

# Migration Project Testing

A White Paper By

**Don Estes**

Chief Technology Officer  
2000 Technologies Corporation

© Copyright 2004

**2000 Technologies Corporation**

679 West Littleton Blvd.

Suite 200

Littleton, CO 80120

(303) 795-1992

## Table of Contents

1	Executive Summary.....	3
1.1	Testing Is Expensive .....	3
1.2	Testing As Risk Management.....	3
1.3	Optimizing Use of Testing Resources .....	4
1.4	Ensuring Realistic Expectations .....	5
1.5	Migration Testing Alternatives .....	5
2	Typical Testing Approaches .....	7
2.1	Comparison Versus Correctness Testing.....	7
2.2	Subject Matter Experts .....	7
2.3	Test Data Versus Production Data .....	8
2.4	Batch Comparison Tests .....	8
2.5	Batch Correctness Tests .....	10
2.6	Online Comparison Tests .....	10
2.7	Online Correctness Tests .....	10
2.8	Stages of Testing.....	10
2.9	Summary .....	11
3	Professional Testing Approaches .....	12
3.1	Positive Versus Negative Testing.....	12
3.2	Coverage Analysis.....	13
3.3	Construct Coverage Analysis .....	14
3.4	Complexity Analysis .....	14
3.5	Risk Optimized Testing.....	15
4	Automation of Migration Testing .....	16
4.1	Source Code Instrumentation.....	16
4.2	Capture Processing .....	17
4.3	Validation Replay.....	17
4.4	Business Rule Replay .....	18
4.5	I/O Replay.....	19
4.6	Datacomm Replay .....	19
4.7	Summary .....	20
5	Conclusion and Recommendations .....	21

# 1 Executive Summary

---

Testing is where the rubber meets the road in a migration project, where we learn whether or not the project was indeed done correctly. And, if the project was not done correctly, re-work and re-testing after failing initial testing is where cost overruns and late delivery begin in a project with problems. Conversely, if the testing is not done correctly, latent or expressed errors will appear in production, resulting in even greater cost and delivery overruns than if they had been found earlier in the project, when they are much less expensive to fix.

## **1.1 Testing Is Expensive**

Testing is also the most expensive part of a project, both in money and in the time of the most knowledgeable personnel who we would prefer to be applying their skills in tasks creating new value rather than preserving old value. Depending on the methodology employed for both migration and testing and on the accuracy requirement for the testing, migration project testing can consume 40% to 80% of a project's total cost, including both external and internal expenses.

In other words, for very high accuracy testing requirements, there can be \$4 or more spent on testing for every \$1 spent on everything else. Yet, when planning a project, testing is rarely given a proportionally prominent place in the project design process. This is true even when choosing an alternate project strategy could have dramatic impacts on the total project cost, time frame and risk.

There is a direct relationship between cost and accuracy. We have a tendency to think of a program as being "tested" or "not tested", whereas the reality is that a program may be 30% tested, or 50% tested, or 90% tested. Except for very simple programs, it is never the case in practice that a program will be 100% tested. The increasing costs of testing ever more obscure and seemingly unlikely cases invokes the law of diminishing returns, ensuring that we will declare a program to be "good enough" tested long before we get close to 100%.

However, to the extent that a program is not 100% tested, there is a residual risk that a latent error in the untested part of the program will be expressed in production. And it is difficult or impossible to predict in advance how significant that latent error may be when it is finally expressed. Thus, budgeting for testing is an exercise in balancing known, present day costs of testing against unknown costs from possible future failures, which is a recipe for either undertesting or overtesting. There is no objective, scientific standard by which anyone can define what is "good enough." It is always to a greater or lesser extent intuitive and subjective.

## **1.2 Testing As Risk Management**

Ultimately, then, migration project testing is an exercise in risk management. However, technicians, both internal and external, rarely think in terms of risk management. Indeed, it is very uncommon for technicians who are not testing specialists to even measure the extent of their testing, a well established process known as "test coverage analysis" or simply "coverage analysis." If you even don't measure your testing, how could you know how accurate it is?

Most testing is conducted with an ad hoc, seat of the pants methodology, based on a combination of knowledge of the application system internals and feedback from users of the application. This is not to denigrate such efforts, as this is the way most testing is done, and it is usually sufficient. But when it is found to be insufficient in a given project, it is always too late to do anything else except more of the same until the result is finally deemed “good enough,” or until it is too late in the day and the system has to go into production regardless, bugs and all. So, luck is a wild card element in the results of every migration project, with a significant dependency on the quality and complexity of the code being migrated, on the migration solutions being applied to that code, and the talent and experience of the project team.

### **1.3 Optimizing Use of Testing Resources**

So what is management to think of all this? Management typically is less schooled in testing theory and practice than the technicians, but management is also better than most technicians at optimizing the use of business resources and engaging in organized risk management in the conduct of a project. Therefore, we recommend to our clients that senior management be engaged at the highest level of planning a migration project, and that progress be measured in a way that allows for resource optimization and maximal risk reduction for the resources available.

