f16_attack  /* name of rulebase */
target f16      /* kind of entity to execute*/
begin
  if         /* first "if" condition */
    ( airspeed > 100.00 )
  then       /* first consequent */
    throttle = throttle + 10.0;
  if         /* next "if" condition */
    ( ( number_of_contacts > 5 ) &&
      ( missiles_left < 3 ) )
  then       /* next consequent */
    ( begin_turn();
      signal_wingman(TURNING); )
end
```

To generate these production rules the Domain Expert must have a defined set of testable conditions from which the "if" portions of the rules are constructed, and a defined set of executable functions from which the "then" consequents are built. In the above example, the testable conditions are airspeed, number_of_contacts, and missiles_left. The executable functions are begin_turn() and signal_wingman(). Throttle is a variable whose value can be changed. For the process to be convenient and efficient, a method must be available to present these options to the user as the rules are being written. A multi-window interface might be appropriate here, with testable items listed in one window, executable functions in another, and the edited rules in a third.
To ease the burden on the Domain Expert, a template similar to the format above should be provided whenever the user wishes to generate a new rule. The text editor should provide the Domain Expert the capability to modify or delete rules, save rulebases in files, merge files, etc. Section 4 gives descriptions of an automated interface that provides for these features as well as for a rulebase editor which is multi-windowed and allows for templates.

While creating the rules, the Domain Expert must keep in mind which testable conditions and executable functions have been previously defined. One method of keeping track of these items is to have them listed in files that would be readily available to the Domain Expert. The Domain Expert must also recognize the need to create new executable functions or testable values. These executable functions would most likely be the product of a programmer who has intimate knowledge of the source code and system design. This knowledge would allow the programmer to create functions that would make the most efficient use of code that is already in place. The programmer must also be familiar with the entity's dynamics and the modeling of the environment to generate code to produce the necessary testable value. This condition illustrates the inescapable interdependency between the content of the rules and the structure of the supporting system.

3.3.2 RULEBASE LANGUAGE (RBL) PREPROCESSOR

In IST's testbed, the preprocessing of a rulebase was performed by a filter called the rbl preprocessor. This is written in the "C" programming language and was intended to run under DOS 3.3.

The rulebase preprocessor is embedded in the rulebase builder which is simply a program that invokes other programs, using one's output as another's input. An alternative to using the rulebase builder is to first use the rbl preprocessor program and then a "C" compiler. The following is a description of the execution of the rulebase builder.

The rulebase builder is invoked with the rulebase file name as a command line argument (i.e. rbl rulebase.rbl). The rulebase builder invokes the rbl preprocessor giving it the rulebase file name as input. The rbl preprocessor then generates a file containing the C source code translation of the rulebase file, a listing file which contains the rules of the rulebase followed by their C source code translations, and a link file which contains linking information for automatically linking the rulebase into the simulation; the actual linking capability is not fully developed. An error file is created only if there are errors found in the rulebase. The four files have the same base file name as the rulebase file with extensions of "c", "lst", "lnk", and "err" respectively (i.e. rulebase.c, rulebase.lst, rulebase.lnk, rulebase.err).
If the rulebase preprocessor does not successfully preprocess the rulebase (i.e. creates an error file), then the rulebase builder will exit, indicating that the rulebase file needs to be corrected. In the resulting error file, an error location (line number) and error message is given for each error found in the rulebase file. The user should view the error file in order to determine and correct the errors in the rulebase file. Having corrected the errors, the user should attempt to build the rulebase again.

After the rulebase is successfully preprocessed, the rulebase builder will invoke the rulebase compiler giving it the file name of the C source code translation of the rulebase. The rulebase compiler simply invokes a C compiler; its purpose is to trap any errors and connect them back to the rulebase file. The rulebase compiler produces the object code representation of the rulebase. This file is used to link with the simulation code to create a simulation executable.

Again, if the rulebase compiler does not successfully compile the rulebase C source file, then the rulebase builder will exit, indicating that the rulebase file needs to be corrected. In the resulting error file, an error location (rule number and beginning line number) and error message is given for each error found in the rulebase C source file. The user should view the error file in order to determine and correct the errors in the rulebase file. Having corrected the errors, the user should attempt to build the rulebase again.

The end products of the successful preprocessing and compiling of the rulebase file are a rulebase object file (rulebase.obj), a rulebase link file (rulebase.lnk), a rulebase C file (rulebase.c), and a rulebase list file (rulebase.lst). The rulebase object file and the rulebase link file are used to link the rulebase with the testbed software. The rulebase C file contains the preprocessed rulebase. The rulebase list file has the rulebase code intermixed with the resulting C code; this file can be used for debugging.

When the preprocessor translates the rules to C code, syntax checking must be done to ensure that the rules are in the proper format. Design of the preprocessor presented difficult choices about where this syntax checking should be done. The assumption was made that the rules entered would follow strict C programming syntax (as if no one makes a typing mistake). The syntax checking in the preprocessor was limited to that syntax specific to the preprocessor, not to the C compiler. Therefore, no C syntax checking was carried out in the preprocessor. It was not until compiling the ".c" file that C syntax errors would be found, even though preprocessing was complete. To correct these errors, the rulebase file would be edited, sent to preprocessing, and then sent back to compiling. Catching these C syntax errors during the preprocessing stage is more desirable because it would
eliminate unnecessary execution of the compiler until the rulebase file was syntactically correct.

Each rule in the input file (rulebase.rbl) translates into two C routines, one to check the antecedent and another to perform the consequent of the rule. The translated rules for the example input file are:

```c
int f16_attack_antecedent_0000()
{
    return
    (airspeed > 100.00);
}

void f16_attack_consequent_0000()
{
    throttle = throttle+10.0;
}

int f16_attack_antecedent_0001()
{
    return
    ((number_of_contacts > 5) &&
    (missiles_left < 3));
}

void f16_attack_consequent_0001()
{
    begin_turn();
    signal_wingman(TURNING);
}
```

Two data structures are generated to allow indirect access to these routines. A rule array contains the address of each pair of routines and the primary data structure contains the address of the rule array and the number of rules it contains. These structures are defined as:

```c
typedef struct
{
    char ALREADY FIRED; /* A flag used at run-time */
    int (*qualifications)(); /* address of test routine */
    void (*consequences)(); /* address of consequences routine */
    char *ruletext /* rule text for explanation if */
    ) RULE;
    /* tracing is turned on */

    typedef struct
{
    int count; /* number of rules in this database*/
    RULE *rules; /* address of array of rules */
} RULE_LIST;
```
The structures that are generated from the above input file are:

```
RULE f16_attack_rule_list[2] =
{
    (0, f16_attack_antecedent_0000, f16_attack_consequent_0000),
    (0, f16_attack_antecedent_0001, f16_attack_consequent_0001),
};
```

