ORGANIZING THE SEQUENCING OF PROCESSES

Fred Lesh
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91103/USA

ABSTRACT

The difficult part of many computer applications is the design and implementation of control mechanisms which allow the necessary calculations to be performed at exactly the right time or in exactly the right logical sequence. Designing systems of this kind involves the designer in a variety of unusually interesting problems, and requires blending classical design approaches with techniques peculiar to computational control applications. This paper presents three examples of such applications and describes the approach adopted for each. The classical design steps are then discussed, and elements peculiar to computational control are highlighted by illustrations drawn from the three sample systems.

I. INTRODUCTION

Many kinds of complexity plague designers of software systems. They are overwhelmed by the number and sophistication of the equations of motion for an interplanetary spacecraft, innundated with the sheer volume of options and special cases of a business application, or puzzled how to achieve a convergent algorithm in the solution of coupled sets of partial differential equations.

But there is a wide class of computer applications in which the primary complexity stems not from the difficulty or variety of the underlying computations but from the need to make those computations in the right order or at the right time. Systems developed to achieve this kind of ordering and timing are called "computational control systems." Three examples of such systems are discussed in Section II.

Computational control systems are usually written assuming that they will be used more than once for a variety of applications within a certain class. Programmers who use an already developed control system for a new application are called "users." But a computational control system may be developed for a single application in order to achieve a flexible system architecture which avoids large reprogramming efforts during the system checkout and acceptance period. The example of Section II-A is such a case.

The architecture chosen for a computation control system can make the difference between a chaotic, inefficient program — difficult to understand and maintain — and an orderly, even elegant program which pleases the maintenance staff as well as the designer. The task of arriving at an orderly architecture for computational control systems blends standard system design concepts with other concepts peculiar to problems of computational control. Section III describes a set of standard steps in system design and some novel features of the standard steps which arise from peculiarities of computational control problems. The novel features and their impact on the standard steps are presented in terms of illustrations drawn from the examples of Section II.

II. EXAMPLES OF COMPUTATIONAL CONTROL SYSTEMS

Sections A, B, and C below contain examples of computer applications leading to interesting and difficult computational control problems. A brief description of the problem is given in each case, followed by an indication of the kind of system developed to solve the problem.

A. Mosaic Tiling

Figure 1 shows a test frame produced by a program written to process TV picture data from the camera aboard the Viking Lander spacecraft, which in late 1975 transmitted to earth the first pictures of the rock-strewn plains of Mars. This picture is a mosaic of tiny tiles — so small that they can be seen in the 20 x 20 centimeter original glossy only under a strong magnifying lens. The total frame is 1024 tiles across and 1024 tiles deep. The tiles range from black to white in 64 shades of gray. Unlike real ceramic tiles, the tiles originate aboard the spacecraft as 6-bit integers which represent gray levels, reside for a while in computers as 8-bit bytes of data, and end up as tiny spots on a photographic film. The tiles are called "pixels."

Processing of images, a complicated task that has been the subject of many papers, is not the concern here. Instead, this paper concentrates on the problem of producing the total frame — image plus borders, annotation, histograms, gray-level calibration scale, etc. In this process, the image is assumed to be stored as data bytes in bulk storage where it can easily be retained and inserted into the frame as required.

The complications in compiling the frame arise from the fact that the number of pixels in the image is too large (10^6) to store conveniently in random access memory. Instead, it is necessary to generate the picture one line at a time and write the lines on magnetic tape in the format needed by the film recorder, which produces the final hard copy. The Viking Lander camera produced vertical data lines that sweep from bottom to top instead of the horizontal lines familiar from commercial TV that sweep from left to right, so it is natural to produce vertical lines for the entire frame also.

Figure 2 is an approximate expansion of the bottom right-hand corner of one of the histograms which appears along the right edge of Figure 1. The area of Figure 2

