UAH RESEARCH REPORT NO. 752

# LANGUAGES FOR ARTIFICIAL INTELLIGENCE: IMPLEMENTING A SCHEDULER IN LISP AND IN ADA

Prepared by:

Dan Hays
Psychology Department
The University of Alabama in Huntsville
Huntsville, AL 35805

Prepared for:

John Wolfsberger
System Software Branch
Information and Electronics Systems Lab
George C. Marshall Space Flight Center
National Aeronautics and Space Administration
Marshall Space Flight Center, AL 35812

October 1988

# LANGUAGES FOR ARTIFICIAL INTELLIGENCE:
## Implementing a Scheduler in LISP and in Ada

*Abstract*

A prototype scheduler for space experiments originally programmed in a dialect of LISP using some of the more traditional techniques of that language, was recast using an object-oriented LISP, Common LISP with Flavors on the Symbolics. This object-structured version was in turn partially implemented in Ada. The Flavors version showed a decided improvement in both speed of execution and readability of code. The recasting into Ada involved various practical problems of implementation as well as certain challenges of reconceptualization in going from one language to the other. Advantages were realized, however, in greater clarity of the code, especially where more standard flow of control was used. This exercise raised issues about the influence of programming language on the design of flexible and sensitive programs such as schedule planners, and called attention to the importance of factors external to the languages themselves such as system embeddedness, hardware context, and programmer practice.

## Table of Contents

# LANGUAGES FOR ARTIFICIAL INTELLIGENCE:
## Implementing a Scheduler in LISP and in Ada

### I. Introduction

This report documents research to explore implementation of an artificial intelligence application, a scheduler for space-resident activities, in various computer languages. This task is one from a larger research effort supported by Grant No. NAG8-641 from NASA's Marshall Space Flight Center to the Johnson Research Center at the University of Alabama in Huntsville. Donnie Ford is Principal Investigator on this grant. John Wolfsberger at Marshall Space Flight Center, is the Contract Monitor. Dan Hays is Task Leader for the research described in this document and the author of this report.[1] The report covers the project year.

*Project Overview.* A scheduler for experiments and other resource-consuming activities in a Space setting, originally developed by Floyd and Ford, was reprogrammed in two computer languages.[2] The original was written in ZetaLISP, using classical techniques of LISP programming. The other two languages used were Common LISP with Flavors, an object oriented language, and Ada. The suitability of yet other languages was also examined briefly.

---

[1] Comments on this report or questions about the research may be addressed to Dr. Hays at 135 Morton Hall, The University of Alabama in Huntsville, Huntsville, Alabama 35899, or to Dr. Ford at Johnson Research Center, Research Institute Building, The University of Alabama in Huntsville, Huntsville, Alabama 35899.

[2] An earlier report from this project was delivered as a paper: D. Hays, S. Davis, and J. Wolfsberger, "An Object-Oriented Implementation of a Scheduler in LISP and in Ada", to the 1988 Conference on Automation for Military and Space Activities, Huntsville, AL.

The scheduler incorporated various features of "artificial intelligence" programs, such as extended and variable searches and flexible input. Since it had been shown to work with data drawn from space missions, it qualified as a practical application of AI programming techniques. Thus, the scheduler provided a sense of realism that a more schematic or *ad hoc* program might not be able to.

As it turned out, the reprogramming became much more a matter of reconceptualization, in going from one computer language to another, than of simple translation (if there is such a thing, even in the world of digital devices). In particular, the recasting of the scheduler from LISP to Ada provided a number of lessons of implementation.

*Research Staff.* Persons at UAH specifically involved in the investigation reported here have included the following:

Dan Hays, PhD. Research Scientist at the Johnson Research Center, and Associate Professor of Psychology. Dr. Hays served as Team Leader for this investigation.

Stephen W. Davis. Research Associate, Johnson Research Center[3]. Mr. Davis was responsible for the programming reported, as well as for aspects of the conceptual development.

Professors Donnie Ford and Stephen Floyd of the Johnson Center and the Management Science/Management Information Systems Department of the School of Administrative Science, provided an essential ingredient of this task by supplying the original scheduling program (see below). They also assisted this project by commenting

[3] Now at Advanced Technology Inc., in Huntsville, AL.

program (see below). They also assisted this project by commenting on the structure of their scheduler and their intentions in its design. Discussion of various points was also provided by other personnel of the Cognitive Systems Laboratory of the Johnson Center. In addition, Mr. Davis was able to talk with several programmers in other organizations who were familiar with Ada implementations, during that part of the project.

*Summary Conclusions.* Though the discussion on which they rest is contained subsequently in this report, several conclusions from this project seem worth calling attention to here.

1. The difference is considerable between brief "benchmarks" and programs meant for use in situations.

2. "Translation" of practical programs will probably not be straightforward if you are trying to make a program that works well.

3. "Object-oriented" techniques may offer advantages to conceptualizing computations such as those involved in the scheduler investigated here. They can also result in improvements in performance.

4. What an "artificial intelligence technique" is, becomes less clear after being examined.

5. Viewed as a case study, the adaptation of the scheduler to Ada involved a number of problems that were in a sense external to the language itself as an abstract structure. These ranged from differences in implementation of memory allocation, to

"mundane" but sometimes frustrating problems of program editors, hardware, and system features.

6. Finally, possibly because the project investigated a program meant for real use, attention was called to contextual factors, including the role of the programmer.

## II. The AI Language Problem

From a machine-centered point of view, the reasons to worry about the choice of a programming language would seem to be few: whether or not one can symbolize a problem for processing by machine, and the efficiency of the resulting machine-level code, thus performance of the program.

But even for fairly standard programming problems, such as straightforward numerical computation, or retrieval of specific information from fully specified locations in an electronically stored corpus, *possibility* and *efficiency* are not enough, so that *individual preference* and *social factors* influence choice of computer languages. The relation of the programmer to the language used is not just one of simple preference. For example, habitual usage is certainly a major source of preference for a language, but it also contributes to expertise in using the language.. Some language features—and features of the facilities that are associated with a language, such as editors, compilers, libraries of code, and so on—may make a programming system harder or easier to use, more or less prone to some kinds of oversight, easier or problematic when other programs must be coordinated, and easier to read or more opaque. Such features, which might collectively be called *human suitability* factors, are now much more widely recognized than they were in earlier decades. For example, features intended to protect writers of programs from certain kinds of errors of reference were important in the design of Ada. Other languages have constrained unconditional branching commands, a source of problems in many programs. Again, to take a simple example, the difficulty that most persons have in counting nested parentheses without putting a finger or a pencil on them has lead to graphic aids for pairing parentheses

in LISP editors.

These five kinds of factors—possibility of expression, efficiency of realization, individual preference, human suitability, and social motives and constraints—apply to languages in which one may write an "artificial intelligence" program, as well as to other applications. Each will be discussed to some extent in this report.

The matter of languages for heuristic or otherwise "smart" programs evokes more particular issues at this point in time, however.

The question of which computer language to use for an AI program is sometimes confounded in current thinking with the issue of *the acceptability of AI programs in the first place*. Thus, some may feel that a program written in a language familiar to their applications, such as C or Fortran (depending), lends acceptability to a program incorporating heuristic strategies, whereas a program to accomplish the same results but written in a language that is not well known to them, such as LISP or Prolog, adds unfamiliarity to uncertainty. The acceptability of one or another kind of AI programming concept is a serious question in some quarters. Lack of understanding of what is involved in an AI program of one or another sort may underlie reluctance to adopt AI techniques. Understanding has not been helped in some instances by broad claims made for AI programs by proponents or entrepreneurs. A better defined issue is program validation. Many managers have raised questions about program reliability. Since even programs which seem to promise determinate or simple mathematical results have been known to fail, incorporating programs which offer heuristic, or complexly reasoned, solutions, seems to overextend program validation procedures which are already expensive and error-prone enough, even

using the apparently simpler programs. If it can be said that, "After all, this program is written in C," some reassurance may be felt.[4]

It is probably no accident that many people today are considering the incorporation of programs involving AI techniques at the same time that they are contemplating *much more extensively linked software-hardware systems* than they have previously had to develop. A group might of course consider using isolated expert systems in parts of their operations, running on small computers not connected to one another; or adding more logical capability to a decision-making program running on an isolated mainframe or minicomputer. But it frequently seems to be the case that questions about informationally interdependent computer systems seem to be of concern to the groups that are also worried about how best to incorporate heuristic programs.

*We think that this connection is not at all accidental. We believe that distributed computing that has to be flexible, and that is fairly complex, will probably have to use some of the techniques of "intelligent" programs in order to accommodate its own complexity.*

That programs might be part of larger systems, or might have to *run on different hardware at different times or in different settings,* is a major external motivation for entertaining the idea of using of one language rather than another. For these reasons and some others, pressure has been spreading, at least in the U. S. and Britain, to adopt Ada as a standard computer language, since part of its design intention was portability among devices. During the past year and even during the past several months, the Ada question has

[4] At least by those who are reassured by the use of C.

become a real one for many organizations that had previously only had incidental opinions on the matter.

Because of the timeliness of the issue, we chose to examine Ada ahead of some other languages that might have been looked at.

If, as noted above, the use of AI programs is unfamiliar to managers and programmers in many settings, it is conversely true that *persons who have been involved in the development of artificial intelligence methods have not often had to have their programs work in the operational settings of government and business*. At this point, limited expert advisory systems do work in probably hundreds of settings, but many other heuristic programs are still in the laboratory stage, or demonstrate certain techniques without incorporating them into rugged, responsive systems.

The step from demonstration to operation may be a large one, and it is related to the language issue in certain ways. To take a simple example, if one is using a language whose implementation requires that the computer stop now and then for "garbage collection" of released storage allocations, what may be a mild nuisance in the computer lab can become a faulty interface feature in an operational setting.[5]

Other issues relating language choice to operational status of a program are likely to be more subtle. For example, even if use of a certain language produces more efficient code for a run-time system—one thinks of some cases where AI programs originally worked out in LISP were redone in carefully optimized C—it may or may not be easy to update or correct bugs in these programs in the

[5] Stopping for garbage collection is not a problem with computing machines such as the Symbolics which are designed especially to run list-processing languages.

apparently leaner language.

These issues, implicated in the choice of languages for AI programs, are large and in some cases difficult ones. This investigation will not resolve them, but may add to their understanding in a selected context.

## III. Versions of the Scheduler

### A. The Original Version

The goal of the project was to explore the usefulness of one or another computer language in programming a moderately complex problem involving some of the organizing techniques of machine intelligence applications. To accomplish this, a prototype scheduler for space module activities developed by Donnie R. Ford and Stephen A. Floyd[6] was chosen for reprogramming . This scheduler had several things to recommend it to this project.

- It addressed a problem of practical importance: scheduling resource-consuming activities in the constrained setting of an orbital Space facility. By extension, it could apply to formally similar situations of scheduling.

- The program was large enough to be interesting: it actually did something useful. At the same time, it was small enough to be manageable for this exercise. (The moderate size and complexity did have implications for issues discussed below. Briefly, a more complex version would have had the chance to be more flexible and more "intelligent" in certain senses; but would have involved much more reimplementation time.)

- The scheduler worked. It had been demonstrated on a number of occasions, and was known to be rugged. In particular, it had been checked with realistic data from Space missions (see the listing in Table 1, taken from of Floyd & Ford, 1986).

[6] Stephen Floyd and Donnie Ford, "A Knowledge-based Decision Support System for Payload Scheduling", Proceedings of the 1986 Conference on Artificial Intelligence for Space, pp. 69-78. This paper is reprinted as Appendix E.

- It was written in the same lab we were in, though by different persons. Thus, the original developers were close at hand to resolve questions about how it was constructed.

Making comparisons among similar programs is not easy. If any changes at all are made, it may be questioned whether the program is really the same. Even in the area of limited-scope benchmarks, reimplementing essentially the same procedure in another dialect of a language, or preparing it to run on another machine, may raise questions about the detailed correspondence of parts. If the program has any complexity, and the languages, compilers, or machines are at variance in any way, then exact correspondence is not possible, not for that matter desirable, unless for some reason direct emulation at issue.

In the versions prepared here, we were concerned to maintain the major functioning of the original program, so that the various versions would do about the same thing, though perhaps not in just the same way. Such a task requires that judgements be made. It should be clear that different approaches to the reprogramming might have been undertaken. An attempt was made, though, to preserve the major behavior of the original program. In addition, both the original and subsequent programs were structured so as to be open to further development. Thus, the coding might have been simpler or more optimized if intentions for future development.had not been kept in mind.

The prototype scheduler, in its 1986 version, was dubbed *Dypas,* for Dynamic Payload Scheduler. It was written in ZetaLisp for the Symbolics™ computing machine. It was programmed using what might be called traditional LISP techniques. Property lists (a kind

of structure available in most traditional LISP implementations) were used fairly heavily for information structures and for specifying rules and evaluating them. Generation of possible solutions involved a fair amount of concatenation of lists, and searching over such structures.

Many varieties of scheduling programs exist, varying in scope of constraints, in how they are tailored to likely process mixes, in reliance on tightly algorithmic as contrasted with diverse and heuristic bases, and so on. This scheduler was designed for space tasks, and thus emphasized power requirements. Also, the processes to be scheduled were mostly one of a kind, as contrasted to some kinds of job-shop scheduling where the same kind of task might be repeated. Because of the way space experiments are prepared to be more or less self-contained, they did not rely on a variety of resources. Various schedule planners for space activities have been described. One that appears to be something like this one in general scope is the MAESTRO system[7]. Another is the scheduler of Bahrami and his colleagues[8].

Floyd and Ford (1986) describe their routine in part as follows:

> During the preparation phase the individual experiment information is provided to the system from [a] data base and appropriately stored, also the working memory is organized and then prioritized for the scheduling phase. This is accomplished using a priority scheme developed from user input. In the

[7] A. L. Geoffroy, Daniel L. Britt, Ellen A Bailey, and John Gohring, "Power and Resource Management Scheduling for Scientific Space Platform Applications", Proceedings, AIAA, 1987, pp. 660-664.

[8] K. A. Bahrami, E. Biefeld, L. Costello, and J. W. Klein, "Space IPower System Scheduling Using an Expert System", American Chemical Society Proceedings, 1986, pp. 1813-1818.

scheduling phase, the experiments are scheduled under [a] heuristic procedure *[see next quote]* and the schedule is created. The schedule itself is part of the knowledge base and is represented as frames. As experiments are scheduled, the subintervals required by the heuristic procedure are defined by start and stop times of the experiments. For each interval the power available, crew available and the experiments that are currently on-going are determined and stored. This information is required for the remaining two phases, namely operation and rescheduling (1986, p. 73).

After setting the schedule, the 1986 Dypas displayed the progression of the scheduled tasks, which included both routine operations and experiments, using a graphical representation. This "operation" phase was not part of the reimplementation of this paper, which stops with a somewhat simpler display of information. It may be noted that the information presentation of the original scheduler, which used familiar Gantt chart formatting, was one of its interesting and useful features.