```
RULE_LIST f16_attack_rules =
    (2, &f16_attack_rule_list[0]);
```

Once the "obj" file is created, it can be linked with the simulation code. For initial testing, the name of a rulebase was hardcoded into the simulation and put into an array of rulebases associated with an entity. For the above example, the name `f16_attack_rules` was added to the array, which stores the address for this structure. One goal of the testbed was to automate this process so that when a rulebase is converted into the object file format, its address is automatically put into an external list. Then, prior to simulation, the user could select the rulebases from this list to be used by any particular entity. These rulebases (addresses) would then be put into the array used by the entity. This automation process is discussed in more detail in Section 4.

A full task description of the preprocessor is located in Appendix A, Rule Base Language Preprocessor Task Documentation.

### 3.3.3 PRODUCTION RULE INTERPRETER

The production rule interpreter was a module in the software testbed that would accept a rulebase ID (or address) from the array of rulebases and then fire the rules in the rulebase in some manner. Each entity has a control block that holds persistent data specific to that individual instantiation. Whenever the control block is using the rulebase interpreter, it contains a pointer to an array of rulebase structure addresses. An improvement to the design would have allowed the specification of a rule processing algorithm per entity. Selection of the specific algorithm would take into account the method of execution desired by the designer of the rulebase.

When a set of rules is written, it may be the intent of the author that each rule be checked only once per invocation, or perhaps that rules will be checked repeatedly, but only executed once per invocation, until a complete pass is made through the rules without a single firing. By matching the execution algorithm to the rules, problems such as endless cycles, deadlocks, and undesired interactions may be reduced. The processing algorithm must also allow for switching between rulebases in the cases of entities that have multiple
rulebases. The switch may be triggered by some external event or the firing of a rule.

Several possible implementations for a rule processor algorithm were evaluated. The algorithms naturally divided into two categories. The first category consisted of those algorithms that continuously fire rules. This might require a processor or process to be dedicated to each rule interpreter because of the enormous amount of processing time required.

The second category of algorithms consisted of those that would invoke the rule interpreter periodically, based on a time interval that could be set dynamically or was based on event triggering. Each time the rule interpreter was invoked, it would fire the rules in some order, for some fixed number of cycles, and then return. In this category, the algorithm might cycle once through the rules and exit, or it might allow for multiple cycles and exit on some explicit condition.

One multiple cycle algorithm implemented in the testbed used a two-pass system in which rules were evaluated in the first pass before being fired in the second. At each invocation of the rulebase, all rules were marked as not having fired. On the first pass of each pair of passes, all rules that hadn't been fired under the current invocation were tested and were flagged if their test conditions were satisfied. During the second pass, flagged rules were executed and marked as having fired in the current invocation. The rule interpreter repeated this two-pass sequence until it made one test pass where no unfired rules qualified to be fired. It then returned and awaited its next invocation.

Two possible implementations for one cycle algorithms have been considered. The first is to test the conditions for a rule and fire it immediately if it succeeds. The second is to test all conditions for all rules on one pass, marking the rules whose conditions are satisfied, and then to fire the marked rules on the next pass. Testing conditions for all rules before firing any was intended to reduce the interdependency of the rules. It did not, however, prevent all conflicts and collisions.

Based on experience with the rule interpreter and various rule processing algorithms, it was learned that writing rules for this type of system is very difficult. When writing rules one must always consider the effect of the execution of some rules on the testable conditions of others. As the simulated behavior becomes more complicated, it may be impossible for a single individual to mentally keep track of all possible rule/condition conflicts.

Another problem we encountered involved the storage/implementation of temporary and persistent variables. With the current design, there is no way to associate a per-
entity copy of a variable with a rulebase. One solution would be to store persistent data in the entity's control block, such as the variable **throttle** in the example above. However, each time a new rulebase were to be added to the system, the control block structure would have to be modified. Storing a pointer in the control block to a generic variable array may be another solution; however, the maximum number and types of variables would be limited and a way to correlate the variables with the active rulebases would be required.

### 3.3.4 USE OF THE RULEBASE TOOLS

When the rule interpreter was still in active use within the testbed software, it was used for maneuvering control, such as the avoidance of multiple obstacles. The rule interpreter was removed from use primarily because of the large amount of overhead required to test and fire the rules. This resulted from the fact that the actions being modeled by rules were more efficiently and appropriately handled by an FSM.

The maneuver rulebase was designed by first imagining an FSM for maneuvering and then creating rules that, when executed, would sequence through the imagined states. Rules affecting all states were lumped into one rulebase list despite the fact that logically separate subsets existed. This setup required complex if-then conditions on every rule to test all rules for all states on each iteration. This was an important source of inefficiency. Separating the rulebase into subsets was tried. This required at least one rule in each subset to cause a state transition. This resulted in far greater efficiency and began to resemble traditional state machines. Later, the rule structure was deleted and the actions were recorded much more efficiently as an FSM.

The rule interpreter may be useful in describing higher levels of behavior where decisions are more in the line of major situation changes, such as high level command and strategy decisions. It may be incorrect, however, to presume that modeling behavior at a high enough level will involve decisions that are less complicated than those of a lower level. At this time, the rule interpreter produces too much overhead to be an effective behavior modeling tool for the testbed.

### 3.4 INFEREN CE ENGINES AND EXPERT SYSTEMS

An **Inference Engine** is a software facility for making judgements based on the presence or absence of facts and on rules of implication. Typically, an inference engine is used to drive a software system, called an **expert system**, which contains information and reasoning tools. An expert system contains the knowledge of an expert and can, given factual information, make recommendations with a percentage of certainty. Many expert system shells, such as CLIPS and
NEXPERT, are commercially available for generating and using databases of rules constructed for this purpose. IST has attempted to use CLIPS to express some generalized rules for tactical movement of vehicles.

3.4.1 AUTOMATING KNOWLEDGE ACQUISITION

Finding ways to automate knowledge acquisition has progressed further for expert systems than for the other areas discussed above. Many systems currently exist and are available commercially or via government channels.

3.4.1.1 COMMERCIAL KNOWLEDGE ACQUISITION SYSTEMS

Some existing commercially available knowledge acquisition systems are tailored to meet needs in certain areas and for specific types of expert systems [11]. For example, the knowledge acquisition system MORE [7] interviews domain experts for the kind of knowledge they use to solve diagnostic problems. The SALT [12] knowledge acquisition system builds expert systems that construct rather than select solutions. These expert systems perform design tasks by proposing values for pieces of a design, identify constraints on the design, and use domain knowledge to revise decisions if constraints in the proposal are violated. The tool SIZZLE [11] builds expert systems that perform quantitative sizing tasks; that is, the systems take descriptions of factors influencing resource demands and recommend the quantities of resources needed.

Each of the above mentioned systems may be able to accomplish the knowledge acquisition needs for the testbed if the corresponding expert system or inference engine is used. These systems, however, may not be easily integrated into the testbed. Currently, size constraints, pricing, and hardware and software requirements of these systems have not been investigated. One goal of any project is affordability; many of these particular systems may not meet this goal.

An alternative is to use an existing commercial expert system, such as NEXPERT OBJECT (supplied by Neuron Data), which has its own automated (or partially automated) knowledge acquisition interface. The knowledge acquisition interface is part of the complete system which includes an inference engine. NEXPERT OBJECT has a library of artificial intelligence routines which can be embedded into any conventional programming language application, such as C. The ability to embed an expert system into the testbed software, i.e. link the expert system software with the testbed software and execute the expert system via a subroutine call within the testbed source code, is a very important feature and makes NEXPERT OBJECT an attractive alternative. NEXPERT OBJECT also interfaces to many existing software packages and runs on a variety of operating systems.

Investigation will continue into these types of systems.
3.4.1.2 REPERTORY GRIDS AS A KNOWLEDGE ACQUISITION TECHNIQUE

Another method for automated knowledge acquisition that has been investigated is the basis for a system developed at the University of West Florida [5]. The tool, ICONKAT [21], uses repertory grids and concept mapping to guide the Domain Expert in defining his domain knowledge. When the knowledge is defined in terms of the repertory grids, it can then be translated automatically into rules which will be used by an inference engine.

Repertory grids were first introduced by George Kelly [8] as a means of expressing his theoretical notions of personal constructs. Kelly's personal construct theory can be briefly stated as [4]:

- A person's thought processes are psychologically channelized by the ways in which he anticipates events.
- A person anticipates events by evaluating them in the context of similar previous experiences and construing their replications. ("In the construing or interpreting of events a person notes features of a series of events which characterize some and are particularly uncharacteristic of others. In doing so he erects constructs involving similarity and contrast, both similarity and contrast are inherent features in any construct."[4])
- Each person characteristically evolves, for his convenience in anticipating events, a construction system embracing ordinal relationships between constructs.

Each person sets up a hierarchical system of constructs where some constructs are more important than others. These constructs are "bi-polar" (sic), personal, and interrelated. It is from the interrelationships between constructs that predictions are made. Constructs can also be extended when a person chooses an alternative expression of the construct in anticipation of greater insight to the things already known.
Constructs are used in repertory grids as a means of rating something. For example, a manager may use the bi-polar construct "intelligent - dim" to rate an employee. The manager should then rate the employee on a scale of 1 to 5 where 1 means the employee is most like the left hand pole (LHP), or intelligent, and 5 means the employee is most like the right hand pole (RHP), or dim. This scale is better defined as a valued scale of percentage certainties:

<table>
<thead>
<tr>
<th></th>
<th>LHP</th>
<th>RHP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>.75</td>
<td>.75</td>
</tr>
<tr>
<td>3</td>
<td>.50</td>
<td>.50</td>
</tr>
<tr>
<td>4</td>
<td>.75</td>
<td>.75</td>
</tr>
<tr>
<td>5</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Figure 3 Repertory Grid Scale

Using the constructs to rate several items, conclusions can be heuristically drawn from the expert's concept of the situation. It is from these conclusions that rules are produced which reflect the heuristic nature of the expert's decisions.

It may or may not be necessary for a Knowledge Engineer to aid a Domain Expert in formalizing the Domain Expert's knowledge into constructs. In an attempt to automate this process, IKONCAT uses the technique of concept mapping developed by Novak and Gowin [16] to aid students in learning. Concept maps are intended to represent relationships between concepts in the form of propositions. Propositions are two or more concept labels linked by words in a semantic unit. For example "sky is blue" would represent a simple concept map forming a valid proposition about the concepts "sky" and "blue".

In IKONKAT, objects, classes and properties from the concept maps are mapped into an object-oriented knowledge base associated with the performance environment. These objects are then evaluated via a repertory grid where the constructs are the properties of these objects.

Concept maps and repertory grids could prove to be valuable techniques for knowledge acquisition for any expert system that may be used in the software testbed. Investigation should continue to see if any of the other knowledge representations mentioned above could also be modeled using this technique.

3.5 LISTS AND TABLES

Some aspects of CGF behavior involving the generation of sequential actions and the directed invocation or canceling of simultaneous actions are suited to implementations using data
structures such as lists and tables. Mission planning and route specification and following are especially dependent upon the generation of lists of actions which must take place serially or in parallel.

IST is developing the concept of a "Mission Script" which may be implemented as groups of lists containing information about actions to be taken, areas to be traversed, or messages to be sent, etc. These lists may be statically defined, as for a standard mission such as a road march, but at times, dynamic analysis of situations may require modification or replacement of lists as conditions mandate changes in plans, routes, or responses to events. The types of tasks defined in the lists should include those that are sequential, continuous, and event driven.

Using this concept, an entity will create a mission script in response to an order or directive, and a "mission manager" module will then take the script and create a mission plan composed of the lower level sequential, continuous, and event driven operations. The entity should use some intelligent software mechanism (yet to be defined) to create the scripts. The orders given the entity may be in the form of terse military op-orders that a user can select from a set of directive menus. The need for immediate intervention on behalf of an entity, such as changing its course or speed, may be satisfied by allowing the user to create a mission script and interactively assign it to the entity.

As testbed development advances, realism could be gained by the use of natural language to enter directives and mission script commands. The investigation of natural language processing will be left as a possible future endeavor because natural language understanding is probably not well enough developed.

An alternative approach is to create a simplified language for defining mission scripts. This language would encourage a more controlled environment for the development and implementation of the mission manager module.

The mission script concept will be described in greater detail in a future technical report.

3.6 THE KREME KNOWLEDGE ACQUISITION SYSTEM

One factor common to all of these knowledge acquisition and representation packages is that each must be tailored to the specific environment toward which the knowledge is targeted. Currently, no available system can accommodate all of the types of knowledge required by a CGF, although an attempt to integrate a number of approaches has been made by BBN in the development of the KREME Knowledge Representation Editing and Modeling Environment [2].
KREME deals with problems of knowledge representation and knowledge acquisition by organizing multiple representation languages and multiple knowledge editors inside a coherent global environment. KREME contains knowledge editors for four representation languages; one for frames, one for rules, one for procedures and one for attached behaviors or methods, defined as functions. The rule, procedure, and functional method editors are accessible through a global mechanism that treats these types of knowledge as forms of procedural attachment to frames. (Frames are used to model knowledge in expert systems.) These knowledge representation techniques are implemented in an object oriented environment to promote the notion of meta-level knowledge where incorporation of new representations and their editors would be straightforward.

The KREME system provides an initial step toward knowledge acquisition for multiple representations of knowledge. However, the goal of KREME is to facilitate knowledge acquisition toward building larger knowledge bases for use with expert systems. KREME, unfortunately, is not directed toward knowledge acquisition for other types of knowledge (or behavior) modeling. For KREME or any knowledge acquisition system, the definition of the different types of required behavior must be completed before benefits will be gained by development of specific tools to create these different types of behavioral databases.

The next section presents ideas for a system similar to KREME. The incorporation of knowledge acquisition tools for knowledge representations that will be used with multiple behavior models is the primary goal in the design of this system. General comments rather than specific details are given about areas of the system design. This leaves the floor open to future ideas that may be incorporated in the final testbed.

4. CONTROLLING THE DEVELOPMENT PROCESS

Development of behavioral assemblages will likely require a Domain Expert to repeatedly specify behavioral databases, test them by observing running simulations, and modify them as required. This cycle could be streamlined by use of a Development Executive which would allow the expert to specify whether he wanted to generate databases, assign them to entities, develop or modify start-up scenarios (specifying entities, their characteristics and positions), or select a scenario and invoke its simulation. Features could be developed such as the ability to checkpoint a running simulation, halt it, and restart the simulation at a named checkpoint. This would permit comparison of behavior resulting from different versions of a behavioral database.
In an attempt to define the development environment where experts can specify behavior in a variety of forms, a system configuration was designed. The design incorporates the user friendliness and the knowledge acquisition techniques discussed above. The functional specification for this system is given in Appendix B, SAFOR System Functional Overview.

This proposed system was originally designed to use multiple rulebases/behavioral databases. At that time, the investigation team decided that rulebases alone could not
provide an accurate behavior model and therefore multiple behavior models had to be considered. This implied that a development executive must incorporate multiple means of knowledge acquisition. The proposed system design is modular in nature and could expand to incorporate different knowledge acquisition systems.

In the following sections, general issues important to any system design and implementation are discussed in relation to the proposed development executive. The incorporation of multiple knowledge acquisition systems will also be discussed.

4.1 KNOWLEDGE ACQUISITION COMPONENTS

The proposed system includes many pieces used for knowledge acquisition, some of which have already been implemented as stand-alone modules which can be easily plugged into the interface. The proposed system includes designs for editors, behavioral database management utilities, and simulation generation components. This section will discuss how the design relates to the knowledge acquisition techniques discussed above and to existing components of the system.

4.1.1 EDITORS

Editors, be they graphical or text based, are used to enter information into the computer. A graphical editor may be used to create the circles and lines of a state diagram for a finite state machine or a Petri net. This editor could be menu driven or accept commands from the keyboard. The editor designed for creating rulebase lists would be text based but would include a template for the user to follow. Multiple templates would allow the creation of different types of rulebases.

A text editor may be the best way to add new algorithms to a simulation system. A series of menus would guide the user to an editor which would allow said user to add or modify source code. Usually, a qualified programmer would perform this task. Automatic code generation systems exist with which a person can graphically specify the flow of an algorithm and automatically generate the source code. This may be another worthwhile area of investigation.

A simple text editor may be sufficient for creating mission scripts. The language of the scripts could then be validated by another component of the system prior to simulation. Another possibility is for the editor to have access to libraries of legal words and statements of the mission script language. The editor could then immediately validate as the user creates the scripts and force the user to make corrections during this process.

Editors may be very simple or may be tailored to a specific application. The determination of the type of editor to use
depends on how the system is designed; one issue being whether validation takes place within the editor or is done by other system components.

4.1.2 BEHAVIORAL DATABASE MANAGEMENT

Once rulebases, scripts, graphs, or grids are created, there must be a way to store them on-line and access them in an organized manner. The method of access would vary with the intended use of the information. For example, graphs and grids may go through translation routines which in turn create behavioral databases. The original graphs and grids must be stored in some manner to allow for easy access for modification. Meanwhile, the behavioral databases must be stored in such a way that their incorporation into a simulation is a trivial process. Each of these types of knowledge representation should have a unique database dedicated to the organization and access of the files. A commercial database utility, such as DBASE III, could provide the tools by which such a system could be built. Database tools could also be custom designed to better meet the needs of each knowledge representation database.

The proposed system, as stated above, was originally designed around the use of rulebases. When the first rulebases are created, a global list manager will create a list of these rulebases that are to be used in a simulation. The global list manager would then maintain the list by the addition or deletion of rulebases when necessary. The lists were essentially database files that consist of descriptive data about the rulebase and a key field to be used for database functions such as sorting, retrieval, report generation, etc.

Although a commercial database product might be used to manage this list, it would have to be called/executed from the simulation executive. Unfortunately, most commercial database managers contain extra non-needed features and occupy a large amount of disk space and memory. For these reasons, IST has created its own global list manager (described in Appendix C, SAFOR Global List Manager). This manager is currently a stand-alone module that can be called from the operating system prompt or from a program depending on the need. At this point, only the minimal requirements for list management have been implemented.

4.2 USE OF BEHAVIOR MODELS AND SCENARIO GENERATION

At present, algorithms, finite state machines, grids, mission scripts, petri-nets, inference engines, rulebases, ladder diagrams, and tables have been discussed in reference to modeling behavior. Recall section 3.3 which discusses the implementation of the rulebases and their incorporation into an entity via a list of rulebase addresses that the control block structure pointed to. This method of incorporation is fairly straightforward. The same type of mechanism may apply
to tables, ladder diagrams, or mission scripts. This method may also be used by an FSM that is stored as a series of tables. Algorithms are usually incorporated directly into the source code as part of some module.

Behavior modeling will occur in some hierarchical fashion. Algorithms, being at the lowest level of the hierarchy, will be implemented as part of the source code. Models that build on top of algorithms, such as rulebases, mission scripts, tables, and FSMs, will be dynamically linked to entities as appropriate. In other words, not all behavior will be associated with an entity all of the time.

Taking this concept further, an expert may decide that some aspect of an entity's behavior is incorrect and may desire to modify the model for that behavior or create a new model. Upon running the simulation, the expert may decide to test the new behavior by associating it with one entity while leaving the old behavior associated with another entity. This allows the expert to observe identically entities in comparison as they demonstrate different behavior.

Associating behavior models with entities can become a complex task. Referring back to the hierarchical design, should a person initializing a simulation be concerned with assigning behavior to the entity at each level of the hierarchy? In other words, should the user be concerned with different implementations of vehicle dynamics as well as different implementations of a high level command tactic such as a road march? If the user must deal with these decisions for each entity, it may require days to initialize a simulation. The ideal solution may require low level functions such as vehicle dynamics to be set or defaulted for each entity and require the user to be concerned only with assignment of higher level behavior. However, the ability to modify and assign low level behavior must still remain a function of the system.

With these considerations in mind, the proposed system must allow for modeling of behavior in such a manner that assignment of that behavior to entities is a simple process, regardless of the level of behavior. The concept of generic behavioral databases was created to account for all types of behavior modeling. The databases will contain lists of behavior files as they can be applied to a simulation.

It may be desired that only a subset collection of these databases be applied to the simulation. Collection could imply using a series of screens and menus to select behavioral databases from the lists to be used in the simulation. Those behavioral databases selected might then be automatically put into an array that would be used by a scenario generation mechanism to allow the user to associate those databases with certain entities.
Scenario generation is the last action prior to running the simulation. Automation of the scenario creation process is needed because of the large number of choices required for the creation and placement of entities in the simulation and the desire to be able to save and repeat a scenario. Scenarios should be created using screens and a menu driven interface. When a scenario is created, the user should then be able to store it, in a file, with a unique name and recall it by name to run a simulation. By storing the scenario, the simulation may be rerun to be viewed from an alternate vantage point or to be slightly modified to show some alternate behavior.

During the creation of a scenario, the user may be required to specify such items as which terrain database to use and where to place each entity within the terrain. For each entity created, the following choices must be made:

- what type of entity is it (vehicle, individual, etc.)
- where should it be placed (x,y,z,yaw,pitch,roll)
- what is its operational affiliation
- what behavioral databases should be assigned per entity type
- what initial missions, standing orders, etc.
- what physical characteristics (fuel and ammo levels, damage)
- appearance status (on fire or not, standing, kneeling, or prone for an individual)

Once a scenario is created, the user can then choose that scenario for the simulation. The simulation is started and initialized by using the scenario file chosen. As the simulation starts, each entity is created just as specified. If anything was not specified, the system should accommodate with default values.

4.3 SIMULATION INTERACTION

While a simulation is active, dynamic modification of entities, their characteristics, missions, or orientations may be necessary. Intervention during a simulation may be desired for a multitude of reasons. During a simulation, an operator interface may allow the individual running the simulation to intervene with commands such as

- op orders to unit
- formation change
- resupply orders.
A debugging interface may be used to trace problems during some stage of the system design and implementation. A programmer would find certain commands valuable for tracking down system operation problems, such as:

- print rule numbers, as executed, for driver in vehicle 3
- print indications of all illegal PDU types
- print amount of memory used for terrain on node 4.

A third type of intervention interface for battlemasters would accommodate control of the overall simulation. These commands may be in the general format of

- set exerciseID
- Start/Stop simulation exercise
- Synchronize clocks on all nodes.

An interactive menu system may allow the user to make the above changes. This menu system may be keyboard or mouse driven. The user should be able to locate and identify the entity(s) he wishes to modify via the mouse or menu. This may imply the use of a planview display or a diagram of force simulation. Changes would not occur until verified with a mouse click on an "OK to continue" menu item or entering "Y"es to a verify question.

4.4 THE COMPLETE SYSTEM

As envisioned, the testbed software should operate from a completely menu driven environment which allows the user to proceed from one phase of the process to the next. The user or Domain Expert would first select a menu item to build behavioral databases by using any of the knowledge representation editors listed in the following set of menus. If some low level function were found to be missing, a programmer would be asked to choose another menu path and use another editor to create the low level routine. Once the behavioral databases were created (and automatically put into lists by the global list manager), the user would select another menu item to collect the set of behavioral databases to be used in the scenario generation. The user would then go to the scenario generation application through a menu and build a scenario. The user would then save the scenario, exit the scenario generator, and select the menu for running the simulation. Here, the user would be prompted for the name of the scenario to use. The name of the scenario would be entered and the simulation would begin.
4.5 GENERAL DESIGN CONSIDERATIONS

4.5.1 INTERFACES

Standard and user friendly interfaces are desirable for all of the functions described above in order to remove the requirement for a user to learn multiple protocols. The interface could consist of a series of screens and menus. This interface would allow the calling of routines from a menu item to accomplish some task, such as file creation or field validation. With a totally menu driven system, all input from the user could be validated and unreasonable choices excluded.

The use of a mouse is anticipated so that the user will have only to point to a menu item and click it to activate that item.

The design of an interface must account for the users of the system and the tasks to be performed. The designer must consider screen layout, menu sizes and positions, multiple or single windows, color schemes, and icon use and placement. Human factors experts have done a great number of studies on the most effective methods of interface design and this information is readily available in technical journals and publications (see [6] for an example).

The implementation of the interface is another issue requiring research. The hardware and existing as well as future software must be considered for compatibility. The testbed being developed at IST is programmed in the "C" language and all other software considered must be integrable with this. The C-scape interface toolset product (written and distributed by the Oakland Group) was chosen as the foundation for implementing the interface. C-scape provides the "C" functions to create windows and a variety of menu forms as well as allow other functions to be embedded in its code. Currently, a small set of test windows and menus have been implemented using C-scape.

4.5.2 HELP

The operator would also benefit from inclusion of "HELP" screens. These text screens would be accessed via HELP menu items and would explain specific aspects of the system. The Quick Help feature of the AKATS system is being considered as an alternative design for offering help to the user.

4.5.3 ERROR HANDLING

As mentioned above, a totally menu driven system would validate all input and would not allow the user to make incorrect choices. This type of error handling would either not allow the incorrect choice to be made or would stop the user after the choice is made with a message stating what must be done next. If the system is not menu driven, error
messages should be issued anytime an error condition occurs, such as hitting the wrong key, and the user may be forced to correct it or be given a help screen explaining the correct procedures. Error handling may also have to be tailored to the specific procedure.
BIBLIOGRAPHY


Appendix A
Rule Base Language Preprocessor
TABLE OF CONTENTS

1.0 OVERVIEW ................................................. A-1

2.0 TASK SPECIFICATION ................................. A-2
  2.1 Task Description ........................................ A-2
  2.2 Task Requirements ....................................... A-2
    2.2.1 Input Description ................................... A-2
      2.2.1.1 Keywords ........................................ A-2
      2.2.1.2 Identification .................................... A-2
      2.2.1.3 Target ............................................ A-3
      2.2.1.4 Declaration ....................................... A-3
      2.2.1.5 Rules ............................................. A-3
      2.2.1.5.1 Antecedents .................................... A-4
      2.2.1.5.2 Consequent ...................................... A-4
    2.2.2 Output Description .................................. A-5
      2.2.2.1 Rulebase Identification ........................ A-5
      2.2.2.2 Target Identification ........................... A-5
      2.2.2.3 Included Files .................................... A-5
      2.2.2.4 Rule Translation .................................. A-5
      2.2.2.4.1 Antecedent ("if") routines .................. A-5
      2.2.2.4.2 Consequent ("then") routines ................ A-5
      2.2.2.5 Primary Data Structure .......................... A-6
    2.2.3 Implementation ..................................... A-6
    2.2.4 Examples of Files ................................... A-6
      2.2.4.1 Input File Example .............................. A-7
      2.2.4.2 Output File Example ............................. A-7

3.0 FUNCTIONAL OUTLINE ................................. A-9
  3.1 Overview ............................................... A-9
  3.2 Sequencer .............................................. A-9
  3.3 Initializer ............................................. A-9
  3.4 Target Parser .......................................... A-9
  3.5 Name Parser ............................................ A-10
  3.6 Declaration Parser ..................................... A-10
  3.7 Rule Parser ............................................. A-10
  3.8 Header Generator ...................................... A-10
  3.9 Symbol Validator ....................................... A-10
  3.10 Token Manager ......................................... A-11
  3.11 Error Manager ......................................... A-11
  3.12 Input Manager ......................................... A-11
  3.13 Output Manager ........................................ A-11
  3.14 Utility Routines ...................................... A-11

4.0 FUNCTIONAL MODULE DESCRIPTION .................... A-12
  4.1 Sequencer .............................................. A-12
    4.1.1 main() ............................................... A-12
      4.1.1.1 Usage ........................................... A-12
      4.1.1.2 Interface to Data Managers .................... A-12
      4.1.1.3 Global Data Structures ......................... A-12
      4.1.1.4 Local Routines ................................ A-12
4.1.1.5 Library Routines .......... A-12
4.1.1.6 Utility and Other Routines .. A-12
4.1.1.6.1 Parser Routines .......... A-13
4.1.1.6.2 Initialization Routines .... A-13
4.1.1.7 Pseudocode ............... A-13

4.2 Initializer .................. A-13
4.2.1 initialize() ............... A-13
4.2.1.1 Usage ................... A-13
4.2.1.2 Interface to Data Managers A-14
4.2.1.3 Global Data Structures ... A-14
4.2.1.4 Local Routines ........... A-14
4.2.1.5 Library Routines .......... A-14
4.2.1.6 Utility and Other Routines A-14
4.2.1.7 Pseudocode ............... A-14

4.3 Target Parser ............... A-14
4.3.1 parse_target() .......... A-15
4.3.1.1 Usage ................... A-15
4.3.1.2 Interface to Data Managers A-15
4.3.1.3 Global Data Structures ... A-15
4.3.1.4 Local Routines ........... A-15
4.3.1.5 Library Routines .......... A-15
4.3.1.6 Utility and Other Routines A-15
4.3.1.7 Pseudocode ............... A-15

4.4 Name Parser ................. A-16
4.4.1 parse_name() ............. A-16
4.4.1.1 Usage ................... A-16
4.4.1.2 Interface to Data Managers A-16
4.4.1.3 Global Data Structures ... A-16
4.4.1.4 Local Routines ........... A-16
4.4.1.5 Library Routines .......... A-16
4.4.1.6 Utility and Other Routines A-16
4.4.1.7 Pseudocode ............... A-16

4.5 Declaration Parser ......... A-17
4.5.1 parse_declaration() .... A-17
4.5.1.1 Usage ................... A-17
4.5.1.2 Interface to Data Managers A-17
4.5.1.3 Global Data Structures ... A-17
4.5.1.4 Local Routines ........... A-17
4.5.1.5 Library Routines .......... A-17
4.5.1.6 Utility and Other Routines A-18
4.5.1.7 Pseudocode ............... A-18

4.6 Rule Parser ................. A-18
4.6.1 parse_rules() ........... A-18
4.6.1.1 Usage ................... A-18
4.6.1.2 Interface to Data Managers A-18
4.6.1.3 Global Data Structures ... A-19
4.6.1.4 Local Routines ........... A-19
4.6.1.5 Library Routines .......... A-19
4.6.1.6 Utility and Other Routines A-19
4.6.1.7 Pseudocode ............... A-19

4.7 Header Generator .......... A-20
4.7.1 header_generator() ..... A-20
4.7.1.1 Usage ................... A-20
4.7.1.2 Interface to Data Managers A-20
4.7.1.3 Global Data Structures A-20
4.7.1.4 Local Routines A-20
4.7.1.5 Library Routines A-20
4.7.1.6 Utility and Other Routines A-20
4.7.1.7 Pseudocode A-20

4.8 Symbol Validator A-21
4.8.1 valid_symbol() A-21
4.8.1.1 Usage A-21
4.8.1.2 Interface to Data Managers A-21
4.8.1.3 Global Data Structures A-21
4.8.1.4 Local Routines A-21
4.8.1.5 Library Routines A-22
4.8.1.6 Utility and Other Routines A-22
4.8.1.7 Pseudocode A-22

4.9 Token Manager A-22
4.9.1 get_token() A-22
4.9.1.1 Usage A-22
4.9.1.2 Interface to Data Managers A-22
4.9.1.3 Global Data Structures A-23
4.9.1.4 Local Routines A-23
4.9.1.5 Library Routines A-23
4.9.1.6 Utility and Other Routines A-23
4.9.1.7 Pseudocode A-23

4.9.2 span_to_keyword() A-23
4.9.2.1 Usage A-23
4.9.2.2 Interface to Data Managers A-23
4.9.2.3 Global Data Structures A-24
4.9.2.4 Local Routines A-24
4.9.2.5 Library Routines A-24
4.9.2.6 Utility and Other Routines A-24
4.9.2.7 Pseudocode A-24

4.10 Error Manager A-24
4.10.1 error_handler() A-24
4.10.1.1 Usage A-24
4.10.1.2 Interface to Data Managers A-24
4.10.1.3 Global Data Structures A-25
4.10.1.4 Local Routines A-25
4.10.1.5 Library Routines A-25
4.10.1.6 Utility and Other Routines A-25
4.10.1.7 Pseudocode A-25

4.10.2 reset_error_buff() A-25
4.10.2.1 Usage A-25
4.10.2.2 Interface to Data Managers A-25
4.10.2.3 Global Data Structures A-25
4.10.2.4 Local Routines A-25
4.10.2.5 Library Routines A-25
4.10.2.6 Utility and Other Routines A-25
4.10.2.7 Pseudocode A-26

4.11 Input Manager A-26
4.11.1 fill_ibuff() A-26
4.11.1.1 Usage A-26
4.11.1.2 Interface to Data Managers A-26
4.11.1.3 Global Data Structures A-26
4.11.1.4 Local Routines A-26
4.7.1.3 Global Data Structures
4.7.1.4 Local Routines
4.7.1.5 Library Routines
4.7.1.6 Utility and Other Routines
4.7.1.7 Pseudocode

4.8 Symbol Validator
4.8.1 valid_symbol()
4.8.1.1 Usage
4.8.1.2 Interface to Data Managers
4.8.1.3 Global Data Structures
4.8.1.4 Local Routines
4.8.1.5 Library Routines
4.8.1.6 Utility and Other Routines
4.8.1.7 Pseudocode

4.9 Token Manager
4.9.1 get_token()
4.9.1.1 Usage
4.9.1.2 Interface to Data Managers
4.9.1.3 Global Data Structures
4.9.1.4 Local Routines
4.9.1.5 Library Routines
4.9.1.6 Utility and Other Routines
4.9.1.7 Pseudocode

4.10 Error Manager
4.10.1 error_handler()
4.10.1.1 Usage
4.10.1.2 Interface to Data Managers
4.10.1.3 Global Data Structures
4.10.1.4 Local Routines
4.10.1.5 Library Routines
4.10.1.6 Utility and Other Routines
4.10.1.7 Pseudocode

4.11 Input Manager
4.11.1 fill_ibuff()
4.11.1.1 Usage
4.11.1.2 Interface to Data Managers
4.11.1.3 Global Data Structures
4.11.1.4 Local Routines
APPENDIX A.I - Rulebase Builder . . . . . . . . . . . . . . A.I-1
APPENDIX A.II DATA STRUCTURE LIST . . . . . . . . . . . . A.II-1
APPENDIX A.III ERRORS . . . . . . . . . . . . . . . . . . . . . . . . . . A.III-1
APPENDIX A.IV BNF GRAMMAR . . . . . . . . . . . . . . . . . . . . A.IV-1
APPENDIX A.V BLOCK DIAGRAM . . . . . . . . . . . . . . . . . . . . A.V-1
1.0 OVERVIEW

The Rulebase Language (RBL) is a pseudo "C" programming language to express collections of rules governing entity simulation in the Intelligent Simulated Forces (ISF) software. Rulebases written in RBL are translated by a preprocessor into C routines which are compiled and linked with the ISF software. The translation into C and compilation of the C is handled by the rulebase builder package. Chapters 1-4 describe the RBL preprocessor; the rulebase builder package is described in appendix AI.
2.0 TASK SPECIFICATION

2.1 Task Description

The Rulebase Language preprocessor is a front end support utility for rapid modification of the rules defining the behavior of a simulated entity. Its purpose is to generate compilable C source code from an ASCII file containing "rules" written in a pseudo C language called RBL. The "rules" represent the behavior model of an entity to be simulated. The simulation is implemented in C; therefore, preprocessing results in a C source code file that can be compiled and linked with the vehicle simulation.

2.2 Task Requirements

2.2.1 Input Description

The RBL Preprocessor is to accept as input an ASCII text-file produced by any word processor or editor that does not embed control sequences in the text. The filename should end with the suffix ".rbl".

The content of the input file should satisfy rules of syntax of the RBL pseudo C language as described below:

2.2.1.1 Keywords

Keywords appearing in the text are name, target, declare, begin, if, then, and end.

In addition to the above keywords, some tokens may need to be recognized in the parsing of the rulebase file such as variable types, relational/logical/arithmetic/assignment operators, and parentheses; braces are not allowed in the input.

2.2.1.2 Identification

The first statement of the input file begins with the keyword name. The name statement is composed of the name keyword followed by the rulebase name, composed of a string of non-white ASCII characters beginning with an alphabetical character. The rulebase name is used to uniquely identify the rulebase structures that are generated by the RBL preprocessor.

example:

    name f16_attack
2.2.1.3 Target

The second statement of the input file begins with the keyword `target`. The `target` statement is composed of the `target` keyword followed by the rulebase target (a string of non-white ASCII characters). The rulebase target is used to associate the rulebase with a particular entity, such as an M1A1 tank, an F16 fighter jet, or an APACHE attack helicopter.

Example:

```
target f16
```

Note: The rulebase target may be broken down into smaller components of such vehicles such as commander, driver, gunner, loader, or vehicle dynamics model in the case of an M1A1 tank.

2.2.1.4 Declaration

An optional declaration section follows the target statement. The declaration section allows the rulebase editor to define variables for localized data storage. The declaration section begins with the `declare` keyword. The `declare` keyword is followed by zero or more declaration statements. A declaration statement consists of a variable type followed by a variable list terminated by a semicolon. A variable list consists of a list of variables separated by commas (variables are to be a string of alphanumeric characters beginning with an alphabetical character).

2.2.1.5 Rules

The rules section follows the optional declaration section or the target statement. The rule section is marked by the keyword `begin`. Following the occurrence of this keyword, the remaining text is interpreted as rules until the terminating keyword `end` is encountered.

Rules written in RBL are antecedent-consequent statements. The antecedent consists of one or more conditions connected by logical operators. The consequent consists of one or more actions. The execution of an RBL rule proceeds by first evaluating the conditions in the antecedent. If the conditions evaluate to true, then the actions of the consequent are performed.

The syntax of a rule is roughly that of an "if then" statement common to most high level programming languages. The keyword `if` signals the beginning of the antecedent. The keyword `then` signals the beginning of the consequent. Both the antecedent and consequent are enclosed by parentheses.

A rule in the RBL pseudo C language conforms to the following template, where the conditions, actions, and boolean operators are replaced by valid C source code.
if ( condition1 AND/OR condition2 AND/OR ...
    conditionN )
then ( action1 action2 ...
    actionM )

2.2.1.5.1 Antecedents

A line beginning with an if keyword begins the antecedent block of a rule. An antecedent block is enclosed with parentheses and may span several lines of input. Conditions appearing in the antecedent use factors, logical operators, and relational operators. A factor is a variable, a literal, or a function call. The AND (&& in C) and OR (|| in C) logical operators are supported. The full set of "C" relational operators is supported.

A condition may follow one of three forms. A simple condition is a boolean factor. A normal condition is two factors separated by a relational operator. A complex condition is two conditions separated by a logical operator. All conditions should be surrounded by parentheses to insure the correct evaluation of the operators.

The following is an example of an antecedent with a complex condition followed by a normal condition followed by a simple condition:

```c
if ( ( (a > 3) || (b == 3) ) &&
    (x != y) &&
    (t_or_f) )
```

where the complex condition consists of two normal conditions.

2.2.1.5.2 Consequent

A line beginning with the then keyword begins the consequent block of a rule. A consequent block is enclosed with parentheses and may span several lines of input. Actions appearing within a consequent are a set of actions to be performed sequentially if the antecedent evaluates to TRUE. Actions are assignment statements or procedure calls. All actions must be followed by a semicolon.

There may be no if or then keywords appearing within a consequent block.
It is possible that an "else" control might be implemented to be performed if the antecedent evaluates to false. An "else" control could be implemented using the same definition as above, except that it would follow the consequent block and would begin with "else".

2.2.2 Output Description

The preprocessor generates one ASCII file which is compilable C source code and a second ASCII file listing errors. The first file (henceforth the rulebase code file) contains the C code translation of the rules. The second file (henceforth the rulebase error file) contains error messages indicating types and locations of errors in the rulebase input file. The file names should end with the suffix ".c" and ".err" respectively. A description of the contents of the rulebase code file follows.

2.2.2.1 Rulebase Identification

The rulebase code file begins with a C comment containing the rulebase name.

2.2.2.2 Target Identification

A comment containing the name of the rulebase target follows the rulebase identification.

2.2.2.3 Included Files

The code file contains a statement to "#include" a target declarations header file. The target declarations file is described in more detail in subsequent sections and chapters.

2.2.2.4 Rule Translation

Each rule in the input file translates into two C routines to perform the antecedent and consequent parts of the rule. Two data structures are generated to allow indirect access to these pairs of routines. A rule array contains the addresses of two routines for each pair. The primary data structure contains the address of the rule array and the number of rules it contains.

2.2.2.4.1 Antecedent ("if") routines

The antecedent routine is a function which evaluates conditions and returns a result either TRUE or FALSE. The antecedent routines are named sequentially beginning with "antecedent_0000". Prefixed to this name is the name of the rulebase in order to insure unique names.

2.2.2.4.2 Consequent ("then") routines

The consequent routine performs the actions that change global/local data which may affect the behavior and/or other rules. The consequent routines are named sequentially beginning
with "consequent_0000". Prefixed to this name is the name of the rulebase in order to insure uniqueness among routines from other rulebases.

### 2.2.2.5 Primary Data Structure

The primary data structure generated by the preprocessor is of type RULE_LIST. A RULE_LIST structure contains an integer count of the rules and the address of an array of type RULE (of size count). Each entry in a RULE array contains a flag used by the runtime rule processor algorithm, an address for an antecedent routine, and an address for a consequent routine; it might also contain some text for explanation (maybe the text of the rule as found in the input file). The RULE_LIST structure is used by the runtime rule processor to access the antecedent and consequent routines of the rules. The RULE_LIST and the array of RULE structures are declared with the rulebase name prefixed to the string constants RULE and RULE_LIST respectively.

Each RULE entry may, if the need arises, contain other fields for flags which are yet to be determined.

The definitions of RULE_LIST and RULE structures are as follows:

```c
typedef struct {
    char already_fired;  /* A flag used at run-time */
    int (*qualifications)(); /* address of test routine */
    void (*consequences)(); /* address of consequences routine*/
    char *ruletext;    /* rule text for explanation */
} RULE;

typedef struct {
    int count;   /* number of rules in this rulebase */
    RULE *rules; /* address of array of rules */
} RULE_LIST;
```

### 2.2.3 Implementation

The Preprocessor is written in Microsoft C. It is invoked using the standard C language argc/argv convention to pass the input filename and other parameters to the routine.

### 2.2.4 Examples of Files

The examples below include blanks and extra lines for clarity. The spacing is not required in either the input or output. A method for structuring the C output for readability would be desirable. There is a UNIX utility called "cb" (C-beautifier) that can meet this need, if it can be found and supported on the host machine.
2.2.4.1 Input File Example

BOF
name f16_attack
target f16
begin
if
  ( airspeed > 100.00 )
then
  throttle = throttle + 10.0;
if
  ( (number_of_contacts > 5) &&
     (missiles_left < 3) )
then
  begin_retreat();
  signal_wingman(TURNING);
end
EOF

2.2.4.2 Output File Example

This example lists the output generated by the RBL preprocessor from the example in 2.2.4.1 above.

BOF
/************************************************************************************/
* f16_attack rules
***********************************************************************************/
* f16
***********************************************************************************/

#include <f16def.h>

int f16_attack_antecedent_0000()
{
  return
  (  
     airspeed > 100.00
  )
}

void f16_attack_consequent_000()
{
  throttle = throttle+10;
}

int f16_attack_antecedent_0001()
{
  return
  (  
     (number_of_contacts > 5) &&
     (missiles_left < 3)
  )
}
void f16_attack_consequent_001()
{
    begin_retreat();
    signal_wingman(TURNING);
}

RULE f16_attack_rule_list[2] =
{
    {0, f16_attack_antecedent_0000, f16_attack_consequent_0000},
    {0, f16_attack_antecedent_0001, f16_attack_consequent_0001},
};

RULE_LIST f16_attack_rules =
    (2, f16_attack_rule_list[0]);

EOF
3.0 FUNCTIONAL OUTLINE

3.1 Overview

The rulebase preprocessor contains 13 components. There is a sequencer that invokes an initializer, 4 parsers, and a header generator. The parsers are: a name parser, a target parser, a declaration parser, and a rule parser. There are four data managers that support these modules: an input manager, an output manager, an error manager, and a token manager. A symbol validator supports symbol validation. Finally, there is a utility component consisting of those routines which are shared among the parsers and managers.

3.2 Sequencer

The Sequencer controls the invocation of the Initializer, the Name Parser, the Symbol Parser, the Header Generator, and finally, the Rule Parser. It saves the Rulebase input file name and the Global Symbol file name, retrieved from argv, in file name buffers.

3.3 Initializer

The Initializer initializes the data structures used by the rulebase preprocessor. It initializes the input and output buffers, names to be used for files which are created, the local and global symbol tables, and the file pointers.

3.4 Target Parser

The Target Parser parses the target definition section of the rulebase file. It recognizes the target keyword and then collects the target type. The target type identifies the type of entity the rulebase will belong to (i.e. tank, plane, helicopter, etc.). For each, there is an associated file that contains the function, macro, and variable identifiers that are allowed to appear in rules used by that entity. The Target Parser loads these identifiers into the global symbol table for use by the Symbol Validator.
3.5 Name Parser

The Name Parser parses the name definition section of the rulebase file. It takes tokens from the token buffer and produces comment strings for the Rulebase output file. It stores the Rule Base identifier in the name buffer (or the Rulebase input file name if the identifier is invalid). Error messages are sent to the Error Manager when detected. The Name Parser returns the parsing status.

3.6 Declaration Parser

The Declaration Parser parses the local variable declaration section of the rulebase input file. If the DECLARE keyword is present, the declaration parser builds a local symbol table and a local type table using the types and symbols found in the rule declaration section (i.e. between the DECLARE keyword and the BEGIN keyword). The local symbol table is in the form of a hashing array of linked lists and it contains all the symbols found in the declaration section. The local type table is in the form of an array of linked lists which thread through the symbols in the local symbol table. The local type table contains a linked list for each declaration type and symbols of the same type are put in the corresponding type list.

3.7 Rule Parser

The Rule Parser parses the rules section of the rulebase input file. It constructs condition and action routines for each rule in the rulebase input file. A validation of the condition and action expressions includes symbol checking and parenthesis matching only; otherwise, these expressions are assumed to be in correct C syntax. After all the rules are parsed, it constructs the rule list declaration, which contains an assignment record for each rule in the rulebase input file.

3.8 Header Generator

The Header Generator generates the symbol declarations. It constructs local variable declarations for each non-empty declaration list found in the local type table.

3.9 Symbol Validator

The Symbol Validator validates symbols. It receives two arguments, a symbol and a validation selector. Symbols can be validated as appearing in a declaration or in rules. A symbol
appearing in a declaration is invalid if any duplicate symbol is found in either the local or global symbol table. A symbol appearing in a rule is invalid if there are no matching symbols appearing in the tables. In the future, symbol validation may have to be extended in order to do type checking. The Symbol Validator returns the validation status.

3.10 Token Manager

The Token Manager buffers tokens from the input buffer. It copies a token from the input buffer to the token buffer. When the end of a line is encountered it invokes the input buffer to fetch a new line. The Token Manager returns the input buffer status (OK, EOL, EOF).

3.11 Error Manager

The Error Manager handles errors detected by the parsers and the managers. It takes a token, an error code, and a line number as arguments, generates an error message, and appends the error message to the error buffer.

3.12 Input Manager

The Input Manager buffers the text from the rulebase input file. It retrieves a line of text from the rulebase input file and stores it in the input buffer. It opens the input file when initially called. The Input Manager returns the file status (UNABLE_TO_OPEN, NO_EOL, EOF, or OK).

3.13 Output Manager

The Output Manager stores output text to an output buffer, and flushes the output buffer and error buffer to files. It opens an output file on initial call. The Output Manager returns a file status (UNABLE_TO_OPEN or OK).

3.14 Utility Routines

The Utility Routines are general purpose routines that are used by more than one parser and/or manager module. They include a routine to expand the size of a buffer, a routine to hash a string, and a routine to determine if a string is a keyword.
4.0 FUNCTIONAL MODULE DESCRIPTION

4.1 Sequencer

The Sequencer is the main routine of the Rulebase Language Preprocessor. After it copies the name of the rulebase file from the command line arguments, it invokes an initialization routine to initialize the data structures and to open input and output files. It then invokes in sequence four routines to parse and translate the rulebase. The status of each routine is tested before the next routine is invoked.

Composition

1. main()

4.1.1 main()

The routine main() is the driver routine of the preprocessor. It invokes the initialization routine and the four rulebase translation routines.

4.1.1.1 Usage

```c
int main(argc, argv)
    int argc; /* number of command line arguments */
    char *argv[]; /* pointer to argument vector */
```

4.1.1.2 Interface to Data Managers

1. Token Manager: get_token and span_to_keyword.
2. Output Manager: flush_obuff.
3. Error Manager: error_handler.

4.1.1.3 Global Data Structures

1. rb_fname: Rulebase file name from command line argument is copied into rb_fname.

4.1.1.4 Local Routines

None.

4.1.1.5 Library Routines

1. strcpy, strcmp: String manipulation routines.

4.1.1.6 Utility and Other Routines

1. clean_up: closes files and frees memory.
4.1.1.6.1 Parser Routines

1. parse_name: parses and translates the name section of rulebase input file.

2. parse_target: parses and translates the target section of rulebase input file.

3. parse_declare: parses and translates the declaration section of rulebase input file.

4. parse_rule: parses and translates the rule section of rulebase input file.

4.1.1.6.2 Initialization Routines

1. initialize: initialize file pointers and data structures.

4.1.1.7 Pseudocode

- Copy rulebase input file name from command line argument to rulebase input file name buffer.
- Initialize data structures and file pointers.
- Initiate token fetch.
- Parse name section.
- Parse target section.
- Parse declaration section.
- Parse rule section.
- Generate header file.
- Clean up files and data structures.

4.2 Initializer

The Initializer initializes the data structures used by the rulebase preprocessor. It initializes the input and output buffers, file names to be used for opening files, the local and global symbol tables, and the file pointers.

Composition

1. initialize()

4.2.1 initialize()

A driver routine that calls local and external routines to perform the initialization.

4.2.1.1 Usage

int initialize()
4.2.1.2 Interface to Data Managers

1. Input Manager: fill_ibuff.
2. Output Manager: flush_obuff and echo.
3. Error Manager: initial_error_tab.
4. Linklist Manager: Q_init.

4.2.1.3 Global Data Structures

1. rb_base: base name of rulebase input file name.
2. rb_fout: rulebase output file name.
3. rb_header: rulebase header file name.
4. rb_error: rulebase error file name.
5. rb_listing: rulebase listing file name.
6. output_fname: array of pointers to output file names.
7. input_fp: pointer to rulebase input file.
8. output_fp: array of pointers to output files.

4.2.1.4 Local Routines

1. initial_iobuf: initialize input/output/error buffers.
2. initial_fname: initialize input/output file names.
3. initial_file: initialize input/output file pointers.

4.2.1.5 Library Routines

1. string routines: strset and strncat.
2. memory routines: malloc.

4.2.1.6 Utility and Other Routines

1. global_symbol_loader_checker: load global symbol table with symbols found in global symbol file.

4.2.1.7 Pseudocode

- Call initial_iobuf to initialize input/output buffers.
- Call initial_fname to initialize file name buffers.
- Call global_symbol_loader_checker to load global symbol table with contents of global symbol file. Return FAD if file access denied.
- Call initial_file to initialize file pointers. Return FAD if file access denied.

4.3 Target Parser

The Target Parser parses the target definition section of the rulebase file. It recognizes the target keyword and then collects the target type. The target type identifies the model the rulebase is to simulate (i.e. tank, plane, helicopter, etc.). For each simulation model, there is an associated file that contains the function, macro, and variable identifiers that are allowed to appear in rules. The Target Parser loads these
identifiers into the global symbol table for use by the Symbol Validator.

Composition
1. parse_target()

4.3.1 parse_target()

The target section parser.

4.3.1.1 Usage

int parse_target()

4.3.1.2 Interface to Data Managers

1. Calls to Output Manager routines append_to_obuff and flush_obuff.
2. Calls to Error Manager routine error_handler.
3. Calls to token_manager routine get_token.

4.3.1.3 Global Data Structures

1. keyword_list: list of keywords.

4.3.1.4 Local Routines

1. valid_target: validate target type.

4.3.1.5 Library Routines

1. string routines: strlen, strcmp.

4.3.1.6 Utility and Other Routines

1. target_symbol_manager: load the global symbol table.

4.3.1.7 Pseudocode

- If token is not target keyword
  - Return error X_TARGET
- If token is target Keyword
  - Get next token, return if EOF
  - If token is not valid target
    - Return error BAD_TARGET
- Construct target comment and send to Rulebase C source file
- Construct include string and send to Rulebase C source file
- Load symbols into global symbol table
- If failed to load symbols
  - Return error status of target_symbol_manager (FAD or NO_MEMORY)
- Get next token and return
4.4 Name Parser

The Name Parser parses the name definition section of the rulebase file. It takes tokens from the token buffer and produces comment strings for the Rulebase output file. It stores the Rulebase identifier in the name buffer (or the Rulebase input file name if the identifier is invalid). Error messages are sent to the Error Manager when detected. The Name Parser returns the parsing status.

Composition
1. parse_name()

4.4.1 parse_name()

The name section parser.

4.4.1.1 Usage

int parse_name()

4.4.1.2 Interface to Data Managers

1. Calls to Output Manager routines append_to_obuff and flush_obuff.
2. Calls to Error Manager routine error_handler.
3. Calls to token_manager routine get_token.

4.4.1.3 Global Data Structures

1. rb_name: name of rulebase.
2. rb_base: base name of rulebase input file name.

4.4.1.4 Local Routines

None

4.4.1.5 Library Routines

1. string routines: strcpy, strcmp.

4.4.1.6 Utility and Other Routines

1. valid_symbol: validates symbols
4.4.1.7 Pseudocode

- If token equal NAME keyword.
  - Next token should be valid symbol and is copied into rb_name as the rulebase name.

- If token was not NAME keyword.
  - Call error_handler, error x_NAME, and copy rb_base into rb_name as the rulebase name.
  - Construct name comment, append to output buffer, and flush to Rulebase output file.
  - Get next token and return.

4.5 Declaration Parser

The Declaration Parser parses the local variable declaration section of the rulebase input file. If the DECLARE keyword is present, the declaration parser builds a local symbol table from the types and symbols found in the rule declaration section (i.e. between the DECLARE keyword and the BEGIN keyword).

Composition

1. parse_declaration()

4.5.1 parse_declaration()

The declaration section parser.

4.5.1.1 Usage

int parse_declaration()

4.5.1.2 Interface to Data Managers

1. Calls to Output Manager routines append_to_obuff and flush_obuff.
2. Calls to Error Manager routine error_handler.
3. Calls to Token Manager routine get_token.
4. Calls to Linklist Manager routine Q_prepend.

4.5.1.3 Global Data Structures

1. type_list: array of type strings

4.5.1.4 Local Routines

1. add_local_symbol: add a symbol to the local type table.

4.5.1.5 Library Routines

1. string routines: strcmp, strcpy.
2. memory routines: malloc.

4.5.1.6 Utility and Other Routines

1. is_type: compare string for match in type list.
2. valid_symbol: validate symbol.
3. keyword: compare string for match in keyword list.

4.5.1.7 Pseudocode

- If current token is not keyword or type, return (OK if token was a keyword, error otherwise).
- Do the following while token is not a keyword.
  - If token is a type (save type).
  - Get symbols (validate them) and add to local symbol table until end of symbol list (semi-colon), keyword found, or EOF.
  - If token was not type (return OK if keyword found).
  - Get tokens until type or keyword found.

4.6 Rule Parser

The Rule Parser parses the rules section of the rulebase input file. It constructs condition and action routines for each rule in the rulebase input file. A validation of the condition and action expressions includes symbol checking and parenthesis matching only; otherwise, these expression are assumed to be in correct C syntax. After all the rules are parsed, it constructs the rule list declaration, containing an assignment record for each rule in the rulebase input file.

Components

1. parse_rules()

4.6.1 parse_rules()

The rule section parser. It recognizes the IF and THEN keywords and calls an expression evaluator to parse the expression.

4.6.1.1 Usage

int parse_rules()

4.6.1.2 Interface to Data Managers

1. Calls to Output Manager routines append_to_obuff and flush_obuff.
2. Calls to Error Manager routine error_handler.
3. Calls to Token Manager routines get_token and span_to_keyword.
4.6.1.3 Global Data Structures

1. keyword_list: list of keyword strings.
2. line_number: line number of rulebase input file.

4.6.1.4 Local Routines

1. build_rule_tab: builds the rule list declaration.

4.6.1.5 Library Routines

1. string routines: strcpy, strcmp, strchr, strcat.
2. conversion routines: itoa.

4.6.1.6 Utility and Other Routines

1. eval_exp: expression parser and validator.

4.6.1.7 Pseudocode

• If token is not BEGIN keyword, span to BEGIN or IF keyword.
  • Do forever
    • If token is END keyword, build rule table and return.
    • /* Parse IF part of rule */
    • If token is not IF keyword
      • Span to IF or END keyword.
    • If token is END keyword, build rule table and return.
    • /* Token was IF keyword
    • Add count of rules to output string for IF begin routine
    • and append to output buffer.
    • Evaluate IF expression and if it is invalid.
      • If it is unexpected EOF, return UEOF.
      • Log error and continue otherwise.
    • /* Expression was not invalid */
    • Append IF end routine to output buffer and flush to
    • output file.
    • /* Parse THEN part of rule */
    • If token is not THEN keyword.
      • Span to THEN, IF, or END keyword.
      • If token is END keyword, build rule table and
      • return.
      • If token is IF keyword continue to beginning of
      • loop.
    • /* Token was THEN keyword */
    • Add count of rules to output string for THEN begin
    • routine and append to output buffer.
    • Evaluate THEN expression and if it is invalid.
      • Return UEOF if it is unexpected EOF.
      • Log error and continue otherwise.
    • /* Expression was not invalid */
    • Append THEN end routine to output buffer and flush to
    • file.
  • /* End of Do loop */
4.7 Header Generator

The Header Generator generates the declarations. It constructs local variable declaration using the contents of the local type table (threaded list for each type through the local symbol table). It sends these declarations to an output file.

Components

1. header_generator()

4.7.1 header_generator()

It generates the symbol declarations.

4.7.1.1 Usage

int header_generator()

4.7.1.2 Interface to Data Managers

1. Calls to Output Manager routines append_to_obuff and flush_obuff.
2. Calls to Linklist Manager routine Q_getnxt.

4.7.1.3 Global Data Structures

1. rb_name: base name of rulebase input file.
2. loc_sym_tab[]: local symbol table.

4.7.1.4 Local Routines

None

4.7.1.5 Library Routines

1. string routines: stropy, strlen.

4.7.1.6 Utility and Other Routines

None.

4.7.1.7 Pseudocode

- Prepare prefix for symbol declarations (rulebase name prefixed to local symbol names)
- For each type in loc_type_tab do the following.
  - If the head of the list is not empty.
    - Reset line length.
    - Append the type to the output buffer.
    - While not end of list do the following.
      - If the line length exceeds maximum.
        - Reset line length.
        - Append new line to the output buffer.
- Flush the output buffer to the output file.
- Increment line length by size of string to output.
- Copy symbol to end of prefix and append symbol to output buffer.
- Get next symbol.
- If next symbol exist, append delimiter to output buffer.
- Append declaration list terminator to output buffer and flush output buffer to output file.

4.8 Symbol Validator

The Symbol Validator validates symbols. It receives two arguments, a symbol and a validation selector. Symbols can be validated as appearing in declaration or in rules. A symbol appearing in a declaration is invalid if any duplicate symbol is found in either the local or global symbol table. A symbol appearing in a rule is invalid if there are no matching symbols appearing in the tables. Symbol validation may need to be extended to do type checking. The Symbol Validator returns the validation status.

Components

1. valid_symbol()

4.8.1 valid_symbol()

It validates symbols.

4.8.1.1 Usage