In a typical project where management is not engaged in the process, technical staff will be asked, “how much do you need to budget for testing?” Since technical staff will tend to be optimistic in most cases, and also because they anticipate resistance to a significant budget request, there will be a strong tendency to underbudget for testing. And indeed, in many cases management will resist allocating even an insufficient budget, because of the lack of black and white criteria from which an optimal budget can be derived. It is a murky, gray process, governed by opinions rather than objective facts. By the time a given budget is proven insufficient in practice, the project will be too far along to cancel, and the additional budget will simply have to be allocated.

For smaller projects, this may not be a bad algorithm. Simply let the technicians test until the rate of faults found comes to an acceptable level. If that occurs before a testing budget is exhausted, well and good, but if not then there are no surprises and no recriminations. In effect, in this case senior management is accepting all the risk of faults in the process, with the pleasant result that the price bid from migration vendors will be minimized.

However, for larger projects, management may want their migration vendor to share in the project risk or even to accept all the project risk. Alternatively, management may be unsure about the necessity or affordability of the project. Without locking down the total cost, management may be unwilling to commit to the project at all. Conversely, if a vendor is going to accept part or all of the project risk, they will have to charge accordingly, which can result in a very high price, particularly when the risk cannot be accurately quantified in advance. A vendor is then forced to price on a worst case basis, although there are vendors who will underprice the risk in order to get a contract and then renegotiate if the project runs into trouble. In this case, the client is left with the alternative of paying more or litigating, and usually the costs, risks and delays from litigating are greater than just paying up and getting the job finished.

For those projects where risk is to be taken totally in-house or shared with a vendor, we recommend that business management in consultation with technical staff set the budget for testing, and set that budget based on *primarily business not technical criteria*. Always, the question should be, “what could it cost us if we miss a fault in one of the migrated programs?”

From the answer to that question, summed over the whole of the application being migrated, should come the amount that management is willing to spend to avoid the impact of such a fault. In other words, the decision should be made in exactly the same way that insurance coverage and acceptable premium payments are decided upon. Then, management should exercise oversight to ensure that the budget is spent to yield the greatest risk reduction for the resources expended. We discuss how to do this in more detail in the body of this paper.

#### **1.4 Ensuring Realistic Expectations**

Both technical and senior management may also have unrealistic expectations of a migration vendor. Some of the faults revealed by testing will have been in the programs for some time, but not expressed for various technical reasons. When the migration process incidentally removes the barriers that previously prevented their expression, faults will then appear in production even though they were not introduced by the migration process.

When a migration is done wholly or partly manually, there will be variations in the way that solutions to migration issues are implemented that subtly vary from one migration technician to another and from one program to the next. In some cases these variations will be the source of errors introduced by the migration process.

When automation is used to partly or wholly introduce the solutions to migration issues into programs, there is a tendency to expect that the automation will be perfect. This is unrealistic. When mapping from one software/hardware environment to another or from one language to another, there are always interacting complexities that can never be perfectly specified in general case transformation algorithms. Although automated migration processes do vary in quality, principally based on the project experience and technical sophistication of the automation employed, none will ever do the job perfectly because none can ever anticipate every possible combination of circumstances that require a specific solution. Every project reveals a combination that has never been seen before in that exact expression, and this will occur even with tools that have absorbed the experience of several hundred projects. The technical problem is simply too complex, regardless of what sales people might lead you to believe. This is why all automated projects have an analysis and setup phase where the tool is adapted to the specific circumstances of the particular set of programs undergoing migration.

The advantages of automation are real and compelling, in the elimination of much manual effort, in the consistency of the results, and in the time it takes to complete the project. These advantages are expressed in sharply reduced costs and in the sophistication of the resulting transformed programs. However, there will be faults, both faults inadvertently introduced into programs by the process and previously latent faults in the programs that will now be expressed. The question is how to find those faults quickly and efficiently in order to control both cost and risk.

#### **1.5 Migration Testing Alternatives**

Ultimately, testing is rarely done scientifically except by professional testers. In the sections that follow we will describe "typical" testing and then thorough, high accuracy scientific testing approaches. Both of these use conventional approaches to migration testing. Then we will describe a non-conventional approach to migration testing utilizing test automation, one that is uniquely applicable only to migration projects and that can be used to produce highly accurate results at lower cost.

## Migration Project Testing

We recommend that senior management carefully consider the business and technical risks involved in a migration project, and then choose the best strategy for both migration and testing. In many cases, particularly smaller projects or projects where an undetected fault poses minimal business risk, it may be appropriate to accept all the project risk, which will result in the lowest price from a migration vendor. In these cases, typical testing may be the best choice.

In other cases, particularly larger projects and projects where an undetected fault poses significant risk to human life and/or the viability of the business, it may be appropriate to share the project risk with the migration vendor, or to completely outsource the project risk by engaging a professional testing organization. In these cases, automation of migration testing can provide a compelling alternative to the high cost of professional, scientific testing.

