AN IMPROVED INTERFACE SIMULATION ARCHITECTURE

Technical Report 94-13

Shannon Wyeth Price

Department of Computer Science and Engineering
Auburn University
Auburn University, Alabama 36849-5347

September 7, 1994

An Improved Interface Simulation Architecture

Final Project for
Master of Computer Science and Engineering Degree
Auburn University
Department of Computer Science and Engineering

Shannon Wyeth Price
Submitted 4 August 1994

Committee Members
Dr. L. A. Moore, Chair
Dr. Kai Chang
Dr. C. L. McCreary

Introduction

Changes made to system requirements late in development are more costly than revisions made early in development. There is, therefore, significant motivation to reduce the number of changes to a proposed system as development proceeds. Simulating a system allows developers to clarify and iteratively refine requirements specifications early in development. NASA has requested that a simulation architecture be developed that will allow developers to demonstrate the functionality of proposed systems and to evaluate the effectiveness of the user interface during the requirements definition phase of development. The process, which will be applied to on-board displays for Space Station Freedom [Moore,1992], will allow developers to iteratively refine and clarify requirements before full-scale development (i.e. flight-code development) begins. Moore [1993a] has described a simulation architecture which allows for design of the displays, building of a low fidelity simulation of the system, establishment of a real-time connection between the interface and the simulator, and integration of an interface evaluation tool. Later, the interface that has been developed will be connected to a high fidelity simulator and finally to the actual on-board flight software. Minimizing the number of changes that need to be made during flight-code development will significantly reduce development costs.

A Simulation Architecture

Two common problems that arise in any developed system (computer system or otherwise) are incomplete or inaccurate conceptions of the system requirements by developers and lack of focus on users' needs as the main goal of design. A lack of understanding of the requirements leads to an error-ridden system or a system that hinders user productivity. If a computer system is developed to file tax returns can the system be considered an asset if calculations have to be double checked? The feel and perceived usefulness of a system are major factors when users begin to use a system. If users do not feel comfortable or feel hindered using a system, they will not use it effectively (if at all) regardless of the quality of the system.

Simulation of a system provides a means of studying a system with minimal expense or risk and an opportunity to study the effects of variations in the system. A system simulation is implemented after the requirements specification (a user-oriented analysis) and prior to the design specification (a system-oriented implementation of the requirements specification). It therefore facilitates a deeper understanding of the behavior of the system in that it tests the

developer's understanding of the problem at hand as well as providing a "bridge" between users and system developers. If the developer overlooked certain possibilities or inaccurately represented certain aspects of the system, the simulation would help to identify the oversights.

The success of a system depends to a great extent on the comfort of the user. If users are unhappy with the feel of the system it will impact the perceived usefulness of the system. Implementing a prototype of the user interface as the system simulation is developed allows evaluation of various designs for the layout of the interface. The developer is able to study the effectiveness of the user interface earlier in development, when users needs are the primary focus. If users are unhappy with some portion of the interface, the developer will be able to change that portion to better meet the users' needs. Ideally, some interface design options would be left undecided at simulator development time (e.g. whether interaction will come from a keystroke on a keyboard or from the press of a mouse button) and the user would be able to choose the most preferred option. The interface simulation will be replaced by a high-fidelity model later in development. Beginning user interface evaluation earlier with a low-fidelity model will allow developers to clarify system specifications and develop a more effective interface.

A system to develop simulations to provide the analysis mentioned above can be illustrated by the diagram shown in Figure1. This structure, or architecture, keeps the system to be modeled

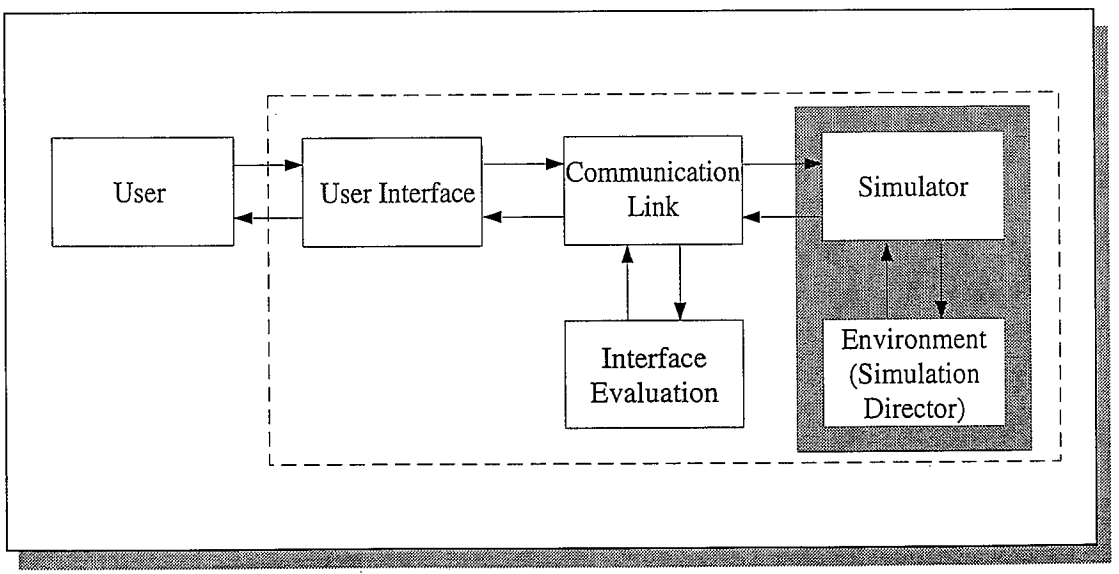

Figure 1 - Simulation Architecture

separate from the user interface to facilitate the comprehensive analysis desired. The user interface is able to send inputs to and accept outputs from the model. User interaction will be recorded and analyzed by a separate evaluation component. The external environment may also affect the model and, as shown in the figure, the simulator may also be exchanging information with the environment. Notice that the system environment is not embedded in the simulator. The environment will be controlled by a "simulation director" who will sit at a separate terminal and simulate environmental changes (e.g. weather and other random events including failure of system components).

Clearly, there are four components of the architecture:

- user interface prototype

- system simulation (including the system model and the environmental model)

- a dynamic, interactive interface between the interface and the simulator to allow for real-time interaction.

- an evaluation component for analyzing the effectiveness of the user interface design.

The user interface and system simulations will be evaluated separately. The systems will be prototyped early in development to clarify any misunderstood requirements. The interface prototype will be analyzed using an interface evaluation tool developed at Auburn and iteratively refined until it meets all use and system requirements. The system simulation will be thoroughly tested to verify correctness.

The project outlined in this paper will demonstrate the simulation component of the architecture. The system to be modeled will be an automobile. Studying this familiar system will show how an interface (dashboard and controls) and a low fidelity simulation (of the engine) may be quickly developed, executed in real-time,and evaluated.

Literature Review

New and well-established techniques exist for effective simulation development and evaluation. Care should be exercised early in the development effort to decide if a simulation is appropriate, to identify the goals of the simulation, and to choose the proper implementation and evaluation techniques. Recognizing the capabilities and limitations of simulations will make their use more effective.

