New Paradigms For Automating Batch Job Processing

Michael A. Casteel
Unison Software
Mtn. View, CA 94043

Since the introduction of the Series II in 1976, the HP 3000 has steadily gained installed base and stature as a business computer. It is no longer just a minicomputer—the Series 70 delivers mini-mainframe power, and HP's new Precision Architecture offers the potential for full-scale mainframe performance in a supercharged HP 3000 chassis. More and more HP 3000 users find themselves running a full-fledged data center, and their companies' demands for computing productivity translate into demands on the computer operations staff.

Like their counterparts elsewhere in the company, data center and operations managers are now looking to the computer to aid them in the performance of their work, and they're using a new breed of software designed for data center management. The operations staff, like new computer users in other departments, must learn and adjust to new ways of doing their jobs. This is never easy, but change is necessary in the pursuit of greater efficiency and productivity. This paper attempts to ease the way by presenting the principles involved in automating one of the most important operations tasks: control of batch job processing.

Although clearly not conceived as a batch processor in the traditional sense, the HP 3000 has had to assume a batch processing burden commensurate with its growth as a general business computer. A number of HP 3000 data centers report they are now processing more than 20,000 batch jobs per month, with the jobs subject to many interdependencies. How can this volume of batch processing be reliably accomplished with the MPE :STREAM facility?

The answer is, it can't. The basic MPE facility only allows the scheduling of jobs based on time dependencies, e.g., hold a job until 8:00 p.m. before running it. There is no built-in provision for MPE running a job every day, every month, or every quarter, nor a way to ensure that a sequence of jobs is run in a certain order. In order to handle all their batch processing, HP 3000 users have resorted to a combination of manual operation and software tools such as SLEEPER from the INTEREX Contributed Software Library. However, this combination is inadequate to handle the jobs on a single Series 70, where six, eight, or even more jobs must be kept running to effectively utilize the machine's capacity.

Over the past few years specialized software tools have become available, mostly from third-party software suppliers, which provide the automation necessary to support production batch processing. These tools put the computer to work for the data center operations staff, just as existing applications serve the end-user. This means the operations staff must adapt to a new working environment, as the end-users already have. This paper presents five major functions which are fundamental to batch job automation, regardless of the particular implementation. They are:

scheduling	- to determine the jobs to be run on a given day
sequencing	- to ensure the jobs run in the correct order
constraining	- to keep conflicting jobs from interfering with one another
synchronizing	- to coordinate with external events
recovering	- because something will still go wrong, at least occasionally.

Scheduling

The first step in production batch processing is scheduling; that is, deciding which jobs are to be run this day. This is also the first, and perhaps most difficult, task to automate. It often takes many months before all the production jobs in a large shop can be identified and cataloged into an automated scheduling system, a phenomenon which underscores the need for automation in such shops.

Of course, not all jobs are scheduled in advance. For example, a programmer's compile job runs when the programmer has completed work on a section of a program. In a production environment, however, hundreds of jobs are scheduled on some regular basis, e.g., every Friday, every work day, or every month-end.

One of the most challenging problems in scheduling regular production jobs is the definition of the *calendars* used to determine when they run. Some processing might be scheduled with regard to the ordinary calendar (e.g., every Friday), and some by an artificial calendar, such as month-end processing for a calendar where each 13-week quarter contains one 5-week and two 4-week months. A shop may require references to several calendars in order to schedule, say, Manufacturing, Financial, and Payroll month-end. The scheduling task is likely to be further complicated by holidays, which may cause processing to either be skipped or rescheduled before or after the holiday.

An automated scheduling system must therefore provide for several different calendars, customizable to the company's holidays and other special requirements. Setting up the calendars is a prerequisite to scheduling jobs properly. Before trying to automate batch job scheduling, it is important to identify the scheduling calendars used, and then study the calendar functions offered in the automated scheduler to obtain the best fit.

In order to handle irregular jobs, which don't fit any calendar, it helps to be able to specify a list of specific dates on which to run each job, rather than calendar intervals. This way, if it is at all possible to predict the schedule on which an irregular job is to be run, it can be documented in the scheduling system and processed automatically. Examples of such scheduling are jobs which run at the Full Moon (I don't know of any automated scheduler which includes the Lunar calendar!), or jobs which run two weeks before each Interex conference.

Of course, every shop is likely to have a substantial amount of *ad hoc* scheduling, where user departments submit jobs or job requests on some basis known only to themselves. To minimize overhead, it helps if the automated scheduler will allow users to submit their requests directly to the scheduling system rather than to the operations staff. Once the users have been trained, their special processing requests can be integrated into the production schedule automatically, while operations concentrates on monitoring and controlling the batch process.

Another benefit of automated scheduling is the ability to schedule jobs *reliably* over long intervals, particularly quarter- and year-end processing. Manual scheduling of infrequent jobs is complicated by lapses of memory and staff turnover, while the computer never forgets.