## 2 Typical Testing Approaches

---

### 2.1 *Comparison Versus Correctness Testing*

Basically, migration testing can answer two questions. “Does the migrated system produce the correct results?” Or, “does the migrated system produce the same results as the old system?” Answering the first question is the more typical approach to software testing in general, and may be referred to as correctness or expected results testing, if presented as a distinct form of testing at all. Most commonly, when someone refers to software testing of any kind, they mean the first: testing that produces results that can be shown to be correct or that produces the results that are expected, given a controlled set of inputs.

The process of answering the second question may be referred to as comparison testing, equivalence testing, or regression testing. In this less common form of testing, the same set of inputs are given to both the old and the newly migrated sets of programs, and the results of the new are compared or “regressed” against the results of the old. If they are the same, which is *not* the same as being correct, then the test is considered to have been passed. If the old programs produced a result that is considered to be incorrect in some fashion, that is of no concern so long as the new set of programs are incorrect in the same way.

### 2.2 *Subject Matter Experts*

When a test plan specifies correctness testing, then subject matter experts, people deeply knowledgeable in the use of the system and perhaps in the implementation of the system, will have to be intimately involved in the definition of the tests and, to a lesser extent, the test executions as well. When a test plan specifies comparison testing, subject matter experts will still be required, but to a significantly lesser extent. In this case it will be often be sufficient to demonstrate the use of the programs. In many cases, normal everyday users of the system can be sufficient subject matter experts for the definition of the tests.

The timely availability of subject matter experts at key points of the project is critical to maintaining schedule and controlling costs, which is why migration vendors may contractually specify certain levels of availability and responsiveness. This is particularly true with regard to test definition and execution. Subject matter experts are frequently overscheduled, and can become a costly bottleneck during the course of the project.

It is significantly more difficult when subject matter experts are largely unavailable, or when there are no subject matter experts available at all. Written documentation is frequently incomplete, out of date, or simply wrong, if indeed any documentation is available at all. In this circumstance a significant amount of time will need to be spent learning the system before correctness tests can be defined. This time may be impossible to quantify accurately in advance. As a result, comparison tests will be easier to define, generally from system operations. Even though easier, this will still be a considerable effort, primarily in the logistics of assembling and executing the tests.

### **2.3 Test Data Versus Production Data**

An question that has to be considered early in the planning with all approaches to testing is whether to use specially constructed test data or to use production data. Where the run times of batch programs are very large, it will be preferable to use test data. Not only will test data be much smaller in volume, resulting in less time waiting for a test run to finish, but it can be constructed to include combinations of input data that only rarely occur in production data. For example, there may be transactions that only occur at end of month or end of year that could be defined specifically for the test data.

Using production data for testing online programs might seem to be less of an issue, since online programs typically access only the data that is required, ignoring the bulk of the data. While it is true that the run time of online tests will be no different with production or test data, the setup times for the tests may be considerable. Even after the one time conversion of data from the old to the new data store has executed and been secured, which can be considerable in many cases, we have to consider that the test image of production data in use will have to be restored before each test. This can take many hours, as opposed to a few minutes or less for low volume test data, in direct proportion to the data volumes involved. These database restoration times, summed over the total number of tests, sharply limits of the number of tests that can be executed in a work week and can have a dramatically negative impact on the project delivery time and cost of testing.

On the other hand, producing low volume test data that accurately reflects all of the circumstances that will be experienced in testing with production data can be an extremely complex and time consuming task, more than many technicians wish to invest even when there will be a significant payback for doing so. Typical testing will therefore consist of production data throughout, or whatever test data can be derived quickly for early tests and production data used for later tests. Therefore, typical testing will tend to use either test or production data based on the concerns of knowledgeable technicians and the time pressures that they are under at the moment. We recommend that the decision about which to employ and when should be a subject for the planning phase of a project.

### **2.4 Batch Comparison Tests**

Testing batch programs tends in be easier than testing online programs because there is rarely any manual input required, whereas online program testing demands a great deal of manual input. However, easier does not necessarily mean easy.

All formal tests are described in scripts, both batch scripts and online scripts. A batch script will be simpler than an online script:

- Specify the set of files and any database, noting any case where the program can decide to access any other files based on its own internal algorithms or via alternate JCL decks.
- Secure an image of all files definitely or possibly input to the program and any subroutines; this is referred to as the “before” image.
- Secure an image of the source to the program, including all copy books and any subroutines with their copy books, then compile the program and subroutines using these sources prior to execution. This particular set of sources should be input to the migration process.

- Secure the exact execution parameters, and execute the program such that it is the only program updating shared files and/or databases; if this is not possible then the programs that have to run simultaneously will have to be run as a controlled set of programs in a single script.
- Sometimes, to minimize effort, a series of programs will be run and only the results from the final program compared in the tests. However, all intermediate files have to be secured in this case anyway so that any discrepancies can be traced back to the creating program.
- Secure the “after” image of all output files plus any database and/or any files that might be updated during the execution; if in doubt secure them all.
- Set up a post execution audit using system logs to ensure that all files and/or databases actually used during execution were secured, and that the correct images (before and/or after) were secured. This can be tedious and time consuming, but otherwise project management must expect that a number of test scripts will be invalid and will have to be recreated. Alternatively, the test script data can be loaded onto a test machine and the test re-executed, comparing against the secured output. This script validation process is less tedious but also time consuming.