Simulation is a risk reduction method. It is typically used as a design tool for clarifying requirements between users and developers, testing proposed solutions and modeling design alternatives. Taylor and Brown [1990] state that developers should have a clear understanding of what the simulation is trying to achieve, making as explicit as possible which system components are taken as given and which aspects of the system they are trying to clarify. Understanding the types of conclusions which may be validly inferred from simulations is a critical skill for simulation design.

Taylor [1990] insists that the simulation should attempt to find fault in a model or design rather than attempting to reinforce the designers' system conception or subtly coerce users' opinions. The users should evaluate the interface subjectively stating what they like or dislike about it. The design should assist communication between the developer and users allowing the developer to acquire a clearer conception of the user's wishes and needs and allowing the user to review the proposed system.

As developers evaluate the simulation, some aspects of the system may be better understood, some may be better approximated, while some should not be considered. Simulations may be used for selection and training of users, examining the interdependence of different functions within the system, comparing design alternatives, and identifying characteristic errors or failure modes. Qualities such as capacity of the system or optimum strategies may be analyzed, however, results should be viewed as very rough approximations. Qualities such as effects of the system on human stress levels and how far the user will trust the system should not be included in analysis [Hopkin, 1990].

Simulation and development of the human-computer interface(HCI) must receive special consideration. Downton [1991] asserts that while design methods for conventional engineering and computer science problems are for the most part well developed and documented, effective

communication with users seems to present a challenging problem whose solution insists that special attention be paid to the indeterminacy of human behavior. HCI analysis incorporates other fields of study including: psychology, ergonomics, linguistics, sociology, anthropology, graphic design, and typography. Consider how communication is affected by human vision. Developers must consider how luminance, contrast, brightness, visual angles, visual fields, and colors affect user learning, comprehension and speed of performance. The quality of information affects the ease of learning, and insight from the fields mentioned will improve the quality [Downton, 1991].

Users may also be able to gain experience with the simulated system. This experience is not practice for the user, but facilitates a more in-depth evaluation of user performance. Developers may observe how users deal with unexpected events to understand the strategies users prefer and the types of mistakes users make consistently. Normally, individual differences are an unwanted source of variance, but evaluation may show that training and design can be tailored to individual strengths and weaknesses [Hopkin, 1990]. Poor implementation choices will prove costly and eliminate the risk reduction advantages of simulation. Life [1990] suggests that one alternative developers may choose is to model the system after a system the users are already familiar with. Such a simulation takes full advantage of the skills users already have. Users have already developed interaction strategies and have more positive attitudes toward the model.

When users have developed strategies for different situations, they are able to react to each situation better and better. System events become familiar and while they may occur randomly, they are not unexpected. As the users work with the interface simulation, they will come to understand how different variables are related and how likely variables are to change. This understanding is crucial to an interface. Variables that do not change often will not be monitored as closely as other values. Also, if two variables are related (one value may be derived from knowledge of the other), only one of the variables can be displayed and the other value can be monitored without the user actually viewing it [Bainbridge, 1990]. This type of informative analysis can only be obtained from real system interaction. Simulation of the interface allows this type of analysis with much reduced risk.

Clearly, simulation focuses system development on the users' needs. Previously, system development has been system-centered. The users' needs were only a starting point. Thereafter, these needs were coerced to match designs and subsequent re-designs. The result was a completed system which did not fully meet the needs of the users. Floyd [1984] asserts

that the user-centered results obtained from design and evaluation of a simulation should become commitments for final system development. Serious problems regarding acceptance of the design should be expected if features of the user interface are modified without the consent of the user. After the goals of simulation have been established, the model may be implemented. Knowing how the simulation is to be used will help simulation designers decide on a proper level of detail (fidelity) for the model. There is a tendency to include too much detail in a simulation. Often only a portion of the system may warrant simulation. Partial system simulation should be approached cautiously, however, since task interdependencies could be neglected [Taylor, 1990]. Diaper [1990] mentions that it is important to separate the critical aspects of the system from those with little effect on the performance of the simulation and real system. The simplest models may consist solely of interviews with users (perhaps in the form of questionnaires). High fidelity models may use graphical displays and animation to test various system qualities or to appear more realistic. Developers must remain objective. There is a strong tendency to equate elaborateness of the simulation with the validity of the findings obtained. The quest for more realism can sometimes be pursued for its own sake, especially if there is evidence that a low fidelity model would be adequate to obtain valid findings [Hopkins, 1990]. A low fidelity simulation will be adequate for clarifying system requirements and demonstrating the effectiveness of the interface. Later, higher fidelity simulations will be built using the conclusions drawn from this initial design and more detailed evaluation and refinement will be possible.

After gathering system requirements and concluding that a system simlation would be helpful, developers must choose a simulation technique to develop a prototype. Simulation languages and tools are in widespread use and offer many different features. Simulation languages have been in use since the 1950's. Languages such as Simscript II.6 and GPSS provide timing routines, statistic-gathering mechanisms, automatic report generation for easy implementation and analysis of models [Bulgren, 1982]. Many developers prefer traditional programming languages such as Pascal, FORTRAN or C because of familiarity, flexibility or availability. Recently, simulation packages have been developed which are easy to use, provide timing, data-collection, report generation routines, and usually use graphics and animation to make development and execution of the model easier to comprehend [McHaney, 1991]. The automobile simulator will be developed using a rule-based language, CLIPS and a Petri net-based system, PERCNET. CLIPS is completely compatible with the Ada and C languages. PERCNET uses a graphical interface for rapid modeling of systems.

A major misconception is that the rapid development of simulations leads to poor design choices (especially in a simulation language). In actuality, the need to rapidly modify a model leads to a stronger tendency for a well-defined structure and readability in the simulator than in the real system. When traditional languages are used to implement the simulation, this well-designed code may be re-used as development begins [Riddle, 1984].

When the simulation has been completed, its execution results must be evaluated. Evaluation must be restricted to the simulation goals which were identified prior to development (Hopkin [1990] provides additional insight to the propriety of simulation goals). Simulation results may be analyzed in a variety of ways. Mathematical evaluations identify the nature of the occurrence of system events. Events may be random, discrete, continuous or some combination. Each of these circumstances has its own mathematical methodologies for evaluation. Furthermore, each of these is supported by simulation languages and tools [Kreutzer, 1986]. Thorough testing (using a structured programming approach) and reviewing automatically generated reports and execution traces are other methods of analyzing simulations [McHaney, 1991].

Sathre [1988] believes that a simulation engineer will help make the best choices as simulation data is evaluated. This person should have knowledge of and experience with different simulation techniques. He outlines three areas that make up a qualified simulation engineer: knowledge base, skills and aptitudes. The knowledge base includes all subjects learned previously by the engineer. Mathematics has traditionally been the most important field in the simulation profession, since simulation events can be described in terms of distribution and set theories. Skills include proficiencies or expertise in areas covered by the model. Knowledge of the system to be modeled is the most important skill. Desirable aptitudes, or personal qualities, are good judgement, adaptability to change and good communication skills.