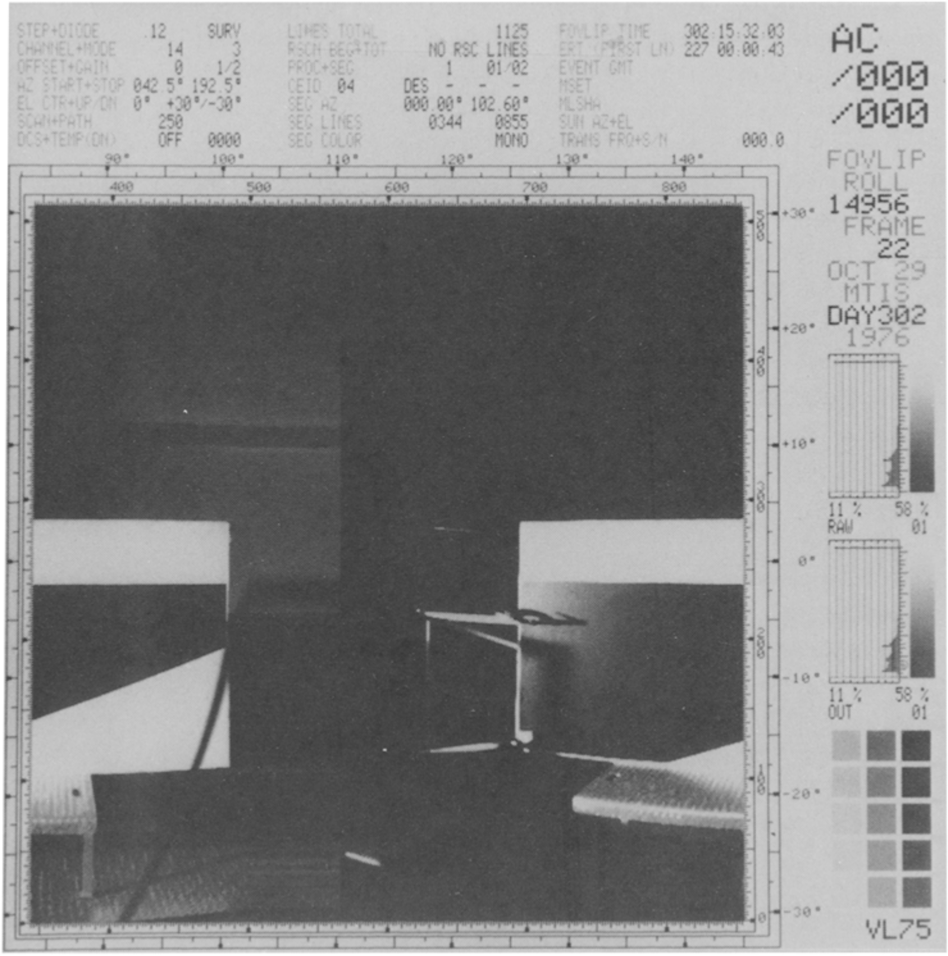


Figure 1. Test Frame From a Mosaic Tiling Program

is shown broken into six subareas. Each of these subareas is associated with a program which generates the pixels for that area. These programs are called "pixel generators." The breakdown of the total area of Figure 2 into six subareas is necessary to make each of the pixel generators a simple, logical entity. Even with this much breakdown, the pixel generators for subareas 1 and 6 are far from trivial since the histogram and annotation both vary from one picture to the next.

The difficulty in building a single line of the frame in Figure 1 is that any single vertical line consists of pixels from many different areas. To produce the necessary pixels for a given vertical line, many different pixel generators must run in the right order, and that order is different for some vertical lines than for

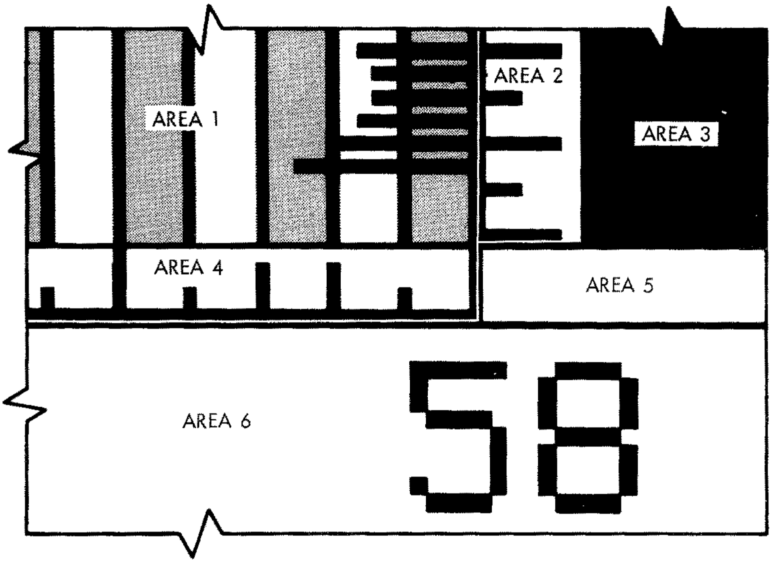


Figure 2. Lower Right Corner of a Histogram

others. Each pixel generator must produce only a small subset of its total pixels (those for one line through the area associated with the pixel generator) and must then somehow remember where it was until its turn comes to produce pixels for the next line. The problem of coordinating the activities of all the pixel generators is an exercise in computational control.

The control scheme adopted for this problem is table-driven, as illustrated in Figure 3, which, for explanatory purposes, shows a simple pattern of areas unrelated to those of Figures 1 and 2. But areas 3 and 5 require the same pixel generator (G_3), and this is exactly the situation which arises with the histograms that appear along the middle of the right-hand side of Figure 1. Even though areas 3 and 5 require the same pixel generator, they need different data.

