SYSTEM SUPPORT FOR THE STANFORD HAND-EYE SYSTEM *

Jerome A. Feldman

Stanford University
Stanford, California

Robert F. Sproull

National Institutes of Health
Bethesda, Maryland

Abstract

The Stanford hand-eye system is implemented as several separate tasks, each executing under a timesharing executive. Development of a programming language (SAIL) and augmentation of the timesharing system were required to provide the necessary data sharing and control flow among the tasks. The SAIL language provides facilities for "associative processing," and is extended to serve the data sharing and communication needs of the hand-eye system. Several user facilities are designed to aid running and debugging the system.

Keywords

associative processing, programming languages, symbolic processing, timesharing, segmentation, interprocess communication, debugging.

*This work was supported by the Advanced Research Projects Agency (SD-183).

1. Introduction

The second Stanford hand-eye system is the result of a design based on the previous solutions to hand-eye subproblems, but with the various problems considered more broadly. The systems and language services which the new system utilizes are based on, among other things, the requirement that several programs or cooperating tasks be able to operate concurrently. Further, each subproblem task is to be under the supervision of a strategy monitor: a strategy for solving a preception or manipulation problem is generated and subsequently executed. During execution, the strategy may call for various subgoals to be pursued with various priorities, for subgoal progress reports, and possibly for alteration of the strategy itself.

The tasks, or "modules," which constitute the system are largely independent but cooperate with the strategy program to solve the perception and manipulation problems put to the system. Examples of tasks are: (1) the TV camera calibration routines, (2) the camera positioning program, (3) the color discriminator, (4) the edge follower, (5) the program to optimize feature discrimination by the TV, (6) the body recognizer, (7) the program to compute an arm trajectory, (8) the task which actually moves the arm over a trajectory, and (9) the strategy task itself. Each of these modules is a complicated computer program, but requires only a fairly simple interaction with the other system modules. The resource and execution requirements vary among the tasks: Some are executed periodically throughout a run (e.g. strategy task); some are used only during initialization (e.g. camera calibration); some must deliver real-time service to hardware devices (e.g. arm mover).

Since the tasks are reasonably self-contained, each was designed by a separate programmer. Each module was to be written and debugged in isolation and then merged into the system. This merging effort is minimized by requiring all programmers to observe the programming conventions of a high level language, SAIL, (see section 2). SAIL was designed with the coding of the hand-eye system in mind, and includes several features to ease that coding.

Even though these tasks are merged into a full system, they remain quite independent. The strategy program monitors and controls each task separately, and may allow concurrent execution of several tasks. For example, the edge follower may be working on a cube at the same time the arm is being positioned. Major changes in the hand-eye system may require new tasks to be added to the system repertoire, or old ones to be deleted. The requirements of flexibility, independence, and concurrent operation are most easily met by executing each task as a separate job in the PDP-10 timesharing system (1). The status of each task as a timesharing job means that existing timesharing facilities (e.g. file system, teletype communication, storage management) are available to each task. The timesharing job scheduler, acting on information supplied by the strategy program, can be used to allocate hardware resources to each task. The alternative to using the timesharing system was to build a scheduler and swapper into a monitor for the hand-eye system which could allocate resources according to priorities set by the strategy task. Since this method would duplicate many of the functions of the timesharing system, we preferred changing the existing system to starting afresh.

An easy means of sharing certain system data among all tasks and of communicating between tasks is necessary, but is not provided by the timesharing system. Data sharing is facilitated by the PIP-10 memory address mapping feature, which allows a collection of jobs to have a common portion of their address spaces reference a common segment of memory. Such a shareable segment of memory is called a "second segment" (as opposed to a "first segment," which is private for each timesharing job) and is generally used to store a pure procedure which is common to several jobs. The hand-eye system, however, uses a second segment to store common data as well as common procedures. The SAIL language provides method of referencing data in this common area (see section 3). Also, a "message procedure" facility was built into SAIL to handle intertask communications (see section4) which could not be accomplished easily with the data sharing mechanisms.