User interfaces may be further evaluated in a number of ways. If a user interface expert is available, heuristic evaluations, which identify properties known to lead to usability problems, are the best evaluation method. Usability tests may be designed to evaluate the performance of the interface under "real-world" conditions. Published guidelines, which supply evaluators with recommendations about the design of an interface, such as how the contents of a screen should be organized, can assist evaluators who are not user interface experts. Finally, the cognitive walkthrough method allows developers to walk through the interface with the users identifying discrepancies between the users expectations and the proposed interface [Jeffries, et al., 1991].

PROJECT

The need for and benefits of the architecture described in Figure 1 may be further understood with an example. A system that most people are familiar with is an automobile. Most people are so familiar with it that calling it a system seems to stretch the imagination, but consider that an automobile is a machine with many interacting components that performs a task. The driver (or user) monitors and controls the automobile's performance using pedals, levers, gauges and of course, a steering wheel. The dashboard and controls are the user interface and the engine is the main part of the system. Mapping the automobile system to the simulation architecture calls for a model of the dashboard and driver controls and a separate model of the engine.

The dashboard and controls have been modeled using Sammi, a user interface simulation tool developed by Kinesix. Sammi runs in the Unix environment and uses an X windows interface. Using the Sammi format editor, developers may draw, design and arrange user interfaces quickly (Figure 2). After the appearance and behavior of each component have been chosen, the format may be "executed" using another portion of the Sammi package.

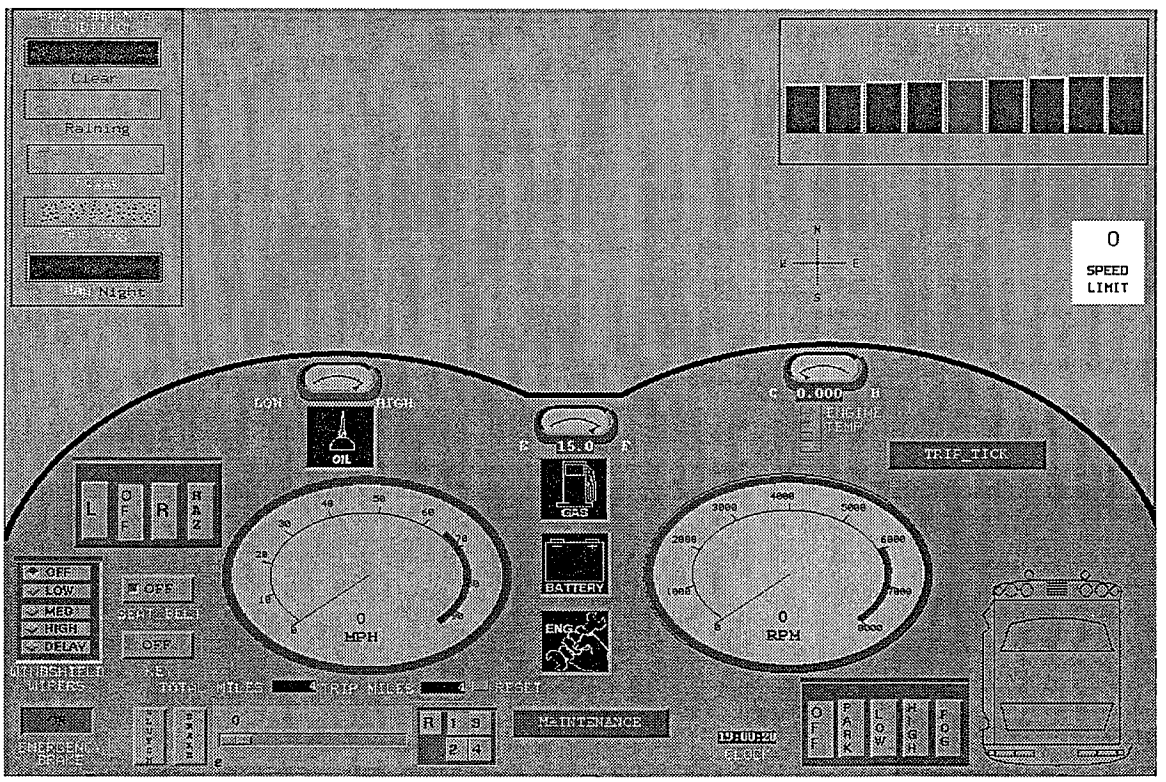

Figure 2 - Dashboard Interface Design

Sammi connects with other applications locally or on a network. As the display executes, each component may send data to or receive data from databases or those applications [Sammi UsersManual, 1993]. Kinesix has provided extensive descriptions of how to design an application for communication with a Sammi format. These descriptions include "skeleton" code for various application needs. These descriptions provided the detail required by this project for communication with the dashboard design.

The main thrust of this project has been to study two approaches for implementing the simulation component and communication link between the interface and the simulation. The engine and its components have been modeled using CLIP, a rule-based language and environment primarily used for the design of expert systems, and PERCNET, a software tool that uses petri nets to model systems. CLIPS executes in a non-procedural fashion making it ideal for representing random and concurrent events. CLIPS is designed for complete integration with the C and Ada programming languages, both of which are widely used. CLIPS also supports an object-oriented approach to programming - a paradigm that has been growing in popularity. PERCNET uses a graphical display to describe the system and then uses animation to display the execution of the model. PERCNET provides several simulation analysis options including workload profiles, time-based performance profiles, and a means of viewing all simulation data [PERCNET User Manual, 1992]. The two simulations were evaluated and compared for ease-of-use, flexibility, performance and cost-effectiveness.

The communication link was originally designed by John Noll of Perceptronics, Inc. and has been significantly modified to work with the PERCNET/Sammi environment. This original design was developed from skeleton code provided by Kinesix (Sammi) and was extended to pass all information to and from PERCNET. The communication link developed for the CLIPS/Sammi environment uses the blackboard paradigm to improve modularity, flexibility and efficiency. This form of data management stores all information in a central location (the blackboard). Processes communicate by posting and retrieving information from the blackboard. The blackboard server manages the blackboard, allowing applications to retrieve current values from the board and to request that a value be changed. The server accepts write requests from valid sources and changes values. The comparison of the two environments goes much further than comparing the two simulation designs. The design of the communication link significantly affects the performance and flexibility of the architecture. A close look at the communication link designs will show which features are most important.

PERCNET

PERCNET is a very powerful systems analysis software package designed by Perceptronics, Inc. It provides an easy-to-use, graphical interface which allows users to quickly lay out a petri net model of the system. Petri nets are similar to state-transition diagrams, but are more "formal." They allow for modeling concurrency and have been studied extensively. PERCNET actually uses "modified" petri-nets. These modifications allow each state to describe pre-conditions for state transitions, modify global variables, perform function calls and maintain a global simulation time. Most features are easy-to-learn. While some study of petri-net theory would benefit designers, much could be done with very minimal knowledge of petri-nets. One difficulty in working with PERCNET is the lack of available documentation on the Tool Command Language(TCL). All function calls, calculations, communication and ad-hoc programming are done using this widely-used language. Perceptronics provides only minimal documentation on the use of the language making it very difficult to perform anything more than the most rudimentary operations.

