Instant Quality Control of Large Batch Processing Jobs

By: Niv Ahituv
Faculty of Management
Tel Aviv University
Tel Aviv, Israel

By: Meir Zelek
Manager of Systems Programming
Bank Leumi LeIsrael Ltd.
Tel Aviv, Israel

Abstract
The most common way to identify success or failure of a job running in a batch-processing mode is by examining a completion code sent by the job to the host operating system. Yet, for a variety of reasons the completion code may inaccurately indicate a successful termination of the job. This article describes a different approach to monitoring the quality of batch processing jobs while in operation. A pattern of behavior is suggested for a program. The pattern reflects ratios of consumption of various hardware resources. The ratios are determined by collecting historical performance variables of the job and analyzing the data by means of statistical methods. Once a pattern is set, the performance variables of every individual run of the program are compared with the precalculated pattern of behavior and if the deviation is beyond certain limits an alarm is triggered.

The proposed quality control technique has been tested on real applications, as well as on some artificial programs. The findings suggest that the technique is reliable in that it successfully distinguishes between proper and malfunctioning runs of a program.

Keywords: Quality control, batch processing, computing management

ACM Categories: D.4.5., D.4.8, K.6.2

Introduction
The rapid proliferation of online computer applications has not eliminated the wide-spread use of batch processing jobs. Batch processing is still the backbone of many computer installations, operating the mass production transaction processing such as payroll, bookkeeping, invoicing, and the like.

Yet what assurance does the IS manager have that transaction processing is functioning reliably? An answer to this question is crucial in large batch-processing operations. A large batch-processing application is usually characterized by a long sequence of jobs, where the successful completion of earlier jobs is prerequisite to the beginning of subsequent ones. Failure to recognize a malfunction could result in high costs later, when a long series of dependent jobs must be rerun and files containing wrong data must be corrected.

Many organizations maintain a large application base of batch jobs and there is extreme pressure to manage these effectively and to keep costs down. Many batch jobs are run at night with minimal human supervision. Much effort is, therefore, invested in quality assurance of batch-processing programs. Basically, the effort is directed along two avenues: quality assurance during the development cycle and quality control while the job is in operation.

The first category of quality assurance relates to various software engineering methodologies dealing with system analysis and design, programming, and testing [6] (e.g., structured analysis and design, structured programming, structured walkthrough, chief programming team, and documentation). Practices of managing the development process of IS may also be included in that category. Although these methodologies and practices undoubtedly improve the quality of the end product (i.e., the program), they cannot guarantee that a program transferred from development to operation is foolproof. For example, a program that processes three inputs, each composed of only 16 bits, and consumes 100 milliseconds to produce an output record, will require about 900 years of CPU time to check all the possible input combinations.

The second category, quality control, relates to manual and automated measures imposed
on routine operations. Manual provisions include operating procedures. Automated procedures include backup and recovery routines, and completion-code (CC) indication. The latter is probably the most common measure. It is based on a code generated by the application program upon completion of its task and checked by the operating system (OS). If the code indicates normal termination (CC = 0 in most IBM mainframes), the OS will initiate the next job; otherwise, the job stream is halted and a message is sent to the operator.

Again, none of the above measures can guarantee that a malfunctioning program will attract immediate attention. Obviously, when the CC does not equal zero, instant reaction is assured. However, a program can terminate while generating a zero CC. A wrong file is mounted (e.g., an old generation of a master-file); the operator mounts only part of a multi-volume file; a program undergoes some minor changes and is not perfectly debugged; a very high rate of invalid input data is rejected and diverted to a suspense file, in all these instances a program may still run smoothly.

In addition to embedded completion code, the software industry offers special packages called production monitors that assist in managing the production of large batch-processing computer installations (e.g., [4]). These monitors help in job scheduling and performance analysis, but they cannot stop the process when the completion code is zero.

The area of quality control (QC) provides some concepts that might be helpful in understanding the problem. One of the principal concepts is "reliability." Robertson [3] defines reliability as the ability of a product to function successfully under required conditions for a predesignated period. Vaughn [5] added the notion of probability, and defined reliability as "the probability of function within certain specified limits for a required length of time under given environmental conditions" [5, p. 198]. Common measurements of reliability are mean time between failures (MTBF), and mean time to failure (MTTF). The former measures the average time between two consecutive failures, while the latter accounts for the time elapsing from the beginning of operation to the detection of the first failure [2].