The large number and complexity of tasks necessitate specific aids to debugging the system. Because the tasks are fairly independent, each can be checked out before being merged into the system. In many cases, the system environment for a task can be simulated by another task for purposes of debugging. Debugging the full system is difficult since as many as a dozen tasks are active concurrently; specific debugging and operating aids were constructed to simplify this process. For example, the user may control the entire system from a single display console (see section 5) and may request traces of task-to-task and user-to-task communications (see sections 4 and 5).

2. SAIL

SAIL (2), the language used for most of the programming in the hand-eye system, is a dialect of ALGOL 60 augmented by a complete character-string facility, by file-handling operations tailored to the PEP-10 operating system, and by LEAP facilities, which are a set of associative data storage conventions and operations. A one-pass compiler and a runtime executive were built for this language. The compiler produces programs to be loaded by a linking loader, which can include other SAIL, FORTRAN, or machine-language procedures. We adopted the philosophy that the language should change to reflect the collective needs of the hand-eye system programmers. Thus we insisted that the methods adopted for data-sharing and intertask communication be supported directly by the SAIL language. The compiler was constructed so that such modifications to the language might be made easily.

3. Data Sharing

Data that is needed by more than one module of the hand-eye system is declared as GLOBAL in each module so that it may be stored in the second segment portion of the hand-eye system's PIP-10 address space.

The global data facilities are used for global system variables (e.g. a boolean debug flag to instruct tasks to record debugging information), and particularly a "world model" containing information about the position and identity of all objects recognized by the system, the positions of the arm and camera, etc. This world model is the system's best estimate of the state of the problem solution, and is used to influence strategy decisions. Each module may, with appropriate precautions, read or update this model.

Shared data is referred to by name in SAIL just as are any other SAIL variables. A dictionary of globally declared names and their data types is used by the SAIL compiler during compilation of each module. Declaring a name as GLOBAL allows SAIL to assign to that global variable an address in the second segment. Such declarations are written as:

```
GLOBAL INTEGER DEBUG_FLAC;
GLOBAL REAL ARRAY CAMERAJTRANSFORM [1:6,1:6];
```

Since the global data area is shared among all tasks, an interlock mechanism is required to prevent concurrent attempts to allocate storage in the second segment at runtime or to change some sensitive SAIL data structure (e.g. SETs). The SAIL runtime routines provide this interlock with a Dijkstra P-V operation (5).

Interlocking the runtime routines is not sufficient to prevent several separate tasks from making conflicting modifications in the global data. Updates should be performed "all at once" since another task may be reading the same global data concurrently, and may not be able to tolerate an inconsistent global model. This update-access interlocking is essentially a scheduling function and is administered by the strategy program which allows access to a portion of the data by only one reader or writer at a time. This access privilege is not enforced by hardware protection, but rather by requesting permission from the strategy task, making the modifications, and announcing to the strategist that the modifications are complete. In fact, the strategist may choose to allow modifications only after some validation tests of the new data have been performed (the strategy program might activate two concurrent, but different, attempts to compute a result and compare results, have a validation process check the data, etc). The communication with the strategy task to establish permission to write in the global model is accomplished with message procedures (see section 4). These modification protocols are not built into the SAIL language, but are coded explicitly in each module which may make sensitive modifications.

Much of the shared data may not require, and hence is not subject to, this close supervision. Some data is declared GLOBAL simply so that two sympathetic tasks might reference it. For instance, since both the color discriminator module and the edge following module use information about the current sensitivity of the TV camera, these parameters are stored as global data.

3.1 The Global LEAP Model

LEAP facilities are used to process much of the symbolic data in the hand-eye system. A full description of LEAP is presented elsewhere (4), and a brief review will suffice here. LEAP is a collection of SAIL syntax and runtime routines for manipulating ITEMs and associatioas of ITEMS. An item is merely a number in the range 0-4096 which has a symbolic use (c.f. ATOMs in LISP). Items may be associated together as triples (or associations) stored in the "associative store." In fact a triple may itself be considered as an item and may occur as a component in a further association. Triples are added to the associative store with the MAKE statement. For example:

```
MAKE father g>john E joe ;
MAKE endpoint ® x E point1 ;
```

The three items in a triple are referred to as the attribute, the object, and the value respectively. This suggests that a triple be reas as: "endpoint of x is pointl." Associations are removed from the associative store with the ERASE statement, as:

```
ERASE iteml ® item2 = item3;
```

and the associative store is seardied with the FOREACH statement:

```
FOREACH x SUCH 1HAT endpoint ® x E pointl do
<statement>;
```

The <statement> is executed once for each value of the itemvar x which occurs in the associative store in an association endpointflxE pointl.