The "heuristic procedure" referred to above that was central to the scheduling was described by Floyd and Ford as follows:

The inference engine performs only forward chaining. This was determined from the structure of the problem. There is an abundance of related facts and information at the beginning of the problem solving process which in turn accommodates the forward chaining process. The conflict resolution problem is solved by allowing the first rule that is satisfied to be implemented. This necessitates an ordering of the rules. This resolution method was chosen because of the short time frame for delivering a [prototype] system. This also facilitates the search through the

working memory." (1986, p. 72)

They defined a procedure whereby an experiment (or other power-consuming task) is "placed on the chart", with its beginning and ending points tagging times for the possible start of other processes. "By updating as each experiment is scheduled, one can maintain for each subinterval of time the information necessary in determining the time slot for the next experiment to be scheduled. The determination of which experiment is to be scheduled next is based on the user predefined priority structure in effect at the time of the scheduling or rescheduling procedure." (1986, p. 72)

In practice this meant that long tasks, which usually also had high priority, were scheduled early in the procedure, based on the "first process to be satisfied" arrangement. Then, others were fit in, as various priorities might be satisfied without exceeding power capabilities. This kind of procedure seems reasonable for the mix of tasks that would be present in space work, where frequently the long-term uses of resources would be those that were important to environmental stability and overall mission success.

Thus, the scheduler took into account important features of space-resident tasks, and proceeded in a fairly direct way to obtaining a feasible schedule that satisfied various constraints. After system setup, the program calculated a schedule for the tasks submitted to it as input databases, then displayed them graphically as a Gantt chart on the screen of the Symbolics monitor.

The version of the scheduler essentially described above was demonstrated on a number of occasions, both locally and in agency settings, and at the 1986 American Association for Artificial

Intelligence meeting in Philadelphia.

LABORATORY MODULE - SPACE STATION

DYNAMIC PAYLOAD SCHEDULER

SUBSYSTEMS:

| NAME | POWER WATTS | PRIORITY CLASS | AGENCY | DURATION | CREW |
|---|---|---|---|---|---|
| ECLSS | 6200 | I | NASA | CONTINUOUS W/20 MIN. LAPSES OKAY EVERY 4 HRS. | 0 |
| COMMUNICATIONS | 1480 | I | NASA | CONTINUOUS | 0 |
| THERMAL CONTROL | 600 | I | NASA | CONTINUOUS-REDUCES LINEARLY TO 400W FOR 10 KW POWER LEVELS | 0 |
| HOUSEKEEPING(MISC) | 6000 | I | NASA | CONTINUOUS | 1 |
| PAYLOAD/EXPERIMENTS: | | | | | |
| DOD/PAYLOAD 1 | 890 | II | DOD | 48 HRS | 1 |
| ESA PAYLOAD 1 | 1845 | II | ESA | 214 HRS | 1 |
| IPS | 165 | II | ESA | 240 HRS | 1 |
| ELECT DIAG STA | 435 | II | NASA | 10 MIN OF EVERY HR | 1 |
| IECM | 480 | III | NASA | 200 HRS | 0 |
| CRNE | 930 | IV | U.K. | 240 HRS | 0 |
| GEN PURPOSE COMP | 383 | III | NASA | CONTINUOUS (5 MIN/HR) | 1 |
| SOLID POLYMER ELECT | 415 | IV | 3M | 36 HRS 1-5 MIN. EVERY 3 HRS | |
| IEF | 125 | IV | NASA | 6 HRS | 0 |
| MLR | 350 | IV | U/IOWQ | 20 HRS | 0 |
| FES-VCGS | 600 | III | NASA | 15 HRS | 0 |
| ROTI | 36 | IV | UAH | 43 HRS | 0 |
| SEM | 2648 | IV | NASA | 6 HRS | 0 |
| RTG | 94 | IV | NASA | 12 HRS | 0 |
| TAPE RECORDER 1 | 85 | II | NASA | CONTINUOUS | 0 |
| TIME CODE GEN | 32 | II | NASA | CONTINUOUS | 0 |
| MASS SPECTROMETER | 215 | IV | JAPAN | 2 HRS | 0 |
| TOOL CHARGER | 50 | II | NASA | CONTINUOUS | 0 |
| FILM PROCESSOR | 163 | II | NASA | 1 HR/DAY | 1 |
| SUPER FURNANCE | 7840 | II | G.E. | 32 HRS | 0 |
| SILICON WAFER PROD | 4760 | III | INTEL | 14 HRS | 0 |
| TAPE RECORDER 2 | 85 | III | NASA | CONTINUOUS | 0 |
| TGA | 612 | IV | ESA | 8 HRS | 0 |
| MEA | 1800 | IV | NASA | 14 HRS | 0 |
| WELDING EXP | 1610 | IV | NASA | 4 HRS | 0 |
| CFES | 890 | III | NASA | 36 HRS | 0 |
| 3-AAL | 500 | IV | NASA | 10 HRS | 0 |
| EML | 420 | IV | NASA | 2 HRS | 0 |
| GFFC | 375 | IV | NASA | 6 HRS | 0 |
| ADSF | 480 | IV | NASA | 48 HRS | 1 |
| ARC | 215 | IV | NASA | 8 HRS | 0 |
| SAFE | 400 | III | NASA | 15 HRS | 1 |
| SOLAR OBS | 375 | II | NASA | ORBIT/DAYTIME ONLY FOR 36 ORBITS | 0 |
| LIGHTNING DET | 125 | IV | NASA | ORBIT NIGHTTIME ONLY FOR 12 ORBITS | 0 |
| CRYSTAL GROWTH | 1200 | II | NASA | 1 HR | 0 |
| COMET SEARCH | 650 | III | JAPAN | ORBIT NIGHTTIME ONLY FOR 40 ORBITS | 0 |
| LIFE SCI 1 | 135 | III | A&M | 36 HRS | 0 |
| LIFE SCI 2 | 1145 | III | UAB | 22 HRS | 0 |
| LIFE SCI 3 | 842 | III | UAB | 66 HRS | 0 |
| CLASSIFIED 1 | 1300 | II | DOD | 8 HRS | 0 |
| CLASSIFIED 2 | 645 | II | DOD | 18 HRS | 0 |
| MAPPING (WEATHER) | 300 | III | USWS | CONTINUOUS (CAN BE INTERRUPTED ANYTIME) | 0 |
| MAPPING (GEO) | 690 | III | USGS | 60 HRS | 0 |
| ORBITER DOCKING | 6500 | ** | NASA | 24-72 HRS, WILL BE GIVEN 6 HRS NOTICE | 0 |
| ORBITER SERVICER | 2400 | ** | NASA | 4-10 HRS, WILL BE GIVEN 2 HRS NOTICE | 0 |

** WILL BE GIVEN TOP PAYLOAD PRIORITY WHEN NEEDED

Table 1.

Space-Resident Tasks Used to Check Dypas Scheduling Programs (from Floyd and Ford, 1986)

## B. The Common LISP with Flavors Version

What was taken to be the main functioning of the 1986 Dypas scheduler was reprogrammed into object-oriented LISP. The base dialect for this reprogramming effort was Common LISP, the standard dialect of LISP that has gained wide acceptance over the past several years. More important to the language issue than the base dialect was the use of the object-oriented programming facilities of Flavors, an extension of LISP. As in the case of the earlier scheduler, this version was developed for the Symbolics computer[9]. The Flavors/Common LISP program was developed under Release 7 of the Symbolics operating system. S. Davis was the programmer.

The processes, that is, the activities to be scheduled, were treated as major objects in the program. They were generated, evaluated, sent messages and in effect queried. The scheduler itself, called a "Dypas", was also treated as an object.

(In fact, the scheduler objects were constructed to run under a multitasking operating system, so that there could be as many as four Dypases working in virtual concurrency, communicating via several state variables. This fact may help explain some of the internal program details, since this version was set up also to explore one kind of (quasi) concurrent organization that we are interested in looking at for future versions of the scheduler.)

A list of names of the main *methods,* the computations to be made, actions to be taken, invoked upon receipt of a message by a computational object may suggest something of the approach taken.

[9] Variously, Symbolics Model 3640 and 3670 machines were used.

The following methods were defined for dypas-processes:

- *make-instance*
- *describe-process* (to provide a consistent format for noting information about the process's priority, current status, 'geometry', next neighbors, and so on)
- *remove-process*
- *create-node*
- *initlz-node*
- *collect-time*

The following methods were among those written for scheduler (dypas) objects:

- *initlz*
- *find-root* (a major method; see below)
- *find-children* (a major method, also discussed below)
- *find-max-power-in-interval*
- *power-at* (more particular)
- *graphical representation*
- *presentation-print*
- *print-power@*
- *print-name*
- *draw-tree* (last several for communicating results to user)
- *initlz-hash*
- *build-hash* (to set information up for quick access)

The labeling of the methods suggests what the program is mainly about, namely, setting up computational entities, manipulating them in familiar kinds of structures such as trees, displaying them, and so on.

They also indicate that this program, as an example of object-oriented realization, is structured more as what one of us (Hays) refers to as "standard computational modularization" or even "standard proceduralization" rather than "radical objectification", where the objects are treated in more metaphorical terms. Traces of the latter can be found in this program, in references such as "schedule-me" and "schedule-us".

One of the central methods for a dypas, *find-root,* is shown as Figure 1. Even without a tracing out of all of the details included, it may suggest some of the approach taken. The method sorts currently schedulable processes to identify those that need to be attached first, then sets them in place.

A lengthier dypas method, *find-children,* is listed in Figure 2. It looks ahead, so to speak, to try out places for subsequently scheduled processes. In doing so, it consults information on resource needs, priority, and timing. The problematical part of the scheduling for this application is handled by *find-children.*

These two methods roughly reflect the general strategy of the original scheduler of assigning high priority processes first, considering constraints of resource usage (mostly power requirements), then fitting in activities of less importance to the mission, less power drain, and so on. A listing of the Common LISP/Flavors version of the scheduler is given in Appendix B. Instructions for running it on a Symbolics computer are given in Appendix A.

It is interesting that in the mix of space tasks on which these schedulers were tested, the high priority processes tended to be of

long duration and generally required substantial power. Less critical processes were frequently shorter and less demanding of resources, though there was some variation. These real-world correlations are not essential to the performance of either scheduler, but it might be noted that in actual scheduling applications the processes being worked with might very well have particular features such as these.

*Rethinking the Scheduler.* Though broadly similar in the way scheduling proceeded, the original LISP version and the Common LISP/Flavors version differed in various ways, as one might expect.

The Flavors version of the Dypas scheduler is clearly a recasting and not just a translation to another programming system.

Both versions have in common the input of a record-like description of the tasks to be scheduled, though these were formed somewhat differently. The original version used property lists. The Common LISP version used a somewhat leaner record format.

"Rules" for weighting the assignment of tasks to time periods, given constraints of resource availability, were incorporated into the original program as list structures. Neither program as developed contained the overhead needed to ask for and immediately take in arbitrary new rules, or to modify old ones short of entering the appropriate part of the code. (Writing a more or less arbitrary rule processor that impacted scheduling computation would have involved a much greater programming and debugging effort than either project could manage—and was not necessary either to illustrate the principles involved, or to obtain programs that worked. Indeed, to write such a "shell"would have involved somewhat different programming considerations overall.) Even so,

the original version had a fairly strong rules-orientation conceptually, it seems to us. The Common LISP/Flavors version streamlined the "rules" of the earlier version into program logic. That is, the invariant high priority rules of the original program that essentially gave great weight to power requirements, and so on, can be seen as *for*-loops in the central dypas methods of the recast version. Thus, the approach of the second program was to embed some of the semantics in the procedural logic, rather than to include it in LISP structures which were referenced. This approach probably contributed to the efficiency of the second version. But the effect, of weighting certain kinds of factors in the search for a pattern for the processes that would meet constraints, was about the same in both schedulers. Even the original scheduler, though much more given to structures that could be construed as "rules", seemed to opt for computational efficiency in its search-and-fit strategies.

One might not expect, just on the face of it, for an object-oriented program to run efficiently. Though it seems pretty clear that the "objectification" of computational entities that an object-oriented system allows, can make thinking about the parts of a problem easier, the overhead involved could be great enough to trade clarity for bookkeeping overload. This would seem to be especially true if the "objects" were generated in the profusion required for searching and testing the fit of a schedule, or some other problematical pattern of the kind likely to be found in AI applications..

It turned out that the Flavors version of the scheduler was decidedly more efficient than the original version. Running on the same machine, with the tasks of Table 1, the second version ran over 4 times as fast as the original, for the main record-consultation and scheduling part of the program. (The display of items in a Gantt

chart was not implemented in the Common LISP version; nor was this included in the timing figures.)

Thus, reliance on "objects" hardly swamped time or space resources. It is not clear, on the other hand, that the use of object-oriented facilities speeded up the scheduler, since there were several other reasons that the second scheduler would run faster than the original version. Because of these additional factors, we cannot say that the Flavors realization of objects was definitely a factor in speed, but we can at least say that we were left with the strong impression, after going through traces of some of the computation, that the implementation of objects in the Symbolics version of Flavors seemed to work fairly leanly so may well have contributed to the speedier performance.[10]

Other factors[11] contributing to the faster performance of the Common LISP/Flavors version would probably include:

- use of efficient hash addressing schemes for the "data" records consulted by the program in working with tasks to be scheduled;
- incorporation of high priority factors into program logic;
- miscellaneous program features.

Relative to the last point, it should be noted that although both versions of the program were intended to work with practical data in realistic situations, the second version was written with efficiency definitely in mind.

---

10 By contrast, we have experienced sluggish performance of object-oriented programs in some earlier systems, as well as major memory requirements..

11 One difference between the two versions that had little to do with performance differential, so far as we can tell, was the use of ZetaLISP for the first version, and Common LISP as the base dialect of the second version.

It might be noted also that some classical LISP programming techniques may result in time-consuming operation. The use of property lists is a conceptual aid, but is not always recommended, for this and other reasons. Again, because of search implementation, the original scheduler did a very large amount of 'cons-ing' or list concatenation—a familiar kind of behavior of traditional LISP programs.[12]

We were concerned with broadening the facility of the Common LISP scheduler to handle rules in a more direct fashion. Some time was spent in discussing ways to recast the scheduler into yet another version that maintained the DYPAS objectification, but that allowed rule substitution and more of a rule logic. Davis was able to implement a version of the well-known RETE rule-examination algorithm. It is given in Appendix C. The rule-based object-oriented program variant was still in the design stage by the project's end, however.

On the subject of rules, and inference patterns that work with logical structures, it may be remarked that, although one often thinks of "intelligent" programs as being ones which process logic statements (e.g., standard advisory expert systems, or theorem-proving programs), rule-processing is by no means coextensive with the kinds of heuristic, flexible computation we include as examples of "machine intelligence". The case of scheduling of one or another sort, where computations of resource usage are important, suggest that some mix of propositional processing and numeric computation, even numeric modeling, could be useful. Several strategies of

[12] It is often something of an eye-opener for a person who is familiar only with algorithmic programming, to monitor a trace of computation which uses list-processing techniques.

construction are possible in engineering this mix.

*Summary: the Common LISP/Flavors Object Version.* The original scheduler for space-resident tasks, constructed using more traditional techniques of list-processing, was redone, using the object-oriented facilities of Common LISP with Flavors. Both the original and the object version ran on the Symbolics computer. The Flavors version incorporated what was taken to be the main thrust of design of the original scheduler, including a basically forward-chaining logic, search for an acceptable fit of tasks satisfying resource requirements, input of records describing the tasks, and so on. The Flavors version, however, set up computational "objects", and was organized as object-associated"methods" which did the main work of computation based on information or messages passed to the objects, in the usual object-oriented fashion. This program also involved several features, such as hash-addressing of representations of the tasks to be scheduled, aimed at enhancing efficiency. The result was a program that did about the same thing as the original but was much speedier.

```
(defmethod (Find-root dypas) ()
  (loop with processes-sorted =
        (map 'list #'car
             (sort (loop for object in PRESENTATIONS
                         collect (dypas-process-other object))
                   #'> :key #'cdr))
        with available-power = *maximum-wattage*
        with pane = (get-pane 'messages)

        for object in processes-sorted   ;;; MAIN LOOP

        for runpower = (dypas-process-max-power object)

        when (>= available-power runpower)
          do (setf available-power (- available-power runpower))
          and do (push object ROOT)
          and do (push object SCHEDULED)
          and do (build-hash SELF runpower 0
                       (dypas-process-runtime object))
          and do (create-node object T 'Ignore)
          and do (setf (dypas-process-begin object) 0)
          and do (setf (dypas-process-end object)
                       (dypas-process-runtime object))
          and do (newpush (dypas-process-runtime object) START-TIMES)
          and do (setf (dypas-process-condition object) 'scheduled)
          and do (let ((pro (dypas-process-name object)))
                   (dw:with-output-as-presentation  (:stream pane
                                                     :type 'dypas-process
                                                     :object  pro)
                     (format pane "~&~S scheduled with ~D left.~%"
                             pro available-power)))))
```

Figure 1.