When one attempts to apply these concepts to IS, a number of questions immediately arise. First, a computer job can be either correct or erroneous; it does not erode over time nor is it subject to mechanical or physical deformations. Hence the concept of MTTF is not very adequate. Second, an erroneous program remains so forever, unless replaced by a new, correct one. Thus the applicability of MTBF is questionable. Nevertheless, the last section in this article mentions how MTBF can be useful for managerial control of the IS production shop.

The main focus, however, is the immediate detection of malfunctioning jobs. To that end we present a quality control tool that can be easily incorporated into routine batch-processing runs. The tool, which is based on simple statistical analysis, increases the likelihood of detection of malfunctioning batch-processing programs. The detection occurs right after the program terminates, and it does not depend on the completion code. The proposed tool was successfully tested in the computer center of a large bank that processes more than a million transactions per day.

An Overview of the Model

Suppose, hypothetically, that an individual program runs alone on a computer in a batch mode (no multiprogramming). Suppose also that the program is fully "deterministic," (it does not contain any conditional path (IF statements)) and for any input transaction the program always performs the same sequence of instructions. Obviously, such a program is fully predictable. If the number of input transactions is given, one can precisely calculate the consumption of CPU time, the number of channel activities (EXCPs), and the number of printed lines. Moreover, if the program logic suddenly changes and a different sequence of instructions is executed, it is likely that deviations will occur in some or all of the proportions between CPU time, EXCPs, number of lines printed out, and number of input transactions. Had the original proportions been known, the deviations could be detected right after the program is terminated.

In reality, there are two major factors hampering the "deterministic" approach. First, due to the nature of multiprogramming and virtual
storage (VS) environments, some performance variables might be affected by the random mix of programs residing in memory at any moment. Second, programs are not "deterministic" in the sense that they do contain IF statements, thus the execution of various code segments is conditional. Due to the above factors, various runs of the same program probably differ in their performance vectors.

However, for frequent and routine runs of a program in batch mode, we would like to make the following assumptions:

1. The proportions between the various types of input transactions is nearly steady over time; in other words the ratio among various types of transactions do not fluctuate extremely over time.
2. When proportions of transaction types tend to be steady, then a batch-processing program will run under a constant "pattern of behavior" exhibited in terms of a vector presenting the ratios between the consumption of the various resources (e.g., CPU time, EXCPs, lines).

In other words, assumption number 2 states that if assumption number 1 is true, then ratios between various variables of the performance vector will tend to be constant.

If both assumptions are true, we can then build a quality control tool in the following manner. By recording the history of the performance vectors of a given program, we can produce a pattern of behavior by characterizing the dependency of one specific variable over the others. We can then compare the performance vector of each future run of the program with the program's pattern of behavior. This comparison can be done immediately upon program termination and can be performed automatically by a special routine inserted into the OS. If the routine identifies a "significant deviation" (this term will be discussed later) from the precalculated pattern of behavior, it will trigger an alarm.

This, in a nutshell, was the initial model which we tested and validated. We did not make any attempt to statistically validate the first assumption. Instead, we selected some sample programs which run daily in the computer center of a large bank. We knew that the nature of the applications suggests that the relative distribution of input transaction types is steady. This was confirmed by local experts. The research concentrated on validating the second assumption. The findings and some implementation aspects are presented later.

However, before we turn to the empirical findings we would like to make three important comments. First, the major purpose of this article is to introduce a quality control tool into batch-processing environments; statistical methods are used here as a means rather than a goal. We will, therefore, omit prolonged discussion on various statistical variables where the discussion does not contribute to the main theme of the paper.

Second, the model proposed here is made to detect runaway programs and major input flaws. It does not substitute for ill-designed controls nor for poorly developed systems. For example, if a customer withdraws $100 from an account while an amount of $1,000 is erroneously keyed-in, the proposed instrument will not detect it; it is the duty of a balance control mechanism to detect it. This model centers on quality control of given computer programs while in operation.