Each item may have a IftTIJM which is an arithmetic value or array of values associated with the item. Thus the DATUM of an item which is intended to represent the endpoint of a line in three dimensions might be a 3-element array containing values for the x, y and z coordinates of the point. SAIL accesses the DATUMs of items in only 3 PDP-10 instructions, which makes the combined use of arithmetic and symbolic processing very attractive.

The associative store of item triples and the DATUM store of algebraic values associated with items together constitute a "LEAP model." All of the information about positions of bodies in the hand-eye system is represented in such a model. Certain kinds of processing, such as pattern matching, arm collision avoidance, and limited problem-solving make use of this model.

A "global LEAP model" is resident in the shared segment and can be accessed by all tasks. It consists of an associative store and a DATUM store, just as any other LEAP model, but is completely independent of the local LEAP models. Global item numbers are assigned by SAIL in the range 3072-4096, while locals are in the range 0-3071. The assignment of numbers is made either by the compiler as a result of a declaration:

```
GLOBAL ITEM a;
```

or by a runtime function which assigns unique global item numbers:

```
x ← GLOBAL NEW;
```

Local models may contain references to global items, but the global model may not contain references to any task's local items since another task will not be able to interpret such an item number. Global items in the local model are used extensively (e.g. the body recognizer builds in its local LEAP model a hypothetical "body" making use of global items such as face, endpoint, line, etc.). A task may make a local copy of a portion of the global associative store if these associations are to be changed in ways which should be hidden from other tasks.

Of course, the operations MAKE, ERASE, PDREAOI, and DATUM may refer to either the local or global models, and the user is required to remove this ambiguity by specifying as GLOBAL those operations on the global model, as:

```
x ← GLOBAL DATUM ( cube );
GLOBAL MAKE face ⊗ cube = y ;
```

The entire shared data area, consisting of the LEAP global model, global arithmetic variables and shared runtime routines, represents much of the environment in which a task program is executed. Thus the current environment may be extracted simply by copying the second segment area. The copy can then be saved for later study or can be used to restart the system from the point where the copy was made. For example, a rather cumbersome global model initialization procedure was avoided by using a copy of a second segment which has already been initialized.

4. Message Procedures

Intertask communication is accomplished by invoking "message procedures." In spirit, task A may send a message to task B requesting that one of B's procedures be executed (hence the name "message procedures"). For example, task A requests evaluation of the procedure find, located in task B:

```
BEGIN "A"
INTEGER g,i;
```

```
FORWARD MESSAGE PROCEDURE find (INTEGER a,b);

i ← ISSUE ('7,"A","B", MESSAGE find (10,g) );

END "A";
```

In the ISSUE call, task A supplies the name of the task to receive the message ("B") and the procedure call, complete with actual parameter values. Task B is coded as follows:

```
BEGIN "B"

INTEGER mess,i;

MESSAGE PROCEDURE find (INTEGER x,y) ;

BEGIN

 comment ... code for the procedure ...;

END;

mess ← GET_ENTRY ('120,"","B","");

 i ← QUEUE ('600,mess) ;

END "B";
```

The GET_ENTRY routine examines a queue of pending messages for one destined for task "B". The QUEUE call invokes a routine which actually evaluates the procedure find (named in the message) with the actual parameters supplied by task A. Notice that B must explicitly call the message processor runtime routines to scan the queue for messages destined for B. Then if B is disposed to process a particular request found in the queue, it may direct the message processor to apply the named procedure of B to the parameters in the message record. We chose not to automatically interrupt B and execute the message procedure since B may be indisposed to interruption (some data structures or hardware devices may be in sensitive conditions and cannot tolerate immediate execution of the message procedure).

