Structured Programming and Software Engineering

Of Hard-Real-Time Minicomputer Systems

By Robert C. Tausworthe

Abstract

This paper describes a structured approach to the development and implementation of real-time minicomputer software systems. The method adapts sound, proved engineering techniques into software project engineering terms. These software engineering practices presume technical expertise as a necessity, but also utilize top-down design and implementation principles, structured programming concepts, concurrent documentation and implementation activities, management visibility and control mechanisms, software standards, and coordinated project teamwork.

1. Introduction

Real-time program system requirements for today's high-technology minicomputer applications, such as deep-space station tracking, telemetry, command, monitoring, and other functions, together with the lack of a suitable high-level language, practically dictate the use of assembly language coding to achieve needed speed and storage efficiency. On the other hand, top-down structured programming methodology promises product reliability and a host of other potential benefits, could one but bridge the gap between the high-order abstractions at the top and the assembly language code at the bottom. Since assembly language coding is most susceptible to programming error, it stands to reason that programs which must be coded at this low language level stand to benefit the most from a structured approach to development.

Minicomputer real-time systems applications are typified by distributed parallel processing using several function-dedicated mini's, with each subsystem mini processing several interrupt-actuated subfunctions in concurrency. Each mini seldom has more than about 32K words of core memory, and mass storage is either limited and slow, or non-existant. Processing speed is usually critical, and core limitations are constraining. Programming is frequently performed by only a few individuals per subsystem, one of whom may be in charge of the hardware portion of the subsystem, as well. The total system development may require upwards of 2 years and involve 50 to 100 people. System lifetimes may extend 10 years or more. In deep space and other applications, delivery dates are critical, and funding is inelastic.

Even though each of the subsystems can be developed internally almost autonomously, there are yet numerous hardware and software interfaces and protocols, all to be defined, coordinated, and enforced, that make for large numbers of human and machine interactions which must be resolved before those internal matters can even be considered. An engineering approach is clearly warranted if project success is not to be an accident.

I do not intend this paper to be a treatise on the engineering method. Some highlights of a useful software engineering method pertaining to minicomputer systems development may prove interesting, however.

Resource Estimation and Work Breakdown Structure

I read somewhere once a statistic indicating that more software projects go awry for lack of schedule time than for any other resource, and that software developments almost always have cost overruns. Such statistics are probably symptomatic of non-management, to a greater extent than lack of experience in estimating. Few project engineers in any discipline could predict schedules and costs accurately without a careful analysis of all the factors involved.

The top-down approach to software engineering has a benefit aside from those advertised in respect to the reliability it brings to programs: namely, in project resource estimation. The same hierarchy by which program functions are architected into increasingly refined layers of detail provides a basis for the work breakdown of implementation tasks. All activities and resources must be identified in this breakdown and refined by study until all the significant cost and schedule drivers

---

*This work was performed under Contract No. NAS 7-100, National Aeronautics and Space Administration

**Jet Propulsion Laboratory, Pasadena, California
have been determined and their effects evaluated. By amalgamation, if necessary, work can be partitioned into finite manageable tasks of about the same duration each, each with a defined responsibility, set of work requirements for commencement, and work deliverables which signal the completion of that task.

The function of the architectural design is therefore three fold: first, to determine all the structural details of the program as implied by requirements, so that surprises during the detail design and implementation phase are minimized; second, to provide a rendering of the program capabilities suitable for review by customers, sponsors, or users; and third, to develop the work-breakdown information necessary to estimate costs, set schedules, allocate resources, and later, monitor performance.

**Project Visibility**

Management and technical decisions can rarely be made reliably without information in the proper amounts, of the proper kinds, and provided at the proper times. Top-down, structured programming with concurrent documentation provides an inroad toward technical reviewability of the deliverables, and affords some management visibility, as well, when keyed to progress of tasks in the work breakdown structure.

Statistical methods teach us that predictions of a project completion date based on measured total milestone achievements (Fig. 1) can be made accurately (within a predefined error) only if the number of measurable milestones (assumed distributed uniformly by plan over the project lifetime) exceeds a calculable lower bound. For a 10% accuracy estimate to be made one-fourth the way through a project, there must usually be at least 111 about 400 milestones planned, to be reported upon (the number is inversely proportional to the square of the tolerable estimation error).