```c
int valid_symbol(symbol, type_ck)
    char *symbol: symbol to validate
    int type_ck: type of validation
```

4.8.1.2 Interface to Data Managers

1. Calls to Linklist Manager routine Q_getnxt.

4.8.1.3 Global Data Structures

1. loc_sym_tab: local symbol table.
2. glbl_sym_tab: global symbol table.

4.8.1.4 Local Routines

None
4.8.1.5 Library Routines

1. string routines: strcmp
2. character routines: isalpha

4.8.1.6 Utility and Other Routines

1. keyword: test if string is a keyword
2. hash: hash a string.

4.8.1.7 Pseudocode

- If symbol is keyword, return UX_KEYWORD.
- If symbol does not start with an alpha character, return BAD_SYM.
- Hash symbol for index to symbol tables.
- If local symbol table list not empty.
  - While node is not null and symbols do not match.
  - Get next symbol in list.
- If match not found in local symbol table.
- If global symbol table list not empty.
  - While node is not null and symbols do not match.
    - Get next symbol in list.
  - Mark global symbol as used if match occurred.
- Determine result of search and return result.

4.9 Token Manager

The Token Manager buffers tokens from the input buffer. It copies a token from the input buffer to the token buffer. When the end of line is encountered it invokes the input buffer to fetch a new line. The Token Manager returns the input buffer status (OK, UEOL, UEOF).

Components

1. get_token()
2. span_to_keyword()

4.9.1 get_token()

It retrieves tokens from the input buffer.

4.9.1.1 Usage

int get_token()

4.9.1.2 Interface to Data Managers

1. Calls to Input Manager routine fill_ibuff.
2. Calls to Output Manager routine echo.
3. Calls to Error Manager routine error_handler.
4.9.1.3 Global Data Structures

1. token_ptr: pointer to next token in input buffer.
2. token: token buffer.
3. line_number: line number of rulebase input file.

4.9.1.4 Local Routines

None.

4.9.1.5 Library Routines

1. string routines: strchr.

4.9.1.6 Utility and Other Routines

None.

4.9.1.7 Pseudocode

- If token pointer points to end of input buffer.
  - Fill input buffer.
  - Increment line number count.
- Record end of line.
- /* Get token from input buffer */
- While token pointer points to a blank, tab, or new line do the following
  - Copy the character to the token buffer.
- If the character is a special character.
  - Back the character off if it is not the token (i.e. more than one character has been retrieved).
  - Break the loop.
- If the number of characters exceeds the maximum.
  - Log an error.
  - While token pointer does not point to a blank, tab, new line, or a special character skip the character.
- /* Token is retrieved */
- While token pointer points to a blank, tab, or new line skip the character.
- If the token is the EOF token, return UEOF.
- If EOL is recorded, return UEOL.
- Otherwise return OK.

4.9.2 span_to_keyword()

It spans all tokens not keywords.

4.9.2.1 Usage

int span_to_keyword()

4.9.2.2 Interface to Data Managers

1. Calls to Token Manager routine get_token().
4.9.2.3 Global Data Structures

None.

4.9.2.4 Local Routines

None.

4.9.2.5 Library Routines

None.

4.9.2.6 Utility and Other Routines

1. keyword: test if string is a keyword.

4.9.2.7 Pseudocode

- While (get token) not EOF do the following.
- If token is a keyword, return OK.
- /* Token did not match a keyword */
- Return UEOF.

4.10 Error Manager

The Error Manager handles errors detected by the parses and the managers. It takes a token, an error code, and a line number as arguments, generates an error message, and appends the error message to the error buffer.

Components

1. error_handler()
2. reset_error_buff()

4.10.1 error_handler()

It handles error messages.

4.10.1.1 Usage

    int error_handler(err_token, error, err_line)
    char *err_token: pointer to token causing error
    int error: error code
    int err_line: line number error located

4.10.1.2 Interface to Data Managers

None.
4.10.1.3 Global Data Structures

1. `ebuff`: error buffer.
2. `last_line_number`: last line error occurred.

4.10.1.4 Local Routines

None.

4.10.1.5 Library Routines

1. `sprintf`: formatted string writer.

4.10.1.6 Utility and Other Routines

1. `add_to_obuff`: add memory to buffer.

4.10.1.7 Pseudocode

- Increase error buffer allocation if necessary.
- If last line not equal to present line, write line number to error buffer.
- Add error to error buffer.

4.10.2 reset_errorbuff()

It resets the error buffer.

4.10.2.1 Usage

```c
void reset_errorbuff()
```

4.10.2.2 Interface to Data Managers

None.

4.10.2.3 Global Data Structures

1. `ebuff`: error buffer.

4.10.2.4 Local Routines

None.

4.10.2.5 Library Routines

None.

4.10.2.6 Utility and Other Routines

None.
4.10.2.7 Pseudocode

- Reset number of characters to zero.
- Reset error buffer to empty.

4.11 Input Manager

The Input Manager buffers the text from the rulebase input file. It retrieves a line of text from the rulebase input file and stores it in the input buffer. It opens the input file when initially called. The Input Manager returns the file status (unable to open, no EOL character, EOF, or OK).

Components

1. fill_ibuff()

4.11.1 fill_ibuff()

It buffers input from the input file.

4.11.1.1 Usage

int fill_ibuff()

4.11.1.2 Interface to Data Managers

1. Calls to Output Manager routine echo (via local routine).

4.11.1.3 Global Data Structures

1. ibuff: input buffer.
2. rb.fname: rulebase input file name.
3. input_fptr: rulebase input file pointer.

4.11.1.4 Local Routines

1. span_to_blank_lines: Span blank lines in input file.

4.11.1.5 Library Routines

1. file routines: fopen, getc, ungetc.

4.11.1.6 Utility and Other Routines

1. add_to_buff: add memory allocation to buffer.

4.11.1.7 Pseudocode

- If input file pointer is null.
- Open input file (return file access denied if failure).
- Reset number of characters to zero.
- Span blank lines (return EOF if end of file encountered).
- /* Fill input buffer */
While (get a character) not new line or EOF do the following.
  * Copy character to input buffer.
  * If number of characters exceed maximum.
    * Return LONG_LINE.
  * /* New line or EOF found */
  * Copy new line and string terminator to input buffer.
  * Return OK.

4.12 Output Manager

The Output Manager buffers output text to an output buffer and it
flushes the output buffer and error buffer to files. It opens an
output file on initial call. The Output Manager returns a file
status (unable to open or OK).

Components
1. append_to_obuff()
2. flush_obuff()
3. echo()

4.12.1 append_to_obuff()

It appends output strings to the output buffer.

4.12.1.1 Usage

int append_to_obuff(out_str,count)
  char *out_str: points to output string.
  int count: number of characters in output string.

4.12.1.2 Interface to Data Managers

None.

4.12.1.3 Global Data Structures

1. obuff: output buffer.

4.12.1.4 Local Routines

None.

4.12.1.5 Library Routines

1. string routines: strcat.

4.12.1.6 Utility and Other Routines

1. add_to_buff: add memory allocation to buffer.
4.12.1.7 Pseudocode

- While allocation not sufficient do the following.
- Add memory to buffer (return NO_MEMORY if no memory available).
- Concatenate output string to output buffer.

4.12.2 flush_obuff()

It flushes the output buffer to a file.

4.12.2.1 Usage

```
int flush_obuff(f_select)
    int f_select: Index to file pointer list.
```

4.12.2.2 Interface to Data Managers

None.

4.12.2.3 Global Data Structures

1. output_fname: output file name list.
2. output_fp: output file pointer list.
3. obuff: output buffer.

4.12.2.4 Local Routines

None.

4.12.2.5 Library Routines

1. file routines: fopen, fputs.

4.12.2.6 Utility and Other Routines

None.

4.12.2.7 Pseudocode

- If output file pointer is null
- Open output file (return error if file access denied).
- Write output buffer to file.
- If selected file is Rulebase output file.
- Write output and error buffers.
- Reset output buffer to empty.
- Reset error buffer to empty.

4.12.3 echo()

It writes the input buffer to a listing file (not used).
4.12.3.1 Usage

int echo()

4.12.3.2 Interface to Data Managers

None.

4.12.3.3 Global Data Structures

1. ibuff: input buffer.
2. output_fname: output file name list.
3. output_fp: output file pointer list.

4.12.3.4 Local Routines

None.

4.12.3.5 Library Routines

1. file routines: fopen, fprintf, fputs.

4.12.3.6 Utility and Other Routines

None.

4.12.3.7 Pseudocode

- If file pointer to listing file is null
- Open listing file (return FAD if file access denied).
- Write input buffer to listing file.

4.13 Utility Routines

The Utility Routines are general purpose routines that are used by more than one parser and/or manager module.

Components

1. add_to_obuff()
2. hash()
3. keyword()

4.13.1 add_to_obuff()

Add memory to a buffer.

4.13.1.1 Usage

int add_to_obuff(abuff)
    BUFFER *abuff: pointer to a buffer (input, output, or error).
4.13.1.2 Interface to Data Managers

None.

4.13.1.3 Global Data Structures

1. ibuff, obuff, or ebuff: one of the buffers.

4.13.1.4 Local Routines

None.

4.13.1.5 Library Routines

1. memory routines: malloc, free.

4.13.1.6 Utility and Other Routines

None.

4.13.1.7 Pseudocode

- If maximum allocation exceeded, return NO_MEMORY.
- Allocate memory to a temporary buffer.
- Copy contents of buffer to temporary buffer.
- Free old buffer.
- Buffer becomes temporary buffer.

4.13.2 hash()

Hash a string to an integer value.

4.13.2.1 Usage