Each pixel generator is written as a pure procedure operating on a state vector which can be pointed to by a data pointer P. The pixel generator is written as though it begins at line zero, and each time it is called, it outputs a single vertical line, increments the internal line count in its state vector, and returns. The state vector for the pixel generator may contain just the internal line number, but it may also contain data such as that used to produce the histograms at the right of Figure 1.

The first row of the control table of Figure 3 tells the control program to produce lines 0 through 12 of the frame by calling pixel generator G_3 with data pointer P_5, then calling pixel generator G_3 again with data pointer P_3, and finally calling

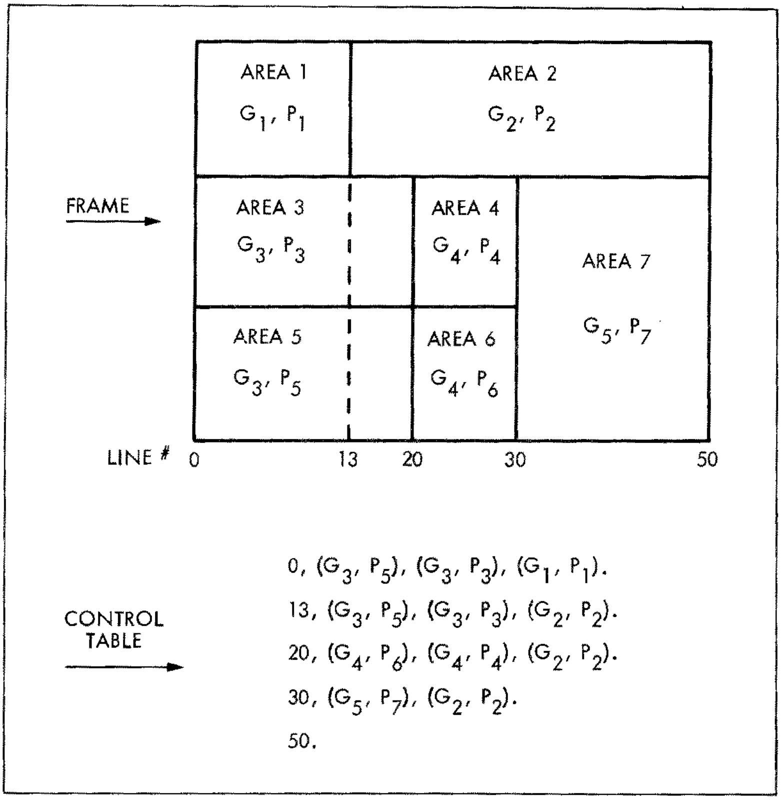

FRAME
AREA 1
G_1, P_1
AREA 2
G_2, P_2
AREA 3
G_3, P_3
AREA 4
G_4, P_4
AREA 5
G_3, P_5
AREA 6
G_4, P_6
AREA 7
G_5, P_7
LINE #
0
13
20
30
50
CONTROL TABLE
$0, (G_3, P_5), (G_3, P_3), (G_1, P_1).$
$13, (G_3, P_5), (G_3, P_3), (G_2, P_2).$
$20, (G_4, P_6), (G_4, P_4), (G_2, P_2).$
$30, (G_5, P_7), (G_2, P_2).$
$50.$

Figure 3. Sample Frame and Control Table for Mosaic Tiling

pixel generator G_1 with data pointer P_1. The control program simply produces lines according to the specifications of the control table, and writes the lines one at a time on magnetic tape.

B. Spacecraft Control

An interplanetary spacecraft typically consists of as many as 20 subsystems such as the power subsystem, the propulsion subsystem, the attitude control subsystem, and the television subsystem. From a computer software point of view, operating a spacecraft is similar to operating an oil refinery or an automotive assembly plant. The operations to be performed are usually such simple ones as closing a switch. The complexity of the problem derives entirely from the necessity to close the right switch at exactly the right time.

At the Jet Propulsion Laboratory, the natural and historical partitioning of spacecraft into subsystems led easily to consideration of distributed systems of microprocessors for spacecraft control and data handling, and a breadboard system was constructed to test the concept [1, 2]. In the breadboard, each subsystem is assigned a microprocessor with just enough capacity to control that subsystem and to acquire and format its data. All the processors are tied together with a common data bus.

Even though the spacecraft control job is distributed, each microprocessor still has several programs which have to run concurrently. The TV subsystem, for example, has one program to drive a TV camera through a complex cycle of erasures needed to eliminate the ghost of the old picture from the vidicon, another program to simultaneously drive data lines out of the alternate camera, and a third program to gather miscellaneous engineering data needed by the system controller to monitor the health of the TV subsystem.