Several forms of system analysis are possible including workload profiles, time-based performance profiles and various system event reports. PERCNET has a blackboard feature that allows designers to monitor all (or selected) global values. The graphical interface allows a system to be described in a hierarchical fashion - with the highest levels containing very little detail. This makes system design modular.

As this project began, PERCNET was a closed package, that is, there was no provision for communicating with other applications. NASA contracted Perceptronics to modify PERCNET to allow for such a feature. The final result was a revision of PERCNET which would allow communication with another application through a socket. This application would be allowed to request that global variables be retrieved and/or modified. PERCNET would essentially open it's blackboard to one other application. The other application is the communications link or server.

SERVER

"Server" is really a misnomer for the program providing the communication link. As mentioned above, PERCNET is actually the data server for the environment. The server only provides a mechanism for passing information between PERCNET and other applications. The

server is connected to PERCNET by a socket and the server is actually on the "client" end of the connection-oriented socket.

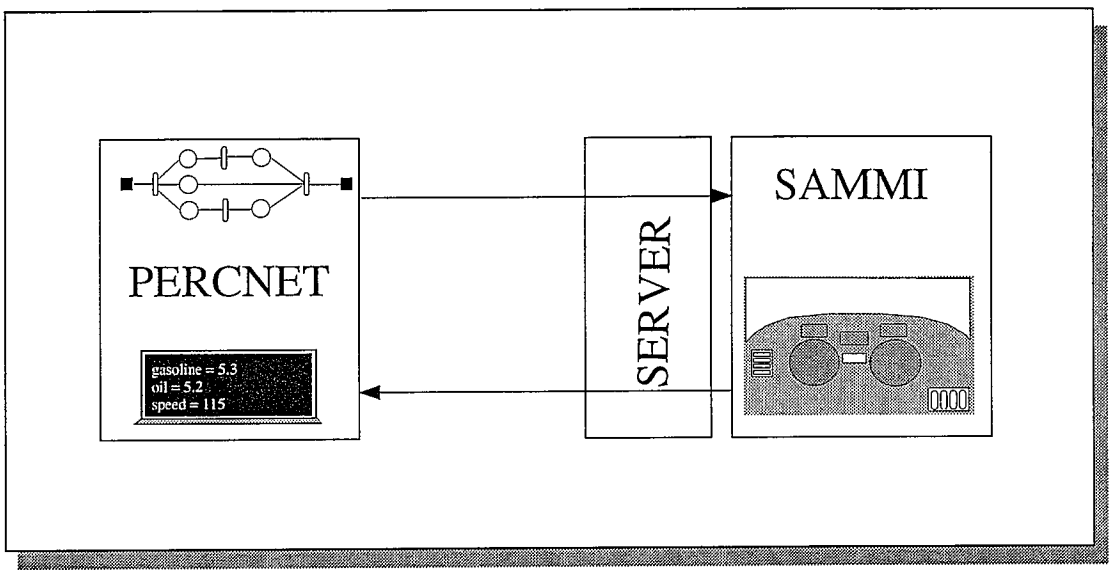

Figure 3 - PERCNET - SERVER - SAMMI

The server establishes connections with PERCNET and Sammi and then alternately receives information from each. Any data or commands received from Sammi are passed immediately to PERCNET. Commands from PERCNET for Sammi are passed immediately through, as well. Finally, the server sends Sammi copies of all variables.

A closer look at the internal workings of the server reveals that the server is actually an extension of Sammi with a socket for reading and writing to PERCNET (Figure 3). The server maintains an array of variables of its own. These variables are those required by Sammi. The server builds the array as it receives variables from Sammi (when the connection with Sammi is initialized). PERCNET may pass many values to the server that are never used by the interface and the server ensures that no attempt is made to send such values to Sammi. This is, however, the only attempt to minimize the amount of communication that passes through the communication link. Now that we have an idea of how the PERCNET-server-Sammi environment works, we should look close at how well it works.

PERFORMANCE OF PERCNET-SERVER-SAMMI

Finally we see the true motivation for the project. Performance is terrible. The original modifications made to the server were not enough to improve the performance to acceptable levels and this project will determine whether an alternate approach would improve the system. PERCNET's performance was most affected. Every system-to-network configuration ended after only a short time in a segmentation fault by PERCNET. The modifications made to PERCNET apparently caused PERCNET to consume more swap space than before. Before long, the machine on which PERCNET was running would have no swap space available and PERCNET would crash. Sustained execution would be requisite for performing any meaningful evaluation. Sammi was also affected by the configuration. Sammi execution slowed drastically when connected to PERCNET and the server. Often Sammi's displays would freeze for more than ten seconds.

Early analysis attempted to find the cause of the poor performance. Since PERCNET code was unavailable, we could only speculate about what was actually happening to cause the consistent system crashes. Performance tests were conducted over a period of several weeks and the results were submitted to Perceptronics for review. Analysis work continued to determine other causes of poor performance. It was determined that the cause of much of the problem was that PERCNET was trying to do too much. Since PERCNET is the blackboard server, as well as the simulator, PERCNET's performance would naturally be affected by the added burden. The two most glaring inefficiencies involved the overhead included with every read or write from Sammi and the amount of redundant information passed between applications.

Each time Sammi requests a value, it sends it's request to the server. The server passes the request to PERCNET. PERCNET finds the current value and passes it to the server. The server marks the variable to be sent to Sammi. Before the variable can be sent, however, Sammi is polled for a pointer to the data structure to receive the value. Sammi returns this pointer and the server inserts the value. The steps are the same for both reads and writes.

Lastly, the method provided for sending variables to Sammi was terribly inefficient. When a calculation was performed in the simulation model for a variable that was needed by Sammi, that variable was passed to Sammi whether or not it's value had changed from the previous iteration. No mechanism was provided for restricting the number of redundant values

passed across the communication link. As a result, PERCNET passed every value back to the server when only a few had actually changed.

Each of these limitations has been addressed in the design of the blackboard in the CLIPS/blackboard/Sammi combination. CLIPS is presented first, followed by the blackboard design and a description of the final performance.

ALTERNATIVE APPROACH: CLIPS

CLIPS is a rule-based language. This means that there may be a larger learning curve than there is with PERCNET's point-and-click interface. After the initial learning stages, however, CLIPS leaves a developer with an emensely powerful simulation tool. The main advantage is flexibility - if there is a feature that you want or need and you can describe to CLIPS how to perform that feature, you can have that feature. CLIPS was written in the C programming language and is completely compatible and extendible with C functions. Knowing C in advance can significantly lessen the learning curve. There was apparently some influence from LISP in the design of CLIPS, because many of the "non-C" features of CLIPS resemble LISP (The name CLIPS closely resembles a merging of "C"and "LISP").