The issues in setting up batch comparison test scripts are primarily logistical. Securing *all* the files and databases, and ensuring that the correct versions were secured without contamination is a non-trivial task. A single error invalidates the test. Then the secured batch test scripts must be archived and tracked, at least one per program being migrated and sometimes several, when a program can be executed in different modes or with varying inputs.

Then when the source secured with the batch test script has been migrated, we can run the comparison test.

- All of the before data and the after data must be converted to the new database and/or system platform, with the after data segregated to use for post-execution comparisons.
- Then the program or programs from the script must be executed using the converted before data as input, updating the converted files and databases as appropriate.
- Then, the output files and any updated files and/or database tables must be compared against the converted after data. The comparison methodology may be a manual inspection of selected records for consistency plus a count of the records, or it may use a specially written program or a comparison utility to automatically compare the files. Sometimes non-meaningful differences (e.g., timestamps) must be disregarded by the comparison tool in order to avoid false positives.
- If the program fails the test, when it is corrected the test needs to be run again until it passes.

## **2.5 Batch Correctness Tests**

A correctness test on a batch program avoids the logistical headache of having to convert the data twice and run the comparison program or utility against the various data stores. However, the batch test script must define how the tester will determine whether the output is correct, perhaps using a control total or some other aspect of the program's execution that is known to the programmer or operator creating the batch test script. Note that this method is dependent on the depth of knowledge of the person creating the test script for the accuracy of its results.

## **2.6 Online Comparison Tests**

Online comparison test scripts must do what a batch test script does with regard to sources and before/after images of the necessary data stores. In addition, the online test script will almost always involve many programs being tested as a group, and it is important to ensure that all possible types of transactions are input to all the programs. The precise screen inputs and the precise screen outputs need to be secured, either to hard copy or in a software tool designed for the purpose.

Then, during the testing of the migrated programs, the screen inputs must be precisely the same in precisely the same sequence, and the outputs compared against the output screen images. A number of software testing tools will automate this process, first of capturing the data entered, and then replaying that data exactly into the screen. However, it is important to note that these basic, online capture/replay tools apply *only* to the screen image, not the database access or update, and they do nothing whatsoever for batch testing.

Note that these screen images can become voluminous very quickly. When the script is completed, then the updated files and/or databases are compared as in the batch comparison script.

## **2.7 Online Correctness Tests**

If done rigorously, online correctness tests will both compare the screen outputs and some type of algorithm to determine whether the files and/or databases have their expected values updated. However, these tests are frequently abbreviated to comparing only the screen outputs, disregarding any validation of the data stored. These tests are somewhat less difficult for testing personnel because a keystroke error may be disregarded where it would require the test to be restarted for a rigorous comparison test, but they require a greater knowledge of the programs to be done to the same level of accuracy.

## **2.8 Stages of Testing**

Testing is generally divided into 5 stages:

1. Unit tests, typically single program tests often with test data if available.
2. System or run unit tests, in which programs that normally run in a job stream or concurrently in an online environment are run together in a test simulating production usage. Unit test scripts may be combined, or new run unit test scripts created, as appropriate, using test data if available and production data if not.

3. Parallel test, in which the old and new systems are run in parallel with production data, with the same work being done on both systems and the results compared.
4. Acceptance test, a formal sign-off that the system's parallel test has completed satisfactorily, usually involving an extended demonstration to stakeholders not directly involved in the parallel testing. The project is usually considered complete at this point.
5. Stress tests, in which the system is artificially driven at a higher than normal transaction rate to determine whether any bottlenecks exist that require tuning and to determine the effective capacity of the new environment. These tests are seldom performed in a migration project.

Interestingly, it is usually unit testing where the greatest efforts are expended, and the greatest percentage of errors found. It is not unusual for unit testing to consume 50-75% or more of all testing, because of the logistical effort of creating, validating and executing the tests.

## **2.9 Summary**

When typical testing is being done by testers with little knowledge of the application programs, comparison testing is generally preferred because the testers will not know how to determine whether the results are correct. If the test scripts are prepared in sufficient detail by knowledgeable staff, then testers with less knowledge can execute the tests. However, the greater effort is usually in creating the test scripts rather than in executing them, so avoiding the use of subject matter experts in creating detailed test scripts necessarily involves significant compromise of testing accuracy. This may be appropriate, but it should be done knowingly. In general, comparison tests are preferred if subject matter experts cannot be regularly involved in the details of testing and test evaluation

This description will perhaps give the flavor of why even typical testing is expensive. There are a lot of hours required to secure and verify each test script, and then to execute them, particularly if using production data volumes. Since there will be some number of faults, depending on the complexity of the programs being migrated and the precise nature of the migration process, programs failing their test will have to be re-worked and then re-tested, perhaps more than once.