The find_root method from the scheduler, Common LISP/Flavors version.

```
(defmethod (Find-children dypas) (&key (stream *standard-output*)
                                       (verbose nil))


    (let ((clock-begin (get-universal-time))
          (schedule-us (map 'list #'car
                            (sort (loop for object in PRESENTATIONS
                                        when (not (member object ROOT))
                                          collect (dypas-process-other object))
                                  #'> :key #'cdr))))
      (loop for schedule-me in schedule-us  ;;; MAIN LOOP
            do (if verbose
                   (format stream "~&~%~'bScheduling ~S~"
                           (dypas-process-name schedule-me)))
               (let* ((those-scheduled (map 'list #'car
                                            (sort (loop for object in SCHEDULED
                                                        collect (cons object
                                                                      (collect-time object)))
                                                  #'< :key #'cdr))))
                 (fresh-line stream)
                 (loop for process-before in those-scheduled ;;; SUB LOOP
                       do
                   (if verbose
                       (format stream "~&  Behind  ~S"
                               (dypas-process-name process-before)))
                   (let* ((begin-time (collect-time process-before))
                          (required-power (dypas-process-max-power schedule-me))
                          (available-power (- *maximum-wattage*
                                              (find-maximum-power-in-interval
                                                SELF schedule-me begin-time))))
                     (if verbose
                         (format stream "~~%Available-Power  = ~D~
                                         ~% Required-Power  = ~D~
                                         ~%     Begin-Time  = ~D"
                                 Available-Power Required-Power  Begin-Time))
```

Figure 2.

The find_children method (begins)

```
(cond ((>= available-power required-power)
       (setf (dypas-process-begin schedule-me) begin-time)
       (setf (dypas-process-end schedule-me)
             (+ begin-time (dypas-process-runtime schedule-me)))
       (create-node schedule-me process-before 'ignore)
       (build-hash SELF required-power begin-time
              (+ begin-time
                 (dypas-process-runtime schedule-me)))
       (push schedule-me SCHEDULED)
       (newpush begin-time START-TIMES)
       (newpush (dypas-process-end schedule-me) START-TIMES)
       (setf (dypas-process-condition schedule-me)
             'scheduled)
       (return
         (if verbose
             (presentation-print *program*
                    (dypas-process-name schedule-me) stream))))
       (verbose (format stream "~%  Look-beyond ~S"
                        (dypas-process-name process-before))))))))
 (format stream "~&~%Scheduling took ~\\time-interval\\"
         (time-difference (get-universal-time) clock-begin)))
(let* ((sum-usage-area (loop for object in PRESENTATIONS
                             summing (* (dypas-process-max-power object)
                                        (dypas-process-runtime object))))
       (longest-runtime (eval `(max ,@START-TIMES)))
       (validation (/ sum-usage-area (* longest-runtime *maximum-wattage*))))

  (format stream "~%Longest process runtime ~\\time-interval\\"
          longest-runtime)
  (format stream "~%Power to time ratio ~2,5F" validation)))
```

Figure 2.

The find_children method
(cont'd)

## C. The Ada Version

The Flavors version of the scheduler was chosen for implementation using Ada, the language and programming system of the U. S. Department of Defense.

This came at a time, early 1988, when many groups in work organizations were very much concerned with questions of adopting Ada for some of their own programming activities. At the time, the drive to adopt Ada, or at least to evaluate it for adoption, had extended well beyond those groups who were writing programs directly for Department of Defense projects.

Given a choice of programming languages to explore for an AI problem, Ada was most interesting to us, and it seemed probably most interesting in general, for several reasons.

One reason was, of course, timeliness. The push to use Ada, or to say why you weren't going to, was underway.

It seemed also especially important to include in the evaluation of Ada some of the programming techniques of machine intelligence. When the design of Ada was worked out, several years earlier, AI programs were probably not so much considered in its design[13] Since that time, a few people in the field have been concerned with just this issue. Some of their work has been reported in the annual

[13] This is a conjecture. Among the many persons who were concerned with Ada in its stage of design and initial negotiation, some must have been aware of the sorts of requirements of programs that were heavy on search and dynamically generated, evanescent structures. But since AI applications were much more in exploratory stages at that earlier time, it seems unlikely that they would have been considered in the same arena as control programs, database programs, and other more standard applications.

AI and Ada conferences (or AIDA, to use the more memorable abbreviation) which were held the past two years. But in the mainstream of developmental work in artificial intelligence, judging from the publications and major conferences of the American Association for Artificial Intelligence, Ada has not been a language that has been used, and the question of its adoption was not one that people who were primarily concerned with the development of AI methodology were concerned with. This would be quite proper, since implementation details operate at a somewhat different level than problems of program logic or concept.[14] Since Ada was rapidly being adopted, it seemed especially important to see how well it suited the needs of AI programming. There was even a note of urgency. A programming system not specifically designed for heuristic programming gave indications of being adopted, largely because of external pressures[15]. At the time, AI programs were just gaining a foothold in many organizations, but the foothold was tenuous in some cases. If Ada turned out to be widely adopted, but unsuitable for the kinds of procedures useful to machine intelligence applications, their benefits could not be realized.

Even if the adoption of Ada had not been an externally important issue, Ada would have been interesting to explore. It is a very rich programming system. And, it differs considerably from list-processing languages.

*Ada and LISP: a Note on their Locale.* LISP is today a highly evolved family of programming languages, with direct lineage to the language formulated by John McCarthy over two decades ago. Ada, as

[14] The issues are not entirely separate, since a programming system can both set constraints on the realization of techniques and also make it easier to use certain kinds.

[15] Which might be merited in the main. The overall suitability of Ada was a larger question than we were prepared to address.

a living language, so to speak, is in its early stages[16]. As an evolutionary product, most of its variation and selection has been on the conceptual level, though there are indications of evolutionary variation in its implementations over the past couple of years. Conceptually, Ada might be said to have drawn on the forms of previously existing computer languages, especially those which carefully controlled the definitions of variables and the transfer and binding of information. With its concern for portability and communication among routines, and its intentions of comprehensive usefulness (the evolutionary equivalent of territorial dominance, one is tempted to theorize), Ada seems to emerge as something of a super-species, or at least as a candidate for such an ecological position. By contrast, LISP dialects, despite a certain level of general capability, if not a certain aggrandizement of form and posture over the past decade, have remained ecologically fairly specialized. They are very much "niche" languages, though the size of the niches has increased lately with the rapid growth of interest in artificial intelligence and other symbolic processing.[17]

*Ada and LISP: some Language Features.* The details of each language are voluminous. There are some immediately contrastive features, though.

Figure 3 lists instructions for a simple kind of computation, first in generic LISP, then in Ada. The Ada code is realized as an Ada function; the LISP code is a LISP function. Each version assumes that certain terms have been defined and given values, and that niceties to insure program acceptance have been observed.

---

[16] Persons who have been involved in the lengthy history of formulating, revising, and supervising early implementations of Ada might disagree.
[17] There are signs that the expanded arena of supercomputing with tightly interconnected processors will be another somewhat specialized niche for LISP variants.

Both examples use recursion.

Perhaps the most obvious difference between the two is that the LISP version looks like LISP, and the Ada version looks something like Pascal or various other languages that trace certain amounts of their syntax and notation to Algol. Neither are exactly "natural" ways to tell a computer to do something—no one talks like this—but each can be figured out. At a superficial level, the LISP function seems replete with parentheses. We are left wondering, in each case, about how recursion and other features are handled during actual computation.

Apart from these impressions, a more substantial difference in the languages is indicated by this example, that is, the emphasis on control of keywords and variables in Ada. There are reserved terms in LISP, even if they are not conventionally given in boldface, but it is a much more easily extensible language. Ada is more the cousin of Pascal or Modula-2 in being "strongly typed", that is in specifying or carefully restricting the definition of kinds and conventions of elementary data units. Interestingly, in the treatment of typing, numbers receive more attention than other conceivable data units, belying the traditional, math-procedural concerns of these languages. Along with the typing, Ada constrains very carefully, if not severely, what can be passed as information to a routine. Some of the limitations of reference and binding are likely to be experienced as frustrations by persons used to programming in list-processing languages. They are used to handling problems that arise in working with the dynamically generated structures of symbolic computation by layering or nesting references to whatever structures may have come about. Such references are a bit tighter in Ada, but also probably less prone to unexpected errors.

Figure 4 shows Ada code for pushing an item onto a stack and popping it up. This kind of action is familiar in many kinds of programming, including assembly language routines. Pushing down and popping up are frequently used list-processing operations.

In this example, the two stack-manipulation operations are contained in an Ada "package" along with some definitional material. The package concept is not limited to Ada, but is one of its distinctive facilities. In a package, one can gather together, or "wrap", related computational entities such as constants, types, or procedures. The package is a convenient place to place the code. It can be found there if it needs to be revised or expanded. A subtler, related benefit is that within the package, referencing is simpler, so that less overhead of cross-referencing is involved in using the items wrapped within.

This push/pop package works on data items of type "integer". Some kind of floating point numeric representation, or character representations, would need additional code. However, Ada does allow terms for such operations to be "loaded" so that they could be applicable to more than one type, assuming that the proper code were available and had been suitably referenced within the system.

Davis chose to program the integer push/pop routines as taking place in a vector array. An alternative, rather more bulky approach would have been to try to reimplement LISP-like data units in Ada. The latter course would have provided needless overhead for most situations. However, using the built-in array addressing of Ada (or Pascal, or Modula-2, or whatever) does mean that the array must be large enough to hold the data items that will be needed, and not so large that too much memory relative will be taken relative to other

needs of the program.

This choice brings up another matter involved in doing list-processing in languages that provide handier facilities for other kinds of operations or memory allocation. Frequently, it is the best course of action to adapt facilities of the language one is working with, rather than to try to rewrite or comprehensively simulate the source language in the target language. Choices of just how to use the facilities of the resident language may have to be made after surveying just what is needed to program the problem at hand.

Finally, in connection with Figure 4, note the code, here unamplified, to raise error conditions when an attempt is made to exceed stack depth or to pop an item from an empty stack. Though it is not apparent from the example, Ada's error handling facilities are generally thought to be very convenient in comparison to those of other contemporary languages.

The extensive features of Ada are discussed in a number of publications. Sources that we have found especially accessible are the texts by G. Booch; various works written by N. Gehani; and I. Sommerville's book on *Software Engineering with Ada,* to name just a few. Though some of the material was written fairly early in the short life history of Ada, comments in various articles contained in *Comparing and Assessing Programming Languages: Ada, C, Pascal,* edited by A. Feuer and N. Gehani (Prentice-Hall, 1984), are clear and still relevant.

So far as discussions of LISP, the features of the language are described clearly for programmers in a number of textbooks, often containing "AI" in the title (texts by Winston, Brooks, Anderson et

al., and Tanimoto come to mind). More technical discussions of strategies of implementation, and issues of how variables are accessed and transferred are contained in comments in G. Steele's basic reference on Common LISP, and in Abelson and Sussman's *Structure and Interpretation of Computer Programs.*

Direct comparisons of LISP, or other list-processing languages, with Ada, or for that matter other languages such as Pascal or Modula-2, are not so easy to come by, though the discussions in the last AIDA conferences are relevant.

Another example of Ada code for a list-processing job is given in Figure 5. The two parts of Figure 5 show how the LISP function, *maplist*, could be implemented in Ada. This example is taken from Gehani's 1983 book on advanced programming in Ada.[18]

*Maplist* applies an operation (function) to each of the members of a list that is specified. This is the kind of thing that one frequently needs to do in symbolic programming. It is convenient not to have to specify anything about the list, such as its number of members, or the kind of members that are in the list.

The Ada code is presented for a "generic" routine, one that has meaning in broad contexts. Gehani only sketches the actual Ada program for this function. In practice a number of details would have to be taken care of. Note that the definitional part of the routine is much lengthier than the "action code" or function body. A suggestion is given of some of the matters that would have to be accounted for to handle the kinds of lists (sequentially linked, with expandable items, etc.) that are the basis of LISP data structures.

[18] Also from Prentice-Hall.

When working with a dialect of LISP, what one takes as basic functions such as *maplist* or other "map" variants, or ones to traverse lists and match items, pair items, reverse lists, and so on, come to be relied on and thought of as simple operations, though their actual coding may be non-trivial and in some cases may be fairly subtle. This is true of the list manipulating part of Prolog dialects.

When faced with handling list processing in languages with Algol-like statement syntax and different kinds of management of information items, the usefulness of such functions becomes felt very definitely. One impulse is to reimplement them, necessarily together with a basic list representation, in the new language. Another approach is to implement just some of the more important ones. In either case, some accommodation will have been made to the facilities of the target language. Yet another approach is to try to rethink the problem as one whose solution has to take place in the second language, without reference to the way it would have been solved in a list processing programming system. If the target language is one in which it is difficult to do some things, such as run-time storage allocation and deallocation, problems at the conceptual level may result.

*Recasting the Scheduler into Ada.* It would have been a much tougher job to reprogram the Ford-Floyd Dypas scheduler more or less directly into Ada, than to work with the Davis Common LISP/Flavors version.

The original scheduler relied heavily on traditional LISP programming techniques and facilities. Though it is not unorderly in overall structure, it looks much less modular than the Flavors version, and at least to a programmer unfamiliar with its underlying

conceptualization, would seem to be far from the intense modularization demanded by Ada programming.

Thus, the fairly "procedularized" recasting, using certain of the object oriented facilities of Flavors, lent itself to translation into Ada, at least at the level of major program modularization. It was not difficult to sketch a fairly direct correspondence between the methods defined for the Flavors objects, and Ada procedures or generic functions.

It did appear,though, that problems might be encountered in storage allocation overhead, and possibly also in straining bounds of readily available memory. These are matters where language facility and system convention and coordination interact, so that knowledge just of Ada language specifications could not tell us exactly what would come about in doing large-scale searches in a program implemented for various systems running Ada.[19]

Figure 6 shows Ada code for setting up the major objects of the conceptualization of the scheduler used in the Flavors version. The Dypas_process was taken as the main object. (Treating individual schedulers as objects, as in the Flavors version, was not considered to be a good idea here. Instead, in a possible version that would use concurrent processing we would intend it to be handled more directly by the concurrent facilities built into Ada.)

Figure 7 lists the central procedure, FIND_CHILDREN, in Ada. In comparison with the LISP/Flavors version, shown in Figure 2, the Ada version is leaner. Its logic is revealed almost skeletally. The

[19] It is interesting that relative to a programming system and language that is meant to be machine independent to a large extent, we so quickly encountered a system problem in our planning.

work of computation is mostly handled by calls to routines. Some of the routines are unique to the scheduler, such as SCHEDULE_ME. Others reflect more basic functions such as CREATE_NODE or PUSH.