Most real-time systems are completely interrupt-driven in the sense that all activity initiations and completions are signaled by a processor interrupt. Interrupt-handling programs turn on flags requesting that computations be performed, and processing is then done on a highest-priority-first basis. Because it is important to be able to easily diagnose malfunctions on a spacecraft operating millions of miles from earth, JPL's breadboard operates in a completely different manner. The only interrupt is one which occurs simultaneously in all processors every 2.5 milliseconds. There is nothing special about 2.5 milliseconds — it is simply an interval chosen to give time resolution more accurately than any which would be needed for spacecraft control, yet long enough to allow completion of calculations which need to be done at a given point in time.

This 2.5-millisecond interrupt serves two related purposes. The first purpose is to act as the escapement for a software clock. The 2.5-millisecond interval between interrupts is called a "tick." There are 400 ticks to a second, and a software clock in each processor increments time (hours, minutes, seconds, ticks) at each interrupt. The second purpose is to strobe out all input or output signals which have been set up by the operation of software during the tick.

The interface circuitry between processors and their instruments is organized in such a way that electrical control signals produced by software operation during a given 2.5-millisecond interval are buffered until the end of the interval before being sent to the external hardware. If programs A and B both run and produce output signals during a 2.5-millisecond interval, the output signals are identical whether program A or B runs first. The order of program execution within a 2.5-millisecond interval is therefore irrelevant, and the programs are thought of as operating simultaneously and instantaneously.

Each program running in this system is required to do specific (usually very simple) things at exactly the right times as kept by the software clock. The computation control problem in this application centers around the questions: How can several programs — each of which must time its operations exactly — be run concurrently?

Can application programs be written in such a way that each is transparent to the others during development and operation? Can they be designed and written using standard DO, IF, and CALL structures?

It was originally proposed to organize the software system around time-event tables. But it is difficult to achieve the effect of DO loops in a time-event system. Also, it was desired to do structured design and coding.

The adopted solution to this problem involves two constructs, called WAIT and WHEN, which allow for program timing. If, for example, a programmer wishes to delay the execution of a sequence code defining some action A until the first tick for which his software clock reads SEC = 40, he writes:

```
WHEN SEC = 40
DO ACTION A
```

If the programmer wishes to delay the execution of a sequence of code defining action B for exactly 75 ticks, he writes:

```
WAIT 75 TICKS
DO ACTION B
```

These constructs can be written anywhere inside or outside of the ranges of DO, IF, or CALL statements. Consider, for example, the case in which it is desired to drive data from a 600-line TV camera. Assume that the camera must start taking the picture at the exact beginning of each new minute (when SEC = 0 and TICK = 0), that the camera takes exactly 10 lines of data each second, and that the line sweep requires 90 milliseconds followed by a 10-millisecond period during which the vidicon beam "flies back" to the beginning of the next line. The control loop for this operation is illustrated in Figure 4.

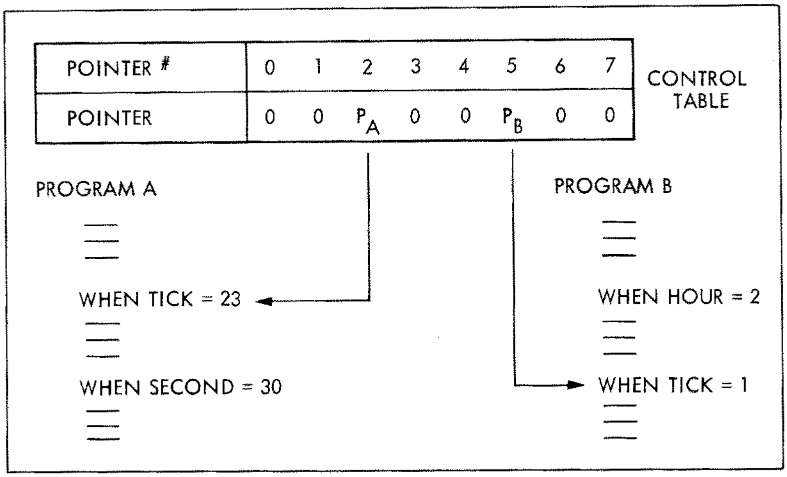
```
WHEN SEC = 0, TICK = 0
DO FOR LINE = 0, 1, 2,. . . , 799
        START LINE READOUT
        WAIT 36 TICKS
        STOP LINE READOUT
        START FLYBACK
        WAIT 4 TICKS
ENDDO
```

Figure 4. Design Language for Spacecraft Control Applications

All application programs written in this fashion are run under an executive. The WHEN and WAIT constructs are CALLs to WHEN and WAIT subroutines which operate as part of the executive. Each time an application program comes to a WAIT or WHEN, the executive takes over, remembers where control came from, and, on each subsequent TICK, tests whether the conditions stated on the WAIT or WHEN are satisfied. Only when they are satisfied does the executive return control to the application program at the location immediately following the stated test conditions.