## 3 Professional Testing Approaches

---

With the description of typical testing in the preceding section, it would seem that nothing else should be necessary. And if that testing is expensive enough, why should management consider spending even more money on testing?

The answer is accuracy. Typical testing allows significant parts of a program to go untested. Typical testing is like single entry bookkeeping, there is no way to check that it was done as well as you think it was done. If the business cost of an undetected fault is sufficiently high, then the accuracy of typical testing may be insufficient to cover the business risk. If it does make business sense to extend the accuracy of testing, then there are several techniques that can be used to do so in addition to simply doing more typical testing, and these are discussed below. Conversely, if the business cost of an undetected fault is acceptable, then no further effort *should* be expended.

### 3.1 Positive Versus Negative Testing

Typical testing usually covers only positive cases. In other words, we enter this correct transaction data and we see the same result in the migrated program as in the original program.

Professional testing will also cover negative cases. We put in every conceivable variant of an incorrect transaction, and ensure that it is properly rejected, from both the original and migrated versions. This is particularly evident in testing online programs, where each field should have both no entry and incorrect data submitted for processing. Where cross-field editing is done, each combination should be permuted across the affected fields. Note that in executing negative testing, it frequently will be the case that faults in the original programs will be uncovered.

### 3.2 Coverage Analysis

If typical testing is like single entry bookkeeping, then using coverage analysis in your testing is like double entry bookkeeping - you have a method to use to check whether you have actually tested what you think you have tested. An example coverage analysis report follows:

1000-INITIALIZATION.	
PERFORM 1100-GET-TODAYS-DATE	Exec
THRU 1100-EXIT.	Exec
INITIALIZE T-ZONE-TABLE.	Exec
OPEN INPUT CARDIN.	Exec
MOVE 'N' TO S-CARDIN-EOF.	Exec
MOVE +0 TO A-RULE-TOT.	Exec
PERFORM 1200-READ-CARDIN THRU 1200-EXIT	Exec
UNTIL S-CARDIN-EOF = 'Y'.	Exec
CLOSE CARDIN.	Exec
IF W-CARD-1-BATCH-NBR > ZEROES	Exec
MOVE W-CARD-1-BATCH-NBR TO PARC8703-BATCH-NBR	=No=
PERFORM 1300-MODIFY-BATCH THRU 1300-EXIT	=No=
END-IF.	=No=
IF W-CARD-1-BATCH-NBR-8 > ZEROES	Exec
MOVE W-CARD-1-BATCH-NBR-8 TO PARC8703-BATCH-NBR	=No=
PERFORM 1300-MODIFY-BATCH THRU 1300-EXIT	=No=
END-IF.	=No=
PERFORM 1400-BIND-AND-READY THRU 1400-EXIT.	Exec
PERFORM 1500-CONVERT-DATES THRU 1500-EXIT.	Exec
DISPLAY ' '.	Exec
1000-EXIT.	Exec
EXIT.	Exec

This report shows exactly which lines of code in this section of the program have been executed ("Exec" in the example) and which not ("=No=" in the example). This shows us that the input record in the CARDIN file did not contain a case where "W-CARD-1-BATCH-NBR > ZEROES" is true, so we know we need to re-execute this program with that field containing a non-zero value in order to exercise the program code in paragraphs 1300-MODIFY-BATCH THRU 1300-EXIT.

This form of coverage analysis does not ensure that all lines of code will be executed. It only provides a way to measure the amount of coverage, and allows the knowledgeable application specialist to decide whether or not the coverage is sufficient. His or her judgment might be incorrect, saying on the one hand that additional coverage is necessary when it is not (resulting in higher than necessary costs) or on the other hand that additional coverage is not necessary when it is (leading to a latent fault not being discovered until production testing when it is much more expensive to correct). However, it does provide checks and balances on the testing process.

There are other forms of coverage analysis besides this report showing lines of code coverage, which is also known as branch and path coverage. More sophisticated analyses can determine whether all permutations of branch paths have been taken, known as logical test path coverage analysis. Other analyses can determine whether arithmetic statements have been executed with a full range of values. And there are even more obscure situations that can be measured. The type of coverage and the degree of coverage are suitable topics for a risk management discussion during migration project planning, but typically branch and path coverage will be sufficient for all but a very few cases.

Reviewing coverage analysis reports can be time consuming, and requires the attention of application knowledgeable people whose time is in heavy demand. These report reviews are difficult to relegate to contract or staff personnel who do not have detailed knowledge of the application. However, it is possible to optimize this and other aspects of project testing, which we will discuss in the next sections.

### **3.3 Construct Coverage Analysis**

It is possible to say that certain constructs that occur within a given program or within the entire suite of programs should always be covered. For example, if the migration project is going to be replacing indexed file access or an older database with a relational database management system, then all of the unique accesses of each file from the old data stores should be tested at least once amongst the coverage reports. Or, programs that are updating the database should test all arithmetic constructs. Et cetera.

These kind of construct coverage rules can be derived during project planning, and can be applied before submitting a final set of coverage analysis reports to subject matter experts. These rules can be used to create a list of constructs that must be covered, and an analysis report created showing which mandatory constructs have not been covered at least once.

