# Testing Algorithmically Complex Software using Model Programs

by

**Liviu-Iulian Manolache**

Submitted in partial fulfilment of the requirements for the degree of MSc in Computer Science
in the Faculty of Science
University of Pretoria
Pretoria
January 2000

Title: Testing Algorithmically Complex Software using Model Programs
Student: Liviu-Iulian Manolache
Supervisor: Prof. Derrick G. Kourie
Department: Computer Science
Degree: MSc in Computer Science

## SUMMARY

This dissertation examines, based on a case study, the feasibility of using model programs as a practical solution to the oracle problem in software testing. The case study pertains especially to testing algorithmically complex software and it evaluates the approach proposed in this dissertation against testing that is based on manual outcome prediction. In essence, the experiment entailed developing a model program for testing a medium-size industrial application that implements a complex scheduling algorithm.

One of the most difficult tasks in software testing is to adjudicate on whether a program passed or failed a test. Because that usually requires "predicting" the correct program outcome, the problem of devising a mechanism for correctness checking (i.e., a "test oracle") is usually referred to as the "oracle problem". In practice, the most direct solution to the oracle problem is to pre-calculate manually the expected program outcomes. However, especially for algorithmically complex software, that is usually time consuming and error-prone. Although alternatives to the manual approach have been suggested in the testing literature, only few formal experiments have been conducted to evaluate them.

A potential alternative to manual outcome prediction, which is evaluated in this dissertation, is to write one or more model programs that conform to the same functional specification (or parts of that specification) as the primary program (i.e., the software to be delivered). Subjected to the same input, the programs should produce identical outputs. Disagreements indicate either the presence of software faults or specification defects. The absence of disagreements does not guarantee the correctness of the results since the programs may erroneously agree on outputs. However, if the test data is adequate and the implementations are diverse, it is unlikely that the programs will consistently fail and still reach agreement. This testing approach is based on a principle that is applied primarily in software fault-tolerance: "N-version diversity". In this dissertation, the approach is called "testing using M model programs" or, in short, "M-mp testing".

The advantage of M-mp testing is that the programs, together, constitute an approximate, but continuously perfecting, test oracle. Human assistance is required only to analyse and arbitrate program disagreements. Consequently, the testing process can be automated to a very large degree. The main disadvantage of the approach is the extra effort required for constructing and maintaining the model programs.

The case study that is presented in this dissertation provides *prima facie* evidence to suggest that the M-mp approach may be more cost-effective than testing based on manual outcome prediction. Of course, the validity of such a conclusion is dependent upon the specific context in which the experiment was carried out. However, there are good indications that the results of the experiment are generally applicable to testing algorithmically complex software.

# OPSOMMING

Hierdie verhandeling ondersoek die uitvoerbaarheid, gebaseer op 'n gevallestudie, van die gebruik van modelprogramme as 'n praktiese oplossing tot die orakelprobleem in programmatuurtoetsing. Die gevallestudie het spesifiek betrekking op die toetsing van algoritmies komplekse programmatuur en dit evalueer die benadering wat in hierdie verhandeling voorgestel word teenoor toetsing wat op handberekende uitkomsvoorspelling gebaseer is. In essensie, het die eksperiment die ontwikkeling van 'n modelprogram behels vir die toetsing van 'n mediumgrootte industriële toepassing wat 'n komplekse skeduleringsalgoritme implementeer.

Een van die moeilikste take in programmatuurtoetsing is om te oordeel of 'n program 'n toets geslaag het al dan nie. Omdat hierdie gewoonlik vereis dat die korrekte programuitkoms "voorspel" moet word, staan die probleem om 'n meganisme vir korrektheidskontrolering (d.w.s. 'n "toetsorakel") daar te stel, gewoonlik bekend as die "orakelprobleem". In die praktyk is die mees direkte oplossing vir die orakelprobleem om die verwagte programuitkomste vooraf met die hand te bereken. Vir algoritmies komplekse algoritmes is so iets egter gewoonlik tydrowend en geneig om foute te bevat. Hoewel alternatiewe tot die handberekende benadering in die toetsingsliteratuur voorgestel is, is daar weinig formele eksperimente uitgevoer om hierdie alternatiewe te evalueer.

'n Potensiële alternatief tot handberekende uitkomsvoorspelling wat in hierdie verhandeling evalueer word, is om een of meer modelprogramme te skryf wat voldoen aan dieselfde funksionele spesifikasie (of gedeeltes van daardie spesifikasie) van die primêre program (d.w.s., die programmatuur wat afgelewer moet word). Indien aan dieselfde invoer onderwerp is, behoort die programme identiese uitvoer te lewer. Teenstrydigheid dui óf op die teenwoordigheid van programmatuurfoute, óf op spesifikasiegebreke. Die afwesigheid van teenstrydighede waarborg egter nie die korrektheid van die resultate nie, aangesien die programme foutiewelik op die uitkomste mag ooreenstem. Indien die toetsdata egter voldoende is, en die implementasies uiteenlopend is, dan is dit onwaarskynlik dat die programme konsekwent sal misluk maar tog steeds ooreenkom. Hierdie toetsing is op 'n beginsel gebaseer wat primêr in programmatuur fouttoleransie toegepas word: "N-weergawe diversiteit". In hierdie verhandeling word die benadering "toetsing met gebruik van M modelprogramme" genoem, of afgekort na "M-mp toetsing".