Since CLIPS is rule-based, it is completely non-procedural. Furthermore, it allows programmers to pick the strategy by which successive rule-firings are chosen. Certain rules may be designated by different priority levels (rules with the highest priority fire before rules with lower priority). Other rule-selection strategies govern how rules with equal priority are selected.

In this project, CLIPS has been extended to include communication capabilities. CLIPS communicates with the blackboard using sockets. Two sockets have been provided for reading and writing respectively. C functions have been developed to eliminate redundant information from the messages passed to the blackboard. This code has been incorporated into a main CLIPS executable and is ready for use with other simulations.

Another improvement compiled into the CLIPS executable has been a control process that allows a user to start, stop and quit CLIPS execution at will. The control process also comes with a graphical interface providing the same features. The real intent of the graphical interface is to provide a starting point for an extended graphical interface to the CLIPS simulation development environment.

The project also demonstrates some programming techniques used in CLIPS to support the simulation. A global simulation time should be maintained and a mechanism for keeping simulation execution time has been demonstrated. Another important feature that makes use of the timer is the periodic update feature. This ensures that CLIPS execution pauses every few seconds to send and receive information from the server. One interesting bonus with this feature is that the frequency of updates is easy to modify. Another feature in the automobile simulation is an example of how to force CLIPS to execute procedurally when needed.

BLACKBOARD (SAMMI-CLIPS SERVER)

While efficiency has been the primary goal of blackboard development, modularity and readability have been given careful consideration. The program may be divided into three portions: blackboard management, Sammi routines, CLIPS routines. The Sammi and CLIPS routines are provided to communicate with the respective applications. These routines map data into a special "blackboard entry" form and pass the data to the blackboard management routines. The blackboard routines also return information to the Sammi and CLIPS routines for routing back to the applications (Figure 4).

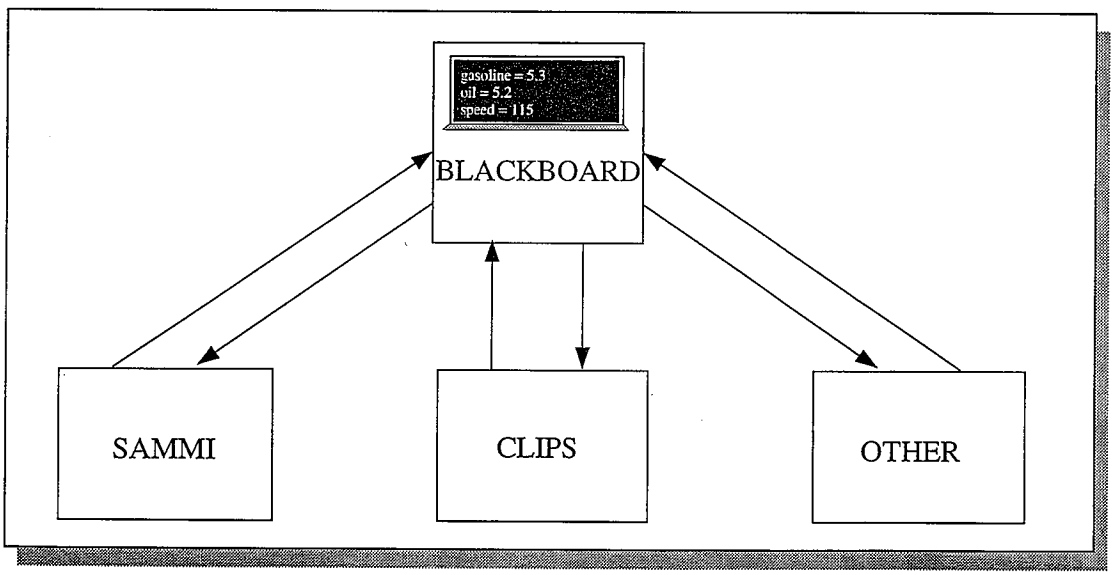

Figure 4 - CLIPS - BLACKBOARD - SAMMI

The modularity of the design will promote further improvements to the program. If changes are required to the CLIPS interface, then only the CLIPS processing routines will be affected. No effect will be made on the other portions of the blackboard. New applications may

be added by following the same format CLIPS and Sammi use. The new application should provide routines for initializing, sending and receiving information with the blackboard routines and disconnecting.

The blackboard management routines require that each application (many more applications may be supported) register itself initially. Applications are assigned application identification numbers which are used for all subsequent transactions. This application number allows the blackboard to closely monitor which variable values each application needs to see. It also provides a mechanism for installing a priority scheme for updates. Currently there is no need for such a scheme. All applications are treated equally.

PERFORMANCE OF CLIPS-BLACKBOARD-SAMMI

The overwhelming advantage of the CLIPS and blackboard combination is the flexibility and potential they provide. Features outlined in Appendix A will show how small modifications can be made that will affect performance for different effects. Also outlined in Appendix A is how the programs may be expanded to include other features or other strategies.

The ability to tune the performance has allowed the simulation architecture to be tailored to specific running conditions (e.g. machine limitations, network traffic and complexity of the interface being simulated). Several parameters (described in Appendix A) may be modified to alter performance. Tuning tests have apparently improved performance drastically. More detailed performance testing will be needed to verify the results.

The blackboard treats all applications equally. Each application sends updates and requests to the blackboard via the blackboard management routines. The blackboard responds to each application with updated information on variables needed by the application. Compare this to the reading and writing procedures provided with PERCNET and you can see why performance appears to have improved so dramatically.

SUMMARY

PERCNET's graphical interface is much more appealing to users. It is very intuitive to use and makes design very easy. For someone to use CLIPS, they must pour through manuals until they understand the language well enough to proceed. Thereafter, working with CLIPS is exactly like working with other languages. Development will be hardly intuitive.

When faced with system performance problems and unsupported features, Perceptronics answers and solves problems as they are able. With CLIPS, developers are able to perform fine-tuning themselves. Additional features may often be quickly developed.

NASA pays for the license to use PERCNET. CLIPS, however, was developed at Johnson Space Center in Houston and is owned by NASA. Considerable savings may further encourage NASA to use an internal package rather than a commercial package.

CONCLUSIONS

PERCNET's ease-of-use is currently it's only advantage over CLIPS. CLIPS overcomes this with power and flexibility. Modification and maintenance of CLIPS and the blackboard will require someone with experience in C and unix network programming. In my designs I have made every effort to make this job easier. I have also provided thorough documentation throughout the code and in Appendix A to explain my design choices.

While the server could have been modified to perform the same function as the blackboard, it is so closely tied to PERCNET that a complete re-write was necessary. The result has been a modular, flexible design providing capabilities far beyond those provided by the server.

The goal of the architecture has been to simulate user interfaces so that they may be evaluated and designed quickly. The most important portion of the simulation architecture is the evaluation component. Performance is the deciding factor. Prior to the conception of this project, the simulation architecture appeared destined to fall short of it's high goals. Now, with a realistic interface, more extensive evaluation is possible.