Note that the migration methodology impinges on testing at this point. If the migration is being conducted using a fully automated process, then a single instance of each construct may be sufficient to prove the transformation of that construct correct. If, however, the process is only partially automated, or if it is wholly manual, then there is the opportunity for the same construct to be transformed in different ways in different programs. In this case, construct coverage analysis may be of limited if any value.

### **3.4 Complexity Analysis**

Some programs require more testing than other programs, but is there a way to determine which ones those are? Certainly, programs with more complex logic in the expression of their business rules will have more branch paths over which to ensure coverage. However, it can also be argued that fairly straightforward reporting programs do not need the analytical scrutiny of primary transaction processing program.

It is possible to process program sources and derive various metrics regarding their complexity. Of the many such metrics, perhaps the best known is McCabe's Cyclomatic Complexity.<sup>1</sup> If we derive this or an alternative metric for the entire library of sources under consideration, then we can apply the greatest testing to the most complex programs, and proportionally less testing for the less complex programs.

---

<sup>1</sup> See references at [www.mccabe.com](http://www.mccabe.com).

### **3.5 Risk Optimized Testing**

Risk optimized testing says that the most testing should be applied to the programs where a fault will create the greatest negative impact. Arguably, a report program need not have much more testing than a correctness test of the output, matching control totals and perhaps checking the sequencing and number of pages. A program that extracts data for analysis or reporting by other programs probably needs more testing since an error in that program could propagate to other downstream programs. And a program that updates the primary database should receive the most testing attention, with a careful analysis of the coverage reports. Such a risk index can be derived fairly simply and can be combined with the complexity analysis discussed above. The exact algorithm for risk optimized testing will be unique to each project, but if the algorithm is derived as part of discussions during project planning it offers the opportunity for increased accuracy while controlling testing costs.

## 4 Automation of Migration Testing

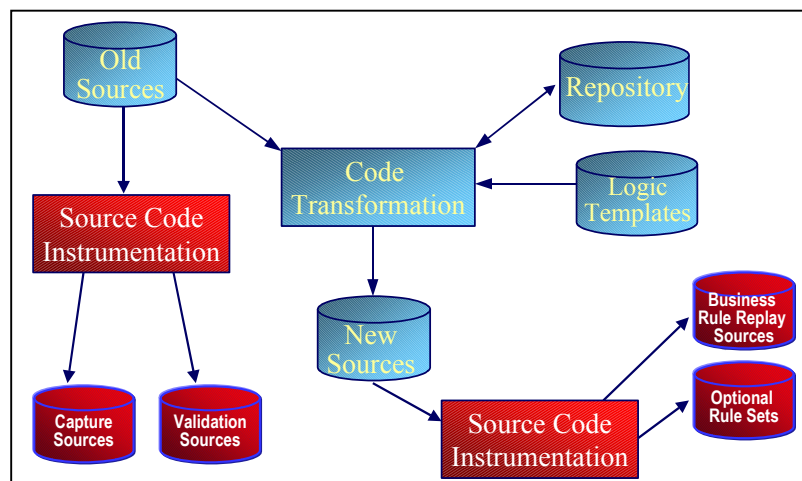
Since testing is such an expensive part of any project, one might reasonably suppose that there are many tools allowing automation of testing. Indeed, there are a few testing products, most notably the online data entry capture and replay systems mentioned above. However, general case automated testing is difficult to do because in normal development testing you always have to do correctness testing. There is no comparison standard against which to regress, and how do you program a piece of software to determine that another program is correct? Having a practical answer to that question would allow significant progress in this area, but sadly none has been found to date.

Automation of migration testing is a different story. Whereas normal development almost always involves a change of business rules, a migration project very carefully avoids changing any business rules, (or at least it does so if it wishes to minimize the overall cost of the project). Changing the infrastructure under which those rules execute is the essence of a migration project, whether it be a platform change, a compiler change, or a database change, or all three. And so long as the business rules do not change, it is possible to fully automate migration testing using a rigorous comparison testing methodology.

### 4.1 Source Code Instrumentation

Source code instrumentation is a process by which software (usually referred to as a “parser”) introduces new program logic into the source code of a given program. Typically, this new logic is functionally neutral in that it does not change the business logic of the program, but the new logic is used for some analytic purpose to understand the operation of the program itself. Coverage analysis is an example of using source code instrumentation to determine what logic in a program is and is not executed. Similarly, performance analysis within a program can be established by timing each paragraph, each branch path, or each instruction in the program and thereby pinpointing where excessive resource consumption is originating.

Automated migration testing utilizes source instrumentation in several forms, initially for capture of test data, and then for one or more different replay scenarios depending on what part of the program is being tested. This summarizes the whole process:



In an automated migration the old sources are processed through a parsing engine that modifies the program source in a controlled fashion to produce the new sources for the new environment. Automated testing augments this by using two forms of instrumentation against the old sources, and one or more forms of instrumentation against the new sources. In the next sections, we will discuss the different source code instrumentation sets and their purposes.