The modularization and readability of Ada is seen very nicely here. By comparison, the parallel Common LISP procedure, even in the organizing context of some kind of object-oriented structuring, seems to have more little details to take care of.

Not visible in this central procedure are the complexities of specification and reference of the related declarations and routines.

*About Ada Objects.* Ada is sometimes referred to as a language that is based on computational "objects". Booch, for example, in both editions of his text, emphasizes the importance of Ada objects.

This terminology may be confusing to someone not familiar with the exact facilities of the language, since the term "object-oriented programming" or "object-oriented programming system" (OOPS, for short) is probably more frequently used today to refer to facilities such as those in Smalltalk-80[20], or Flavors extensions for LISP, and several more recent languages.

The usage is closer to the terminology of "first-class objects" or "second-class objects" in discussions of symbolic programming languages, which have to do with matters of reference within a program.

Both Ada objects and OOPS objects involve "information-hiding", as do the procedures of Modula-2 and other languages when internal

[20] The language and programming system developed by Xerox Palo Alto Research Center, and now available for various machines.

terminology is not meaningful outside of procedural boundaries.

The inheritance of properties by instantiated objects is not easy to manage in Ada, though it is a common and useful feature of languages such as Smalltalk. The Generic definition facilities for Types in Ada do not allow such referencing. Thus, hierarchies of objects are not possible (unless simulated somehow), much less multiple inheritance, where an object can draw characteristics from more than one branch.

For this reason, translation from Flavors into Ada may run into constraints fairly quickly, if many features of object-oriented programming were included in the original.

*Problems of Implementation.* In redoing the Flavors version of the scheduler using Ada, we had two kinds of problems. The first were basically *matters of language differences,* such as different ways of handling objects. The second, actually more serious difficulties were experienced in areas peripheral to the language itself. Of these, *systems-related problems* were very important. These often interacted with what might be termed *mundane problems* of working with specific language systems and hardware.

Some of these problems may have been unique to this particular project, though we suspect that the sorts of problems that we encountered were not.

Our situation in starting to work with the Ada scheduler may also have been not exactly unique. The person doing the bulk of the programming was not experienced in using Ada for any kind of application. He had a sort of textbook familiarity with it, and shared programmer lore, but had not programmed anything more than

a few exercises using the language. However, he would qualify both as "an experienced programmer" and as "an experienced AI programmer". Besides LISP dialects, he had programmed in Pascal and in C, so was familiar with the sorts of syntax and referencing in those languages. Besides these, he was familiar with several AI programming "shells". In other words, he was both experienced in programming, and used to some variety of programming systems.[21]

Thus, the experience of recasting the scheduler into Ada could be taken as a kind of case study in language adoption.

Relative to learning to use Ada *as a language* the following comments can be made from this particular case:

- Textbook material was easily available, and generally clear, though some of the more available and better known texts talked *about* the language somewhat more than getting down to examples that were worked out in thorough detail.[22] Such a level of discussion was not a particular problem for someone experienced in programming and knowledgeable about computer systems, but might be puzzling to persons less experienced. (The level of discussion may have been a function of the relative novelty of the language, and possibly also was traceable to the scarce availability of validated compilers for Ada at the time some of the textbooks were written. Since that time, we have noticed somewhat more detailed and less discursive material available.)
- We expected Ada not to be so restricted in its referencing as it was. Possibly because the problem being worked with needed somewhat flexible referencing, we found that it was easy to

21 By contrast, other accomplished programmers specialize in just one language.

22 It will not surprise persons used to such material that some of the examples in the texts did not work.

assume that such was available.

- As the conventions of the language became more familiar, the value of Ada's modularity for clear communication and structuring of program logic was very pleasant to discover.
- The facilities of the language for error handling were good to have in an explicit form.
- Though primarily used in planning developments of the Ada version, the facilities for concurrency seemed adequate.
- Generally, relative to ease of learning, Ada had extensive facilities, each replete with particular conventions and restrictions, so the information load was fairly heavy.

So far as learning Ada, features of the language itself were not nearly so important as the computational facilities associated with one or another implementation of the language. Ironically, Ada is a language whose realization in compiler form has been a matter of explicit early discussion and subsequent attention by the Department of Defense, its genitor. Compilers must be validated, to check that the numerous specifications are met. Along with the concern for validation is the well-known insistence on multiple-machine usage. When taken together with the care in cross-referencing routines and information items, and the safeguards for internal consistency that is also part of the language design, a large amount of compilation time seems to be taken up with checking the code for such matters. Thus, to compile even short procedures frequently involved substantial turn-around.[23]

Turnaround time is turnaround time, and can be adjusted to. When taken together with sparse documentation and diagnostics, though,

23 The suspicion was also raised in some cases that compilation time tended to be lengthy because of the complexity of the language; or perhaps even because some of the compilers had not been well tuned.

the delays for compilation seemed especially frustrating. In some cases, no information was given regarding the reasons for an unsuccessful compilation, and a search of the documentation did not readily provide clues. We were left wondering

- if the cause of the problem were an error or faulty assumption in the program (likely),
- if so, what the error or inconsistency was,
- if in fact the problem might have resulted from not doing what the editor or compilation front-end expected rather than being a language error (more likely that we would have thought at the outset),
- if the problem were in the procedure itself or in the linkage checking,
- if we had run into a compiler flaw (not as likely as we thought at times, but with some of the earlier versions of compilers which we worked with, a possibility that could not be dismissed).

Problems with compilers, editors, linkage checkers, and so on were encountered both with the Symbolics (which in general has a superior user interface and system facilities) and with IBM Personal Computers and their clones (which are not known for either). Ada on the Symbolics would of course be written on top of the basic list-processing structure of the machine and its system software. We felt that this made it not the ideal machine to benchmark Ada on because of this indirectness of implementation. Even so, because of the Symbolics' system facilities, we had thought that its Ada would be easier to use. The main problem that we had with it (which may also have been to some extent a function of our place on the learning curve) was inadequate diagnostics. We are curious, however, as to the future of an Ada implementation on this machine, since the Symbolics systems software has already developed smooth-running

multi-tasking, and has a number of other excellent software and interface features to recommend it.

Generally speaking, Vaxes are not as convenient to use in preparing and editing programs (though they have been steps ahead of other minicomputers and various mainframes). We certainly had no quarrel with Vax Ada itself, which must be something of a landmark. Unfortunately, because of mundane reasons, such as chronic hardware problems on one machine, and uneven access then memory inadequacy on the other Vax, for much of the time we were unable to use one of these machines.

Thus, our early plan of developing the Ada version both for the Symbolics and for the Vax was frustrated, given the time for this part of the project.

Even though there was no chance of programming the complete scheduler for a PC, some time was spent in working with smaller routines using two kinds of Ada on IBM and similar personal computers. Both of the Adas were in "early" versions[24]. Both were fairly frustrating to use. (We have since used a revised version of one of the PC Adas, and do not believe we would have had the same problems with it.)

*Status of the Implementation.* The complete scheduler had not been implemented into fully functioning Ada by the close of the project period. We felt that we had learned a considerable amount, but also felt a sense of frustration that so many "peripheral" matters, such as hardware access, capability of editing software, and especially Ada-related system software (either compilers or

[24] A relative term. It appeared that an "early" version of an Ada implementation was one that did not work very well.

closely related system features), had been sources of problems.

These are some interim conclusions regarding the Ada programming experience that we felt could be offered at the end of the project period:

- Ada is a large and capable language,
- Partly because of language complexity, and partly because of system and compiler problems, using Ada for a reasonably complex program involved various frustrations and delays.
- It seems to take somewhat longer to implement and integrate a program of moderate complexity than would be the case with a language of more traditional scope of integration.
- Ada routines are easy to read, in general.
- The transition to using Ada confidently, when other languages have been used, may take longer and use more resources than one anticipates.
- Translating AI applications such as this scheduler into Ada are easier if they are "proceduralized".
- We remained suspicious of the ability of computer systems operating with Ada as their language/programming systems to efficiently use memory in the voluminous and dynamic ways that often seem necessary for AI problems.

the collect-time function in LISP

```
(defun  collect-time (process)
   (if  (eq T (process-before process))   ; if this is true
     (process-runtime process)            ; then do this
     (+ (process-runtime process)         ; else do this
        (collect-time (process-before process)))))
```

the collect_time function in Ada (recursive version)

```
function COLLECT_TIME(P:  in out PROCESS)  is
    begin
        if  T  =  PROCESS.BEFORE
            then return PROCESS.RUNTIME;
        elseif return  PROCESS.RUNTIME +
                          COLLECT_TIME(PROCESS.BEFORE);
end  COLLECT_TIME;
```

Figure 3.

Illustration of a function programmed in generic LISP, and in Ada.

```
package body SI is
  MAX: constant:= 1000;
   A: array(1..TOP) of INTEGER;
   TOP: INTEGER range 0..MAX;

    procedure PUSH(X:INTEGER) is
    begin
        if TOP = MAX then
            raise ERROR;
        end if;
        TOP:=TOP+1;
        A(TOP):=X;
    end PUSH;

    function POP return INTEGER is
    begin
        if TOP = 0 then
            raise ERROR;
        end if;
        TOP:=TOP-1;
        return S(TOP+1);
    end POP;

begin
  TOP:=0;
end STACK;
```

Figure 4.

Ada Package for Push and Pop Operations (Implemented for an Array of Integers)

MAPLIST

```
generic
   type LIST is private;
   type ELEM is private;

   with function F(E: in ELEM) return ELEM;
            -- This is the function being extended.

      --The following operations for values of the type LIST must
      --be supplied or be available in the context of the instantiation
      --since they are necessary for extending the function F.

         with function HEAD(L: in LIST) return ELEM is <>;
                  --first element of the list L
         with function ADD_LAST(L: in out LIST; X: in ELEM) is <>;
         with function TAIL(L: in LIST) return LIST is <>;
                  --the list that is the same as L
                  --but minus the first element.

         with function EMPTY(L: in LIST) return BOOLEAN is <>;
         with function CREATE return LIST is <>;
                  --an empty list

function MAPLIST(L: in LIST) return LIST;
```

Figure 5. Sketch of Code for LISP Maplist in Ada (from Gehani, 1983)

MAPLIST body

```
function MAPLIST(L: in LIST) return LIST is
  A:= L;
   RESULT:= CREATE;      --the empty list.
   while not EMPTY(A) loop
      ADD_LAST(RESULT, F(HEAD(A)));
      A:= TAIL(A);
   end loop;
   return RESULT;
end MAPLIST;
```

The generic function MAPLIST may be instantiated for a list of integers and the function SQUARE as

```
function SQUARE_INTEGER_LIST is
            new MAPLIST(INTEGER_LIST, INTEGER, SQUARE);
```

Figure 5 (contd) - Sketch of Code for LISP Maplist in Ada (from Gehani, 1983)

declaring access type DYPAS_PROCESS

```
type PROCESS;

type DYPAS_PROCESS is access PROCESS;

type PROCESS is
      record
        MAX_POWER: INTEGER;
              BEGIN: INTEGER;
                 END: INTEGER;
           CONDITION: STRING(1.. 30);
              BEFORE: DYPAS_PROCESS;
               AFTER: DYPAS_PROCESS;
      end  record;
```

then declaring the objects is as follows

```
      ECLSS: DYPAS_PROCESS;
            -- object ECLSS created with slots set to null.

      IEFF: DYPAS_PROCESS (END => 100);
            -- object IEFF created with slot END set to integer 100.
```

Figure 6. Setting Up Main Objects (DYPAS_PROCESSes) for Ada Version of Scheduler.

```
procedure FIND_CHILDREN is
   CLOCK_BEGIN: INTEGER;
   SCHEDULE_ME, PROCESS_BEFORE: DYPAS_PROCESS;
            --The global variables SCHEDULED and SCHEDULE_US
            --are declare in DYPAS package specification.
      begin
        CLOCK_BEGIN:= TIME_IO.GET_UNIVERSAL_TIME;

      MAIN_LOOP:
               for SCHEDULE_ME in SCHEDULE_US loop
                    TEXT_IO.new_line;
      SUB_LOOP:
               for PROCESS_BEFORE in THOSE_SCHEDULED loop
                    BEGIN_TIME:= COLLECT_TIME(PROCESS_BEFORE);
                    REQUIRED_POWER:= SCHEDULE_ME.MAX_POWER;
                     AVAILABLE_POWER:= MAX_WATTAGE
                                          - FIND_MAX_POWER (SCHEDULE_ME BEGIN_TIME);
                    if AVAILABLE_POWER >= REQUIRED_POWER then
                        SCHEDULE_ME.BEGIN:= BEGIN_TIME;
                        SCHEDULE_ME.END:= BEGIN_TIME + SCHEDULE_ME.RUNTIME;
                        CREATE_NODE(SCHEDULE_MEPROCESS_BEFORE);
                        BUILD_HASH(REQUIRED_POWER BEGIN_TIMESCHEDULE_ME.END);
                        PUSH(SCHEDULE_ME SCHEDULED);
                        PUSH(BEGIN_TIME START_TIMES);
                        PUSH(SCHEDULE_ME.END START_TIMES);
                        SCHEDULE_ME.CONDITION = "Scheduled";
                     end  if

      end loop SUB_LOOP;
      end loop MAIN_LOOP;

end FIND_CHILDREN;
```

Figure 7. The Central FIND_CHILDREN Procedure in Ada.

## IV. Issues and Conclusions

*What was Learned.* Probably the most interesting finding to emerge from the various attempts to recast the original LISP scheduler into other programming contexts, was the *efficiency and speed of the object-oriented version.* The improvement in performance was substantial, and though it seemed to result from several factors (program organization, speedy addressing schemes, and so on), the programs were comparable in some important ways. Apart from the difficulties of comparing any two complex programs, the original scheduler and the Flavors version ran on the same computing machine, and both were within the LISP programming milieu. Thus, questions of recasting list-processing programs to an essentially non-list-processing environment, as in the later exploration of Ada, were irrelevant here.

The improvement in performance was especially interesting since object-oriented techniques are commonly offered as aids to clearer and more communicable conceptualization of a problem, but are not usually suggested as leading to markedly improved efficiency. We found here that at least in this case, the "objectification" at the very least did not detract from performance and may well have contributed to the improvement.

In relation to the above, we also learned more clearly the relation between ordinary object-oriented techniques, which in the literature are often discussed as just radical cases of "information hiding" in procedures, and the tradition of *very modular procedural organization* of programs—which of course is a strong cultural feature found in the design of Ada[25].

25 And other current languages in that tradition such as Pascal, Modula-2, and others.

It should be noted, though, that *very tight modularization is not necessarily a programming ideal for AI programs,* especially when it implies, as it often does, extensive constraints on symbolic reference, ability to pass on information among procedures, and kinds of data structures that are convenient to use within a particular system. We ran into these constraints almost immediately in setting up the recasting of the scheduler into Ada, in the restrictions on generic types. (We think we would have had even more problems with a more narrowly conceived language such as Modula-2, despite its conceptual neatness.)

This is not to say that we have anything against clarity of organization of programs[26]. Both clarity and effectiveness can come from other organizational principles, and other language/system bases. For example, the kind of organization into inference routines and modifiable knowledge *corpora* that has emerged in some machine intelligence work, is relatively clear and certainly useful, but is different in approach from what would be recommended by the dogmas of the Wirth school of thought.