References

Bainbridge, Lisanne (1990). "Extrapolating From One Task to Another," Simulation and the User Interface, New York, New York: Taylor and Francis Ltd.

Bulgren, William G (1982). Discrete System Simulation, Englewood Cliffs, New Jersey: Prentice-Hall, Inc.

Diaper, Dan (1990). "Simulation: A Stepping-Stone Between Requirements and Design," Simulation and the User Interface, New York, New York:Taylor and Francis Ltd.

Downton, Andy (1991). "Engineering the Human-Computer Interface: An Overview," Engineering the Human-Computer Interface, New, York, New York: McGraw-Hill Book Company.

Downton, Andy and Leedham, Graham (1991). "Human Aspects of Human-Computer Interaction," Engineering the Human-Computer Interface, New York,New York: McGraw-Hill Book Company.

Floyd, Chritiane (1984). "A Systematic Look at Prototyping," Approaches to Prototyping, New York, New York: Springer-Verlag.

Hopkin, V. David (1990). "Operational Evaluation," Simulation and the User Interface, New York, New York: Taylor and Francis Ltd.

Jeffries, Robin, Miller, James R., Wharton, Cathleen and Uyeda, Kathy (1991). "User Interface Evaluation in the Real World: A Comparison of Four Techniques," Human Factors in Computing: Reaching Through Technology - CHI '91 Conference Proceedings, New York, New York: Addison-Wesley Publishing Co. Ltd.

Kreutzer, Wolfgang (1986). System Simulation: Programming Styles and Languages, Reading, Massachusetts: Addison-Wesley Publishing Co.Ltd.

Life, M. Andrew (1990). "Simulation and the User Interface: An Introduction," Simulation and the User Interface, New York, New York: Taylor and Francis Ltd.

McHaney, Roger (1991). Computer Simulation: A Practical Perspective, San Diego, California: Academic Press, Inc.

Moore, Loretta A. (1992). A Process for Prototyping Onboard Payload Displays for Space Station Freedom. In M. Freeman, R. Chappell, F. Six, & G. Karr (Eds.), Research Reports - 1992 NASA/ASEE Summer Faculty Fellowship Program (Report No. NASA-CR-184505, pp. XXXVI.1 - XXXVI.4). MSFC, Alabama: NASA,George C. Marshall Space Flight Center.

Moore, Loretta A. (1993a). Assessment of a Human Computer Interface Prototyping Environment (Contract No. NAS8-39131). MSFC, Alabama: NASA, George C. Marshall Space Flight Center.

Moore, Loretta A. (1993b). Integration and Evaluation of a Simulator Designed to be Used Within a Dynamic Prototyping Environment. In G. Karr,M. Freeman, R. Chappell, and F. Six (Eds.), Research Reports - 1993 NASA/ASEE Summer Faculty Fellowship Program.

MSFC, Alabama: NASA, George C. Marshall Space Flight Center.

Percnet User's Manual (1993). Woodland Hills, California: Perceptronics, Inc.

Riddle, William E. (1984) "Advancing the State of the Art in Software System Prototyping," Approaches to Prototyping, New York, New York: Springer-Verlag.

SAMMI User's Guide (1992). Houston, Texas: Kinesix Corporation.

Sathre, B.A. (1988). "A Study of the Skills, Aptitudes, and Knowledge Bases Required for a Simulation Software Engineer," Tools for the Simulation Profession 1988, San Diego, California: Simulation Councils Inc.

Taylor, Andrew and Browne, Julie (1990). "Indivisibility, Inseparability, and Indiscernibility: Some Problems with Simulations for Complex Domains," Simlation and the User Interface, New York, New York: Francis and Taylor, Ltd.

APPENDIX

APPENDIX A - Maintaining/Modifying CLIPS/BLACKBOARD

This section provides a more in-depth discussion of the design of both CLIPS and the blackboard. Every attempt has been made to make each portion of the design well-documented, well-formatted and clearly designed. Anyone willing to take a small amount of time will be able to familiarize themselves with the design enough to make any modifications necessary. Several design choices are discussed.

CLIPS has been a tremendous surprise to work with. Learning a new programming language has little appeal for many programmers. The surprise came when it was found how quickly one could learn to do very useful things with the language. Writing the rules for the simulation was actually the easiest part of the whole project. As proficiency with the language developed, more advanced features provided tremendous possibilities. The manuals present the language in a very easy to read format, contained extensive reference sections and sample code. Furthermore, the manuals outline how CLIPS may be easily extended to include C (and other) functions written by programmers

A basic proficiency with CLIPS may be gained quickly. A quick overview of the modifications to the CLIPS executable will explain how the system executes. Next the constructs are described which have provided to control the flow of the simulation. Using these guidelines, will enable rapid development of any system simulation.

The CLIPS main routine was modified following the guidelines provided in the Advanced Programming Guide in the chapter entitled *Embedding CLIPS*. The main routine initializes clips, loads the clips file to be executed and then begins execution. When execution has stopped, control is returned to main. The strategy chosen is to construct the rules such that they force CLIPS to pause periodically (i.e. no rules may fire). When this happens, control returns to the main routine. From the main routine, communication with the blackboard takes place (sending and receiving variable updates) and a rule is asserted re-activating CLIPS.

Writing CLIPS programs to take advantage of this strategy requires the incorporation of several techniques. These techniques include rules, variables and functions which may be used in subsequent simulation designs. The first choice involves determining which values will be passed to or received from the blackboard. All global variables (defined using the "defglobal"

command) are passed to the blackboard. No other values are passed. Facts and local variables may be used to store value which do not need to be passed to the blackboard. It will be show later how communication has been further streamlined for efficiency. The most important rule is the clock rule:

```
(defrule CLOCK
        (declare (salience -10))          ; decrease priority
        ?tick <- (clock_tick)             ; helps rules with all "test" preconditions fire
        ?update <- (updated TRUE)         ; keeps clock active - re-asserted by main routine
        ?cont <- (continue TRUE)          ; restarts main loop
=>

        ; start_time is the sytem time when the reset command is given
        ; the simulation time is calculated with respect to start_time

        (bind ?new_time (- (integer (time)) ?*start_time*))        ; calculate current sim time


        (if (<> ?*time* ?new_time)                                 ; don't print time again if
            then                                                   ; it hasn't changed
                (bind ?*time* ?new_time)
                (printout t "TIME = " ?new_time crlf)

        )

        ; disable clock via "updated" rule once every "update_interval" seconds
        (if (>= ?*time* ?*next_update*)
                then
                  (bind ?*next_update* ( + ?*time* ?*update_interval*))
                  (retract ?update)
                  (assert (updated FALSE))
                else
                  (retract ?update)
                  (assert (updated TRUE))
        )

        (retract ?tick)
        (assert (clock_tick))
        (retract ?cont)
        (assert (continue TRUE))
)
```

The clock rule stays ready at all times, but because the salience (i.e. priority) of the rule is kept low, it will not block the firing of other rules. Without this declaration, the clock rule could fire continuously with no other rules firing (even though they are ready to fire). When execution begins, the current system time is retrieved and stored. The current simulation time is always known by retrieving the system time and comparing it to the starting time. Notice that the new simulation time is temporarily stored in a variable called "new_time". This value is compared to

the last calculated time. If the two values are the same, then the clock rule has fired more than once within one second. In that case, the time is not printed and facts are reset to allow the clock rule to fire again.