## **4.2 Capture Processing**

In capture instrumentation, the old sources have logic added that captures the before and after contents of all data areas relating to I/O operations, plus all relevant state information, all written to a capture file. In addition, the dynamic logic path through the program is captured to be used for coverage analysis and for logic path testing. All of the captured information is placed into a capture file for subsequent use in replay processing. Note that "I/O" in this sense refers to any operation that causes data to move into or out of the module under test. Thus, a date call to the operating system, a subroutine call, or passed parameters to a called module are equally an I/O as a READ, WRITE, SQL or CICS command.

In effect, for the purpose of validation and business rule replay, the capture file comprises the complete batch or online test script. All inputs and all outputs are recorded automatically, as well as all execution parameters without the opportunity for manual error in securing the scripts and without the involvement of subject matter experts. Note that this all but eliminates the greatest sources of both logistical effort and logistical error in the definition of test scripts.

For I/O replay and datacomm replay, any data store that is randomly updated (typically database files and indexed files) must also be secured at the start of data capture and coordinated with the capture data. Optionally, they can be secured after data capture as well. None of the sequential files need be secured as their full contents will be recorded in the capture file, including printer output and all terminal traffic. When multiple programs (including subroutines) multi-process during capture, it is necessary to include at least a high resolution date/time stamp to provide unique sequencing to the order of I/O's issued, or a sequencing broker introduced to force serialization of I/O's if date/time is insufficient to ensure uniqueness.

## **4.3 Validation Replay**

Validation replay is the inverse of the capture processing. The old sources have logic added to suppress all normal I/O and to take all input I/O data from the capture file and to compare any output I/O against the previous output I/O as stored in the capture file. The logic path followed in the capture program is compared against the logic path followed during replay processing. No external I/O takes place at all except to the replay parameter file and to the replay report file.

This initial replay step ensures that the captured data was secured without processing or handling errors, and also produces a cumulative coverage analysis report by program. If any program requires additional coverage, then the capture program only has to be re-executed with the additional data conditions, and the additional coverage data added to that already captured for that program.

The capture must always occur on the source platform, using the original sources, compiler, database and other components. Depending on the source and target environments, the validation replay may have to occur on the same platform, or it can occur on an inexpensive Intel workstation, for significantly greater efficiency and lower cost.

#### 4.4 Business Rule Replay

Batch programs have two logical layers: the business rule layer and the data access layer. Online programs have three: the same as batch plus the user interface layer. Business rule replay tests the business rule layer of both batch and online programs. The new sources are instrumented with the same logic as inserted into the old sources for validation replay, then the replay is executed. This will occur in the target environment in many cases, but in some an Intel workstation can be used for business rule replay as well as validation replay.

Any errors introduced into the program business logic by the migration process, or any discrepancies created by the change in environment or the change in compilers will appear on the replay report. These discrepancies take two forms: data discrepancy (including state information discrepancy) and logic path discrepancy.

A data discrepancy is found when any referenced field has even a single bit error. The following is an example of a data discrepancy taken from a replay report<sup>2</sup>:

```

Input#          236 Verb# 0150 Verb TRANSFER Record PARW8580-W01-REC
                Return Code 0000 File Status
                Compare=BEFORE Input Seq 006130 COBOL Seq 006128

0001-0002  Cap  ..
  Hi 4Bits  00
  Lo 4Bits  0D
-----+-----1-----+-----2-----+-----3-----+-----4-----+-----5
0001-0002  Rep  ..
  Hi 4Bits  00
  Lo 4Bits  00
!!!!!!!!!!  =X
-----+-----1-----+-----2-----+-----3-----+-----4-----+-----5
    
```

In the captured data, this 2 byte field had a value of X'000D', but in the replay it had a value of X'0000'. All fields, regardless of size, are compared and any discrepancy displayed in this manner. An "X" indicates the byte or bytes that differ in the field.

Similarly, a logic path discrepancy produces an entry in the replay report like the following example:

```

Path Error   Input#          63
  Capture Path# 000009  Cap Src Seq # 001019
  Replay Path # 000011  Cap Src Rec # 001026 Rep Src Rec # 004783
    
```

In this example, during the capture execution the program took branch path #9, whereas in the replay execution the program took branch path #11. Sometimes a logic path discrepancy does

<sup>2</sup> All examples taken from the Test Harness 2000 product from 2000 Technologies Corporation.

not result in a data discrepancy, but indicates that there is a possible latent error that might be expressed as a data error under other conditions. For example, the wrong calculation might be used but the data involved included multiplying by zero, so the result was the same in either calculation.

At other times a logic path discrepancy will pinpoint the exact point in a program that was the ultimate cause of a data discrepancy, allowing the source of the problem to be found in seconds rather than hours.

#### **4.5 I/O Replay**

I/O replay is one of two optional rule sets that can be applied to the new sources. I/O replay tests the I/O interface logic only, not the business rule logic. Since in this case the database and indexed file update I/O's will actually be expressed, it is necessary to secure the before image of the database (and optionally the after image as well) along with the capture file. The before image must be converted to the target database prior to initiating I/O replay. Optionally, the after image can be compared to the resulting database after completing I/O replay.