Using this scheme, the executive runs, in any tick, only the short segment of a given user program between two logically consecutive WAIT or WHEN statements. The programmer of an application program never needs to be aware that this is going on, but it allows his programs to time their operations and simultaneously enables the executive to maintain control so it can run several application programs concurrently [3].

The only control table needed by the executive is shown in Figure 5 and consists of a set of eight pointers. Each application is assigned to one pointer. If the application program is not running, its pointer is zero. If it is running, its pointer points to the test which must be satisfied before control is returned to the program.



POINTER #	0	1	2	3	4	5	6	7
POINTER	0	0	P_A	0	0	P_B	0	0

Figure 5. Control Table for Concurrent Program Operation

C. Spacecraft Simulation

The primary product of a spacecraft simulation is a real-time simulated telemetry stream. Months before a spacecraft is launched, teams of engineers and scientists spend hundreds of hours rehearsing their roles in monitoring and controlling its operation. The effectiveness of these rehearsals depends on realistic simulation of the spacecraft and its telemetry stream.

Spacecraft simulation is mostly a discrete problem. Commands arrive at the spacecraft and cause state changes; instruments turn on and off; the telemetry mode changes. All these are discrete events which can be queued and executed in a predetermined order. But if it is to be realistic, spacecraft simulation must also involve things like the three angles of spacecraft attitude, the battery charge, and the instrument temperatures, all of which change continuously. Continuous variables may be calculable directly as functions of time, but more often they are obtainable only as solutions to a simple set of differential equations. So the spacecraft simulation problem is essentially a continuous/discrete problem [4].

One complication in the computational control of simulation arises from the magnitude of the programming task. Programs which simulate given spacecraft subsystems are called "models," and usually 10 to 20 models running simultaneously are required to produce an effective telemetry simulation. A given programmer/analyst is often assigned to write and check out one to five models. Generally, no one programmer/analyst understands all the ramifications of certain spacecraft events.

For example, a program simulating subsystem A may set a logical variable with a given name. That logical variable may affect operation of subsystems B, C, D, and E. The programmer of the subsystem A model may know about some of these instructions, but not all. If he neglects to notify any subsystem model that he has changed a logical variable which affects it, the ramifications of the change will not be propagated correctly.

A second complication in this kind of problem is that there are often a variety of simple actions which must be performed at specified values of the continued variables. In simulating a spacecraft attitude control system, for example, there is a continuous variable — usually a combined position sensor and rate sensor output — which determines whether or not the tiny cold-nitrogen jets which control spacecraft attitude should be on or off. The jet turns on when this variable reaches one value and then shuts off when it reaches another. Simulating the turn-on or shut-off of the jet is a simple matter of changing the value of a parameter used in the differential equations which describe the spacecraft's rotatory motion. But that action must be taken at exactly the right point, and it is not possible to predict in advance the time associated with that point.

A third complication peculiar to spacecraft simulation is that simulated time must be kept close to real time, though not as close as might at first be thought necessary. The reason for this is illustrated in Figure 6. At distances where interplanetary spacecraft spend most of their time (lunar distances or greater), it takes many seconds for a command to pass from the transmitter on earth to the receiver on

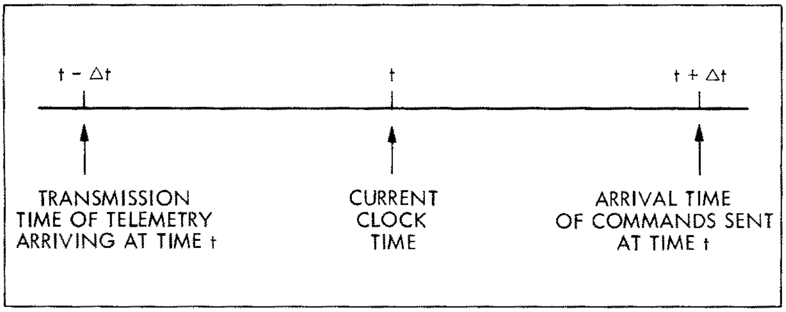

t - Δt
t
t + Δt
TRANSMISSION TIME OF TELEMETRY ARRIVING AT TIME t
CURRENT CLOCK TIME
ARRIVAL TIME OF COMMANDS SENT AT TIME t

Figure 6. The Real-Time Window

the spacecraft. Telemetry changes resulting from any command require a similar time to return to earth. Therefore, it is necessary only that simulated time τ remain between $t - \Delta t$ and $t + \Delta t$, where t is real time and ΔT is the light travel time. This scheme requires that commands sent at a time t_o be buffered until $\tau = t_o + \Delta t$ and that telemetry data produced at τ_d be buffered until $\tau_d = t - \Delta t$. But the rate of command transmission is very low, and telemetry data buffers can be kept small by maintaining τ near $t - \Delta t$.