Almost all forms of SAIL data may be passed as actual parameters in message procedure calls (i.e. with the ISSUE function): STRINGS, INTEGERS, REALS, ARRAYS, SETs, ITEMs, and ITEMVARs. If the variable is in the private address space of the caller, a copy will be made in the shared space so that the called task may reference it. No data is ever copied back into the caller's private space after evaluation of the message (i.e. no call by reference for private variables). If data is to be returned from the message procedure to its caller, the called task may leave information in the global data area (i.e. pass global variables by reference) or may ISSUE a return message to the caller which contains the appropriate data.

Associated with the message data is a mechanism to control execution of the caller and called tasks. A task is said to be "active" (being executed or able to be executed) or "blocked" (execution suspended). A task such as the camera servoing module may request to be blocked waiting for messages. When a message procedure call is issued to a message procedure in a blocked task, the message processor instructs the timesharing system to resume execution of that task. The activated task may scan the queue, evaluate the procedure and then request to be blocked while awaiting more messages. Similarly, the task which issues the message may request to be blocked until the called task has completed execution of the message procedure, at which point an ACKNOWLEDGEment is produced by the called task, and the timesharing system resumes execution of the calling task.

The details of the implementation of message procedures allow considerably more flexibility than described in the preceding discussion. Both the sender and receiver must declare the names and parameter types for any message calls to be passed between them. A typical declaration is:

```
FORWARD MESSAGE PROCEDURE find (INTEGER x; REAL y) ;
```

FORWARD specifies that the text of the procedure body does not accompany this declaration. The module which will process messages requesting evaluation of "find" must contain somewhere the text of "find."

The sender of a message to "find" makes the following call:

```
  x ← ISSUE (directive, "A", "B", MESSAGE find
(10,g) );
```

ISSUE is a call to the message processor which composes a message from the task named A to the task named B to evaluate the procedure "find" with arguments 10 and the value of g. The message record is composed and placed in the message queue. The sender has further options, specified by the value of the directive code: any or all of the following may be specified in combination, to be interpreted in the order given here:

1). SEND the message to the task named B. If B is blocked awaiting receipt of a message, the message processor directs the timesharing system to unblock the job which corresponds to the message destination B. Races inherent in the decision to unblock are avoided by interlocking portions of the shared message processor code.

2). WAIT for ACKNOWLEDGEMENT of the message. Since it is common practice to ACKNOWLEDGE a message after evaluation is complete, WAITing normally means blocking execution of the calling task until the called task has evaluated the message procedure.

3). KILL the record of the message (i.e. delete the record from the message queue). If KILL is not specified, the message record will remain in the queue even after the message has been received and processed by the called task. KILLing is not automatic, since the calling task may wish to examine this record.

The ISSUE function returns a descriptor for the message issued which can be used for further inquiries. For example, A need not wait for B to complete evaluation of the procedure, but may wish to continue execution. However, at some later time, A may wish to verify that the message procedure has been invoked and completed, and may use this descriptor to make such inquiries of the message processor.

The receiving task, B, must process messages sent to it. Whenever the program for B wishes to scan the message queue, it executes the following statement:

```
x ← GET_NTRY (directive, "A","B", "find") ;
```

which invokes a search of the message queue for messages destined for B from A requesting evaluation of procedure "find." The receiving task may be indifferent about the names of caller and message procedure, and may specify in the directive code that this portion of the match is to be ignored. The receiver may also specify that his execution is to be blocked until an appropriate message arrives. The value of GET_ENTRY is the descriptor of a message.

The receiving task may use this message descriptor to invoke several message processor functions with the QUEUE call:

1). ACTIVATE the message. The parameters in the message record are used to evaluate the message procedure test in the receiver's program.
2). ACKNOWLEDGE the message. This is usually an indication to the caller that the message procedure has been evaluated. If the calling program was WAITing for such an ACKNOWLEDGEMENT, its execution is now resumed.
3). KILL the message record.

The message processor has a feature which allows tracing of messages. When a message is ISSUEd, a message record is composed but not SENT. The message processor sends another message to a task named 'TRACE." This new message is a request for evaluation of the message procedure "TRACE" with the descriptor of the message record being traced as parameter. The original ISSUEr is blocked until the TRACE message is ACKNOWLEDGED, at which point the original message is finally SENT.