In the case of a single batch program execution with no concurrent programs or subroutines participating the I/O's, business rule replay and I/O replay testing can be combined in a single replay execution, which is sometimes logistically convenient.

With batch or online programs executing multiple concurrent programs, the various participating programs are instrumented so that only the I/O routines will execute during replay. Then the serialized I/O's are extracted from the capture data repository in temporal sequence across all programs and a special set of captured data created specifically for I/O replay. An I/O replay driver program will read the captured data and pass it to the appropriate program via a CALL. Then the called program will execute the I/O against the database or indexed file and compare the results with the results obtained from the original I/O. In this way, all the I/O's can be tested without the need for application program logic to be invoked. As an interesting side benefit, this I/O driver arrangement can be used to create an I/O stress test to facilitate database tuning.

It depends on the nature of the migration whether I/O replay will be the most important part of testing or whether business rule replay will predominate. Typically, language migrations and platform migrations will find business rule replay to be more important, whereas database migrations will find I/O replay to be more important. Some projects will find both to be of equal importance.

#### **4.6 Datacomm Replay**

Where the architecture of the target data communications environment allows it, the input and output data communications messages can be extracted in a manner very similar to I/O replay, and a datacomm driver utility used to exercise the online programs in the same temporal sequence as the original messages. This arrangement can also be used to create a stress test of the online environment to determine the maximum throughput of the target configuration.

## **4.7 Summary**

Using a capture/replay methodology, it is possible to fully automate migration regression testing, including branch and path coverage analysis. By separately testing the business rule, I/O and user interface layers of programs, all aspects of the program execution can be accurately tested. However, this separate testing methodology only works because 100% of the data and state information is captured and then compared during replay testing. Any attempt to use less than 100% will cause the tests to fail in short order.

As a practical matter, it should be clear that this methodology produces a much higher degree of accuracy than any attempt at manual testing, and further that the methodology provides its own audit by having fully integrated coverage analysis. It should also be clear that the degree of detail and perfection of process required for this testing to succeed cannot be implemented with fallible human beings conducting the testing and performing the comparisons. Only full automation can succeed. As a result, this is not a method that augments manual testing, which is how some test automation tools are used. This is a method that renders manual testing largely irrelevant, at least at the level of unit and system testing where most project testing effort is expended.

In the final analysis, this method must also be less expensive than anything but the most basic of typical testing, since it largely or completely eliminates the most labor intensive parts of migration project testing, which also happen to be the most tedious and error prone aspects of the testing process. If the automated testing is also integrated with the automated migration process, it may be possible to catch more errors earlier in the cycle, reducing most client and vendor costs while improving the accuracy of the results.

## 5 Conclusion and Recommendations

---

As we have explored over the course of this paper, testing will almost always be the largest expense of a migration project. There is an inverse relationship between total cost and project risk, and technicians are not necessarily the best placed to determine the optimum level of both. Technicians are certainly necessary in designing the tests and selecting the testing strategy or strategies to be employed. However, we assert that technicians are not sufficient to do so, and that business management and other stakeholders must be employed in the process as well, since these are projects where technical and business issues intersect.

When an outside migration vendor is employed to make the necessary changes to the program source code, there are degrees of risk that can be borne by the project owner, by the migration vendor, or shared by both. There is a tendency on the part of project owners to expect too much from a migration vendor, particularly one that employs a highly automated process. There is a reluctance on the part of vendors to challenge these expectations. By being too candid, a vendor may lose the business to a competitor who is content to use unrealistic expectations to get a contract and then push their project managers to cover the slack. Thus there is a race to the bottom that benefits no one.

A vendor's total price must of necessity reflect the degree of risk borne by them, or they will not be in business very long. However, accurately quantifying the risk is a fraught enterprise, since by definition the risk is in what we don't know about the application system. Even establishing the boundaries of what we don't know may be difficult, so that we don't know what we don't know, and how can anyone price that risk effectively and fairly? Actuarial methods do not apply due to insufficient historical data.

There are perhaps two ways to approach effectively dealing with this conundrum. One is to separate migration and testing to two different organizations, the "set a thief to catch a thief" approach. This can work and work well, but only if the testing organization is involved from the very beginning of the project to assure that the migration process and the testing process are properly complementary and not antagonistic.

The second approach is to use a migration vendor with a fully integrated approach to both migration and testing, ideally with complementary automation of both processes. This second approach provides the opportunity to cut vendor costs as well as controlling project risks for both client and vendor. Some of the vendor cost savings can be shared with the client, for a better price proposition.

Which is the best approach for your project? As always, it depends on a number of factors so that a suitable set of guidelines is difficult to list that cover even most cases. Perhaps the only valid generalization is that the more business and technical risk there is in the project, the more conservative the technical strategy should be and the more accurate the testing goals should be. We recommend an objective study of both technical and business requirements for the project, with alternatives clearly laid out for consideration for project planning for both client and vendor.