Effective solution of the computational control problem for spacecraft simulation requires three major elements — one for each of the complications described above.

To handle the complexity and interaction problem, models are organized as shown in Figure 7. The first part of each model handles initialization of all continuous and discrete variables associated with the subsystem. The second part handles discrete variable calculations. Using a method reminiscent of the General Simulation Language (GSL), this part of the model has many entries [5]. The code at each entry normally executes one simple action (event). Figure 7 shows only three entries (10, 20, and 30) for ease of illustration. The third part of the model calculates the continuous variables associated with the subsystem.

Propagation of effects between models is allowed for by providing a class of logical variables which must always be set or reset with the statement

```
CALL LSET (LV,X)
```

where LV is the logical variable and X gives the value (0 or 1) to which LV is to be set. The subroutine LSET changes LV, then checks a list associated with the variable LV to see how to propagate the effects of the variable, and, in effect, enters the

```
                    INITIALIZATION
      SUBROUTINE ACSI
      ______
      ______
      ______

      RETURN
- - - - - - - - - - - - - - - - - - - - - - - - - -
                 DISCRETE EVENT HANDLING
      SUBROUTINE ACS
      GO TO (10, 20, 30) K
   10 ______
      ______
      ______

      RETURN
   20 ______
      ______
      ______

      RETURN
   30 ______
      ______
      ______

      RETURN
- - - - - - - - - - - - - - - - - - - - - - - - - -
             CONTINUOUS VARIABLE CALCULATION
      SUBROUTINE ACSX
      ______
      ______
      ______

      RETURN
```

Figure 7. Structure of a Simulation Model

appropriate discrete event handlers with the correct control parameter k. The list is constructed automatically as the result of calls of the type

CALL LTAB (LV,T/F/B,M,K)

where the cases T, F, and B are defined as:

Case	Meaning
T	call model M with parameter K when logical variable LV goes from F to T
F	call model M with parameter K when logical variable LV goes from T to F
B	call model M with parameter K with logical variable LV changes

Most calls to LTAB are made at initialization time, but they can be and often are made from the discrete event subroutines.

The second major element is required to allow programmers to easily specify and achieve the execution of events at given values of continuous variables. To specify that an action be taken at a given time, the programmer writes:

CALL TSET (T,M,K)

Here T is the value of time (t) at which the action is taken, and the code which performs the action is at entry K of the discrete event handling portion of model M. To specify that an action be taken when a continuous variable V becomes equal to C, the programmer writes:

CALL YSET (V,C,M,K)

The set of items (V,C,M,K) is called a "Z-trigger," and the point at which V = C is called the "Z-trigger 0-point."

The third major element (required to keep simulated time close to real time) is the computational control algorithm itself. This algorithm is shown in Figure 8. The 0-points of Z-triggers are not easy to find, and the search requires iteration. The complexity of this search is completely represented by step 4 in the algorithm of Figure 8.

III. STEPS IN DESIGNING COMPUTATIONAL CONTROL SYSTEMS

System requirements are elusive, and gathering and documenting requirements is a major part of any system development. But the problem of requirements gathering is a subject in itself and too complex to be covered here.

So this paper assumes that a reasonably good statement of the requirements is available and discusses the standard system design steps that remain, emphasizing peculiarities and regularities which arise from restricting the problem class to that of computational control problems.

A. Analysis

The first task of the system designer is to establish the elements from which the system will be constructed. In the case of the mosaic tiling system, for example, the elements are the individual pixel generators in the form of closed subroutines, the control table, and the system control program which uses the pixel generators and the table to construct the picture lines. In the case of simulation, the elements are more complex. There is still a system control program, but it is far more complicated than that for mosaic tiling. Instead of pixel generators, there is a set of models, but each model has a complicated internal structure which needs to be understood in

```
    LET $H$ BE THE MAXIMUM STEP SIZE FOR CONTINUOUS VARIABLE CALCULATIONS
    LET $\delta$ BE THE ONE-WAY LIGHT-TRAVEL TIME
    LET $t$ BE REAL TIME
    LET $\tau$ BE SIMULATED TIME
    LET $\tau_L$ AND $\tau_R$ BE THE LEFT AND RIGHT ENDS OF A SIMULATED TIME INTERVAL
    LET $T_{NEX}$ BE THE SMALLEST T FOR ANY UNSATISFIED TSET CALLS
    LET $\epsilon < \delta/2$
0.  INITIALIZE ALL MODELS
1.  SET $\tau_L = \tau$, $\tau_R = MIN(\tau + \delta/2, \tau + H, T_{NEX})$
2.  WAIT UNTIL $\tau < t - \delta + \epsilon$
3.  CALCULATE CONTINUOUS VARIABLES AT $\tau_R$ FOR ALL MODELS
4.  IF ANY Z-TRIGGER 0-POINTS LIE BETWEEN $\tau_L$ AND $\tau_R$ ADJUST $\tau_R$ UNTIL IT IS THE
    FIRST Z-TRIGGER 0-POINT
5.  INTERPOLATE AND BUFFER TELEMETRY FOR INTERVAL $(\tau_L, \tau_R)$
6.  PERFORM ACTIONS ASSOCIATED WITH ANY TRIGGERS AT $\tau_R$
7.  REPEAT FROM STEP 1.
```

Figure 8. Computational Control Algorithm for Spacecraft Simulation

detail. Instead of a single control table, there are T-triggers, Y-triggers, and lists associated with logical variables.