*Intelligence in Machines.* A curious phenomenon of work in AI is the *disappearance of intelligence* when a process is examined closely, or comes to be understood.[27] The case of now you see it, now you don't 'intelligence' in the routines was something that we experienced in regard to each of the programs that we worked with.

---

[26] Regretably often taken to be the sole property of "structured" algorithmic languages such as Pascal.

[27] Though this has often been pointed out for the case of machines, it also seems to apply at times when human intelligence is carefully scrutinized, and what seems to have been whim or wisdom seems to be more determined by simple causes than not.

What seemed to be more significant was the *embeddedness of intelligence* into the routines. Even in the original scheduler, much is arranged and specified rather than left for inference. The Flavors version of the scheduler in some ways went a little farther in incorporating actual heuristics about the situation of scheduling space tasks into code.

What this implies is that the critical issue is not whether a program is intelligent because it does some reasoning or not, or is smarter or less smart in an absolute situational sense, but the *modifiability of the program to meet new situations* together with the *situational adequately of response.* This makes the matter of program "smartness" much more a matter of both interface and external connectedness to a world of events and meanings, than necessary features of internal procedure. So *flexibility and aptness* become important more than particular formal features that may have been seized upon as indicators of program virtue.

The now you see it phenomenon also is related to issues of programming language features and capabilities. Assume for the moment that only one computer arrangement was being used. Various high level languages, in which programmers would express what needs to be done to solve real problems through computation, must be transformed into machine instructions. The machine will do what it can do. But, it turns out that it is much easier to do some things using some higher-level languages. It is virtually impossible to do some things with some languages. For example, it used to be next to impossible to do any substantial amount of list processing with Fortran, and it is still difficult to do so without special routines in more specific languages.

So the reality of the programming language, with its domain of

meanings and operational possibilities, is very real indeed.

If machines are different from one another in certain ways, the discussion becomes more complex. For example, in this project the difference between the underlying structure of the Symbolics, which has special hardware facilities to help list-processing addressing and even some higher order operations, is different enough from the underlying structure of a Vax, or a PC, that one would want to examine not just language differences when exploring how programs should best be written for these systems.

From the point of view of the humans who use the computers, either the possibilities and constraints of the software, or the possibilities and constraints of the hardware, could be the source of easy problem solution, or of frustration in trying to solve a computational problem. In this project, for example, the major problem was what software would allow or facilitate. It turned out that even on the same machine, it was relatively easy to solve certain problems with some software, and a real challenge to do so with another programming language, even one that is advertised as being diverse and capable.

*Possible Future Investigation.* Further exploration of Ada would seem to be of considerable interest. Both the more limited approach to the scheduling problem, and ones that were only mentioned in passing here, such as multitasking versions, would be interesting to gain more experience with.

Of languages that are in concept more limited than Ada, Modula-2 and C seem especially interesting. C is widely used and has a good reputation for implementing efficient programming solutions. Modula-2 is becoming increasingly appreciated by people who write

serious software, including applications for machine control and coordination. Both have certain restrictions of conceptualization, though.

Because of the results for the object-oriented version, it might also be worthwhile to pursue that kind of technique in future studies.

In addition, various projects have been underway to provide either somewhat general list-processing facilities, or more specialized "expert system" systems written in some of the more classically structured languages. These could be evaluated using problems such as this scheduler.

*Summary.* A realistic application involving the concepts and certain programming techniques of artificial intelligence, was examined in several versions. The original, in classic LISP, contrasted with an object-oriented version. Work on a recasting into Ada, at base a differently structured language, involved a number of both practical and conceptual problems.

# APPENDICES

Appendix

A. Instructions for Common LISP/Flavors DYPAS Scheduler

B. Common LISP/Flavors Scheduler: Listing

C. Rule-Search (RETE) Algorithm Adaptation

D. Routines from Ada Version of the Scheduler

E. Floyd and Ford's Description of their Scheduler

*Appendix A.*

# Instructions for Common LISP/Flavors DYPAS Scheduler

These are the instructions supplied by
S. Davis for running the Common LISP/Flavors
Version of the Scheduler
on the Symbolics computing machine.

To Set Up the Common LISP/Flavors Version of DYPAS -

Begin by restoring the distribution from tape. The best way to do this is by entering the Command Processor command;

Edit File SYS:SITE;DYPAS.TRANSLATIONS

and typing the form

```
(fs:set-logical-pathname-host "DYPAS"
     :translations '(("** ;" ">dypas>**>")))
```

The physical directories do not have to exist. Now, evaluate and save this file, then go back to the Listener.

From the Command Processor do a Restore Distribution, specify the Cart Unit to use and begin restoring.

To load DYPAS into working memory do;

Load System DYPAS

then type <SELECT>-Square.

Running DYPAS is equally simple. Six of the eight commands displayed are mouse sensitive. The two that are not are Describe Process and Remove Process. They are available from the Keyboard. Describe Process is also available by mousing on a process name after selecting Show Schedule. The top four commands should be selected in sequence initially, and then used logically afterwards. A short introduction of these commands follow.

INITIALIZE sets-up program definitions and clears the Frame Base. It is always correct to select this command first. Initialize prepares DYPAS working memory for new processes.

SELECT PROCESSES allows the user to individually select the processes he wants to schedule. When selected it brings up other windows that can be logically followed.

SCHEDULE begins the scheduling process, but does it in the background. You can then actively Describe Processes, Create Processes or Show Schedule to give a dynamic view of what is being scheduled.

SHOW SCHEDULE does its best description after the scheduling is done. You can Describe Processes by mousing on the tree structure for any process name that is sensitive.

Try removing a few processes after scheduling and then select Schedule again. See what kind of schedule differences exist. When you want to begin again select Initialize and start fresh.

NOTE: To allow DYPAS to run quickly, keep the mouse pointer out of the Message window. Some of the processes are mouse sensitive and it will slow down the scheduling when mouse and process conflict.

*Appendix B.*

# Common LISP/Flavors Scheduler: Listing

```
;;; -*- Syntax: Common-lisp; Package: USER; Base: 10; Mode: LISP; Default-character-style: (:FIX :ROMAN :LARGE) -*-
;;; Stephen W. Davis
;;; Johnson Research Center 205-895-6257


(defmethod (Make-Instance DYPAS-PROCESS) (&rest ignore)

  "A make-instance method for Dypas-Process, that is used by
   DefDypas constructor. Make-instance is never called directly."

;;; Check and see if process is already in correct program.
  (unless (member NAME
                  (map 'list
                       #'(lambda (x)
                           (dypas-process-name  x))
                       (dypas-presentations *program*)))
    (let ((stream (get-pane 'messages)))
      (setf PRESENTATION
            (dw:with-output-as-presentation
              (:object NAME
               :type 'dypas-process)
              (format nil "~A" NAME)))
      (set '*last-process* NAME)
      (set *last-process* SELF)
      (push SELF (dypas-presentations *program*))
      (setf OTHER (cons SELF (+ RUNTIME MAX-POWER))) ;;; weight
      (cond ((neq NAME 'root-process)
             (fresh-line stream)
             (dw:with-output-as-presentation
               (:stream stream :type 'dypas-process :object NAME
                        :allow-sensitive-inferiors t)
               (format stream "~& ~S added as a Process." NAME)))
            ((eq NAME 'root-process)
             (setf (dypas-presentations *program*)
                   (remove self (dypas-presentations *program*))))))))


(defmethod (Describe-Process dypas-process) (pane &Key (verbose t))

  (format pane  "~&~%Priority    : ~D~%~
                     Sponsor     : ~S~%~
                     Max-power   : ~D~%~
                     Runtime     : ~D"
          PRIORITY SPONSOR MAX-POWER RUNTIME)

  (format pane "~%Crew~7@T: ~:[not needed~;needed~]" CREW)
```

```
    (format pane "~%Continuous : ~:[no~;yes~]" CONTINUOUS)

    (when verbose
      (format pane "~%Time Frame : ~(~S~)~%~
                    Status     : ~(~S~)~%~
                    Condition  : ~(~S~)~%~
                    Before     : ~S~%~
                    After      : ~S~%"
              TIME-FRAME STATUS CONDITION
              (if (or (eq BEFORE nil)
                      (eq BEFORE t))
                  'Root
                  (dypas-process-name BEFORE))
              (loop for object in AFTER
                    collect (dypas-process-name object))))

    (format pane "~&Begins at  : ~\\time-interval\\~%~
                  Ends at    : ~\\time-interval\\~%"
            BEGIN END))


(defmethod (Remove-Process dypas-process) (&rest ignore)
      (setf (dypas-presentations *program*)
        (remove SELF (dypas-presentations *program*)))
  (let ((stream (get-pane 'messages)))
    (format stream "~&~%~S removed from Frame Base~%" NAME)))


(defmethod (Create-node Dypas-process) (process-before process-after)
  "This method should be called when you or Jo want to parse the tree"
  (if (eq process-before T)
      (setf BEFORE T) ;signifies a root process.
      (newpush SELF (dypas-process-after process-before))
      (setf BEFORE (eval process-before))
      )
  (if (eq process-after 'ignore)
      ()
      (setf (dypas-process-before process-after) SELF)
      (newpush (eval process-after) AFTER)))


(defmethod (Initlz-node dypas-process) ()
  (setf AFTER nil)
  (setf BEFORE nil))
```

```
(defmethod (Collect-time dypas-process) ()
   (if (eq T BEFORE)
        RUNTIME
         (+ RUNTIME (collect-time BEFORE))))

#|
    ****************************************
    *                                      *
    *  Framework defined flavor methods.   *
    *                                      *
    ****************************************
|#


(defmethod (Initlz dypas) (&key (all t))
  (setf STATE '(+)) (setf ROOT nil)
  (clrhash HASH-TABLE)
  (make-root-process self)
  (setf START-TIMES '(0)) (setf SCHEDULED '())
  (cond (all
         (loop for object in PRESENTATIONS
                   do (progn (eval '(location-makunbound
                                      (locf ,(dypas-process-name object))))))
         (setf PRESENTATIONS nil))))



(defmethod (Build dypas) (mag begin end)
  (setf STATE (append
                STATE
                (read-from-string
                  (format nil "((* ~d (- (U (- CT ~d))
                                         (U (- CT ~d)))))"
                          mag begin end)))))


(defmethod (Initlz-all-nodes dypas)()
  (loop for object in PRESENTATIONS
        do (setf (dypas-process-after object) nil)
           (setf (dypas-process-before object) nil)
           (setf (dypas-process-end object) nil)
           (setf (dypas-process-begin object) nil)
           (setf (dypas-process-condition object) 'UNCOMPLETED))
 T)
```

```
(defmethod (Find-root dypas) ()
  (loop with processes-sorted =
        (map 'list #'car
             (sort (loop for object in PRESENTATIONS
                         collect (dypas-process-other object))
                   #'> :key #'cdr))
        with available-power = *maximum-wattage*
        with pane = (get-pane 'messages)

        for object in processes-sorted   ;;; MAIN LOOP

        for runpower = (dypas-process-max-power object)

        when (≥ available-power runpower)
          do (setf available-power (- available-power runpower))
          and do (push object ROOT)
          and do (push object SCHEDULED)
          and do (build-hash SELF runpower 0
                             (dypas-process-runtime object))
          and do (create-node object T 'Ignore)
          and do (setf (dypas-process-begin object) 0)
          and do (setf (dypas-process-end object)
                       (dypas-process-runtime object))
          and do (newpush (dypas-process-runtime object) START-TIMES)
          and do (setf (dypas-process-condition object) 'scheduled)
          and do (let ((pro (dypas-process-name object)))
                   (dw:with-output-as-presentation  (:stream pane
                                                     :type 'dypas-process
                                                     :object  pro)
                     (format pane "~&~S scheduled with ~D left.~%"
                             pro available-power)))))


(defmethod (Find-children dypas) (&key (stream *standard-output*)
                                       (verbose nil))
  "Ok hold your breath, this is not going to be pretty."

  (let ((clock-begin (get-universal-time))
        (schedule-us (map 'list #'car
                          (sort (loop for object in PRESENTATIONS
                                      when (not (member object ROOT))
                                        collect (dypas-process-other object))
                                #'> :key #'cdr))))
    (loop for schedule-me in schedule-us  ;;; MAIN LOOP
          do (if verbose
                 (format stream "~&~%~'bCScheduling ~S~D"
```

```
                    (dypas-process-name schedule-me)))
            (let* ((those-scheduled (map 'list #'car
                                         (sort (loop for object in SCHEDULED
                                                     collect (cons object
                                                                   (collect-time object)))
                                               #'< :key #'cdr))))
              (fresh-line stream)
              (loop for process-before in those-scheduled ;;; SUB LOOP
                    do
                (if verbose
                    (format stream "~&  Behind  ~S"
                            (dypas-process-name process-before)))
                (let* ((begin-time (collect-time process-before))
                       (required-power (dypas-process-max-power schedule-me))
                       (available-power (- *maximum-wattage*
                                           (find-maximum-power-in-interval
                                             SELF schedule-me begin-time))))
                  (if verbose
                      (format stream "~%Available-Power  = ~D~
                                      ~% Required-Power  = ~D~
                                      ~%     Begin-Time  = ~D"
                              Available-Power Required-Power  Begin-Time))
                  (cond ((≥ available-power required-power)
                         (setf (dypas-process-begin schedule-me) begin-time)
                         (setf (dypas-process-end schedule-me)
                               (+ begin-time (dypas-process-runtime schedule-me)))
                         (create-node schedule-me process-before 'ignore)
                         (build-hash SELF required-power begin-time
                                     (+ begin-time
                                        (dypas-process-runtime schedule-me)))
                         (push schedule-me SCHEDULED)
                         (newpush begin-time START-TIMES)
                         (newpush (dypas-process-end schedule-me) START-TIMES)
                         (setf (dypas-process-condition schedule-me)
                               'scheduled)
                         (return
                           (if verbose
                               (presentation-print *program*
                                                   (dypas-process-name schedule-me) stream))))
                        (verbose (format stream "~%  Look-beyond ~S"
                                         (dypas-process-name process-before)))))))))
      (format stream "~&~%Scheduling took ~\\time-interval\\"
              (time-difference (get-universal-time) clock-begin)))
    (let* ((sum-usage-area (loop for object in PRESENTATIONS
                                 summing (* (dypas-process-max-power object)
                                            (dypas-process-runtime object))))
```

```
        (longest-runtime (eval '(max ,@START-TIMES)))
        (validation (/ sum-usage-area (* longest-runtime *maximum-wattage*))))
    (format stream "~%Longest process runtime ~\\time-interval\\"
            longest-runtime)
    (format stream "~%Power to time ratio ~2,5F" validation)))


(defmethod (graphical-representation dypas)(longest-runtime)
  (let ((pane (get-pane 'description)))
    (send pane :clear-history t nil)
    (multiple-value-bind (width height) (send pane :size)
      (let ((power-factor (/ height *maximum-power))
            (time-factor  (/ width longest-runtime)))

        (do ((times (stable-sort START-TIMES #'<) (cdr times))
             (y1 0)
             (x0 (first times))
             (x1 (- width  (* time-factor (second times))))
             (y1 (- height (* power-factor (powerat self (first times))))))
            ((= (length times) 1))
          (graphics:draw-rectangle x0 y0 x1 y1
                                   :stream pane
                                   :filled t))))))

(defmethod (Presentation-print dypas) (name stream)
  (dw:with-output-as-presentation
    (:stream stream
    :type 'dypas-process
    :object name
    :allow-sensitive-inferiors nil)
  (format stream "~&~'bC ~S scheduled~3" name)))


(defmethod (Make-Root-process dypas) (&rest ignore)
  (Defdypas 'Root-process
                :pretty-name "Begin Processing"
                :priority 10
                :sponsor "Johnson Research Center"
                :max-power *maximum-wattage*
                :runtime 0
                :crew t
                :continuous t
                :other (cons *program* *program-frame*)
                :time-frame "Wait for Steve Davis"
                :status 'off
                :condition 'process-await)
```

```
  (setf (dypas-process-begin (eval 'root-process)) nil)
  (setf (dypas-process-end (eval 'root-process)) nil)
  (setf (dypas-process-after (eval 'root-process)) ROOT)
  (setf PRESENTATIONS (remove (eval 'root-process) PRESENTATIONS))
  (setf (dypas-process-before (eval 'root-process)) nil))


(defmethod (find-maximum-power-in-interval dypas) (schedule-me begin-time )
  (declare (special end search-start-times-list))
  (setq end (+ begin-time (dypas-process-runtime schedule-me)))
  (setq search-start-times-list (newremove end START-TIMES #'<))
  (push begin-time search-start-times-list)
  (setq search-start-times-list
        (newremove begin-time search-start-times-list #'>))

    (loop for time in search-start-times-list
        maximize (powerat *program* time ))) ;(eval `(power@ ,time ,SELF))))


(defmethod (Print-Power@ dypas) (time)
          (format (get-pane 'messages) "~%Time= ~D , Power@ = ~D"
                  time (powerat self time)))


(defmethod (Print-Names dypas) (schedule-us)
  (loop for object in schedule-us
        do (present (dypas-process-name object) 'dypas-process
                    :stream (dw:get-program-pane 'messages))))

(defmethod (Draw-Tree dypas) (root-process &key (stream *standard-output*)
                                                (describe nil))

  (labels ((component-processes (process)
             (let* ((pro (dypas-process-after (eval process)))
                    (pro-name
                      (cond ((listp pro)
                             (map 'list #'(lambda (x)
                                            (dypas-process-name  x))
                                  pro))
                            (t (dypas-process-name pro)))))

               pro-name)))

    (format-graph-from-root (dypas-process-name (eval root-process))
                            #'(lambda (process stream)
                                (if describe
                                    (describe-process (eval process) stream
```

```
                                                    :verbose nil)
                                          (present process 'dypas-process
                                                   :stream stream)))

                                    #'component-processes
                                    :stream stream
                                    :orientation :horizontal
                                    :border :oval
                                    :border  3
                                    :within-column-spacing 25
                                    :dont-draw-duplicates t)))


(defmethod (Powerat dypas) (ct)
  (let ((return-val 0))
    (maphash #'(lambda (time pulse-value)
                 (if (≤ time ct)
                     (setq return-val (+ return-val pulse-value))
                     ))
             HASH-TABLE)
    return-val))


(defmethod (Initlz-Hash dypas) ()

  (if (hash-table-p HASH-TABLE)
      (clrhash HASH-TABLE)
      (setf HASH-TABLE (make-hash-table :test '=))))


(defmethod (Build-Hash dypas) (mag begin end)
  (multiple-value-bind
    (value found) (gethash begin HASH-TABLE)
    (if found
        (setf (gethash begin HASH-TABLE) (+ value mag))
        (setf (gethash begin HASH-TABLE) mag)))
  (multiple-value-bind
    (value found) (gethash end HASH-TABLE)
    (if found
        (setf (gethash end HASH-TABLE) (- value mag))
        (setf (gethash end HASH-TABLE) (- 0 mag)))))
```