The last comment relates to the determination of dependent and independent variables. The large samples that can be obtained in this case call for the use of statistical regression (which, eventually, was found to be very adequate). However, in order to calculate linear regression coefficients in the model we had to select a dependent variable and independent variables. Almost arbitrarily, the CPU time was selected for a dependent variable. The rationale for this decision was that in business applications the "driving forces" are input/output operations, while memory activities are relatively limited and triggered by the amount of peripheral activities. This decision, however, is merely technical. We believe that the major ideas of the paper still hold if another variable is chosen for the role of a dependent variable.

**Empirical Results**

The empirical data was sampled from a large computer installation of the following configuration:

- 1 CPU IBM 3033, 12MB memory
Quality Control

1 CPU IBM 3081, 16MB memory
about 100 disk drives IBM 3350 (317.5 MB each)
16 tape drives IBM 3240
operating system MVS/SP 1.1

Data from three different daily programs were collected over three months and analyzed by the statistical package SAS (Statistical Analysis System). Exceptional business days (e.g., end of a month, end of a quarter) were excluded.

We will now present, in detail, the analysis of one program (labelled NESHM):
- the number of runs in the sample was 49
- the number of input transactions per run (EXCPIN) ranged from 3,000 to 16,000;
- the number of tape read accesses per run (TAPEIN) ranged from 0 to 148,000;
- the number of disk read accesses per run (DASDIN) ranged from 24 to 35;
- the number of disk write accesses per run (DASDOUT) ranged from 20,000 to 110,000.

We selected the CPU time to be a dependent variable and computed its regression with every other individual variable as well as with all combinations of two, three, four, and five variables. Table 1 displays the resulting R-squares.

As can be observed from Table 1, the "best" explaining variable among the individual variables is EXCPIN (input transactions). The "best" multiple regression is the last one where all the independent variables are taken into account (although the contribution of TAPEIN is very minor). We decided to focus on these two models.

**Model 1** (multiple regression):
\[
CPUTM = 0.00021849 \times TAPEIN + 0.00066801 \times TAPEOUT + 0.62825266 \times DASDIN + 0.00139555 \times DASDOUT + 0.01158308 \times EXCPIN + 328.128672
\]
with standard error of estimate of CPUTM SEE = 21.84; \( R^2 = 0.925 \) (rounded)

**Model 2** (single variable regression):
\[
CPUTM = 0.01739043 \times EXCPIN + 372.06276489
\]
SEE = 32.80; \( R^2 = 0.81 \)

We turn now to the major purpose of this experiment, that is, to check whether the statistical model can be used as an instant quality control tool. Four runs of the same program were executed:

1. Control run: a complete run without any modification.
2. Incomplete master-file: half of the master file was ignored (note that this case is not imaginary and can happen when a multi-volume file is processed).
3. Bug: a "small" bug was deliberately introduced into the program (a < sign was replaced by a > sign in an IF statement).
4. Invalid transactions: a large number of invalid transactions were incorporated into the input batch (80% of the batch, compared to an average of 20%). Note that the program does identify and reject the invalid data, however, the completion code upon termination is still zero. One would not like to proceed to subsequent jobs under such circumstances.

Table 2 displays the values of the various variables of the four runs. Table 3 presents the deviations between the predicted CPU times and the actual ones.

The analysis of Table 3 provides the following observations:

1. Both models are sensitive to changes caused by program malfunctioning. This is especially exhibited when the deviations between actual and predicted CPU times are expressed in units of standard error of estimate (SEE units).
2. A minor change in the program (run number 3) highly affects the performance variables.
3. A large number of invalid transactions (run number 4) is detected by Model 1 much better than by Model 2. This is explained by the structure of the models: while Model 2 is affected only by the number of input transactions, Model 1 is affected also by changes in the distribution of various transaction types. Thus Model 1 is more sensitive to an unusual amount of invalid transactions. In an environment where the ratios between various transaction types tend to be steady, model 1 can serve as an alarm to detect deterioration in input quality.
Table 1. R-Squares for the Regressions of CPU Time and All Other Variables in NESHM Program