It is important to note that in spite of the diversity of applications in Sections II-A through C, the systems have a certain regularity of structure. In each case, there is a control program which runs the system, one or more control tables used by the control program to determine what to do next, and a set of slave programs (pixel generators, application programs, or models) which do the desired calculations under the direction of the control program. These elements can be expected to appear in almost any computational control system, and recognition of this fact gives the system designer a quick start in drawing his system diagrams.

In the mosaic tiling example, the control table completely defines the sequence of calculations from start to finish. Sequencing does not depend on calculated results. In the spacecraft control and simulation examples, however, sequencing by the control program depends on the results of calculations or real-time interactions and cannot be predicted in advance. As a result, each of these last cases requires the development of mechanisms by which the user can communicate new sequencing requirements through the slave programs to the control program. In each case, these mechanisms involve language elements. In the spacecraft control case, the language elements used are the WHEN and WAIT. In the simulation case, the language elements used are CALL TSET, CALL YSET, CALL LSET and CALL LTAB.

These language elements can be thought of as statements which pass necessary sequencing information to the control program. In each case, they translate as simple CALLs to subroutines. Only in the case of the WAIT and WHEN is the control more complicated. These cases, too, are CALLs to subroutines, but the subroutines do not return control immediately as subroutines normally would.

So the designer can expect that if the sequence of control depends on calculated results, he will need some simple language elements to pass sequencing information from the slave programs to the control program. He can also expect the language elements to translate to subroutine CALLs.

B. Synthesis

Synthesis is the process of putting the diverse elements of the system back together again, and consists primarily of designing the algorithms used by the control program. Even though analysis and synthesis seem like entirely different activities, they are so mixed together in early design stages that it would be almost impossible to say where one began and the other left off. They are like two sides of a coin of design that the designer flips from time to time to decide which to do next. Consider, for example, developing an algorithm to control operation of application programs in the spacecraft control system. The question immediately arises: How does the control program know when the application programs need control again and to what location control should go? Obviously, some kind of control table is needed. But there are countless table structures which can be used, and each different table structure requires a different algorithm. Deciding which tables to use and what structures they should have must be done concurrently with deciding on the algorithm to be used in the control program. One effective procedure involves writing down a clear picture of a control table, sketching an algorithm which uses it, and then looking for ways to simplify and speed up the algorithm by changing the control table. Several iterations of this procedure with constantly increasing precision of definition usually suffice to produce something good enough to serve as a starting point for the system architecture.

There are many ways to depict algorithms. The use of flow charts is an old, well understood technique and is perfectly acceptable. A better system involves the use of structured languages such as "Program Design Language" and the processor developed by Caine, Farber, and Gordon, Inc., of Los Angeles [6]. Restricting programming structures to DO, IF, and CALL generally leads to better program organization than allowing unrestricted branching. Structured English language statements are far more compact and efficient than flow charts, and — given a processor — far easier to change.

Users of structured languages should always be aware that the rules of structured usage are intended to simplify programming structures and make them easier to understand. Whenever adherence to structured usage clearly complicates programs or makes them more difficult to understand, the rules should be violated. A good example of this violation occurs in the WAIT and WHEN statements of the spacecraft control example. These subroutines do not return control to the user in the normal way, but instead return control to the control program.

C. Experiment

The system designer eventually completes his analysis and synthesis, has his control tables structured, has a specification for the language elements (if any) used by the slave programs to communicate control information back to the control program, and has some form of representation of the control program algorithm. It would seem that the next step should be coding. It probably is not. The next step should usually be for the designer to put himself in the shoes of a user unfamiliar with the system and try to develop some sample slave programs for the system. This may turn out to be just as easy as the designer thinks it will be. If so, it will not take long. But the designer may run into some surprises — particularly if he is lucky enough to be able to talk a friend into trying to develop slave programs. Because of his unfamiliarity with the underlying assumptions concerning the problem, the friend may try to do things which never even occurred to the designer.