*Appendix C.*

# Rule-Search (RETE) Algorithm Adaptation

This is an adaptation by S. Davis of the
RETE algorithm of Charles Forgy
for a future version
of the Common LISP/Flavors scheduler

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: RETE; Base: 10.; Default-character-style: (:FIX :ROMAN
:LARGE); -*-
;;; Stephen W. Davis
;;; Johnson Research Center 205-895-6257

(defun rule-interpreter nil
  (let (choice)
    (do ((add-list database (cdr add-list))
         (conflict-set nil (append (enter-clause (car add-list)) conflict-set)))
        ((and (null add-list)(null conflict-set)) database)
      (cond ((null add-list)
             (setq choice (conflict-resolution conflict-set))
             (perform-action choice)
             (setq add-list (cdr choice))
             (setq conflict-set (remove-repeats conflict-set (second choice))))
            ))))

(defun conflict-resolution (conflict-set) (first conflict-set))


(defun perform-action (action)
  (let ((rule (first action))
        (pred (first (second action)))
        (actor (second (second action)))
        (lesser (third (second action))))
    (format t "~%Firing rule ~A --> ~A is ~:[a~;the~] ~A~@[ of ~A~]." rule actor lesser
pred lesser)
    (setq database (cons (second action) database))
    (values)))

(defun remove-repeats (conflict-set action)
  (do ((temp conflict-set (cdr temp))
       (result nil (cond ((equal (cadar temp) action) result)
                         (t (cons (car temp) result)))))
      ((null temp) result)))

(defun enter-clause (clause)
  (let* ((pred (car clause))
         (args (cdr clause)))
    (putprop pred (cons args (get pred 'bindings)) 'bindings)
    (append
      (mapcan #'(lambda (prod)(enter-first args prod)) (get pred 'first))
      (mapcan #'(lambda (prod)(enter-second args prod)) (get pred 'second))
      (mapcan #'(lambda (prod)(enter-third args prod)) (get pred 'third)))))

(defun enter-first (args prod)
  (mapcan #'(lambda (x)(match-and-execute x prod))
          (cross-product (list args)(clauses prod 'second)(clauses prod 'third))))

(defun clauses (prod position)
  (or (get (get prod position) 'bindings)
      (list nil)))

(defun cross-product (lis1 lis2 lis3)
  (mapCAN #'(lambda (x)
              (mapCAN #'(lambda (y)
                          (mapCAR #'(lambda (z)(list x y z)) lis3))
                      lis2))
          lis1))

(defun match-and-execute (args prod)
  (cond ((apply (get prod 'condition) args)
```

```
        (check-for-dups prod (apply (get prod 'action) args)))
       (t nil)))

(defun check-for-dups (prod action)
  (cond ((member* action database) nil)
        (t (list (list prod action)))))

(defun member* (target lis)
  (do ((temp lis (cdr temp)))
      ((null temp) nil)
    (cond ((equal (car temp) target)(return temp)))))

(defun enter-second (args prod)
  (mapcan #'(lambda (x)(match-and-execute x prod))
          (cross-product (clauses prod 'first)(list args)(clauses prod 'third))))

(defun enter-third (args prod)
  (mapcan #'(lambda (x)(match-and-execute x prod))
          (cross-product (clauses prod 'first)(clauses prod 'second)(list args))))

(defun traceon2 ()
  (trace enter-first enter-clause remove-repeats perform-action
         conflict-resolution rule-interpreter clauses enter-third enter-second
         member* check-for-dups match-and-execute cross-product putprop ))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: RETE; Base: 10.; Default-character-style: (:FIX :ROMAN
:LARGE); -*-
;;; Stephen W. Davis  home: 205-533-7308
;;; Johnson Research Center 205-895-6257

(defun encode-productions nil (setq initlist '()) (mapcar 'encode-a-production product
ions))

(defun encode-a-production (production)
  (let ((name (first production))
        (condition (second production))
        (action (fourth production)))
    (let ((pred1 (first (first condition)))
          (pred2 (first (second condition)))
          (pred3 (first (third condition)))
          (predact (first action)))
      (when  (notmember predact initlist)
        (setf (symbol-plist predact) nil)
        (push predact initlist))
      (setf (symbol-plist name) nil)
      (cond ((and pred1 (notmember pred1 initlist))
             (setf (symbol-plist pred1) nil)
             (push pred1 initlist)
             (cond ((and pred2 (notmember pred2 initlist))
                    (setf (symbol-plist pred2) nil)
                    (push pred2 initlist)
                    (cond ((and pred3 (notmember pred3 initlist))
                           (setf (symbol-plist pred3) nil)
                           (push pred3 initlist)))

                    )))))
      (encode-condition condition name)
      (putprop name (build-action (mapcar 'cdr condition) action) 'action)))

(defun encode-condition (conditions prodname)
  (let (first second third)
    (setq first (first conditions))
    (update prodname (first first) 'first)
    (cond ((cdr conditions)
           (setq second (second conditions))
           (update prodname (first second) 'second)))
    (cond ((cddr conditions)
           (setq third (third conditions))
           (update prodname (first third) 'third)))
    (putprop prodname (build-condition (cdr first) (cdr second) (cdr third))
             'condition)))

(defun update (prodname pred relation)
  (putprop prodname pred relation)
  (putprop pred (cons prodname (get pred relation)) relation))

(defun build-condition (vars1 vars2 vars3)
  `(lambda (first second third) ,(build-test vars1 vars2 vars3)))

(defun build-test (vars1 vars2 vars3)
  (let ((test (append (encode-pair vars1 vars2 'first 'second)
                      (encode-pair vars1 vars3 'first 'third)
                      (encode-pair vars2 vars3 'second 'third))))
    (cond ((null test) t)
          ((null (cdr test)) (first test))
          (t (cons 'and test)))))
```

```
;; ;ENCODE-PAIR (=X =Y) (=Z =X) FIRST SECOND
;;; ENCODE-PAIR ((EQ (FIRST FIRST) (SECOND SECOND))

(defun encode-pair (vars1 vars2 name1 name2)
  (and vars1 vars2 (cond ((eq (first vars1)(first vars2))
                          '((eq (first ,name1) (first ,name2))))
                         ((and (cdr vars1)(eq (second vars1)(first vars2)))
                          '((eq (second ,name1)(first ,name2))))
                         ((and (cdr vars2)(eq (first vars1)(second vars2)))
                          '((eq (first ,name1)(second ,name2))))
)                        ((and (cdr vars1)(cdr vars2)(eq (second vars1)(second vars2))

                          '((eq (second ,name1)(second ,name2)))))))

(defun build-action (condition action)
  '(lambda (first second third) (list (quote ,(first action))
                                      ,@(describe-args condition (cdr action)))))

(defun describe-args (condition action)
  (let ((args1 (first condition))
        (args2 (second condition))
        (args3 (third condition)))
    (mapcar #'(lambda (x) (cond ((equal x (car args1)) '(first first))
                                ((and (cdr args1)(equal x (second args1))) '(second fi
rst))

                                ((equal x (first args2)) '(first second))
                                ((and (cdr args2)(equal x (second args2))) '(second se
cond))

                                ((equal x (first args3)) '(first third))
                                ((and (cdr args3)(equal x (second args3))) '(second th
ird))))
            action)))

(defun traceon nil
  (trace describe-args build-action encode-pair build-test build-condition
         update encode-condition encode-a-production encode-productions))
```

```
;;; -*- Syntax: Common-Lisp; Package: RETE; Base: 10; Mode: LISP; Default-character-st
yle: (:FIX :ROMAN :LARGE) -*-

(cp:define-command (com-production-system :command-table "User")
    ((trace 'scl:boolean
            :prompt "Trace forms"
            :default nil
            :display-default t
            :documentation "Trace all encoding and firing forms."))
   (if trace (progn (traceon)(traceon2))
             (untrace))
   (production-system))

(cp:define-command (com-encode-productions :command-table "User")
    ((trace 'scl:boolean
            :prompt "Trace forms"
            :default nil
            :display-default t
            :documentation "Trace all encoding and firing forms."))
   (if trace (traceon)
       (untrace))
   (encode-productions))

(cp:define-command (com-rule-interpreter :command-table "User")
    ((trace 'scl:boolean
            :prompt "Trace forms"
            :default nil
            :display-default t
            :documentation "Trace all encoding and firing forms."))
   (if trace (traceon2)
       (untrace))
   (rule-interpreter))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: RETE; Base: 10.; Default-character-s
tyle: (:FIX :ROMAN :LARGE); -*-
;;; Stephen W. Davis  home: 205-533-7308
;;; Johnson Research Center 205-895-6257


(defun Production-System nil  (in-package 'rete) (initvars) (encode-productions) (rule
-interpreter))

(proclaim '(special productions database initlist))

(defun putprop (predicate value slot) (setf (get predicate slot) value))


(defun initvars ()
  (setq productions '((p1 ((father =x =y)(wife =z =x))  implies  (mother =z =y))
                      (p2 ((mother =x =y)(husband =z =x))  implies  (father =z =y))
                      (p3 ((wife =x =y))  implies  (husband =y =x))
                      (p4 ((husband =x =y))  implies  (wife =y =x))
                      (p5 ((father =x =z)(mother =y =z))  implies  (husband =x =y))
                      (p6 ((father =x =z)(mother =y =z))  implies  (wife =y =x))
                      (p7 ((husband =x =y))  implies  (male =x))
                      (p8 ((wife =x =y))  implies  (female =x))
                      ))

  (setq database '((father Alan Alta) (wife Alice Alan)
                   (mother Bertha Bet) (husband Bob Bertha)
                   (wife Caitlin Carter)
                   (husband David Dianne)
                   (father Earnest Earnie)(mother Evelyn Earnie)
                   (father Frank Fred)(mother Felicia Fred)
                   (husband George Georgia)
                   (wife Helen Herbert))))

(Defvar *hook-level* 0)

(defun hook (x)
  (let ((*evalhook* 'eval-hook-function))
    (eval x)))

(defun eval-hook-function (form &optional env)
  (let ((*hook-level* (incf *hook-level*)))
    (format *trace-output* "~%~v@TForm:  ~S"
            (* *hook-level* 2) form)
    (let ((values (multiple-value-list
                    (evalhook form
                              #'eval-hook-function
                              nil
                              env))))
      (format *trace-output* "~%~v@TValue: ~{~S~}"
              (* *hook-level* 2) values)
      (values-list values))))
```

*Appendix D.*

Routines from Ada Version of the Scheduler

These routines were programmed
by S. Davis
as part of the Ada adaptation
of the scheduler.

```
procedure FIND_CHILDREN is
   CLOCK_BEGIN: INTEGER;
   SCHEDULE_ME, PROCESS_BEFORE: DYPAS_PROCESS;
            --The global variables SCHEDULED and SCHEDULE_US
            --are declare in DYPAS package specification.
      begin
        CLOCK_BEGIN:= TIME_IO.GET_UNIVERSAL_TIME;

      MAIN_LOOP:
            for SCHEDULE_ME in SCHEDULE_US loop
                TEXT_IO.new_line;
      SUB_LOOP:
            for PROCESS_BEFORE in THOSE_SCHEDULED loop
                BEGIN_TIME:= COLLECT_TIME(PROCESS_BEFORE);
                REQUIRED_POWER:= SCHEDULE_ME.MAX_POWER;
                 AVAILABLE_POWER:= MAX_WATTAGE
                                          - FIND_MAX_POWER (SCHEDULE_ME BEGIN_TIME);
                if AVAILABLE_POWER >= REQUIRED_POWER then
                    SCHEDULE_ME.BEGIN:= BEGIN_TIME;
                    SCHEDULE_ME.END:= BEGIN_TIME + SCHEDULE_ME.RUNTIME;
                    CREATE_NODE(SCHEDULE_MEPROCESS_BEFORE);
                    BUILD_HASH(REQUIRED_POWER BEGIN_TIMESCHEDULE_ME.END);
                    PUSH(SCHEDULE_ME SCHEDULED);
                    PUSH(BEGIN_TIME START_TIMES);
                    PUSH(SCHEDULE_ME.END START_TIMES);
                    SCHEDULE_ME.CONDITION = "Scheduled";
                 end  if

      end loop SUB_LOOP;
      end loop MAIN_LOOP;

end FIND_CHILDREN;
```

declaring access type DYPAS_PROCESS

```
type PROCESS;

type DYPAS_PROCESS is access PROCESS;

type PROCESS is
      record
        MAX_POWER: INTEGER;
              BEGIN: INTEGER;
                END: INTEGER;
          CONDITION: STRING(1.. 30);
             BEFORE: DYPAS_PROCESS;
              AFTER: DYPAS_PROCESS;
      end  record;
```

then declaring the objects is as follows

```
      ECLSS: DYPAS_PROCESS;
            -- object ECLSS created with slots set to null.