<table>
<thead>
<tr>
<th>NUMBER OF INDEPENDENT VARIABLES IN MODEL</th>
<th>VARIABLE IN MODEL</th>
<th>R-SQUARE (ROUNDED)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TAPEIN</td>
<td>0.00</td>
</tr>
<tr>
<td>1</td>
<td>DASDIN</td>
<td>0.05</td>
</tr>
<tr>
<td>1</td>
<td>TAPEOUT</td>
<td>0.29</td>
</tr>
<tr>
<td>1</td>
<td>DASDOUT</td>
<td>0.39</td>
</tr>
<tr>
<td>1</td>
<td>EXCPIN</td>
<td>0.81</td>
</tr>
<tr>
<td>2</td>
<td>TAPEIN DASDIN</td>
<td>0.05</td>
</tr>
<tr>
<td>2</td>
<td>TAPEIN TAPEOUT</td>
<td>0.29</td>
</tr>
<tr>
<td>2</td>
<td>TAPEOUT DASDIN</td>
<td>0.31</td>
</tr>
<tr>
<td>2</td>
<td>TAPEIN DASDOUT</td>
<td>0.39</td>
</tr>
<tr>
<td>2</td>
<td>TAPEOUT DASDOUT</td>
<td>0.75</td>
</tr>
<tr>
<td>2</td>
<td>TAPEIN EXCPIN</td>
<td>0.81</td>
</tr>
<tr>
<td>2</td>
<td>TAPEOUT EXCPIN</td>
<td>0.82</td>
</tr>
<tr>
<td>2</td>
<td>DASDIN EXCPIN</td>
<td>0.84</td>
</tr>
<tr>
<td>2</td>
<td>DASDOUT EXCPIN</td>
<td>0.88</td>
</tr>
<tr>
<td>3</td>
<td>TAPEIN TAPEOUT DASDIN</td>
<td>0.31</td>
</tr>
<tr>
<td>3</td>
<td>TAPEIN DASDIN DASDOUT</td>
<td>0.39</td>
</tr>
<tr>
<td>3</td>
<td>TAPEOUT DASDIN DASDOUT</td>
<td>0.76</td>
</tr>
<tr>
<td>3</td>
<td>TAPEOUT DASDIN DASDOUT</td>
<td>0.76</td>
</tr>
<tr>
<td>3</td>
<td>TAPEIN TAPEOUT EXCPIN</td>
<td>0.82</td>
</tr>
<tr>
<td>3</td>
<td>TAPEIN DASDIN EXCPIN</td>
<td>0.84</td>
</tr>
<tr>
<td>3</td>
<td>TAPEOUT DASDIN EXCPIN</td>
<td>0.84</td>
</tr>
<tr>
<td>3</td>
<td>TAPEIN DASDOUT EXCPIN</td>
<td>0.88</td>
</tr>
<tr>
<td>3</td>
<td>DASDIN DASDOUT EXCPIN</td>
<td>0.89</td>
</tr>
<tr>
<td>3</td>
<td>TAPEOUT DASDOUT EXCPIN</td>
<td>0.923</td>
</tr>
<tr>
<td>4</td>
<td>TAPEIN TAPEOUT DASDIN DASDOUT</td>
<td>0.76</td>
</tr>
<tr>
<td>4</td>
<td>TAPEIN TAPEOUT DASDIN EXCPIN</td>
<td>0.84</td>
</tr>
<tr>
<td>4</td>
<td>TAPEIN DASDIN DASDOUT EXCPIN</td>
<td>0.89</td>
</tr>
<tr>
<td>4</td>
<td>TAPEIN TAPEOUT DASDOUT EXCPIN</td>
<td>0.924</td>
</tr>
<tr>
<td>4</td>
<td>TAPEOUT DASDIN DASDOUT EXCPIN</td>
<td>0.924</td>
</tr>
<tr>
<td>5</td>
<td>TAPEIN TAPEOUT DASDIN DASDOUT EXCPIN</td>
<td>0.925</td>
</tr>
</tbody>
</table>

As mentioned earlier, the analysis was performed on three programs; one of them exhibited on the next page. The results of the other two are not very different. All the results support the basic assumption that programs running in a batch mode on a frequent basis do have a constant "pattern of behavior," and that pattern can be used to detect exceptional cases.

With these interim conclusions in mind, we move to describing the second phase of the research, during which artificial programs were written and analyzed.