Printouts by the 'TRACE" procedure of the message activity which cite times of ISSUE, source, destination, message procedure names and parameter values are a great aid to debugging. The trace facility may also be used by the strategy task, which monitors all requests for message procedure evaluations and approves of them or prematurely ACKNOWLEDGES them and sends an indication to the caller that his request was denied. The strategy program administers resource allocation of the hand-eye system and may wish to deny requests which use special I/0 devices or extensive calculations.

5. User's Control at the Console

Monitoring of the hand-eye system is accomplished with another timesharing job, the "pseudo-teletype controller," which mixes teletype output from all of the system tasks for display at the user's console and allows him to send character strings or timesharing commands to any of the tasks. A typical session will involve starting the controller task, using it to create one timesharing job for each hand-eye system task and assigning each a logical name. To send characters to a task via the controller, the user types:

```
lognam; characters to go to task of name lognam
```

The user may subsequently omit the "lognam;" and strings will go to the most recently specified destination. The user may also specify timesharing-system commands to be executed on behalf of any task. A special command format is implemented since command strings must be directed to the timesharing system command decoder rather than to the running task:

```
lognam: timesharing command for task lognam
 :: command to be sent to all tasks
```

Output from jobs is displayed to the user as:

```
lognam→output characters
```

Task operation with the pseudo-teletype controller task as an intermediary or with a direct connection to the user's console appear identical to the task. Any requests for teletype input or output are routed by the timesharing system to the pseudo-teletype controller. The timesharing system also routes task display output to the graphics display console used by the pseudo-teletype controller. The user may thus interact from a single console with all jobs in the hand-eye system. All transactions of the pseudo-teletype controller may be traced and recorded on a disk file. This record and the message procedure trace provide a convenient post-mortem.

6. Debugging

One troublesome aspect of running this system is that a bug which appears in one run may not reappear subsequently. A little less noise in the camera or position-sensing potentiometers of the arm may prevent recurrence of the error. Many of the tasks keep traces of their performance as the system executes. This information may be used to search for signs of irregularity, or to recreate in a task the conditions of failure and thus permit the programmer's scrutiny.

The organization of the system greatly facilitates debugging it. The modular nature of the system permits the user to replace a defective module with a dummy which records on a disk file the global environment and input information sequence for the task. Later, a small system consisting of the defective module and a "driver" is assembled. The driver uses the disk file to recreate the environment in which the defective module failed; the author of the defective code may now engage in debugging activities without loading the entire hand-eye system. For example, the body recognizer is checked out using disk files containing two-dimensional line representations of objects. These files are created by a two-task system in which a dummy invokes the edge follower to read the TV camera and edge-follow a body, and then saves a file containing the edge data.

In lieu of a special driver for debugging each subsystem, the user may type SAIL statements for immediate execution. A special version of the SAIL compiler was constructed which compiles code into core and executes it on demand. This compiler is initialized with the appropriate global data and message procedure definitions and is executed as one of the tasks in the hand-eye system. A user may construct message procedure calls or strategies on-line, may examine global variables, and may search the global associative store. An example of such an "immediate" SAIL program is:

```
EXECUTE  show_item ( blobs ) END;
```

A file containing all on-line commands is kept which later can be edited to yield a SAIL program for compilation.

A machine-language debugging program, RAID(3), is loaded with each task. If necessary, the user at the console may interrupt a task and use RAID to peek at the memory cells assigned to the task. The loading process leaves a symbol table which has entries for all local and global data names used by the task, as well as indications of procedure entry points, labels, etc. RAID uses this symbol table to create a symbolic display of machine code and variable values.

7. Discussion

The global data, message procedure, console and debugging features described have all been implemented and used in the Stanford hand-eye project. These facilities were used in programming the Instant Insanity puzzle, which required 9 cooperating tasks (6). These tasks were designed to test new approaches to hand-eye solutions, especially the ideas of strategy generation and supervision. However, the strategy program, its protocols for monitoring tasks, and its interaction with the timesharing system have not yet been fully developed.