      IEFF: DYPAS_PROCESS (END => 100);
            -- object IEFF created with slot END set to integer 100.
```

*Appendix E.*

Floyd and Ford's Description of their Scheduler

from the 1986 Proceedings of the Conference
on Applications of Artificial Intelligence to Space

# A Knowledge-based Decision Support System for Payload Scheduling

Stephen Floyd*
and
Donnie Ford**

*Department of Management Information Systems and Management Science
School of Administrative Sciences, University of Alabama in Huntsville

**Cognitive Systems Laboratory - Johnson Research Center
University of Alabama in Huntsville

## ABSTRACT

The purpose of this paper is to illustrate the role that artificial intelligence/expert systems technologies can play in the development and implementation of effective decision support systems. A recently developed prototype system for supporting the scheduling of subsystems and payloads/experiments for NASA's space station program is presented and serves to highlight various concepts. The potential integration of knowledge based systems and decision support systems which has been proposed in several recent articles and presentations is illustrated.

## 1. INTRODUCTION

At the Sixth International DSS Conference (DSS-86) Peter Keen in the closing plenary address entitled "DSS: The Next Decade" discussed what he perceived as the important roles of current and future AI/ES technology in extending the field of decision support systems. Among his perceptions was the fact that the field of AI could play a major role in the development of systems to support the tougher, ill-structured types of problems. He also viewed current AI/ES hardware and software technology as "power tools" for DSS development. A few months earlier John Little in an article entitled "Research Opportunities in the Decision and Management Sciences" promoted similar observations while discussing research priorities of NSF's Decision and Management Science program [9]. Major among these priorities was the role that expert systems technology could play in advancing the Decision Sciences. Similar ideas have been expressed over the past few year by other reseachers in articles and at major conferences such as ORSA/TIMS, DSS-86, IDS and AAAI [8], [13], [16], [17]. This paper supports these observations by describing knowledge-based DSS for scheduling payloads for NASA's space station program. The payload scheduling system serves to illustrate the potential integration of DSS and ES as it involves the addition of a knowledge based component to a system which currently provides decision support via extensive interaction between scheduling personnel and more traditional scheduling techniques. It is the authors' hope that the following discussion of the scheduling system will help other researchers in establishing the applicability of the new "power tools" in DSS development.

This paper concerns the development of a solution procedure and interactive system for scheduling subsystems and payloads/experiments for the National Aeronautics and Space Administration space station program. Traditionally, scheduling problems have been viewed as static in nature (i.e., a schedule is developed for a particular planning horizon and adhered to) and were cast as having one or more clearly defined objectives (e.g., minimize overall completion time,

maximize resource utilization, etc.). As such, these problems were most commonly solved via application of optimal seeking algorithms, heuristics or simulation analysis [1] [4] [6] [7] [15]. The payload scheduling problem, in contrast, is representative of a class of scheduling problems which are highly dynamic in nature. Not only may the various parameters change at any time, but the objectives themselves may change also. As will be illustrated in this paper, the nature of this class of problems is such that they can be most effectively solved by knowledge based expert systems [2] [3] [5] [11] [18] [19].

Provided in the first section of the paper is a detailed description of the class of problems under investigation. After an overview of the problem domain, the specifics are provided for the NASA problem which lead to the development of the system. The third section discusses the initial dynamic scheduler solution strategy that was developed for the prototype system. The details of this prototype expert system and its development are provided in the fourth section. The fifth section discusses future enhancements that have been identified for the system. The final section of the paper provides some concluding remarks on the research to date, and some suggestions for future research in the area of dynamic scheduling.

## 2. PROBLEM DESCRIPTION

The application addressed in this paper concerns development of a system for the scheduling of subsystems and payloads aboard the space station. Subsystems are systems which function to support space station on an ongoing basis. These include such subsystems as life support systems, communications systems, and various "housekeeping" systems. Aboard space station will also be various payloads and experiments. These will be sponsored not only by NASA but also by other U.S. and foreign government agencies, universities and private industries.

Each of the subsystems or payloads has a certain set of characteristics and requirements which must be considered in determining when during the mission it should be scheduled. For example, each subsystem and most of the payload/experiments will draw operating power from Space Station's limited power supply. Additionally, certain of them will require astronaut intervention either on a continuous basis for the duration of the experiment or for specified subintervals of time. Some subsystems and experiments are continuous in nature and run uninterupted throughout the entire mission. Still others operate either continuously or intermittently for only a specified subinterval of the mission time window. The nature of some experiments will require that they be conducted only during certain phases of the mission (e.g., during day orbit, during night orbit, during certain orientations of space station, etc.). These example characteristics, as well as others which will not be detailed here, coupled with the fact that payload/experiments are placed in priority classes which must be reflected in the schedule, form the basic criteria for establishing feasible schedules.

The complexity of the scheduling problem is compounded further by the fact that events which will be occurring during the mission will serve either directly or indirectly to upset current schedules and/or influence future ones. For example, at any time during the mission an ongoing experiment may fail or be aborted for some reason, a scheduled experiment may be withdrawn from the schedule, an experiment or entire class of experiments may be added and/or experiment priorities changed. The scheduler must be designed to handle such dynamic changes via interaction with various mission personnel, including astronauts, mission planning specialists and principal investigators of affected experiments.

As mentioned previously, each subsystem and payload/experiment will consume various resources. Major among these will be energy from the Space Station's power supply and manpower provided by the astronauts on board. Such limited resources place constraints on what systems and experiments can be concurrently ongoing. Additionally, and this is another of the dynamic aspects of the problem, the power and manpower allotments themselves may change at various times throughout the mission. In some instances the change notification will provide lead time for scheduling adjustments, whereas in others no lead time will be provided. Changes will occur, for example, when vehicles dock with Space Station. Such changes result from the fact that the docking will usually draw on such resources as the power and manpower supply. In light of the above mentioned characteristics, the scheduler must have capabilities beyond the generation of traditional static feasible schedules. The dynamic scheduler must have the capacity to respond interactively to such changes and, when required, maintain feasibility via a rescheduling procedure.

A final characteristic of the payload scheduling problem is that the scheduling objectives are variable. During the course of a Space Station mission, mission specialists may re-structure the scheduling objectives. For example, it might be that early in a mission a resource leveling strategy is adopted which will maintain a fairly constant and conservative power consumption rate. Such an objective would naturally "stretch out" the scheduling of experiments over some designated planning horizon. Later in the mission cycle, however, factors may change this objective to one of scheduling as many payloads/experiments as possible (subject to the maximum power availability and other constraints) in a given time frame. These characteristics then establish the need to develop a system which is capable of not only establishing static schedules but also of dynamically maintaining feasible payload/experiment schedules which reflect the varying parameters of the problem.

## 3. SOLUTION STRATEGY

Sample data around which the prototype system could be constructed was provided by NASA's Power Branch. The data as considered by NASA to be representative of actual scheduling data. As seen from Table 1, four subsystems and forty-five (45) payloads/experiments were included. Provided in the table are the experiment name, the associated power consumption requirements in kilowatts, the sponsoring agency, the time duration (including other specifications such as continuous/intermittent, day orbit/night orbit, etc.) and crew involvlement required. In addition to the data in the table, other problem specifications were also provided. Most pertinent among these were (1) the specification of a normal lab module power level of 25 kilowatts, (2) a priority structure based on the sponsoring agency and the nature of the payloads/experiments, and (3) a two-week scheduling horizon. Additionally, several system requirements pertaining to the actual operation of the scheduler were specified. These provided a framework for the user interface and system output as detailed later in the system description section of the paper.

To prototype an initial system for user evaluation and feedback, a means of generating feasible schedules in the absence of a complete corporate knowledge base had to be developed. This was accomplished via the modification of a scheduling strategy presently used by NASA scheduling personnel which involves conceptualizing schedules using a Gantt chart type format. This heuristic procedure is representative of those that when augumented by various scheduling rules will comprise the scheduling knowledge base of the final system. An example schedule for a simple four experiment problem is given in figure 1. As can be seen, experiments one, two and three are continuous, and experiment four is intermittent.

Given inside the bars, which represent the experiment durations, is the power requirement of the particular experiment. For simplicity these power requirements are assumed constant as long as the experiment is "on." Through the use of this four experiment example, the heuristic will now be described.

As an experiment is placed on the chart, its beginning and ending point(s) serve to divide the overall time window, the x-axis, into intervals as illustrated by the dotted lines in figure 1. By updating as each experiment is scheduled, one can maintain for each subinterval of time the information necessary in determining the time slot for the next experiment to be scheduled. The determination of which experiment is to be scheduled next is based on the user predefined priority structure in effect at the time of the scheduling or rescheduling procedure. For the sake of illustration we will simplify the four experiment example further by assuming a single scheduling objective of maximizing power utilization. Each experiment is scheduled by searching through time on the x-axis in figure 1 from left to right until a subinterval or group of successive subintervals is found which has sufficient duration and power availability to support the given experiment. The experiment is then scheduled and added to the chart in correspondence with this subinterval. Subinterval information is updated to reflect resource availability to reflect resource availability (i.e., power and manpower) and the scheduling procedure continues.

Applying the scheduling heuristic to the representative problem provided by NASA is obviously much more involved than the example provided above as the various experiment characteristics and requirements must be matched to appropriate intervals. As the number of requirements increases for experiments, so too does the amount of infor mation being kept on each subinterval. Additionally, as the number of experiments already scheduled increases, the number of subintervals to be examined during each individual scheduling process also increases. This increase in the number of subintervals is compounded even further when the experiment scheduled is of an intermittent nature. These facts, coupled with the previously mentioned dynamic aspects of the problem, necessitate an automated procedure for generating schedules. The next section of the paper will describe the prototype system developed to accomplish this.

## 4. SYSTEM DESCRIPTION

The prototype system follows the basic production system structure of a knowledge base, inference engine and working memory or global data base. The knowledge base consists of a reduced set of scheduling rules and knowledge pertinent to the example problem. The system utilizes a frame representation scheme which allows for utilization and exploitation of knowledge other than rules. This feature increases the speed and efficiency of the system; in particular, the inferencing process.

The inference engine performs only forward chaining. This was determined from the structure of the problem. There is an abundance of related facts and information at the beginning of the problem solving process which in turn accomodates the forward chaining process. The conflict resolution problem is solved by allowing the first rule that is satisfied to be implemented. This necessitates an ordering of the rules. This resolution method was chosen because of the short time frame for delivering a "demo" system. This also facilitates the search through the working memory.

The working memory consists of a list of experiment names. Associated with

these names are certain facts that are placed into the knowledge base. These include the power requirements, identification number, priority class, sponsoring agency, duration of the experiment, and the required crew involvement. Using this information, a prioritized list of experiments is generated for utilization during the scheduling phase.

The system has been designed and developed in an open-ended fashion to allow system to be extended with only minor adjustments. It contains, in addition to the knowledge-base system structure, an output interface which is at present for demonstration purposes only. This interface will be detailed later. he system itself is dynamic in that it moves through or between different phases of the problem solution. The phases include preparation, scheduling, operation, and rescheduling.