Analysis of Artificial Programs

In order to further inquire into the usefulness of the various models, we wrote two COBOL programs, each performing a file update in a different method. We ran each program a
Table 2. Values of Variables in Four Runs of NESHM

<table>
<thead>
<tr>
<th>Run Number</th>
<th>Run Type</th>
<th>CPUTM</th>
<th>EXCPIN</th>
<th>DASDOUT</th>
<th>DASDIN</th>
<th>TAPEOUT</th>
<th>TAPEIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Control</td>
<td>403.64</td>
<td>4007</td>
<td>22255</td>
<td>35</td>
<td>5377</td>
<td>18603</td>
</tr>
<tr>
<td>2</td>
<td>Incomplete master-file</td>
<td>53.54</td>
<td>4007</td>
<td>3827</td>
<td>35</td>
<td>5377</td>
<td>111</td>
</tr>
<tr>
<td>3</td>
<td>Bug</td>
<td>652.53</td>
<td>4007</td>
<td>22253</td>
<td>37</td>
<td>5377</td>
<td>18602</td>
</tr>
<tr>
<td>4</td>
<td>Invalid transactions</td>
<td>397.47</td>
<td>4007</td>
<td>33396</td>
<td>50</td>
<td>5377</td>
<td>18659</td>
</tr>
</tbody>
</table>

Table 3. Deviations Between Predicted and Actual CPU Times in Four Runs of NESHM

<table>
<thead>
<tr>
<th>Run Number</th>
<th>Run Type</th>
<th>Actual CPU time</th>
<th>Predicted CPU time</th>
<th>Deviation in SEE CPU time</th>
<th>Deviation in SEE units</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Control</td>
<td>403.64</td>
<td>435.245</td>
<td>31.605</td>
<td>1.447</td>
</tr>
<tr>
<td>2</td>
<td>MF</td>
<td>53.54</td>
<td>405.408</td>
<td>351.948</td>
<td>16.116</td>
</tr>
<tr>
<td>3</td>
<td>Bug</td>
<td>652.53</td>
<td>436.499</td>
<td>216.031</td>
<td>9.892</td>
</tr>
<tr>
<td>4</td>
<td>Invalid transactions</td>
<td>397.47</td>
<td>444.739</td>
<td>47.269</td>
<td>2.164</td>
</tr>
</tbody>
</table>

MODEL 1

<table>
<thead>
<tr>
<th>Predicted CPU time</th>
<th>Deviation in SEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>444.746</td>
<td>38.106</td>
</tr>
<tr>
<td>444.746</td>
<td>388.206</td>
</tr>
<tr>
<td>444.746</td>
<td>210.784</td>
</tr>
<tr>
<td>444.746</td>
<td>44.276</td>
</tr>
</tbody>
</table>

MODEL 2

<table>
<thead>
<tr>
<th>Predicted CPU time</th>
<th>Deviation in SEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>444.746</td>
<td>38.106</td>
</tr>
<tr>
<td>444.746</td>
<td>388.206</td>
</tr>
<tr>
<td>444.746</td>
<td>210.784</td>
</tr>
<tr>
<td>444.746</td>
<td>44.276</td>
</tr>
</tbody>
</table>

number of times in order to get sufficiently large samples.

UPDSEQ1

The program labelled UPDSEQ1 is a classic sequential update program. It reads in a sorted transaction file and updates a sorted master file. While the volume of the master file is more or less constant, the number of records in the transaction file is a random variable. Moreover, some of the transaction records are irrelevant to the particular master file and should be ignored, whereas the others undergo several update routines. The proportion between relevant and irrelevant transactions in the transaction file is a random variable.

The number of runs was 18; the number of input transactions per run (EXCPIN) ranged from 918 to 1,100; the number of DASDIN per run ranged from 271 to 1,147; the number of DASDOUT per run ranged from 260 to 1,159. The program did not use magnetic tape files.

Table 4 displays the coefficients of determination ($R^2$) between CPU time (CPUTM) and each of the possible combinations of the other variables in the sample of normal runs.

As can be clearly observed from Table 4, the number of input transactions (EXCPIN) does not highly affect the CPU time because, as mentioned earlier, some of the transactions are irrelevant, hence they do not consume CPU time for updating. All the other individual variables and their combinations highly affect the CPU time.