```c
int hash(string)  
    char *string: string to use for hashing
```

4.13.2.2 Interface to Data Managers

None.

4.13.2.3 Global Data Structures

None.

4.13.2.4 Local Routines

None.

4.13.2.5 Library Routines

None.
4.13.2.6 Utility and Other Routines

None.

4.13.2.7 Pseudocode

- For each character of string
- Sum integer value of character.
- Return sum mod hash size.

4.13.3 keyword()

Test if a string is a keyword.

4.13.3.1 Usage

int keyword(string)
char *string: string to compare for match of keyword.

4.13.3.2 Interface to Data Managers

None.

4.13.3.3 Global Data Structures

1. keyword_list[]: list of keywords.

4.13.3.4 Local Routines

None.

4.13.3.5 Library Routines

1. string routines: strcmp.

4.13.3.6 Utility and Other Routines

None.

4.13.3.7 Pseudocode

- For each keyword in keyword list
- If string matches keyword, return 1.
- /* No keyword matches */
- Return 0.
APPENDIX A.I - Rulebase Builder

In IST's testbed, the preprocessing of a rulebase was performed by a filter called the `rbl preprocessor`. This is written in the "C" programming language and was intended to run under DOS 3.3.

The rulebase preprocessor is embedded in the rulebase builder which is simply a program that invokes other programs, using one's output as another's input. An alternative to using the rulebase builder is to first use the rbl preprocessor program and then a "C" compiler. The following is a description of the execution of the rulebase builder.

The rulebase builder is invoked with the rulebase file name as a command line argument (i.e. `rbl rulebase.rbl`). The rulebase builder invokes the rbl preprocessor giving it the rulebase file name as input. The rbl preprocessor then generates a file containing the C source code translation of the rulebase file, a listing file which contains the rules of the rulebase followed by their C source code translations, and a link file which contains linking information for automatically linking the rulebase into the simulation; the actual linking capability is not fully developed. An error file is created only if there are errors found in the rulebase. The four files have the same base file name as the rulebase file with extensions of ".c", ".lst", ".lnk", and ".err" respectively (i.e. `rulebase.c, rulebase.lst, rulebase.lnk, rulebase.err`).

If the rulebase preprocessor does not successfully preprocess the rulebase (i.e. creates an error file), then the rulebase builder will exit, indicating that the rulebase file needs to be corrected. In the resulting error file, an error location (line number) and error message is given for each error found in the rulebase file. The user should view the error file in order to determine and correct the errors in the rulebase file. Having corrected the errors, the user should attempt to build the rulebase again.

After the rulebase is successfully preprocessed, the rulebase builder will invoke the rulebase compiler giving it the file name of the C source code translation of the rulebase. The rulebase compiler simply invokes a C compiler; its purpose is to trap any errors and connect them back to the rulebase file. The rulebase compiler produces the object code representation of the rulebase. This file is used to link with the simulation code to create a simulation executable.

Again, if the rulebase compiler does not successfully compile the rulebase C source file, then the rulebase builder will exit, indicating that the rulebase file needs to be corrected. In the resulting error file, an error location (rule number and beginning line number) and error message is given for each error found in the rulebase C source file. The user should view the error file in order to determine and correct the errors in the rulebase file. Having corrected the errors, the user should
attempt to build the rulebase again.

The end products of the successful preprocessing and compiling of the rulebase file are a rulebase object file (rulebase.obj), a rulebase link file (rulebase.lnk), a rulebase C file (rulebase.c), and a rulebase list file (rulebase.lst). The rulebase object file and the rulebase link file are used to link the rulebase with the testbed software. The rulebase C file contains the preprocessed rulebase. The rulebase list file has the rulebase code intermixed with the resulting C code; this file can be used for debugging.

When the preprocessor translates the rules to C code, syntax checking must be done to ensure that the rules are in the proper format. Design of the preprocessor presented difficult choices about where this syntax checking should be done. The assumption was made that the rules entered would follow strict C programming syntax (as if no one makes a typing mistake). The syntax checking in the preprocessor was limited to that syntax specific to the preprocessor, not to the C compiler. Therefore, no C syntax checking was carried out in the preprocessor. It was not until compiling the "c" file that C syntax errors would be found, even though preprocessing was complete. To correct these errors, the rulebase file would be edited, sent to preprocessing, and then sent back to compiling. Catching these C syntax errors during the preprocessing stage is more desirable because it would eliminate unnecessary execution of the compiler until the rulebase file was syntactically correct.
APPENDIX A.II DATA STRUCTURE LIST

Rulebase File Name Buffer (char rb_fname[])  
A string copied from command line argument 1, contains the file name of the input rule file. Used to open the input file and to create output file names.

Output File Names (char rb_fout[], rb_error[])  
These two strings contain file names used to open output files. The file names have the same base as the input file and extensions of ".c" and ".err" respectively.

Base File Name (rb_base)  
This string holds the base name of the input file name. It is used to create the output file names by adding appropriate extensions.

Output File Name List (char *output_fname[])  
This array of two string pointers contains pointers to the output file names.

Output File Pointers (FILE *output_fp[])  
This array contains file pointers to the output files.

Input File Pointer (FILE *input_fp)  
File pointer to input file.

Buffer Structure (BUFFER)  
The buffer structure is the structure used to define character buffers. It contains a pointer to character (buff), an integer count of the number of characters (num_char), and an integer count of the size of memory allocation (alloc).

Input Buffer (BUFFER ibuff)  
The input buffer is used to hold lines of input from the rulebase input file. The input buffer holds the currently active input line.

Output Buffer (BUFFER obuff)  
The output buffer is used to hold lines of output produced by the parsers.

Error Buffer (BUFFER ebuff)  
The error buffer is used to hold error messages.

Token Buffer (char *token[])  
The token buffer is used to hold tokens from the input buffer. Tokens are null terminated strings containing non-white-space characters. The token buffer holds the currently active token.

Line Number (int line number)  
The line number holds the current line number of the input file.
Linked List Chaining Structure (CHAIN_LINK)
The linked list chaining structure is a structure containing
pointers that are used in linked lists.

Linked List Head Structure (CHAIN_HEAD)
The linked list head structure is a pointer to a CHAIN_LINK
structure. It is used for defining the heads of linked lists.

Local Symbol Node Structure (LOC_SYM_NODE)
The local symbol node structure is the structure used to define
nodes in the local symbol table. It contains a CHAIN_LINK
variable (chain) for the local symbol table list, a second
CHAIN_LINK variable (type_chain) for the local type table list,
and a character array to store the name of a symbol (symbol).

Global Symbol Node Structure (GLBL_SYM_NODE)
The global symbol node structure is the structure used to define
nodes in the global symbol table. It contains a CHAIN_LINK
variable (chain) for the global symbol table list, and a
character array to store the name of a symbol (symbol).

Local Symbol Table (CHAIN_HEAD loc_sym_tab[])
The local symbol table is an array of linked lists of local
symbol node structures. The local symbol table holds the symbols
found in the declaration section.

Local Symbol Type Table (CHAIN_HEAD loc_type_tab[])
The local symbol type table is an array of linked lists. There
is a linked list for each variable type allowed in the
declaration section. The linked lists are threaded through the
local symbol table (i.e. only one copy of each symbol exist).

Global Symbol Table (CHAIN_HEAD glbl_sym_tab[])
The global symbol table is an array of linked lists of global
symbol structures. The global symbol table holds the symbols
from the global symbol file.

Type List (char *type_list[])
The type list is an array of character strings. Each string will
be a declaration type (i.e. int, char, double, etc.).

Keyword List (char *keyword_list[])
The keyword list is an array of character strings. Each string
will be a keyword.

Rulebase Input File
The rulebase input file contains the user written rulebase. It
is the input file to the rulebase preprocessor.

A.II-2
Global Symbol File
The global symbol file contains all the symbols the user is allowed to use in the rulebase definition. The symbol file contains symbols delimited by white spaces. It can also contain C comments which are ignored. It is the input file for building the global symbol table.

Rulebase Output File
The file generated from the rule file. The output of the rulebase preprocessor is compilable "C" source code.

Error Output File
File containing errors found when translating input file.
APPENDIX A.III ERRORS

FAD
File access denied. Failure to open a file or to complete a I/O operation in general.

UEOF
Unexpected end of file.

UEOL
Unexpected end of line.

LONG_LINE
Input line too long.

X_KEYWOrd
Expecting keyword. Caused by a keyword left out of the grammar. Keyword would be replaced by whatever keyword is expected.

UX_KEYWOrd
Unexpected keyword. Caused by a keyword found when not expected.

DUP_SYMBOL
Duplicate symbol. This error occurs when a symbol in the declaration section matches a previously defined symbol.

UNDEF_SYMBOL
Undefined symbol. This error occurs when a symbol in the rule section does not match a previously defined symbol.

BAD_SYMBOL
Symbol uses invalid construction.

BAD_EXPRESSION
Bad expression. The expression did not have matching parenthesis.

NO_MEMORY
Failure to gain needed memory.
APPENDIX A.IV BNF GRAMMAR

1 Abbreviations

ASGN: ASSIGNMENT
BOOL: BOOLEAN
EXEC: EXECUTE
FUNC: FUNCTION
M_RULE: MULTIPLE RULES
S_RULE: SINGLE RULE
ST: STATEMENT
SUB: SUBROUTINE

2 Grammar

<rulebase> ::= name <identifier>
target <target identifier>
[declare <declarations>]
begin
<rules>
end

<identifier> ::= A string of characters. Most likely will have a limit on number of characters and type of characters.

<target identifier> ::= A string of characters. The name of one of several predetermined simulation objects.

<declarations> ::= <type> <identifier list> | <type> <identifier list> <declarations>
<type> ::= A declaration type. Most likely a restricted set to include integer, floating point, and character.
<identifier list> ::= <identifier> | <identifier> <~> <identifier list>
<rules> ::= <a_rule> | <rules> <a_rule>
<a_rule> ::= if ( <conditions> )
then ( <actions> )
<conditions> ::= <bool_fac> | <bool_expr>
<bool_fac> ::= A boolean variable, function, or literal. A variable must exist in local or global declarations. A function must exist in global declarations.
<bool_expr> ::= An expression of boolean factors connected by appropriate boolean operators. The expression is assumed to be in correct "C" syntax (may include function call).
<actions> ::= <asgn_st>; | <asgn_st>; <actions> | <exec_sub>; | <exec_sub>; <actions>

A.IV-1
<exec_sub> ::= Execute a global subroutine
<asgn_st> ::= <arith_ident> = <arith_expr> | <bool_ident> = <bool_expr>
<arith_ident> ::= An integer or floating point variable.
<arith_expr> ::= An expression of arithmetic factors connected by appropriate arithmetic operators. The expression is assumed to be in correct "C" syntax.
<bool_ident> ::= A boolean variable.
APPENDIX A.V BLOCK DIAGRAM
Appendix B
SAFOR System Functional Overview
TABLE OF CONTENTS

1. Introduction ................................................. B-1

2. Functional Descriptions ..................................... B-2
   2.1 Window Interface ....................................... B-2
   2.2 Help Utilities .......................................... B-2
   2.3 Rule Extractor .......................................... B-2
   2.4 Simulation Constructor ................................. B-4
   2.5 Simulation Component .................................. B-5
   2.6 Compiler/Linker ........................................ B-5
   2.7 Future Features ....................................... B-5

3. Flow of the System .......................................... B-6
   3.1 Flow of Current System Under DOS ................... B-7
   3.2 Flow of Proposed System ............................... B-9
SAFOR System Functional Overview

1. Introduction

The purpose of this document is to give a functional overview and program flow for the proposed SAFOR System. This description is to be used as a basis for the implementation of the system, from building rulebases to designing help and error reporting facilities. This document describes, step-by-step, what the user will do with the system.

Below is a block diagram of the basic high level steps a user would go through to create and use rulebases to define the behavior of an intelligent simulated force.

```
Create Rulebase -> Editor

<table>
<thead>
<tr>
<th>Global List Manager</th>
</tr>
</thead>
</table>

Assign To Target

<table>
<thead>
<tr>
<th>Build Scenario</th>
</tr>
</thead>
</table>

Select Scenario

Run

Pause

Check Point

Run From Check Point
```
The actual system will consist of many more components, such as help screens, a rule base editor, a rule preprocessor, and interface tools for creating the windows, screens, and menus. How the user interacts with these components will also be described.

The last part of the system to be described will be the future work. This includes a graphics editor which will allow the user to create diagrams of controllers for intelligent simulated entities and state diagrams for each controller. These diagrams show the flow of execution of rulebases which will simulate behavior in the entities.

2. Functional Descriptions

2.1 Window Interface

The window interface will provide the ability to view several files at once. This is important during the rulebase creation/modification phase. The screens and menus will enhance the ease of use, especially if the system is totally menu driven. This way, all input from the user will be validated. The interface will allow the calling of routines from a menu item to accomplish some task, such as file creation. This interface will interpret all keyboard and mouse input.

2.2 Help Utilities

The Help Utilities will include a series of text screens which can be accessed via menu items. Each menu item will pertain to one area of the system. Once a help menu item is chosen, another menu may appear for more detailed help. The help will be in the form of a screen of information which can be searched, paged up and down, or position at home or end points.

<table>
<thead>
<tr>
<th>HELP</th>
<th>Help For Editor Commands:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Editor</td>
<td>To insert text, position the cursor at the point where you want to insert the text. Press the 'i' key. Enter your text. Press the &lt;Esc&gt; key to exit the insert mode when you are</td>
</tr>
<tr>
<td>Errors</td>
<td></td>
</tr>
<tr>
<td>Targets</td>
<td></td>
</tr>
</tbody>
</table>

The use of a mouse is anticipated so that the user will only have to point to a menu item and click the mouse button to activate that menu item. We would also like to add a Help Editor so that help screens may be edited and added over time as the system changes.

2.3 Rule Extractor

This large part of the system is the driving force behind the design of the rest of the system. The Rule Extractor consists of two components, the Rule Base Editor and the Rule Base Builder. The Rule Extractor will eventually contain the graphics editor to create the state diagrams, as mentioned above. The purpose of the rule extractor
is to give the SME defining the behavior of some simulated force the
ability to put into the computer some rules about the force's behavior.
To do this, the user must first create a rulebase which will contain
these rules of behavior. This is done using the Rule Base Editor. The
rulebase will then be processed to convert it into C source code to be
used by the simulator. This is done by the Rule Base Builder.

2.3.1 Rule Base Editor

The Rule Base Editor is used to create a file of rules which define
the behavior of some simulated force. The rules will be in the form of
"if...then...". There will also exist two lists that the user can
access while editing the rulebase. The first is a list of testable
values which describe some aspects of the current state of a simulated
entity. The second is a list of routines which perform actions or
modifications of values to generate desired results.

2.3.1.1 Editor

The editor can be an existing software package that is run under a
windowing environment. The basic requirement is that it allow insertion
and deletion of text. Eventually, the editor may be tailored to be menu
driven with the items "ADD RULE", "DELETE RULE", "SAVE", "EXIT". ADD
RULE would prompt the user for the "if" and "then" parts of the rule.
DELETE RULE would prompt the user to select a rule via mouse or arrow
keys to delete. EXIT would question the user about saving the file
before exiting the editor.

2.3.1.2 Target Information Viewer

The target information viewer is the part of the window utilities
which allows the two lists mentioned above to be displayed while the
user is creating the rules in the editor. Because the rulebase contains
rules of behavior for a unique entity (i.e. target), the testable values
and routines which apply to that entity only will be displayed. The
majority of these values and routines will, however, be generic for most
entities.

Whenever a menu item is chosen or a command entered, a window for
the testable values and/or a window for the routines will open on top of
the editing screen. These windows may be moved around the screen and the
user should be able to scroll up and down within a window. Once a
window is open, it can remain on the screen while the user continues to
create rules. The user can choose to close the windows or they will be
automatically closed when the editor is exited.

2.3.1.3 Template

As the editor is invoked for a new rulebase, a template containing
the format of the rules will be inserted into the file as a header, with
instructions for the user to copy the template for each rule created.
If the editor becomes menu driven, then the template will be automated
by asking the user for the filename and each part of the rules.
2.3.1.4 Error Handler

The Error Handler works as a validation tool by notifying the user when an incorrect key has been hit or when the user tries to exit without saving. If the system is menu driven, error handling will be done by the menu and windows utilities. The error handler does not check syntax. This should be done by an editor or by the rbl preprocessor.

2.3.2 Rule Base Builder

The Rule Base Builder accomplishes two tasks. First, if the rulebase is to be used with the RBL inference engine, then the rulebase must be preprocessed. Next, there will exist a global list of all rulebases. Once a rulebase has been created, it must be added to this list or, if a rulebase has been deleted, it must be removed from this list. This list is the base from which a user chooses which rulebases to use in the simulation.

2.3.2.1 Rule Base Preprocessor

The preprocessor works only with rulebases created for use with the RBL inference engine. It is described in detail in Appendix A. For other inference engines, such as CLIPS, no preprocessing is needed, so the rulebase file is used as is. As more expert systems are investigated and added to the simulation, the preprocessor may have to be modified to handle the new rulebases.

2.3.2.2 Rule Base List Manager

The function of the List Manager is to create and maintain a file which lists all rulebases available to be used in the simulation system. A simple database manager may serve as the list manager. The user, at this time, is responsible for adding and deleting the rulebases in the list. The rulebase should be specified by a path and file name and a rulebase name to which the rulebase can be applied. It is the intent that this process be automated, such that when preprocessing is completed, the system will add the rulebase to the list and query the user for the rulebase name. When the user wants to delete a rulebase via a menu item, the rulebase will automatically be deleted from the list. At this point, we impose the restriction that all rulebases must have a unique rulebase name.

2.3.2.4 Error Handler

The error handling for the Rule Base Builder is more complicated because it must handle the errors from the preprocessing. These include link and compile errors. The compile errors must be associated with the rule in which they occur so that users can fix the incorrect sections of code. These error messages will be put into a file so that the user can view them.

The user should choose a menu item to preprocess a rulebase. If there are errors, a message will be displayed informing the user. The user can then choose a menu item to view the error file. Another
desirable feature would be to allow the user to view this file while editing a rulebase, similar to viewing the value/routine files. Linking errors should not occur unless something is mistyped.

2.4 Simulation Constructor

The purpose of the Simulation Constructor is to allow the user to specify the rulebases he wishes to use in the simulation. These rulebases are automatically put into an array in a "C" source code file which is then compiled and linked into the simulation.

2.4.1 Rule Base Collector

The collector allows the user to select which rulebases he wishes to use for the simulation. When invoking the collector from a menu item "SELECT RULEBASES FOR SIMULATION", a window should open for the collection array. Another window should open on the global rulebase list. The user then selects rulebases from the global list, via mouse or arrow keys, to be put into the collection array. The user may also select rulebases to be removed from the collection array. It is the intent that this process be as automatic as possible. This means that upon completion of selecting rules for the collection array, the array will automatically be generated and placed in a C file. Each array item will contain three values: the rulebase name, the address of the rulebase or filename of the rulebase, and the target type for the rulebase. As the rulebase is being added to the collection, the file containing the rulebase will be copied to a simulation compile area. When a rulebase is deleted from the collection, the file containing the rulebase is removed from the compile area.

2.4.2 Error Handler

Error handling in the simulation constructor is inversely proportional to the amount of automation. The use of menus and mouse selection reduces error processing by forcing correct choices. If the rule collection is not automated, then the error handler will be invoked if invalid files are put into the collection array. A check will be done against the global file list to make sure that only valid rulebases are in the collection array.

2.5 Simulation Component

This component includes all the source files to run the simulation. Other documents describe the simulation source modules in detail [1]. All of the source files are compiled and linked to create an executable module. The rulebase files will also be included to be linked in with the executable. The intent is to have a menu item "RUN SIMULATION" which will invoke the executable. At this point, all source files are created in the DOS environment using the VI editor. The creation of routines to describe the physical behavior of an entity is done this way. We would like to see this automated such that the user could pick another menu item to edit a source file, new or existing, and an edit window would open for that file.
After editing, the user could then pick the menu item "CREATE SIMULATION" which would compile and link the source files. Some kind of automation for creating a new entry in a makefile must be added.

2.6 Compiler/Linker

The compiler and linker are Microsoft C utilities. These would be called from a menu item as described above.

2.7 Future Features

2.7.1 Scenario Builder

The scenario builder is a future feature. An initialization file would be created containing information about the entities or force structures the user wants created at the start of the simulation. Information about the entities would include object type, location, yaw, pitch, roll, and rulebase to use, for starters. This information could be entered via menus and user prompts or the user could just edit a file with a template for the format of the force information.

This initialization file would then be copied to the simulation area, a dedicated directory, where it would be compiled and linked in with the simulation executable. Source code would have to be added to create the forces listed in the initialization file at the beginning of the simulation.

2.7.2 Control and State Editor

As stated in the introduction, we wish to allow the user to describe a simulated force in terms of a set of controllers for the force and a set of states for each controller. Each state would in turn have one or more rulebases defined for it. A graphics editor would allow the user to define the controllers by circular icons and the relationships between controllers by lines between the icons. The states would be defined similarly. When describing a simulated force, once the user has created a controller object, he can then select that object by placing the cursor on it and click the mouse or hit a function key to bring up a menu. The menu could contain actions such as "CONNECT TO OBJECT" or "CREATE STATES". If the user chose to create the state diagram by selecting the "CREATE STATES" menu item, another editing window would pop-up. The user would then create the state diagram in the same way he created the controller diagram, by creating state objects and the connections between the state objects. The user might position the cursor on a state object and bring up a menu by a mouse click or function key. The menu might contain the actions "CONNECT TO OBJECT" and "RULEBASE". If the user chooses "RULEBASE", he would get another menu with all possibilities of rulebase management, MODIFY, DELETE, PREPROCESS, BUILD, etc. By this time we hope that most of the rulebase management will be automated.

3. Flow of the System

The flow of the system will be described in this section by the steps a user has to go through to create a rulebase, link the rulebase
into the simulation, and run the simulation. At this time, the following items are not implemented on our system:

1. user interface, including window capabilities
2. help screens
3. a global list of rulebases
4. a collection array of rulebases to use with the simulation
5. a scenario builder

Section 3.1 will describe the flow of the current system with the assumption that the global list of rulebases exists and the simulation source has been changed to use the collection array of rulebases. DOS 3.3 is currently used as the underlying structure.

Section 3.2 will describe the flow of the system when the window user interface, help facilities, and scenario builder have been added.

3.1 Flow of Current System Under DOS

3.1.1 Creating a Rulebase

The user is sitting at a DOS prompt. He wants to create a rulebase to add to the system. He uses an editor (we'll assume VI) to open and create his rulebase file. The first thing he does after opening the file is to read a template file into his file. This gives the user the format of the contents of the rulebase file. He follows this format and inserts the rulebase name, the target, and his rules. If the user wishes to view the values and routines he can use for the rules, he must have printed out the source files or looked at them ahead of time.

3.1.1.1 Preprocessing A Rulebase

The preprocessing of a rule base is performed by the rbl (rule base language) preprocessor. The user should enter the command rbl from the DOS prompt giving the rule base file name as a command line argument (i.e. rbl rulebase.rbl). Three files are produced as output. One file contains the C source code translation of the rule base file. A second file contains any errors found in the rule base; this file will be empty if no errors are found. The third file contains linking information for an automated global list update feature which is as yet undeveloped. The three files have the same base file name as the rule base file with extensions of ".c", ".err", and ".lnk" respectively (i.e. rulebase.c, rulebase.err, rulebase.lnk).

If the error file is not empty, then the rule base file needs to be corrected. In the error file, an error location (line number) and message is given for each error found in the rule base file. The user should view the error file to determine the errors in the rule base file. These errors should be corrected in the rule base file, and the rule base file should be preprocessed until the rbl preprocessor produces an empty error file.

Once the rule base file has been translated into C source code without errors, the C source code file must be compiled. The source code is compiled using the rbl compiler. The user should enter the
command `cmprbl` from the DOS prompt giving the C source code file as a command line argument (i.e. `cmprbl rulebase.c`). Two files are produced as output. One file contains the object code representation of the rule base. This file is used to link with the simulation code to create a simulation executable. The second file contains any errors that are caused by compilation of the rule base C source code. The two files have the same base file name as the rule base file with extensions of ".obj" and ".err" respectively (i.e. `rulebase.obj`, `rulebase.err`).

If the compilation error file is not empty then the `rule base file` needs to be corrected. In the compilation error file, an error location (rule number) and message is given for each error found in the rule base C source file. The error locations produced from compiling the rule base C source file are given as rule numbers of the rule base file, starting with rule 1. The user should view the error file to determine the errors in the `rule base file`. These errors should be corrected in the `rule base file`, and the rule base file should be preprocessed and compiled until both the rbl preprocessor and rbl compiler produce empty error files.

3.1.2 Adding Rulebase to Global List

The rulebase has now been compiled and is ready to use, so it must be added to the global rulebase list. This list resides in a file and contains three entries for each rulebase. The first is the path and filename of the rulebase, the second is the rulebase name, and the third is the target force. Each rulebase must have a unique name. The user edits this file (again with VI) and adds a line for the rulebase. Comments at the beginning of the file will show the format of what the user should insert. A typical entry would be:

```
c:\joe\rulebase\A10_attack.rbl  A10_atk  A10
```

The better way to handle this is to use a small database utility to create a two field database. The user could add a record for each rulebase. The database system would also validate the rulebase name for uniqueness.

3.1.3 Collecting Rulebases For The Simulation

Now the user decides he wants to use his rulebase in the simulation. The user must edit (again using VI) the file which contains the collection array. This file would have comments at the beginning which would indicate the format of the array entries. The assumption is made that the user knows the "C" programming language well enough to add to the array. So the user creates an entry in the array which contains the rulebase name, the address or filename of the rulebase, and the target force type for the rulebase. A list of target types will be available in a header file that the user must look at or know ahead of time.

Next, the user must copy the rulebase file to the simulation compile area. He may have to modify the makefile to link the rulebase in with the simulation. The user must now compile the simulation by
doing a "make". If there are any errors, these will show on the screen. The user may have to re-edit files to fix the errors and then run the make again. This process continues until no errors are displayed.

If the user decides to remove a rulebase, he must delete the rulebase from the collection array by editing the file. The user must then remove the rulebase file from the simulation area by deleting it.

3.1.4 Running The Simulation

An executable file now exists for the simulation, probably named "sim.exe". To run this, the user types the filename at the DOS prompt and the simulation is executed. It is assumed that the routines have been added to allow the user to specify the rulebase the user wants to use with a simulated force. When the user creates a force during simulation, the user specifies the location in x, y, and z coordinates, the yaw (direction facing) of the force, and the rulebase. If the rulebase name the user enters does not match those in the collection, an error will be displayed and he will have to reenter the command to create the force. There should also exist a command which will allow the user to see the names of all the rulebases in the collection.

Another part of the simulation which will be implemented is the creation of a scenario initialization file. This file will contain an array of information about forces which will be created at the beginning of a simulation. The user would create or modify this file after creating the collection array. The user would define for each force the starting location, rulebase, and missions or assignments. Also in this file should be a list of default rulebases for specific target forces. This way, if a rulebase is not defined for a force, the default rulebase will be assigned to it.

3.2 Flow of Proposed System

The computer will be running a window interface and the simulation system will be represented by an icon. To start the system, the user will click on the icon. This will bring up the introductory screen with the initial choice menu. It may be desirable that, when the computer is turned on and the system running, the user immediately see the introductory screen.

The user must now choose an action from the initial menu shown below by using the arrow keys to move up and down the menu and hitting the <Enter> key when his selection is highlighted. If the system has a mouse, the user will use the mouse to move the cursor over the menu selection and click to select that menu item.
3.2.1 HELP

If the user selects HELP, then another screen would appear with descriptions of each of the items in the menu other than HELP. Depending on the amount of description necessary, the user may be asked to select another menu item or item number for more detail. One selection that will be consistent will be the CONTINUE option which will take the user back one level. The screen below could appear if HELP were selected from the main menu item.

HELP FOR MAIN MENU

1. RULEBASE - The RULEBASE selection will bring up a menu of functions which will allow you to create, modify, and delete rulebases as well as add rulebases to the global list.

2. SIMULATION - The SIMULATION selection will bring up a menu of functions which will allow you to choose rulebases for the simulation, create a simulation with the rulebases, and run the simulation. It will also provide editing capabilities for simulation source and physical behavior modeling routines and values. The user may also choose to create or modify an initialization scenario.

3. EXIT - This selection will return you to DOS.

Select a number for more information or <Enter> to CONTINUE.
Another way to allow the user to get more help would be to have another menu to the side of this screen that has the options:

- MORE ON RULEBASE
- MORE ON SIMULATION
- CONTINUE

### 3.2.2 RULEBASE

If the user selects RULEBASE, another screen will appear with the menu of functions that are related to creating and maintaining rulebases. The menu could look like this:

- HELP
- CREATE/MODIFY RULEBASE
- PREPROCESS RULEBASE
- RULEBASE LIST
- PREVIOUS MENU

Again the user would select a menu item via arrow keys and <Enter> or using the mouse to select a menu item by pointing and clicking.

#### 3.2.2.1 HELP

When the user selects the HELP menu item, a screen will appear and it will contain short descriptions of each of the menu items other than HELP. The user may be able to obtain a more detailed description by select menu items to do so, as described in section 3.2.1.

#### 3.2.2.2 CREATE/MODIFY RULEBASE

If the user selects the CREATE/MODIFY RULEBASE menu item, a screen will appear and ask him to enter the filename of the rulebase. The user may also be asked to enter a path name describing where the rulebase would reside. If the system cannot find the file, it will ask the user if he wants to create the file. If the user responds "Y"es, then the system will open an editing window for inserting into the file. This window will also be opened for a file that already exists and the contents of the file will be displayed. For a new file, the user will be asked to enter the initial information of rulebase name and target.
force and these will be displayed in their proper places in the edit window.

The structure of each rulebase file is based on the processing algorithm used to execute the rules. Therefore, it may be necessary to have a menu for different formats of rulebases. These formats would dictate the type of information that is in the template. A generic template may have an "if...then" structure which can be used for all inference engines.

The commands for the editor may be menu driven depending on which editor was selected for the system. If so, the user may have a menu like:

```
HELP
INSERT
DELETE
COPY TEMPLATE
VIEW FILE
SAVE
EXIT
```

HELP would pop-up a screen showing help on the different options. INSERT would allow the user to insert text at the cursor position. DELETE would allow the user to select the text he wants to delete and remove it from the file. COPY TEMPLATE would insert a rule template at the cursor position. SAVE would save the rulebase without exiting. EXIT would ask the user to save and then exit editing and put the user back to the rulebase menu.

VIEW FILE would show another menu of files the user can view. This would include the VALUES, ROUTINES, and ERRORS. The VALUES and ROUTINES files contain the names of values and routines which can be used for the target force of the rulebase. The ERROR file contains errors from preprocessing that the user may view in order to correct the errors. Each window would have a scroll bar which would allow the user to scroll forward and backward through the files. There would be an EXIT on the title bar of the window which would allow the user to close the window if clicked on.

### 3.2.2.3 PREPROCESS RULEBASE

If the user selects this menu item, a screen will appear and ask the user to enter the name of the rulebase to be preprocessed. The system will then display messages about the status of the preprocessing sequence. The messages will indicate if there were any errors in preprocessing and tell the user to correct the error by re-editing the file. The user will be able to view the errors through the editor, as described in the previous section.

If there are no errors, a message will be displayed that will tell the user that the rulebase may now be added to the global list if it is not already there. Another way to handle this is to tell the user that
the rulebase has been added to the rulebase list and have the system automatically do this if there are no errors.

At the completion of preprocessing, with or without errors, the user is put back at the rulebase menu.

3.2.2.4 RULEBASE LIST

If the user selects this menu item, a window will open showing the current rulebase list. The window will have scroll bars so the user can view the list backward and forward. A menu would also be resident on the screen and would contain the following items:

```
HELP
ADD RULEBASE
DELETE RULEBASE
SAVE CHANGES
EXIT
```

The user would select one of the menu items to allow him to modify the list. If the user selects HELP, a help screen would be displayed giving the user descriptions of what the menu items would do and how to use them. If the user wants to add a rulebase to the list, he would select ADD RULEBASE and the system would prompt him for the path and rulebase file name as well as the rulebase name. If the user chooses to delete a rulebase from the list, he must select the rulebase to delete from the list by moving the mouse so that the cursor is on the rulebase and clicking to select it. Then the user must choose DELETE RULEBASE from the menu. The system would check the collection list to see if the rulebase is currently being used by the simulation. If so, a warning would be displayed on the screen indicating this to the user. The user might be told that he must delete the rulebase from the collection list before he could delete it from the global list. Or, the system might ask the user if he wants to delete the rulebase from the collection list. If he responds "Y"es, the system would modify the collection list to delete the rulebase, remove the rulebase from the simulation area, and remove the rulebase from the global list. If the system would not modify the collection list, then it would only remove the rulebase from the global list.

If the user selects SAVE CHANGES, the system would save all changes to the global list. If the user selects EXIT, the system would ask the user if he wants to save, exit modifying the global list, and return the user to the rulebase menu.

3.2.2.5 PREVIOUS MENU

This menu selection will take the user back to the main menu by closing all windows and screens opened after the main menu.
3.2.3 SIMULATION

If the user selects the SIMULATION menu item, then a new screen will appear with a menu of functions related to creating and maintaining the simulation. This menu could look like:

<table>
<thead>
<tr>
<th>HELP</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE SIMULATION</td>
</tr>
<tr>
<td>RUN SIMULATION</td>
</tr>
<tr>
<td>PREVIOUS MENU</td>
</tr>
</tbody>
</table>

The user would select a menu item via arrow keys and <Enter> or using the mouse to select a menu item by pointing and clicking. PREVIOUS MENU will close this menu screen and re-display the main menu.

3.2.3.1 HELP

When the user selects the HELP menu item, a screen will appear which will contain short descriptions of each of the menu items other than HELP. The user may be able to obtain a more detailed description by selecting menu items to do so, as described in section 3.2.1.

3.2.3.2 CREATE SIMULATION

When the user selects this menu item, another screen will appear with a menu of functions involved in the creation of a simulation. This menu could look like:

<table>
<thead>
<tr>
<th>HELP</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODIFY RULE COLLECTION</td>
</tr>
<tr>
<td>CREATE/MODIFY SCENARIO</td>
</tr>
<tr>
<td>SWITCH SCENARIO</td>
</tr>
<tr>
<td>MODIFY SOURCE</td>
</tr>
<tr>
<td>CREATE SIMULATION</td>
</tr>
<tr>
<td>PREVIOUS MENU</td>
</tr>
</tbody>
</table>

The user should select an item by the methods discussed. PREVIOUS MENU will close this window/screen and re-display the SIMULATION menu.
3.2.3.2.1 HELP

The user can select this menu item to get a screen full of information on the other menu items listed. If the user desires, another menu may be selected allowing the user to request more detailed help on that item. The user can press the <Enter> key to continue or select CONTINUE from the detailed help menu to get back to the CREATE SIMULATION menu.

3.2.3.2.2 MODIFY RULE COLLECTION

The user selects this menu item if he wants to change the rulebases used by the simulation. The system will open two windows on the screen. The first window will show the contents of the global rulebase list. The second window will show the contents of the current rule collection. A menu will contain functions which will allow the user to modify the collection. The menu will contain the following:

```
HELP
ADD RULEBASE
DELETE RULEBASE
SAVE CHANGES
EXIT
```

HELP will give the user a screen of help descriptions for each of the menu items. SAVE CHANGES will save the contents of the rule collection list to a file and EXIT will ask the user to save, exit modifying the rule collection, and bring up the CREATE SIMULATION menu.

To add a rulebase to the collection, the user selects the ADD RULEBASE menu item. The user is then told to select the rulebase to add from the global rulebase list by moving the cursor to the rulebase in the global rulebase window and either pressing <Enter> or clicking with the mouse. The system then adds this rulebase to the collection and it will be displayed in the collection window. The system will copy the rulebase from its home directory to the simulation directory and add it to the list of files to be linked in with the simulation executable.

If the user wants to delete a rulebase from the collection, he must select the DELETE RULEBASE menu item. The system will tell the user to select the rulebase to delete from the collection window by arrow keys and <Enter> or mouse and click. The system will remove the rulebase from the collection file, remove the rulebase file from the simulation directory, and remove it from the list of files to be linked in with the simulation executable.

NOTE: One problem of implementation is that there will be certain rulebases used as default rulebases for specific simulated entities. We don't want to allow the user to delete these default rulebases from the
simulation area unless he replaces them with other rulebases.

3.2.3.2.3 CREATE/MODIFY SCENARIO

The user selects this menu item when he wants to create a scenario to be present at the start of the simulation.

3.2.3.2.4 SWITCH SCENARIO

3.2.3.2.5 MODIFY SOURCE

When the user chooses this menu item, a warning will be displayed that only a qualified programmer may modify source. The user will be asked to continue, Yes or No. The modifying of source can be handled in many ways. The user may be restricted to creating or modifying routines that deal with entity behavior and not allowing the user to access any other simulation source code. This is probably the best way. Therefore, if the user answers "Yes" to continue, a menu appears with such items as LIST FILES, EDIT FILE, COMPILE FILE, and EXIT. LIST FILE would show files in the directory that contains the routines and values. Any editor may be used for editing files. The editor should also allow the user to view an error file if there are any errors from compiling. COMPILE FILE will ask the user for the C file name and do a compile of that file, redirecting any errors to an error file. The user will be notified if there are errors. EXIT will return the user to the CREATE SIMULATION menu.

3.2.3.2.6 CREATE SIMULATION

If this menu item is chosen, it is assumed that the user has chosen the rulebases and the scenario to start the simulation with. A message should appear to remind the user to do this if it hasn't already been done. The user should be asked to continue with creation and if the response is "Yes", then the system proceeds to compile files and link them together to form the simulation executable. If the response is "No", the user will be returned to the CREATE SIMULATION menu. If any errors occur during compiling and linking, the user will be notified by a message displayed on the screen. The user will be directed to take action and told that the errors are in a file. At this point, the system should be so automated that the errors would occur only when modifying source. When the user acknowledges the error problem by pressing <Enter>, he is placed back at the CREATE SIMULATION menu.

3.2.3.3 RUN SIMULATION

When the user selects this menu item, the simulation will start running, initially creating all simulated entities specified in the scenario the user chose. The user can then enter commands to manipulate the entities and display or just watch. It is the intent to automate these commands by using menus. When the user wishes to stop the simulation he selects the menu item STOP SIMULATION. This should bring the user back to the SIMULATION menu.

Another way to start the simulation is from a checkpoint. A
checkpoint file is created when a running simulation is stopped and information that defines the simulation is saved. The simulation can be restarted from the checkpoint by using the saved information to re-initialize.

To provide the user or operator with the ability to control a running simulation, the following commands could be part of an interactive simulation menu:

- CANCEL A SIMULATION
- CHECKPOINT AND CONTINUE
- CHECKPOINT AND QUIT
- SUSPEND SIMULATION
- RESTART SIMULATION
- RESTART FROM CHECKPOINT.
Appendix C
SAFORT Global List Manager
## TABLE OF CONTENTS

1. Introduction .............................................. C-1

2. Functional Descriptions .................................. C-1
   2.1 global_list_manager() ................................. C-1
   2.2 add_rulebase_name() ................................. C-3
   2.3 delete_rulebase() .................................... C-4
   2.4 quit_execution() ..................................... C-6
   2.5 open_file() .......................................... C-7
   2.6 enter_names() ........................................ C-8
   2.7 file_insert() ........................................ C-9
   2.8 search_rb() .......................................... C-10
   2.9 remove_rb() .......................................... C-12
   2.10 display_menu() ....................................... C-13
SAFOR Global List Manager

1. Introduction

This document is a specification of the Global List Manager. This module will be part of the Rule Extractor. The Global List Manager maintains and updates a file that contains a list of rulebases available for the simulation. Each record in this "global" list contains three fields. The first field is a unique rulebase name, the second is a rulebase file name with a directory path, and the third is a target name. Each field has a maximum length, the rulebase name is 60 characters, rulebase file name is 256 characters, and the target name is 70 characters. The Global List Manager maintains each record on separate lines. This is done by adding a carriage return at the end of each record. The following is an example of the file layout:

```
Germany    d:\list\ruleB1.kb    target1_name
Italy      d:\list\ruleB2.kb    target2_name
France     d:\list\ruleB3.kb    target3_name
USA        d:\list\ruleB4.kb    target4_name
          
          
Panama     d:\list\ruleB8.kb    target8_name
```

The main functions of the Global List Manager are to add records to the global list and delete records from the global list. The program will provide a simple user interface with a menu. The menu will have three choices to select from, ADD, DELETE and EXIT. The user can select a menu item by entering a number at the prompt. To add a new record, the Global List Manager will prompt the user for a rulebase name, a rulebase file name, and a target name. When the three names are entered they will be stored in a record in the global list. The Global List Manager uses the rulebase name of each record to organize the list in alphabetical order. The system will search for the correct alphabetical position of the new record in the global list file. When the Global List Manager finds the correct position in the global list file, it will add the record at that position. To delete an exiting record from the global list file, the global list file will prompt the user for a rulebase name, search for its position in the global list file, and remove the record by moving up the rest of the records in the global list manager.

2. Functional Descriptions

2.1 global_list_manager()

2.1.1 Function Description

This is the top level routine of the Global List Manager. This module calls a routine that has a menu for the user to select one of the following functions, delete, add and exit. The user must enter the number of a selection at the prompt. This value must match one of the possible selections. Then, a "switch" statement transfers control of the program to the sections of code that handles implements the
selected task. After the selected task has completed, the function will return to the main module. This will be repeated until the user exits the program.

2.1.2 Inputs

None

2.1.3 Outputs

None

2.1.4 Functions

2.1.4.1 Library Functions

1. int printf()

formats and prints a series of characters to a standard output stream.

2.1.4.2 Local Functions

1. void display_menu()
   displays functions and returns selection
2. void add_rulebase_name()
   add a rulebase name to the global list
3. void delete_rulebase_name()
   delete a rulebase name from the global list
4. void quit_execution()
   quit from program
5. void open_file()
   open the global list file

2.1.5 Sample Code

/* This structure would be part of a header file */
struct list_n
{
    char dir_name[256];
    char kb_name[60];
    char target_name[70];
};

void global_list_manager()
{
    File *fp_in;
    /* pointer to a file */
    struct list_n *list;
    /* pointer to structure of names */
    char selection[1];
    /* This array stores the selection */
open_file(fp_in);
    /* opens the file global to read and write */

display_menu(selection);
    /* displays menu selections */

/* Based on the selection the switch statement transfers control to one of the three cases */
switch (selection[0]){
    case '1':
        add_rulebase_name(struct_name);
        break;
    case '2':
        delete_rulebase(struct_name);
        break;
    case '3':
        quit_execution(fp_in);
        break;
}

2.1.6 Called By
None

2.1.7 Header Files

global.h

2.2 add_rulebase_name()

2.2.1 Function Description

This function prompts the user for the rulebase name, rulebase file name and target name. All three names entered are stored in one structure. Then the function calls a search routine to find the correct file position for the new record to be added. Because the global list file is organized by rulebase name in alphabetical order, the search only looks at the first field in the record. Next, a function to insert the new record is called. This function will write the new record at the specified location in the global list file and shift the current records from that position to the end of the file to accommodate the new record.

2.2.2 Inputs
struct list_n *struct_name /* pointer to a structure of names */

2.2.3 Outputs
None
2.2.4 Functions

2.2.4.1 Library Functions

None

2.2.4.2 Local Functions

1. void enter_names()
   scan for the three names for the record and store them in a structure
2. void search()
   find the correct file location
3. void file_insert()
   inserts the record in the global list

2.2.5 Sample Code

/* This structure would be part of a header file */
struct list_n
{
char dir_name[256];
char kb_name[60];
char target_name[70];
};

void add_rulebase_name(struct_name, fp_in)

struct list_n *struct_name;
/* pointer to a structure of names */
File *fp_in;
/* pointer to a file */
{
   enter_names(struct_name);
   /* prompt users to enter names */
   search_rb(rulebase-name, fp_in);
   /* search file for rb name */
   file_insert(struct_name, fp_in);
   /* write new record to file */
}

2.2.6 Called By

global_list_manager()

2.2.7 Header Files

global.h

2.3 delete_rulebase()
2.3.1 Function Description

This function allows the user to delete a rulebase from the global list file. The system will prompt the user for a rulebase name. Then the function calls a search routine to find the location of the record in the file. Because the global list file is organized by rulebase name in alphabetical order, the search only looks at the first field in the record. Once the correct record is found, a remove routine will be called to remove the record from the global list file. This will be implemented by reading in all records that are located after the record to be deleted. These records are stored and written back to the global list file starting at the location of the record to be deleted. Hence, the record will be removed from the list by writing over.

2.3.2 Inputs

1. struct list_n *struct_name
   pointer to structure with names.
2. FILE *fp_in
   pointer to the file.

2.3.3 Outputs

None

2.3.4 Functions

1. int printf()
   formats and prints a series of characters to a standard output stream.

2.3.4.1 Local Functions

1. void search_rb()
   find the rulebase name in the list.
2. void remove_rb()
   remove the rulebase name record from the list.

2.3.4.2 Library Functions

1. int printf()
   formats and prints a series of characters to a standard output stream.
2.3.5 Sample Code

    /* This structure would be part of a header file */
    struct list_n
    {
        char dir_name[256];
        char kb_name[60];
        char target_name[70];
    }

    #define MAX 200   /* Maximum number of name in the database */

    void delete_rulebase(struct_name,fp)
    {
        struct list_n names[MAX];
        /* array of structures */

        printf("\nEnter the name of the rulebase that you want to
delete");

        search_rb(rulebase-name)
        /* return pointer location of string*/

        remove_rb(fp_in,array)
        /* remove record from global list file */
    }

2.3.6 Called By

    global_list_manager()

2.3.7 Header Files

    global.h

2.4 quit_execution()

2.4.1 Function Description

The user can exit the Global List Manager program by selecting the
exit item from the main menu. The quit_execution() function calls a
function to close the given stream. Then the program exits by
terminating the calling process.

2.4.2 Inputs

    fp_in     /* string file pointer. */

2.4.3 Outputs

    None
2.4.4 Functions

2.4.4.1 Library Functions

1. int fclose() close the file
2. int exit() exit the program

2.4.4.2 Local Functions

2.4.5 Sample Code

quit_execution(fp_in)
{
    fclose(fp_in); /* close file before you leave */
    exit(1); /* terminate program execution */
}

2.4.6 Called By

global_list_manager()

2.4.7 Header Files

None

2.5 open_file()

2.5.1 Function Description

The name of the file containing the global list will be GLOBAL. If the file does not exist the Global List Manager will create the file by calling the open file function which returns the pointer to the file. Otherwise the open_file function will print a message indicating it was not successful in creating the file.

2.5.2 Inputs

char *fn_in string pointer.

2.5.3 Outputs

None

2.5.4 Functions

2.5.4.1 Library Functions

1. int fopen() open file
2. int printf() formats and prints a series of characters to a standard output stream.
2.5.4.2 Local Functions

None

2.5.5 Sample Code

```c
void open_file(fn_in)
{
    FILE *fopen(), *fp_in;
    char fn_in[12];

    fp_in = fopen (fn_in, "w+"); /* open file to read and write */
    if (fp_in == NULL) /* checks if there was a problem opening the file */
        printf("\nThe file %s was not created\n",fp_in);

    else

}
```

2.5.6 Called By

```
global_list_manager()
```

2.5.7 Header File

None

2.6 enter_names()

2.6.1 Function Description

This routine prompts the user for the rulebase name, rulebase file name with a path, and the target name. The function expects as input a pointer to a structure of type list. This structure is a template for the record of three fields where the three names are stored. Each name is stored in one of the fields of the record. There is a limit to the number of characters allowed for the different names.

2.6.2 Inputs

```
struct list *n_list[]        pointer to array of structures.
```

2.6.3 Outputs

```
None.
```
2.6.4 Functions

2.6.4.1 Library Functions

1. printf() print to screen
2. scanf() get input from the user

2.6.4.2 Local Functions

None

2.6.5 Sample Code

/* This structure would be part of a header file */
struct list_n
{
    char dir_name[256];
    char kb_name[60];
    char target_name[70];
};

void enter_names(n_list)
struct list_n *n_list; /* pointer to a structure */
{
    /* This prompts for each name and scans each string entered */
    printf("\nPlease enter the name of rule base\n");
    scanf("%s", n_list->dir_name);

    printf("\nPlease enter the name of the file and a path\n");
    scanf("%s", n_list->dir_name);

    printf("\nPlease enter the target name\n");
    scanf("%s", n_list->target_name);

2.7 file_insert()

2.7.1 Function Description

This function writes a string to a file at the current file pointer location.

2.7.2 Inputs

struct list_n *struct_name[i] struct element in array
File *fp_in file pointer to global list
2.7.3 Outputs

None

2.7.4 Functions

2.7.4.1 Library Functions

1. fprintf() formats and prints a series of characters to output file

2.7.4.2 Local Functions

None

2.7.5 Sample Code

/* This structure would be part of a header file */

```c
struct list_n
{
    char dir_name[256];
    char kb_name[60];
    char target_name[70];
};
```

```c
file_insert(fp_in, struc_name)
{
    File *fp_in;
    struct list_n *struc_name[i];

    fprintf(fp_in, "%s", struc_name[i]->dir_name);
    fprintf(fp_in, "%s", struc_name[i]->kb_name);
    fprintf(fp_in, "%s", struc_name[i]->target_name);
}
```

2.7.6 Called By

add_rulebase()

2.7.7 Header Files

stdio.h

2.8 search_rb()

2.8.1 Function Description

This function searches the file alphabetically using a binary search. First the search finds the location of the root node, this is the middle record in the global list file. This divides the file in two half. The upper half are all the records that are greater than the middle record and the lower half are all the records that are less than the middle record. Secondly the rule base name being searched for is compared to the first field of the middle record. If the two
strings are not equal the search moves on to one of the two half's depending if the rule base name is greater or less than the string it is being compared to. The search will continue dividing the file until it can find the string or the interval becomes empty.

2.8.2 Inputs

\begin{verbatim}
struct list_n *struct_name[i]
    struct element in array

File *fp_in
    file pointer to global list
\end{verbatim}

2.8.3 Outputs

\begin{verbatim}
File *fp_in    file pointer to global list
\end{verbatim}

2.8.4 Functions

2.8.4.1 Library Functions

1. sizeof() returns the length of file in bytes
2. strcmp() compares string to see if they are equal

2.8.4.2 Local Functions

None

2.8.5 Sample Code

\begin{verbatim}
search_rb(rulebase-name, array-of-structures)
char *rulebase-name;
{
    int lo, mid, hi, strcmp; /* integer variable declaration */
    lo = 0; /* initialize to zero */
    /* This gets the number of records in the file */
    hi = sizeof(file)/sizeof(record);

    /* This is the binary search */
    while (lo <= hi) {
        mid = (lo + hi)/2; /* this is the middle value */

        /* This function compares string1 and string2 lexicographically and returns a value indicating their relationship */
        cmpresult = strcmp(field1-record, rulebase-name);
        if (cmpresult < 0) /* check if it is less than */
            lo = mid + 1; /* middle lower half of the file */
        else if (cmpresult > 0) /* check if it is greater than */
            hi = mid - 1; /* middle upper half of the file */
        else
            return mid; /* return the middle value */
    }

    return NOTFOUND; /* if string does not exist return the last node */
}
\end{verbatim}
2.9 remove_rb()

2.9.1 Function Description

This function removes the rulebase name, rulebase file name and target name associated with it. This will be implemented by reading in all records that are located below the record to be deleted. The records read are stored in an array of structures. Then they are written back to the global list file starting at the location of the record to be deleted. Therefore, the deleted record is removed from the file simply by writing over its location in the file.

2.9.2 Inputs

1. FILE *fp_in  
   file pointer to global list

2.9.3 Outputs

None

2.9.4 Functions

2.9.4.1 Library Functions

1. int fread()  
   read a string from a file
2. int fwrite()  
   write a string to a file
3. int fseek()  
   change file pointer

2.9.4.2 Local Functions

None

2.9.5 Sample Code

remove_rb(fp_in)
File *fp_in;
{
    File *fp_out;
    while(ch != EOF){  /* check for end of file */
        while (fp_in != fp of rulebase to be deleted){
            read from in file
            fwrite to out file all rule base name from infile
            if(fp_in = fp location of rulebase to be deleted)
                fp_out = fp_in + increment by one string length
        }
    }
}

2.9.6 Called By
2.9.7 Header Files

    io.h
    global.h

2.10 display_menu()

2.10.1 Function Description

This function displays a simple menu. The menu has three selections for the user. The user can make a selection by entering a number. The number is stored in an array and passed to the next routine.

2.10.2 Inputs

    char selection[]       array to store selection

2.10.3 Outputs

    None

2.10.4 Functions

2.10.4.1 Library Functions

    1. printf()          formats and print a series of characters to a standard output stream
    2. _clearscreen()    clears the screen for text
    3. scanf()           read input from the keyboard

2.10.4.2 Local Functions

    None
2.10.5 Sample Code

```c
display_menu(selection)
char selection[1]; /* This is an array passed to the function */
{
    int i;

    _clearscreen(_GWINDOW); /* clears the screen */

    /* This displays the menu options on the screen */
    printf("\n\n Welcome to the GLOBAL LIST MANAGER.");
    printf("\n Please make any selection by entering a number");
    printf(" 1. Add a rulebase name to the global list\n");
    printf(" 2. Delete a rulebase name from the global list\n");
    printf(" 3. Exit the program");

    /* This scans for one of the input values */
    printf("\n Enter a selection number: ");
    scanf("%d",&i);
    selection[0] = i;
}
```

2.10.6 Called By

1. global_list_manager()

2.10.7 Header Files

```c
stdio.h
```