The detailed work breakdown structure provides the list of tasks, deliverables, and conditions for completion. It thus provides the vehicle for meaningful progress reporting. Compleitions of tasks are reportable milestones, so the minimum number of needed task descriptions can be determined from project schedule variance goals. Management action can be applied to distribute resources to correct any deficient performance areas (Fig. 2).

**Real-Time Structured Programming Conventions**

The principles set forth in Brinch Hansen's text, *Operating Systems* [2], form the basis of structured real-time programming practice. The simple "FORK-JOIN" concurrent program control (Fig. 3) structure resembles the IFTHENELSE used in structured sequential programming (Fig. 4), except that both activities within the structure execute in one case, and only one in the other. The need for restricted use of the "go to" in sequential programs is mirrored by the need for restricted entry into concurrent modes, discriminant activities among concurrent processes, and recognizable reentry into the sequential mode.

Interrupt-driven processes form an important class of program elements rife in minicomputer applications. A simple "WHEN" structure (Fig. 5), much like FORK-JOIN and IFTHENELSE, can be used effectively to restrict indiscriminant arming/disarming of interrupts and standardize passage of control on exit from interrupt routines.

Just as sequential programs sometimes benefit from an occasional unstructured passage of control (escape), concurrent programs sometimes need to deviate from FORK-JOIN and WHEN structures. Such deviations are typically traceable to a need for execution of alternate code (Fig. 6), conditional initiation of concurrent processing, or the initiation of a concurrent mode inside one module, with sequential mode resumed in another (Fig. 7). Relegation of such deviations to exceptional and rare instances, and careful documentation in those instances improve reliability, understandability, and maintainability of the program.

**Documentation Conventions**

Due to the extreme low level of assembly language, program structural information needs to be communicated to a would-be reader of the program in a more deliberate way perhaps than would be necessary had the program been written in a high-level language with explicit structured syntax. Flowcharts and software design languages (SDL's), each augmented by additional descriptive narration, are effective means of describing program structure in such cases. Special flowchart conventions, such as those in Figs. 4 through 8, are need to accommodate concurrent real-time process control functions, such as interrupt handling, process initiation (both conditional and unconditional), restricted forms of unstructured forks and joins, and the like.
Data Protection

The use of data structures shared by two or more concurrent processes must be synchronized if consistent results from the program are to be achieved. A "consistent" program is one that outputs results that are repeatable in a practical sense, even if there are errors in the program. Data synchronization alone does not produce consistent programs, any more than structured programming produces correct programs. But the mere awareness that means for synchronization and consistency must be provided, either through the employing system's program design, promotes programming practices which permit data structure integrity to be evaluated and demonstrated in a reasonable way.

Anomaly Behavior

Almost every software project enters into a phase where it is "90 percent complete", in which it remains seemingly for a very disproportionate length of time before the true completion. Much of this time is spent, as it turns out, discovering and repairing anomalies in the program, operations manuals, or program requirements. The numbers and kinds of anomalies in a software package are matters of fact and not matters of probability; however, since only a relatively small portion of a large program's documentation and response can ever be verified in a practical sense, the process of discovering anomalies appears to be a random process.

Repairing an anomaly requires study, software alteration, and then verification. Because the differing kinds of anomalies exhibit a range of difficulty, and because human interaction is always required, the repair rate also appears to be a random value. I shall only address discovery statistics in this discussion, but discovery and repair models may be found in [3].

One popular model [3,4] for modeling anomaly discovery is the following: the probability of discovering a new anomaly at any instant of time is proportional to the number of anomalies remaining and the applied effort. Testing to find such anomalies in this case assumes that the discovery is not so rapid as to swamp out the efforts of the test team. The anomalies are assumed sparse and independent -- a condition usually well fulfilled during the final tests of well structured software. In fact, although intuitively based, these assumptions appear to be well supported by empirical data, as will be seen a little later.

Figure 9 shows the general theoretical form of discovery status expected as a function of time (assuming constant effort is being applied to testing). The initial find-rates, aA, have been set equal for all the cases shown. Figure 10 then shows the fractional variation in the time required to discover n anomalies about the expected time. Note that the variance is fairly well behaved (decreasing as 1/n, like that of an ordinary random process) until about 85 percent of the anomalies have been found. Then the variance rises very sharply. The conclusions which may be reached from these two sets of curves will be addressed shortly.