We selected two models to test the sensitivity to malfunctioning:

Model 1 (Single variable regression):

\[
\text{CPUTM} = 0.522 + 0.000999 \cdot \text{DASDIN} \\
(\text{SEE} = 0.052; R^2 = 0.968)
\]

Model 2 (Multiple regression):

\[
\text{CPUTM} = 0.00049803 \cdot \text{DASDOUT} + 0.00053858 \cdot \text{DASDIN} + 0.00032436 \cdot \text{EXCPIN}
\]
Table 4. R-Square for the Regressions of CPU Time and All Other Variables in UPDSEQ1 Program

<table>
<thead>
<tr>
<th>NUMBER OF INDEPENDENT VARIABLES IN MODEL</th>
<th>VARIABLE IN MODEL</th>
<th>R-SQUARE (ROUNDED)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EXCPIN</td>
<td>0.048</td>
</tr>
<tr>
<td>1</td>
<td>DASDIN</td>
<td>0.968</td>
</tr>
<tr>
<td>1</td>
<td>DASDOUT</td>
<td>0.970</td>
</tr>
<tr>
<td>2</td>
<td>DASDIN DASDOUT</td>
<td>0.975</td>
</tr>
<tr>
<td>2</td>
<td>DASDOUT EXCPIN</td>
<td>0.986</td>
</tr>
<tr>
<td>2</td>
<td>DASDIN EXCPIN</td>
<td>0.989</td>
</tr>
<tr>
<td>3</td>
<td>DASDIN DASDOUT EXCPIN</td>
<td>0.994</td>
</tr>
</tbody>
</table>

\[ + 0.16206533 \]
\( \text{(SEE} = 0.027; R^2 = 0.99) \)

Just for demonstration purposes, the plot of Model 1 is shown in Figure 1. The asterisks represent actual observations and the dots represent predicted observations (the dots are missing when the predicted and the actual observations concur).

With these models at hand, we performed five experimental runs of UPDSEQ1 as follows:

Run 1: Program logic was changed (bug).
Run 2: Null transaction file (no records in the file).
Run 3: Incomplete master file; complete transaction file (only part of the master file was mounted).
Run 4: Incomplete master file; incomplete transaction file.
Run 5: Normal run.

Table 5 displays the results obtained in all the above runs. The major observation derived from Table 5 is that Model 2 distinguishes between normal and abnormal runs better than Model 1 does. This is due to the parameters of the models: while Model 1 is affected only by changes in the size of the master file and not by the size of the transaction file, Model 2 is sensitive to changes in both files. Had we...
Table 5. Deviations Between Predicted and Actual CPU Times in Five Runs of UPDSEQ1

<table>
<thead>
<tr>
<th>Run Number</th>
<th>Run Type</th>
<th>Actual CPU time</th>
<th>Predicted CPU time</th>
<th>Deviation in SEE CPU time</th>
<th>Model 1 Deviation units</th>
<th>Model 2 Deviation units</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bug</td>
<td>1.46</td>
<td>1.724</td>
<td>.267</td>
<td>5.105</td>
<td>1.542</td>
</tr>
<tr>
<td>3</td>
<td>Incomplete</td>
<td>0.88</td>
<td>1.130</td>
<td>.250</td>
<td>4.789</td>
<td>0.929</td>
</tr>
<tr>
<td>4</td>
<td>Incomplete</td>
<td>0.72</td>
<td>1.130</td>
<td>.410</td>
<td>7.848</td>
<td>0.788</td>
</tr>
<tr>
<td>5</td>
<td>Normal</td>
<td>1.51</td>
<td>1.727</td>
<td>.217</td>
<td>4.149</td>
<td>1.542</td>
</tr>
</tbody>
</table>

selected a deviation of 1.5 units of SEE to be the 'warning' point, Model 2 would have screened out all the abnormal runs but not the normal run.

**UPDSEQ2**

UPDSEQ2 is also a sequential update program, but unlike the previous one, it processes only relevant transaction records. In other words, all the transactions that manage to approach UPDSEQ2 are supposed to update master records. Moreover, the program includes an internal sorting routine that sorts input transaction in an ascending order. Our assumption was that since an internal sorting routine is based on a logarithmic algorithm, a regression model that refers to the logarithmic function of the number of input transactions might provide an adequate pattern of behavior to the program.

The number of runs was 15; the number of input transactions (EXCPIN) per run ranged from 1 to 406; the number of DASDIN per run ranged from 1208 to 1667; the number of DASDOUT per run ranged from 1195 to 1667. The program did not use magnetic tape files.