Sequencing

Once the scheduler has determined *which* jobs should be run, the next step is to define the correct processing sequence. Although some jobs are essentially independent of others, it is critical that most jobs be executed in the proper sequence with respect to other jobs. Consider a batch update and reporting application, in which the processing sequence is: First, a backup (in case something goes wrong); then, an update job posts the

batch transactions to the database; finally, a number of report jobs analyze the updated database.

This case illustrates the need for proper sequencing. If the update does not follow the backup, either it will fail because the database is tied up by the backup, or the backup will be useless because it did not get a complete copy of the database before updating. Furthermore, if the reports do not follow the update, they will either fail or produce incorrect figures.

The essence of job sequencing is this: A job must follow another job if it uses the results of the first job's processing (e.g., reports follow the update), or if it modifies a file which is required by the first job (e.g., update follows the backup). If a job does not use another job's output or erase its input, then it usually needn't follow it; it may be that the two jobs must never run at the same time, but that is a *constraining* issue and is covered in another section.

From this it may seem that the simplest way to define the processing sequence would be to list jobs in the proper order. For example, we would simply list, the backup first, then the update, and then the reports. By following this list, the computer operator can correctly process the jobs, one at a time (assuming nothing goes wrong). Many shops use just this approach for existing manual or semi-automated operations, and MPE even supports it: Just set the job limit to 1 and stream the jobs in the correct order.

Of course, ideally we want the HP 3000 to handle more than one job at a time. After all, it is a multiprogramming computer system. If we have several independent applications, such as accounting and manufacturing, each can have its own list of jobs to be processed independently. Although MPE offers little assistance, a number of such lists could be maintained and executed, depending on the skill of the operator.

But the simple list scheme becomes complicated when we try to take advantage of multiprogramming within one application. In the backup/update/report example, it may be that some or all of the report jobs can be run at the same time. What is needed is a simple means of maintaining job sequence dependencies with the flexibility to permit multiprogramming whenever possible.

Automated job processing typically uses an approach with maximum flexibility: For each job you specify which other job(s) must come before it. If we use the word FOLLOWS to signify this relationship, then we can easily express job sequences as, "UPDATE FOLLOWS BACKUP," and "REPORT FOLLOWS UPDATE." In the example we have been discussing, there may be several reports, all of which "FOLLOW UPDATE."

Clearly, this approach makes it easy for the computer to achieve the highest possible degree of multiprogramming. At any given time, the machine can process all jobs not waiting to "FOLLOW" jobs not yet complete. This approach also accommodates complex interdependencies, in which a job is dependent on *more* than one other job. A report may combine input from two applications, for example, and it need only "FOLLOW" update jobs in each application. This kind of job would be impossible in the listkeeping scheme, but is simply and elegantly handled by stating the individual job dependencies.

As it can take considerable effort to identify all the production jobs to be cataloged into the scheduling system, it can be even harder to determine the correct sequencing rules. One of the disadvantages of the simple list so often used is it doesn't reveal precisely *why* jobs run in a particular order, and this can make it difficult to recognize multiprogramming opportunities. If the list does not distinguish whether the sequence is due to true dependencies or simply because some jobs cannot be run together, it can take a long time to find out the necessary facts. Once rules are determined and then docu-

mented in the automated system, job throughput can be optimized. The information can also be invaluable in application system maintenance.

It is usually not enough simply to cause Job B to FOLLOW Job A. Usually, we require that Job A complete *successfully* before we can permit Job B to run. In our example, the reports should not be run if the update job terminated abnormally, due to something like a work file reaching its capacity. Instead, dependent jobs should wait until the problem has been corrected and necessary processing completed. Ideally, the batch processing software should signal the problem to a designated person, and perhaps proceed with automatic recovery processing.

An important consideration in this regard is how the software can know whether a job has succeeded or failed. Without automation, someone usually has to look at the job's output ($STDLIST) or read a completion message on the console. A common criterion, quite compatible with automation, is to consider a job successful only if it reaches the ":EOJ" command at its end. Although this doesn't hold for all jobs, experience shows most jobs adhere to this rule. The rule also covers those occasions when the job itself doesn't fail, nor does it complete, such as a system failure while the job was in execution.

If "successful" job completion is signified by reaching some point in the job other than the end, it may be possible to modify the job by inserting a utility program at the point which will signal success to the batch controller. Or, the job could be modified to adhere to the ":EOJ" rule, and only reach its end on successful completion. The advantage of the latter approach is consistency with the majority of jobs, always an aid in maintenance.

Finally, it may be that successful completion of a job can only be determined by inspection of the results. Software products now on the market can automate this by scanning reports or listings for tell-tale messages, but in general it may still require a person to check the results. To derive maximum benefit from automation, such applications should be modified to remove the need for inspection and provide a more obvious signal of success or failure.