Another interesting mechanism is provided by the "clock_tick" fact. Several rules in the simulation might have preconditions consisting only of tests and no facts. These rules may not fire when all tests evaluate to true. Rules are easier to use with at least one fact present in their preconditions. The clock_tick fact is used in the preconditions for these rules to allow them to become ready for firing:

```
(defrule TURN_KEY
        ?tick <- (clock_tick)
        (test (= 0 (str-compare ?*key* ON)))
        (test (= 0 (str-compare ?*seatbelt* ON)))
        (test (= ?*gear* 0))
        (test (> ?*battery* 10.0))
        (test (= ?*state* ?*READY*))
=>

        (bind ?*state* ?*STARTER*)
        retract ?tick)
        (assert (clock_tick))

        (printout t "ACTIVATE STARTER (" ?*time* ")" crlf)
        (tick_tocks 2)
        (assert (updated TRUE))
)
```

Without the clock_tick fact, this rule may never fire. Notice that the turn_key rule simply reasserts the clock_tick fact so that it can be used by other rules.

The clock rule also provides a mechanism for performing periodic updates to the server. As explained earlier, CLIPS must ensure that all rules become inactive periodically. This is accomplished within the clock rule. Notice that the fact (updated TRUE) is continually re-asserted by the clock. This fact also triggers the next firing of the clock rule. The clock continually reasserts this fact to keep itself active. The clock rule uses the global variable update_interval to determine when the next update should occur. The value for update_interval has been set to five. Every five seconds, the clock rule retracts the (updated TRUE) fact, disabling itself and all other facts (remember that the clock_tick fact is asserted by the CLOCK as well). When this occurs, control returns to the main routine where updates are performed. Within the main routine, a CLIPS function call - AssertString ("updated TRUE") - is made which

reactivates the clock rule. Next the main re-starts CLIP and execution of the model continues until the next update.

The last feature provided by the clock rule is a technique which will ensure that during the main execution loop (often an infinite loop), execution pauses to fire the clock. Remember that the clock salience has been set to a low value, so as long as other rules remain active (as they would in an infinite loop), the clock will not fire. If the clock does not fire, no updates are performed. The guarantee that the clock will fire has been provided with the (continue TRUE) fact. This fact may be used by any rule as follows:

```
(defrule RUNNING
        ?tick <- (clock_tick)
        (test (= ?*state* ?*RUNNING*))
        ?cont <- (continue ?TRUE)
=>
        (printout t "RUNNING..." crlf)

        (bind ?*state* ?*RUN*)
        (assert (alternator))          ; activates alternator rule
        (assert (fuel_pump))           ; activates fuel_pump rule
        (assert (oil_pump))            ; activates oil_pump rule
        (assert (cool_engine))         ; activates cooling rule
        (assert (continue))            ; activates last rule in main loop
        (assert (stop_running))        ; activates rule to decide if car should stop or stall
        (assert (calculate))           ; activates rule for calculating rpm, speed, etc.
        (retract ?cont)

        (tick_tocks 1)
        (retract ?tick)
        (assert (clock_tick))
)
```

The running rule is activated by the (continue TRUE) fact. When it fires, it activates several other rules, but does not reassert the (continue TRUE) fact. After the other rules fire, execution halts except for the clock rule. This allows for an update of the system time and a possible update.

Another time feature provided is contained in the tick_tocks function. Often a programmer would like to force a rule to consume clock time. This would be necessary if a rule described an event lasting several seconds. A call to the tick_tocks function forces execution to enter a side loop where the required time elapses before execution continues. Programmers should be carful using times much longer than the update_interval. Update performance (and overall system performance) may be affected. Another technique would be required for longer times.

```
(deffunction tick_tocks (?n)

        ; This function does the same thing as the CLOCK rule, but is called from
        ; within a rule and returns to that rule when completed

        (bind ?*target_time* (+ ?n ?*time*))

        (while (< ?*time* ?*target_time*)
                (bind ?new_time (- (integer (time)) ?*start_time*))
                (if (<> ?*time* ?new_time)
                        then
                                (printout t "TIME = " ?new_time crlf)
                                (bind ?*time* ?new_time)
                )
        )
)
```

For ambitious programmers, who would like to add or modify features to the CLIPS executable program, the modifications are described. The control interface is the most useful addition to the program. Users are allowed to stop and start CLIP Sexecution by typing commands. A quit option is also provided. While the initial interface is text-based, a simple graphical interface using Motif widgets has also been provided. These provide the same functionality as the text-based interface, but are intended to give a starting point for future extensions of the program. The control interface is implemented using two separate processes (created with a fork command). These processes communicate via a pipe. The child waits for the "play" command to come through the pipe and thereafter periodically checks the pipe for the "stop" or "quit" command. When the parent (which accepts user input) detects a stop or quit command, it sends the command to the child who resets CLIPS or gracefully terminates for "stop" or "quit" respectively.

Another feature that was implemented in the CLIPS executable program was a mechanism for testing variable values for change before transmitting them. The main routine maintains an array of variable/value information and checks the previous value of a variable against the current value. Searching through the array has been avoided entirely by storing the array index with the variable. When the main routine retrieves global variables, CLIPS always returns the list of variables in the same order (the order listed in the CLIPS program file). When the array is initialized, the index for each variable is recorded. Later when all global variables are received and checked one at a time, a count is kept to quickly match the current global variable with it's corresponding position in the array.

Before variables are passed from CLIPS to the blackboard, they are stored one-by-one in a data structure called clips_msg. This structure records several pieces of information including variable name, type and value. The value field is a union type to support many data types. The index mentioned above is stored as well. This structure allows CLIPS to neatly pack a buffer of variables for transmission to the blackboard. Routines in the blackboard program expect to receive a buffer filled with clips_msg structures. These are easily extracted upon reception.

Communication with the blackboard is performed using two TCP sockets - one for sending and one for receiving. The buffer size for messages has been set to 2048, but may be easily changed in the blackboard.h file, if needed to improve performance. Communication between CLIPS and the blackboard proceeds as follows: CLIPS sends some number of messages and then receives some number of messages. Several approaches were attempted using a single socket for all communication, but two sockets were chosen to ease the synchronization requirements between the two applications. The blackboard's receiving socket has been set to timeout if no information is received within a specified length of time (this time limit is tuneable in the file blackboard.h). This way the server is not left waiting for CLIPS to send information. If CLIPS is not ready, the blackboard performs processing for other applications and tries again later. This is a feature not provided by the PERCNET configuration. If the server is forced to wait for CLIPS, interface events sent to the blackboard are also forced to wait. The result is reduced performance where performance is most noticeable.