The two models that were tested are the following:

**Model 1** (Linear model):

\[ 	ext{CPUTM} = 0.00805126 \times \text{EXCPIN} + 0.00606913 \times \text{DASDIN} \]

![Figure 2. Actual (*) vs. Predicted (+) Observations in Model 1 of UPDSEQ2](image-url)
**Quafity Control**

**CPUTM**

0:00:05.5000
0:00:05.0000
0:00:04.5000
0:00:04.0000
0:00:03.5000
0:00:03.0000
0:00:02.5000
0:00:02.0000
0:00:01.5000
0:00:01.0000
0:00:00.5000
0:00:00.0000

I

NOTE: 2 OBS HIDDEN

**PLOT OF CPUTM*EXCPIN**

**SYMBOL USED IS * +**

**PLOT OF P_CPUTM*EXCPIN**

**SYMBOL USED IS ~**

**Figure 3. Actual (•) vs. Predicted (*) Observations in Model 2 of UPDSEQ2**

0.00486288 * DASDOUT + 0.58578485

(R² = .998; SEE + 0.054)

**Model 2 (Nonlinear model):**

\[
\text{CPUTM} = 0.09934032 \times \log\text{E}(\text{EXCPIN}) + \\
0.00440517 \times \text{DASDIN} + \\
0.01058702 \times \text{DASDOUT} - 5.82153467
\]

(R² = .994; SEE = 0.189)

It is clearly seen that both models nicely depict the pattern of behavior of the program. Model 2 is not superior to Model 1 probably because the internal sorting routine does not consume much CPU time relative to other routines in the program.

**Table 6. Deviations Between Predicted and Actual CPU Times in Four Runs of UPDSEQ2**

<table>
<thead>
<tr>
<th>Run Number</th>
<th>Run Type</th>
<th>Actual CPU time</th>
<th>Predicted CPU time</th>
<th>Deviation in CPU time</th>
<th>Deviation in SEE CPU time</th>
<th>Predicted CPU time</th>
<th>Deviation in CPU time</th>
<th>Deviation in SEE CPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Normal</td>
<td>6.19</td>
<td>6.203</td>
<td>0.013</td>
<td>0.258</td>
<td>6.065</td>
<td>0.125</td>
<td>0.661</td>
</tr>
<tr>
<td>2</td>
<td>Null Trans. File</td>
<td>2.60</td>
<td>2.667</td>
<td>0.067</td>
<td>1.262</td>
<td>4.346</td>
<td>1.746</td>
<td>9.241</td>
</tr>
<tr>
<td>3</td>
<td>Null Master File</td>
<td>2.08</td>
<td>2.252</td>
<td>0.172</td>
<td>3.209</td>
<td>0.195</td>
<td>1.884</td>
<td>9.971</td>
</tr>
<tr>
<td>4</td>
<td>Null TF; Incomplete MF</td>
<td>1.59</td>
<td>1.670</td>
<td>0.080</td>
<td>1.494</td>
<td>0.769</td>
<td>2.356</td>
<td>12.47</td>
</tr>
</tbody>
</table>

Figures 2 and 3 display the predictive power of Models 1 and 2 (respectively). In both figures, the "independent" variable is the number of input transactions (EXCPIN) and the dependent variable is the CPU time. It is graphically seen that there is no significant difference between the two models.

The models were tested in four runs:

- Run 1: Normal run.
- Run 2: Null transaction file (contains only one record).
- Run 3: Null master file (contains only one record).
- Run 4: Null transaction file; incomplete master file (half of the normal size).

Table 6 displays the results obtained in all the above runs.
It is seen in Table 6 that both models distinguish between the normal run and the abnormal runs. However, the non-linear model does it much more sharply. This can be explained by the logarithmic function of EXCPIN, which, in fact, reduces the effect of EXCPIN on CPUTM relative to the other variables.

Conclusions From the Empirical Findings

Based on the analysis of the real applications, as well as of the artificial programs, it is possible to conclude that programs running under normal conditions are typified by a pattern of behavior which can be expressed by means of statistical models. Moreover, the use of a statistical model can help in trapping abnormal runs that might occur due to human error (e.g., incomplete files, bugs, etc.).

However, drawing the borderlines between normal and abnormal cases must be subject to human judgement. If the range allowed for normal behavior is too narrow, there might be too many false alarms; if it is too wide, some exceptional cases will not be recognized on time.