During the preparation phase the individual experiment information is provided to the system from an external data base source and appropriately stored, also the working memory is organized and then prioritized for the scheduling phase. This is accomplished using a priority scheme developed from user input. In the scheduling phase, the experiments are scheduled under the previously explained heuristic procedure and the schedule is created. The schedule itself is part of the knowledge base and is represented as frames. As experiments are scheduled, the subintervals required by the heuristic procedure are defined by start and stop times of the experiments. For each interval the power available, crew available, and the experiments that are currently on-going are determined and stored. This information is required for the remaining two phases, namely operation and rescheduling. The initial schedule is provided to the user for evaluation in a Gantt chart format with appropriate labels (i.e., experiment identification, start and end times, resource loadings, etc.). The user is then afforded an apportunity to make several types of scheduling changes including changing the planning horizon, manually scheduling experiments, reprioritizing experiments, changing rfesource parameters, etc. Based on nthe changes specified, the system determines whether any rule/constraint conflicts exist and if so performs a rescheduling operation as descr5ibed below to resolve the conflicts. In cases where the specified changes do not allow for conflict resolution, the user is so informed.

The output interface during the operation and rescheduling phases is graphical in nature and menu driven. During the operation and rescheduling phases, the system simulates control of the power source for the experiments, i.e., it turns them off or on at the appropriate times indicated by the schedule and updates all the necessary interface information accordingly. The operation phase has two modes: (1) static and (2) dynamic. In the static mode, the system is capable of displaying a power utilization graph for a two week, one week, one day or six hour period of time. Also, the vital information for each experiment (start time, end time, etc.) can be requested by the user simply by using the mouse and a selection menu. In the dynamic mode, the system uses the output interface (see figure 2) to interact with the user through four basic windows - a current status window, a schedule window, a power curve window and a message window. The current status window shows the current status of all the experiments of a payload at a particular point in time. This is accomplished, as illustrated in the window at the top of the screen in figure 2, by representing each experiment as a numbered box. Reverse video is then used to differentiate the on state. Labels placed within the boxes indicate such statuses as aborted, removed, completed, etc. The schedule window (bottom left of screen in figure 2) displays the names of the experiments on separate lines and uses a Gantt chart format similar to that shown in figure 1 to display the scheduling of each experiment. This window simulates movement through time, i.e. as

time passes the bars that represent the experiment move to the left and disappear as the experiment is completed. When the experiment is completed the word "completed" appears next to the experiment name. In the schedule window the experiments are also numbered to provide a cross-reference to the numbered experiment boxes in the current status window. The power curve window (middle right of display screen) plots percent power utilization as it scrolls through time. The remaining window is the message window. This is used for interaction and control purposes.

One of the important capabilities built into this system is its ability to reschedule the experiments when deemed necessary. This is one of the main differentiators of this system when compared to others developed for such scheduling applications. The system is capable of determining when it is necessary to reschedule. When such a determination is made, the experiments affected are identified and removed from the active schedule. A rescheduling is conducted and the new schedule is implemented (i.e., made active). There are, based on the initial problem description, a limited number of occurences that would warrant a reschedule. These include an experiment failure, an experiment abort, a power allotment increase or decrease, or the announced arrival of orbital docking and/or servicing vehicles. The first two occurrences require an automatic rescheduling while the others require the system to check working memory and the knowledge base to determine if rescheduling is in fact necessary. Thus we see the system is capable of moving between the different phases, capable of recognizing where it is and what knowledge is applicable, and dynamic in its ability to genenrate and maintain a schedule that will accomplish the objective or objectives of the mission as specified by mission
planning specialists.

## 5. FUTURE ENHANCEMENTS

While the system demonstrates the potential of using a knowledge base system approach in the area of scheduling, there are several enhancements that have been identified and are currently being implemented to improve the performance and capabilities of the system. First and foremost is that more experiential knowledge needs to be added to the knowledge base. Sessions have been scheduled with the appropriate NASA personnel to begin the task of knowledge engineering [2] [3] [11] [18]. Also, knowledge concerning the determination of alternatives to the schedule instead of just developing a single initial schedule will be added. This will provide the system with the capability of helping NASA personnel in satisfying the dynamic objectives experienced during a mission and will also facilitate the rescheduling process. An example of such an objective is when power allotment reduction forces the schedule to run past the end-of-mission time. Having "knowledge" of alternatives, the system will have a better understanding of which experiments to schedule. Should it continue with the normal rescheduling rules or does some special set of priorities apply? Not only will more objectives be handled in the enhanced system, but the system will be able to handle more constraints, (e.g., fluctuating power requirements of experiments, orientation of the experiments, etc.). Another area of knowledge enhancement concerns the rescheduling function. It has been determined that the system should be capable of performing a quick-fix reschedule when necessary. This will provide the necessary time to perform a more detailed and thorough reschedule in the event of an emergency situation where a temporary "quick fix" is necessary.

The second area of improvement and enhancement to the system is efficiency. Not only is the efficiency of the code being considered, but a more efficient and effective method for searching the schedule and determining experiment slots is

under development. This search process, as was mentioned previously, is complicated by the scheduling of intermittant experiments early in the scheduling process. In particular, one experiment in the sample data is required to be scheduled ten minutes out of every hour that the mission is operating. Under the present system this creates 336 additional time intervals that might have to be checked for power and crew availability in determining a feasible interval for a later experiment. The present system takes 15 minutes to schedule the 47 test experiments. The majority of this time is due to the early scheduling of intermittent experiments. Another efficiency enhancement is for the system itself to determine the mission time horizon. This will reduce any excess time that is added in order to accomodate all the experiments. At present, the time horizon is driven by the number of experiments requireing crew involvement and the number of crew available to work with the experiments. A critical path algorithm is being investigated for application in this area.

The final area of enhancements is in the user interface, both input and output interfaces. On the input side, a query/answer system will be added to allow for easy input of experiment data and knowledge base maintenance. This interface will have a limited, natural language parser [10] [12] and will exploit the use of graphics. On the output side several enhancements will be made. First, the system will have an explanation capability for how and why it chose the schedule it is recommending. This capability will soon be provided since the system is currently being redone using an expert system shell which provides how and why facilities. Also, a hardcopy capability for printing out the schedule in "readable" form will be added. Currently the schedule is only stored in symbiolic form. These output enhancements will facilitate the evaluation of system performance. This should allow the system in turn to gain user acceptance more quickly and will also help facilitate the implementation phase.

## 6. CONCLUSIONS

This paper has detailed a knowledge-based system for solving the NASA space station payload/experiment scheduling problem. The problem is representative of a larger class of dynamic scheduling problems which, for the most part, have been ineffectively handled using more traditional numeric techniques. An expert systems approach allows one to effectively deal with the dynamics and incomplete information which characterize this class of problem. Still in prototype form the system is meeting with wide acceptance and interest not only from the sponsoring agency, but also from other independent sources.

The interest this project has received indicates that there is potential for further research in this area. The wide problem domain encompassed by dynamic scheduling provides many areas for future applications (e.g., project scheduling, production scheduling, manpower scheduling,. etc.). Additionally, as systems are implemented and knowledge engineering continues, there is a good likelihood that commonalities will be established across various scheduling applications. This would allow devlopment of an expert system shell for such problems. Such a shell would allow scheduling systems to be readily developed and implemented.

## REFERENCES

1. Baker, K. R., Introduction to Sequencing and Scheduling, New York: John Wiley and Sons, Inc. 1974.
2. Barr, A. and Fiegenbaum, E., The Handbook of Artificial Intelligence, Volumes I and II, William Kaufmannm Inc., 1982.

3. Davis, Randall, and Douglas B. Lenat, Knowledge - Based Systems in Artificial Intelligence, New York: McGraw - Hill, Inc., 1982.
4. French, S., Sequencing and Scheduling: An Introduction to the Mathematics of the Job Shop, New York: John Wiley and Sons, Inc., 1982.
5. Harmon, Paul, and David King, Expert Systems, New York: John Wiley and Sons, Inc., 1985.
6. Johnson, L. A. and D. C. Montgomery, Operations Research in Production Planning, Scheduling and Inventory Control, New York: John Wiley and Sons, Inc., 1974.
7. Lawler, E. L., J. K. Lenstra and A. H. G. Rinnooy Kan, "Recent Developments in Deterministic Sequencing and Scheduling: A Survey," in M. A. H. Dempster et. al. (Eds.), Deterministic and Stochastic Scheduling, Reidel, Dordrecht, 1982, 35-73.
8. Lehner, P. E. and Donnel, M. L. "Building Decision Aids: Exploiting the Synergy Between Decision Analysis and Artificial Intelligence," Paper at ORSA/TIMS, San Francisco, May 1984.
9. Little, John D. C., "Research Opportunities in the Decision and Management Sciences", Management Science, Vol. 32, No. 1, January 1986.
10. Rauch-Hindin, Wendy, "Natural Language: An Easy Way to Talk to Computers," Systems & Software, January, 1984, pp. 187-230.
11. Riesbeck, C. K. and Roger Schank, "Comprehension by Computer: Expectation-based Analysis of Sentences in Context," in W. J. M. Levelt and G. B. Hores d'Arcais (Eds.),Studies in the Perception of Language. Chichester, England: John Wiley and Sons, 1976, pp. 247-294.
12. Rich, E., Artificial Intelligence, New York: McGraw-Hill, Inc., 1983.
13. Sprague, R., "The Role of Expert Systems in DSS," Paper at ORSA/TIMS, Dallas, Nov. 1984.
14. Symbolics software. Report, Symbolics, Inc., 21150 Califa Street, Woodland Hills, California, 1981.
15. Tersine, Richard J., Production/Operations Management: Concepts, Structure, and Analysis, (2nd ed.), New York: North-Holland Press, 1985.
16. Turban, E., and King, David, "Building Expert Systems For Decision Support," DSS-86 Transactions, Jane Fedorowicz, Editor, 1986.
17. Turban, E., and Watkins, P., "Integrating Expert Systems and Decision Support Systems," MIS Quarterly, June 1986.
18. Waterman, Donald A., A Guide to Expert Systems, Reading, Massachusetts: Addison-Wesley Publishing Company.
19. Winston, Patrick H., Articial Intelligence, (2nd ed.), Reading, Massachusetts: Addison-Wesley Publishing Company, 1984.
20. Winston, P. H., and Horn, B. K. P. LISP, Reading, Massachusetts: Addison-Wesley Publishing Company.

LABORATORY MODULE - SPACE STATION

DYNAMIC PAYLOAD SCHEDULER

SUBSYSTEMS:

| NAME | POWER WATTS | PRIORITY CLASS | AGENCY | DURATION | CREW |
|---|---|---|---|---|---|
| ECLSS | 6200 | I | NASA | CONTINUOUS W/20 MIN. LAPSES OKAY EVERY 4 HRS. | 0 |
| COMMUNICATIONS | 1480 | I | NASA | CONTINUOUS | 0 |
| THERMAL CONTROL | 600 | I | NASA | CONTINUOUS-REDUCES LINEARLY TO 400W FOR 10 KW POWER LEVELS | 0 |
| HOUSEKEEPING(MISC) | 6000 | I | NASA | CONTINUOUS | 1 |
| PAYLOAD/EXPERIMENTS: | | | | | |
| DOD/PAYLOAD 1 | 890 | II | DOD | 48 HRS | 1 |
| ESA PAYLOAD 1 | 1845 | II | ESA | 214 HRS | 1 |
| IPS | 165 | II | ESA | 240 HRS | 1 |
| ELECT DIAG STA | 435 | II | NASA | 10 MIN OF EVERY HR | 1 |
| IECM | 480 | III | NASA | 200 HRS | 0 |
| CRNE | 930 | IV | U.K. | 240 HRS | 0 |
| GEN PURPOSE COMP | 383 | III | NASA | CONTINUOUS | 1 (5 MIN/HR) |
| SOLID POLYMER ELECT | 415 | IV | 3M | 36 HRS | 1-5 MIN. EVERY 3 HRS |
| IEF | 125 | IV | NASA | 6 HRS | 0 |
| MLR | 350 | IV | U/IOWQ | 20 HRS | 0 |
| FES-VCGS | 600 | III | NASA | 15 HRS | 0 |
| ROTI | 36 | IV | UAH | 43 HRS | 0 |
| SEM | 2648 | IV | NASA | 6 HRS | 0 |
| RTG | 94 | IV | NASA | 12 HRS | 0 |
| TAPE RECORDER 1 | 85 | II | NASA | CONTINUOUS | 0 |
| TIME CODE GEN | 32 | II | NASA | CONTINUOUS | 0 |
| MASS SPECTROMETER | 215 | IV | JAPAN | 2 HRS | 0 |
| TOOL CHARGER | 50 | II | NASA | CONTINUOUS | 0 |
| FILM PROCESSOR | 163 | II | NASA | 1 HR/DAY* | 1 |
| SUPER FURNANCE | 7840 | II | G.E. | 32 HRS | 0 |
| SILICON WAFER PROD | 4760 | III | INTEL | 14 HRS | 0 |
| TAPE RECORDER 2 | 85 | III | NASA | CONTINUOUS | 0 |
| TGA | 612 | IV | ESA | 8 HRS | 0 |
| MEA | 1800 | IV | NASA | 14 HRS | 0 |
| WELDING EXP | 1610 | IV | NASA | 4 HRS | 0 |
| CFES | 890 | III | NASA | 36 HRS | 0 |
| 3-AAL | 500 | IV | NASA | 10 HRS | 0 |
| EML | 420 | IV | NASA | 2 HRS | 0 |
| GFFC | 375 | IV | NASA | 6 HRS | 0 |
| ADSF | 480 | IV | NASA | 48 HRS | 1 |
| ARC | 215 | IV | NASA | 8 HRS | 0 |
| SAFE | 400 | III | NASA | 15 HRS | 1 |
| SOLAR OBS | 375 | II | NASA | ORBIT/DAYTIME ONLY FOR 36 ORBITS | 0 |
| LIGHTNING DET | 125 | IV | NASA | ORBIT NIGHTTIME ONLY FOR 12 ORBITS | 0 |
| CRYSTAL GROWTH | 1200 | II | NASA | 1 HR | 0 |
| COMET SEARCH | 650 | III | JAPAN | ORBIT NIGHTTIME ONLY FOR 40 ORBITS | 0 |
| LIFE SCI 1 | 135 | III | A&M | 36 HRS | 0 |
| LIFE SCI 2 | 1145 | III | UAB | 22 HRS | 0 |
| LIFE SCI 3 | 842 | III | UAB | 66 HRS | 0 |
| CLASSIFIED 1 | 1300 | II | DOD | 8 HRS | 0 |
| CLASSIFIED 2 | 645 | II | DOD | 18 HRS | 0 |
| MAPPING (WEATHER) | 300 | III | USWS | CONTINUOUS (CAN BE INTERRUPTED ANYTIME) | 0 |
| MAPPING (GEO) | 690 | III | USGS | 60 HRS | 0 |
| ORBITER DOCKING | 6500 | ** | NASA | 24-72 HRS, WILL BE GIVEN 6 HRS NOTICE | 0 |
| ORBITER SERVICER | 2400 | ** | NASA | 4-10 HRS, WILL BE GIVEN 2 HRS NOTICE | 0 |

** WILL BE GIVEN TOP PAYLOAD PRIORITY WHEN NEEDED

Table 1. Experiment Data

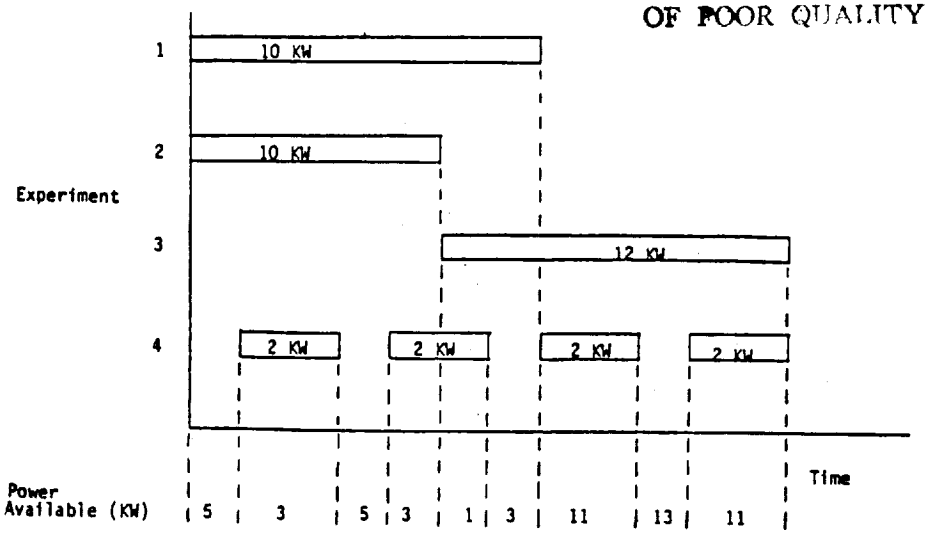


Figure 1. Example Schedule

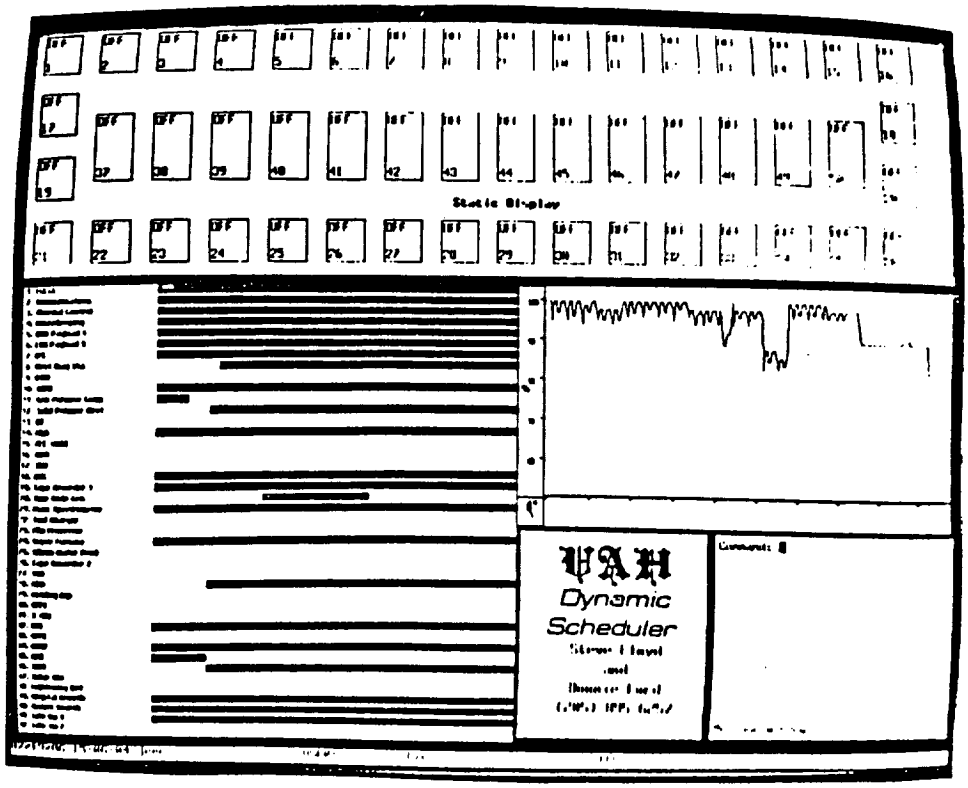


Figure 2. Scheduler Display Screen