An example of the kind of surprises which can hit the system designer at this stage occurred during the design of the spacecraft control system. Originally, the WAIT statement had been designed to specify the wait duration in terms of the number of minutes, seconds, and ticks. These are the elements of system time and are used in the WHEN statement. Using them in the WAIT statement seemed to give the system a pleasing uniformity. But as soon as slave programs were written using the WAIT system, it became apparent that the natural unit for the user to specify was the total number of 2.5-millisecond ticks he wanted to wait, even if that number exceeded 400 (1 second).

D. Articulation

The importance of letting others know the details of a proposed architecture varies greatly with the circumstances of the development. In the mosaic tiling system, there were no anticipated users of the system beyond the designer himself, and the primary concern was simply to get the job done fast. The system was already running before any significant amount of documentation was done, and the documentation, even when published, was ignored.

The other extreme is illustrated by the simulation example. Publication of the first draft of the system design in that case raised a storm of technical and managerial controversy which did not settle down for months. A second draft of the system design document with a considerable number of fundamental changes was required to settle the dust of the technical arguments, but formal management meetings and full-scale slide presentations of the system design were required to finally resolve the managerial disagreements.

So the system designer must try to anticipate the need for communication even as he designs the system. There is almost always a need for some communication of design ideas to some audience, and if the designer keeps that in mind as he draws his system diagrams, he may be able to make them usable for future presentations to his peers.

E. Iteration

It is not enough to iterate the analysis and synthesis during the early design phase, or even to iterate a system design based on inputs from a design review. It should always be planned, if time permits, to iterate a system at least once after it has been completely designed, implemented, and used. The importance of iteration cannot be overstressed — it is the essence of design. Systems of significant complexity can seldom be designed right the first time. A little experience with a system almost always makes glaringly obvious things which could be seen, if at all, only with peripheral vision during design.

Sometimes, because of schedule pressures, it may not be possible to iterate, and the system is left with glaring flaws. The mosaic tiling system contains a good example. That system was implemented so that any single picture line which deviated by even one pixel from the known line length caused a diagnostic message to print and processing to stop. That sounded like a good idea when thought of, but it meant that normally only one error per computer pass could be caught. If the entire picture had been allowed to complete, the finished copy could have pointed out errors in many pixel generators simultaneously.

Although the system does work, system redesign is usually valuable just because several benefits are gained by the redesign. The spacecraft control system affords a good example. This system was originally implemented with a WHEN statement which assumed that each item of system time was normally to be tested and used control bits to tell which items to ignore. Once the system was being used, it became obvious that the WHEN statement should simply name each item to be tested. This made the system independent of the time base used and even allowed programmers to say, in effect,

```
WHEN FLAG F IS SET
CONTINUE
```

where flag F may be set by another program or by a data transmission from another machine. The resulting system was easier to use, was far more flexible, and ran faster than the old one.

IV. CONCLUSION

Systems for computational control almost always consist of a single control program, a number of slave programs, and one or more control tables. The problem of analysis is to identify these elements and their structure. The problem of synthesis is to determine a simple way to tie the elements together with a foolproof algorithm for the control program. Usually, the algorithm needs to be flexible enough to handle an entire class of similar problems. Analysis and synthesis must be tackled together and constitute the early design stage.

But a first design, untested by the fires of criticism and use, seldom represents the designer's best work. Even before his design is cast in code, the designer needs to put himself in the user's place by writing slave programs and testing alternative architectures. He needs to explain his work to an audience of peers and solicit their helpful and honest criticism. And even when a first version of the system is running, the designer should, if time allows, plan to do a final iteration, for experience is an exacting referee, and iteration is the name of the game.

ACKNOWLEDGEMENT

This paper presents the results of one phase of research carried out at the Jet Propulsion Laboratory, California Institute of Technology, under Contract No. NAS7-100, sponsored by the National Aeronautics and Space Administration.

REFERENCES

1. David A. Rennels, Borge Riis-Vestergaard, and Lance C. Tyree, "The Unified Data System: A Distributed Processing Network for Control and Data Handling on a Spacecraft," *NASCON Conference Proceedings*, May 1976.

2. David A. Rennels, "A Distributed Microprocessor System for Spacecraft Control and Data Handling," *MIDCON/77 Conference Proceedings*, 1977.

3. Hansen Per Brinch, "Concurrent Programming Concepts," *Computing Survey*, Vol. 5, No. 4, December 1973.

4. Fred Lesh, "A Continuous/Discrete Simulation for Interplanetary Spacecraft," *Proceedings of the Seventh Annual Pittsburgh Conference on Modeling and Simulation,* April 21-22, 1977, pp. 243-247.

5. Donald G. Golden and James D. Schoefflen, "GSL - A Combined Continuous and Discrete Simulation Language," *Simulation,* Vol. 20, No. 1, January 6, 1973, pp. 1-8.

6. Steven H. Caine and E. Kent Gordon, "PDL - A Tool for Software Design," *AFIPS Conference Proceedings,* Vol. 44, pp. 271-276, National Computer Conference, 1975.