The hand-eye system researchers find this collection of system facilities fairly easy to use. One major shortcoming, however, is the lack of a SAIL debugging system capable of displaying and modifying the various algebraic and symbolic data in a program. Examining LEAP structures is particularly difficult; some users included in their programs procedures for presenting their data structures in a meaningful way (e.g. a display of a perspective projection of the "world" they are processing).

The flexibility of this system steins from the factoring of the hand-eye problem into a set of largely independent tasks. The interface among tasks (message procedures) and shared data are all documented in a small file which contains the SAIL declarations which each task includes in order to access this data. An experimenter who wishes to add a task to the system or to replace an existing task has only to observe the conventions for communication with neighboring tasks. He may then test out his new methods as they are applied during solution of various manipulation problems posed to the system.

The smooth evolution of the system so far suggests that modifications to accomodate future requirements may be quite simple. The message procedure and global data facilities were added easily to SAIL as the system design progressed.

The data sharing and message communication facilities described in this paper represent one solution to the problem of multitasking. The PL/1 language provides a very similar solution, in which procedures may be executed as asynchronous TASKs, each of which is named. Synchronization is accomplished by WAITing for the completion of a named task. Since all tasks are procedures declared in one PL/1 program, data sharing is straightforward: global data is declared in the outer block and is hence accessible to all tasks (procedures). The PL/1 solution was inadequate for the hand-eye system for several reasons: (1) All procedures must reside in core simultaneously, in order that the execution environment for a task can include all outer block data (This is also in keeping with

OS/360 restrictions). The hand-eye system is too large to enjoy this luxury and includes large quantities of code used only for initializtion, for error recovery, or for debugging which need not reside in core during a run (One function of a swapping timesharing system is to move idle tasks to secondary storage). (2) If all tasks are compiled as one PL/1 program, the program must be recompiled each time a local change is made. The tremendous size of the hand-eye system and the large number of programmers at work on it would make this process time-consuming and difficult. (3) Our system permits separate tasks to be substituted at will with a minimum of attention to the environment in which they will execute. (4) Our system requires an explicit scan of the message queue to avoid invoking procedures when inconvenient to the called task. The PL/1 facilities do not directly solve this problem.

Data sharing and message procedure mechanisms are useful to programs other than the hand-eye system. Compelling reasons, both aesthetic and practical, often suggest dividing a programming task into a collection of possibly concurrent processes, but the lack of good ways to coordinate the processes is a sizeable obstacle. We feel that a programming language should provide solutions to these problems appropriate to the operating system used to execute the programs.

Acknowledgements

Much of the development and programming for the system described in this paper were done by Daniel C. Swinehart and Karl K. Pingle. Many users oi' the system offered helpful suggestions.

References

1. Hie Stanford hand-eye system is run on a PDP-10 and PDP-6 computer, with 128k of core memory connected to both processors. The PDP-10 executes timesharing jobs, and the PDP-6 is used for real-tine service and for driving special hardware devices. Hand-eye hardware such as camera motors, ami controls, A-D converters are serviced from the PDP-6. The TV camera is serviced by the PDP-10 timesharing system. Users may work from one of 6 display consoles, from one of several teletypes, or from one of 25 video terminals. The timesharing system is a DEC PDP-10/50 system, extensively modified to service the TV, displays, shareable second segments, and diverse scheduling requests.

2. Swinehart, D.C., and Sproull, R.F. SAIL Manual, Stanford Artificial Intelligence Laboratory Operating Note No. 52.

3. Petit, P.M. RAID Manual, Stanford Artificial Intelligence Laboratory Operating Note No. 58.

4. Feldman, J.A., and Rovner, P.D. An Algol-Based Associative Language, CACM, 12, 8, p. 439 (1969).

5. Dijkstra, E.W. The Structure of "THE" Multiprogramming System, CACM LI, 5, 341.

6. Feldman, et al. The Use of Vision and Manipulation to Solve the Instant Insanity Puzzle, these proceedings.

7. PL/1 Reference Manual, IBM DOC. No. C28-8201-1.