A partial remedy to this problem can be obtained by using two (or more) models for the same program. One model can be more sensitive to certain types of errors while the other model can better serve in identifying other types. A quality assurance model has to be tailored to meet the specifications of the program it is supposed to inspect.

Generally speaking, it appears that multiple regression models are more instrumental in error detection than signal-variable models. This is due to their structure. They take into account several variables simultaneously so they are more sensitive to irregular deviations in any of the variables. A single variable model might not be significantly effected by changes in variables external to the model.

The implementation of a quality control mechanism that is based on the proposed model and a possible extension of its use for managerial control are discussed next.

Management Application

This section discusses two issues: how to implement the QC tool and how to use it for managerial control purposes.

The implementation procedure for a quality control mechanism should undergo the following steps:

1. Select a number of key programs to control during regular batch-processing operations.
2. Collect the statistics of performance variables of the selected programs over a given period of time.
3. For each key program, compute a regression model that most adequately delineates its pattern of behavior.
4. Check the sensitivity of the model to a number of artificial errors (e.g., bugs, incomplete files) to see whether the errors are indeed detected and what should be reasonable border lines between normal and abnormal cases. If necessary, repeat steps 3 and 4 until you obtain satisfactory results.
5. Store the regression coefficients in a special file.
6. In parallel to the preceding steps, write a monitor program that is inserted into the job stream right after each selected key program. The monitor program compares the performance variables of the last run of the key program with those of the stored model, and "raises hell" when the deviation between predicted and actual values is greater than a certain limit.
7. Test the control mechanism for a certain period until it is well tuned.

The proposed quality control mechanism is not problem free. The major hurdle relates to the need to construct a learning mechanism—a mechanism that modifies its coefficients whenever an intended change is introduced into the program under examination. It is most likely that key programs undergo modification and changes every now and then. If these are not taken into account, false alarms might occur often. Based on experience accumulated through experimental runs, our suggestion is to compute the regression coefficients frequently, always based on the last N normal runs (similar to the concept of moving average). This will take into account minor changes introduced into the key program. However, when a major revision is made to a program, new statistics have to be
collected before the mechanism becomes useful again.

Once the quality control tool is installed, its benefit is twofold. Not only is it used for instant inspection of batch jobs, but it can also generate statistical data for managerial control. The extended use for managerial control might open interesting avenues for computer installation managers. Management can accumulate data on incidents where job streams had to be halted due to QC problems. The data can be categorized according to the cause of the incident (e.g., bug in a program, poor quality input, operator error). This will indicate the "weaker" areas in the production shop.

In addition, it is possible to develop a measurement very similar to MTBF (see definition in the first section). For each application one can record the frequency of failures and calculate the average time between incidents. This will enable closer monitoring of the quality of various batch applications. If the quality tends to deteriorate, appropriate provisions should be taken. In extreme cases, the MTBF can signal that a comprehensive evaluation of the application has to be undertaken [1], and perhaps a new system life cycle must be initiated. In fact, once a procedure for MTBF calculation is established, approaches and methods from QC theory can be incorporated into IS. This, of course, requires further research, beyond the scope of this paper.

Acknowledgement

The authors are grateful for very helpful remarks provided by Yoav Benjamini of Tel Aviv University and Magid Igbaria of Drexel University, and by an anonymous referee and an associate editor.

References


About the Authors

Niv Ahituv is an Associate Professor and Chairperson of the Computers and IS Program at the Faculty of Management, Tel Aviv University, Israel. Formerly, he lectured at the Claremont Graduate School, the University of Calgary and the University of British Columbia, and managed the DP department at the Bank of Israel. He holds degrees of B.Sc. in Mathematics, M.B.A., M.Sc. and Ph.D. in Information Systems. His articles have appeared in Computers and Operations Research, The Computer Journal, Information and Management, MIS Quarterly, Interfaces, Decision Sciences, The Journal of Systems Management, and others. He is a member of the editorial board of Human Systems Management. He has co-authored a book, Principles of Information Systems for Management. His main areas of interest are economics of computers, information economics, and information systems management and development.

Meir Zelek is Manager of Systems Programming at Bank Leumi Lelisrael Ltd., Tel Aviv, Israel. He holds a B.Sc. degree in Mathematics and an M.Sc. in Information Systems (Tel Aviv University). He has extensive experience in information systems, particularly in systems programming.