Constraining

Now that the jobs have been scheduled and sequenced, there may be further constraints which need to be applied. One such constraint is illustrated by the MPE job limit: Only so many jobs are allowed to execute at one time. Some automated job controllers refine this function, offering the ability to limit the number of jobs executing for a given application, or in a given account. This can be a handy resource allocation tool. At month-end, for example, you can allow accounting a half dozen jobs at a time, while limiting development to two, facilitating month-end closing.

There are often more specific constraints to be applied, such as making certain that particular jobs never run at the same time. Although this can be (and often is) handled by sequencing one job before another, it can be counterproductive when there is no real reason for the sequence. If one job is arbitrarily selected to go first, any event which delays that job will also delay the second. It is therefore important to distinguish such dependencies from ordinary sequencing.

Exclusive job constraints are typically due to a conflict over some resource in the system, such as a database, a file, or even contention for the CPU. It is helpful to identify these resources explicitly when establishing processing constraints. First, this documentation removes any mystery regarding why certain jobs can't be run together. Second, it permits more effective maintenance, such as when new jobs are developed which require the same resources. Finally, the systematic recognition of resource con-

straints can turn up new opportunities to increase productivity. For example, jobs which require the system's only tape drive can run together successfully, but only one will actually be executing at any time. By identifying such jobs to the job controller, they can be kept from occupying executing job slots while waiting for a resource to become available.

Some automated systems extend this resource-centered model by distinguishing between "shared" and "exclusive" use. For example, jobs which access a database would identify it as a needed resource. Read-only jobs such as reports can specify shared access, which allows several such jobs to run at the same time. If *exclusive* access is specified for update jobs, the system can not only avoid running two update jobs together, it will not run an update while other jobs are using the database. This concept supports the highest degree of multiprogramming while maintaining the constraints necessary for successful processing.

Shared resources can be further controlled by designating the available quantity of the resource. If two tape drives are available, two jobs can be run if they each request only one unit of the "tape drive" resource. This approach can be extended to bulk resources like "CPU time" and "Disc I/O," and used to create a dynamic "job limit" which takes into account the special needs of certain CPU- or I/O-intensive jobs.

Synchronizing

The third facet of job control is synchronization with external events. For example, a certain input/update job must await the arrival by courier of a magnetic tape each night. Since this tape is not visible in the computer, the controller software cannot release the job into execution automatically. Instead, it is usually the operator's job to inform the software when the tape becomes available.

Automated job controllers commonly provide for operator input via prompts or "run book" codes, which the software displays and the operator responds to when the external condition is satisfied. In addition to the availability of input data, such conditions may include the reservation of a peripheral device or the state of an application database (are all on-line users off the system?). This method is sometimes employed for job sequencing, when a pre-dependent job is processed on a remote computer not in the local network, or when the successful completion of a pre-dependent job must be certified by someone before processing can continue.

Sometimes external events are visible to the software and synchronization can be automated. It may be that the availability of input data, or the shutdown of on-line processing, can be determined by the appearance or availability of a file or database in the computer. A number of automated job controllers offer such "file dependencies".

And, it may be that external software is able to determine that a synchronizing event has occurred. In this case, it is useful for a program or job to be able to send a signal, akin to the operator's response, to trigger dependent processing.

Recovering

Finally, there comes a time to face the inevitable: A job fails and some recovery procedure must be performed. The job controller's first contribution may be to announce the job's failure in a timely manner. The sooner the problem is attended to, the better the chance of getting production back on track.

In most shops, the ultimate (sometimes only) recourse is to "call the programmer." Most job control packages will allow you to attach some descriptive information to each job

and make it available on-line, such as the name and telephone number of the programmer. There are even software and hardware packages which can make the phone call automatically!

Although the variety of possible recovery actions is as varied as the possible causes of job failure, it may be possible to establish a highly automated recovery procedure. For example, failure of a particular job could trigger the job controller to launch a restore of the database and continue with other processing, bypassing the failed job until it can be attended to. Other automated actions could include rerunning the failed job, or simply bypassing it without recovery.

Conclusion

The operations staff will succeed in automating their batch job processing without severe difficulty by carefully attending to these few basic elements:

Scheduling – Identify all jobs which are regularly scheduled and determine the basis for the pattern in which they are scheduled. Obtain the calendars used and understand holiday scheduling.

Sequencing – Find out why jobs are run in a particular sequence. Identify where a series of jobs really needs to be processed in a certain order, as opposed to an order established arbitrarily.

Constraining – Document conflicts between jobs which prevent them from running together. Identify system resources which may be at the root of these conflicts.

Synchronizing – Prepare a list of checkpoints or external events which may require operator action to trigger or continue processing.

Recovery – Collect recovery procedures and instructions for any jobs which have them.

Once these basic elements are assembled, batch job automation can be accomplished fairly easily and with a likelihood of significant productivity gains.