The discovery rate is, of course, dependent upon the applied effort, as well as time. It is typical that the effort profile is variable for a number of reasons among which are manpower phasing, availability of computer resources, and availability of software resources. During the early testing, there is usually a lower level of effort being applied toward anomaly discovery than later on, because "things are getting up to speed". Effort is being diverted to planning, coordination, and resource acquisition. Toward the end, effort may drop again, to the level supported by sustaining or operational personnel.

The effects of varying work load can be accommodated in the constant-effort model by merely substituting accumulated man-days for accumulated days. The effects seen in Figs. 9 and 10 are stretching and shrinking of the time axis as the effort profile drops and rises, respectively.

Even when work profile effects are thus factored in, it is frequently the case that the discovery of anomalies takes place in varying environments. This is especially true of real-time systems, where program development is typically done outside the final, full-up operating environment. Simulation of actual interrupts, events, data, and other physical phenomena are used to validate operation of the software prior to installation in its final operating environment.

As a result, those anomalies which are due to unmodeled real-time interactions among processes are undiscoverable by any means until the programs are integrated into their final environment. In such situations, only a portion A1 of the total anomalies A will be found during the first stage of tests, no matter how long testing goes on. If the second stage takes place in the final environment, then A-n become discoverable during this stage.
Typical projects may find only about one-half to two-thirds of the total anomalies in the first stage. This phenomena is illustrated in Figure 11 which shows the actual anomaly history of a large (200,000-line assembly language) system with theoretical curves superimposed. The closeness of fit attests to the validity of the model for prediction purposes. It also shows the correct management decision was made to install the software in the operational environment when discovery in the simulated environment began to level off.

Inasmuch as the model is probably an optimistic one, results based on it tend to point out things which can and cannot be accurately deduced. For one thing, the model shows that one cannot predict accurately the number of total anomalies until a large fraction of them has already been found. It also shows that the final 10-15 percent of the anomalies will require considerably more effort to find than the first 85-90 percent, and the variation in the discovery effort over this interval is likely to be widely distributed about the expected effort. Project budgets and schedules must include resources to accomplish the expected as a matter of course, and must provide contingencies to account for variations in the mean.

Conclusion

A top-down, structured approach to real-time minicomputer software development takes steps to align sound engineering principles with the special considerations required for software implementation. These special considerations include provisions for both technical visibility (readable, understandable documentation and meaningful design reviews) and management information (status and progress, resources, performance, etc. Technical and documentation standards and disciplines for implementing program control and data structures (including those necessary to expand structured programming philosophy to accommodate real-time computation), promote team communication and uniformity of deliverables.

Quantifiable visibility is the only rational basis for sound technical decisions and competent management action. Work breakdown structure, PERT, critical path analysis, rate charts, and other similar management tools have been successfully and widely used in hardware projects for resource estimating, project planning, and progress monitoring. Software projects are beginning to use these techniques to the same advantage.

In principal then, minicomputer software project engineering is, in many respects, inherently no more difficult a task than engineering the associated hardware system, when the same care and effort is given to technical and managerial visibility requirements.

References


Figure 1. Best-fit line to cumulative milestone data, variance area, and typical performance.

Figure 2. Actual performance showing effect of management action taken in June to accelerate completion.

Figure 3. The FORK-JOIN concurrent processing structure (ANSI standard flowcharting display)

Figure 4. Canonc non-real-time structured programming forms.

Figure 5. The WHEN structure for flowcharting trap procedure interrupting a background program

Figure 6. Alternative process selected by dedicated trap actuation resembles IFTHENELSE structure.
Figure 7. Flowchart structure of unconditional and unconditional concurrent processes initiated within module X.

Figure 8. Flowchart of module X showing entries into concurrent modes.

Figure 9. Expected anomaly discovery progress as a function of time (A total anomalies)

Figure 10. Fractional variance statistics during anomaly discovery.

Figure 11. Cascaded testing and level of effort effects on anomaly discovery in Mark-III Data System project (X = Aa).