We know what the other processing is in the case of CLIPS, but have not looked in depth at the blackboard design. A careful look at the main routine will demonstrate the modularity provided:

```
main(argc, argv)
        int argc;
        char **argv;
        {
        int sammi_connected = 0, clips_connected = 0;
        int sammi_id, clips_id;
        int done = 0;

        sammi_connected = connect_sammi (argc, argv);     /* establish Sammi connection */
        if (sammi_connected)
                {
                printf ("SAMMI CONNECTED\n");
                sammi_id = register_app ();
                }

        clips_connected = connect_clips (argc, argv);       /* establish CLIPS connection */
```

```
if (clips_connected)
        {
        printf ("CLIPS CONNECTED\n");
        clips_id = register_app ();
        }

/* Enter the event loop. */
while (!done)
        {
        /*  PROCESS EVENTS FROM ALL ATTACHED APPS */
        if (clips_connected)
                clips_connected = process_clips (clips_id);
        else
                /* CLIPS will control when blackboard server terminates */
                done = 1;

        if (sammi_connected)
                process_sammi(sammi_id);

        /*  UPDATE ALL APPS AS NEEDED */
        if (clips_connected)
                clips_connected = update_clips(clips_id);

        if (sammi_connected)
                refresh_dfds(sammi_id);
        }

sammi_connected = disconnect_sammi ();
} /* end main */
```

Notice that the format is exactly the same for both CLIPS and Sammi. This same format can be followed for adding additional applications. First, establish a connection and upon successfully connecting, register with the blackboard server (allows assignment of application id number). Next, insert function calls in the main event loop to pass information from the application to the blackboard and to pass information from the blackboard to the application. The order of these may be rearranged to affect system performance.

The functions that are called have been written for each specific application. Each function is application specific. For example, the process_clips function receives a message from CLIPS (on the receiving socket) and breaks this message into a number of clips_msg structures. Each clips_msg structure contains information about CLIPS global variables that have changed values (remember that CLIPS only sends variables whose values have changed). For each variable, a blackboard_entry structure is created and the variable information is inserted into the structure. Finally, the function passes the complete blackboard_entry structure to the blackboard's update_entry routine where the server modifies the appropriate blackboard

entry. The process_sammi function works similarly except that communication between Sammi and the process_sammi function occurs via the RPC mechanism.

The most important feature for flexibility is that the interface between any application and the blackboard uses a common data structure - blackboard entry. This data structure maintains all pertinent information for a variable and value. The variable name is the unique identifier of all blackboard entries (as with Unix - variable names are case-sensitive). The blackboard entry structure contains mostly general fields for the variable name, type, size and value, but also contains some application specific fields:

```
typedef struct _blackboard_entry
        {
        char        key[MAX_KEY_LENGTH];
        union    {
                 void *v;
                 short int i;
                 long int l;
                 float f;
                 double d;
                 char *c;
                 } value;
        int         type;
        int            size;
        int         dirty;
        int         last_update; /* application id of last appl to modify */
        int    console;
        desc_list   *desc;
        struct _blackboard_entry *next;
        } blackboard_entry;

typedef struct _desc_list
      {
      int       desc;
      struct _desc_list *next;
      } desc_list;
```

The console and desc_list components have been added to support Sammi. Sammi needed to have more information stored with each value to ensure that all interface components requiring a value received that value. In the case of the automobile, the value of the weather variable may appear in several locations on the screen. Each of these locations must receive a separate value. Using the desc_list, one blackboard_entry can maintain the single value and the multiple destinations.

It is important to see that this modification of the blackboard structure required only minor modifications to the blackboard server routines. The first modification occurs in init_entry.

Init_entry is a function used by every application to acquire a "blank" blackboard entry. Communication with the blackboard entry involves acquiring a blank entry and modifying only the fields known by an application. When the blackboard server receives a blackboard entry, it compares each field in that entry to the default value contained in the blank entry. If a field contains a value other than the default or if the field has changed from its previous value, that field is updated on the blackboard. If the field contains the default value or is not changed, it is left unmodified. Init_entry allows other applications to ignore the Sammi specific fields of the blackboard structure.

Before examining each blackboard routine, the application identification number should be explained further. Remember that after an application connects successfully, it makes a call to the register_app function and is assigned an application number. This number is used in two ways: it allows the blackboard to maintain a record of which variables need to be sent to which applications (via a dirty bit for the application) and it allows for addition of a priority scheme for updating values (whereby an application's request to update a value already modified by another application of higher priority is denied). The priority scheme would be installed in the update_entry routine.

The blackboard server uses a bit mask to record a "dirty-bit" for each application. The bit mask is actually an integer (32 bits) whose bit positions correspond to application id's - a value that is dirty for application one would have a dirty value of 1, while one that is dirty for applications four and two would have a dirty value of ten (001010 in binary). This scheme allows the blackboard to decide which values to send to each application. When an application updates a value, the dirty bit is set for every application registered except the one requesting the update.

A brief look at each blackboard routine will demonstrate the capabilities of the blackboard server.

blackboard_entry init_entry ()

This function returns a blackboard_entry whose fields have been initialized to default values. The update_entry and override_entry compare fields it receives against these default values to determine whether the field should be modified on the blackboard.

int register_app ()

The next available application number is assigned. Currently, a bit mask is maintained indicating which application numbers have been assigned. If one is released, it is reassigned to the next requesting application.

int remove_app (int app_id)

The application number for the requesting application is freed.

int update_entry (blackboard_entry item, int app_id)

An application uses this to request that a variable be changed on the blackboard. This routine will add the entry if it does not already exist. The last_update field for the variable is set to the requesting application number. This provides a way to know which application was the last to modify a value (for use in a priority scheme). Zero is returned on failure or if the update has been denied.

blackboard_entry *update_app (app_id)

This function returns a linked list of blackboard entries. The list is every variable whose dirty bit has been set for the application number. The dirty bit is cleared for that application.

void delete_entry (blackboard_entry item, int app_id)

This function is not used by CLIPS or Sammi, but would allow an application to remove a variable from the blackboard.

void lookup_entry (blackboard_entry *key, int app_id)

When an application only needs to request that a value be sent to it, it can use this routine to have the corresponding dirty bit set for the variable.

int override_entry (blackboard_entry item, int app_id)

If the blackboard denies an update for an application, the application may need to override that denial. Care should be exercised with this function - it has been provided as a work-around in special cases.

int get_entry_type (blackboard_entry item)

Sometimes an application may know the variable name, but not have access to the officially recorded type of the value. This convenience function is provides to return the current type of the variable.

Design intentions were to define a strict form to be used in the blackboard interaction. A blackboard protocol, consisiting of the blackboard_entry data structure and application number is used to provide a consistent interface. The modular form has been used for readability and ease of modification. CLIPS functions have been provided as a jump-start to designing simulations using the language. Additional blackboard functions may extend the design, but care should be taken to adhere to a consistent protocol for all blackboard functions. The source code is available and may be modified as needed. Questions, comments, and modifications should be forwarded to sprice@eng.auburn.edu or moore@eng.auburn.edu.