Die voordeel van M-mp toetsing is dat die programme gesamentlik 'n benaderde maar kontinu verbeterende toetsorakel vorm. Menslike bystand word slegs benodig om program teenstrydighede te ontleed en om oordeel daaroor te vel. Gevolglik kan die toetsproses tot 'n groot mate ge-outomatiseer word. Die hoof nadeel van die benadering is egter die bykomstig poging benodig om die modelprogramme te bou en onderhou.

Die gevallestudie wat in hierdie verhandeling aangebied word lewer *prima facie* getuienis wat aandui dat die M-mp benadering meer koste-effektief mag wees as toetsing wat op handberekende uitkomsvoorspelling gebaseer is. Natuurlik hang die geldigheid van so 'n gevolgtrekking van die spesifieke konteks af waarin die eksperiment uitgevoer is. Daar is egter goeie aanduidings dat die resultate van die eksperiment algemeen van toepassing op algoritmies komplekse programmatuur is.

# TABLE OF CONTENTS

# 1. INTRODUCTION

## 1.1 Motivation and Overview

The gap between research into software quality assurance and practice seems to be problematic. According to Osterweil et al (1996), only a small fraction of the research ideas have been widely applied in practice, and most of these ideas were advanced by the research community more than two decades ago! One of the reasons leading to the gap between research and practice is the lack of solid and well-measured experimental work to evaluate the applicability of research results in practice. Moreover, very little of the work produced by the research seems to be known to the practitioner community.

The core of this dissertation is a realistic and well-measured experiment that evaluates a practical approach to the oracle problem in an important area of software quality, testing. The experiment was conducted in an industrial environment, thus exposing testing practitioners to valuable research results and technologies.

Testing is an indispensable software validation and verification (V&V) technique. In general, other V&V techniques such as walkthroughs, reviews, inspections, and formal verification, although normally cost-effective, cannot be fully relied upon to detect all specification and implementation faults (Osterweil et al (1996)). Moreover, the development and run-time environments are not perfect either and, sometimes, not well understood. Compilers, linkers, interpreters, standard and third party libraries, operating systems, networks, and hardware can also be faulty or just misused. Testing is necessary at least to assess whether the implemented software is operational in a particular environment. Testing also plays an important role in software validation. The user requirements are drawn up based on the user's mental model of the software. Unless the software is exercised, it is difficult to assess how well the user's actual needs are met. Testing is regarded by some authors as the ultimate validation technique. *Chapter 2* will address testing in more detail.

A fundamental and difficult task in software testing is to assess the correctness of the outcomes of a program that is subjected to particular test inputs. In the testing literature, the problem of establishing an appropriate mechanism to adjudicate on whether an outcome is correct or not is usually referred to as "the oracle problem" (e.g., Hamlet (1996), Zhu et al (1997)). Compared to the attention given to other testing aspects, the testing theory has somewhat neglected the oracle problem. Hamlet (1996) points out: "Testing theory, being concerned with the choice of tests and testing methods, usually ignores the oracle problem. It is typically assumed that an oracle exists, and the theoretician then glibly talks about success and failure, while in practice there is no oracle but imperfect human judgement." In testing, an oracle is a means for determining whether a program passed or failed a test[1]. The simplest conceptual form of an oracle is the comparison of the actual output against a pre-calculated expected output. For a small number of simple test inputs, it might be feasible to work out the expected outputs manually. Adequate testing, however, usually requires rather complex and large test data sets. Especially for algorithmically complex software, manual outcome prediction is time consuming and error-prone. Beizer (1995), referring to outcome prediction in control-flow testing, points out: "While your first notion might be to try playing computer and work through the paths manually, don't do it! First of all, it can be very hard work. Second, you're trying to simulate a computer, and that's something none of us humans are very good at: You're likelier to make an error in your manual outcome prediction than the programmer is in programming". It is important to note that his remarks pertain to testing in general and not only to control-flow testing. Richardson et al (1992) alsomention that

[1] Some authors adopt a stricter definition and regard an oracle as a means for predicting the correct program outcomes.

manual result checking is neither reliable nor cost-effective. The next paragraph looks at alternatives to manual prediction of outcomes.

The research community has addressed the oracle problem especially in relationship with formal methods. Stocks and Carrington (1993) discuss the generation of oracle templates from formal specifications. Richardson et al (1992) propose an approach to deriving and using specification-based oracles in the testing process. Peters and Parnas (1994, 1998) developed a test oracle generator that produces a software oracle from relational program documentation. Unfortunately, formal methods have not been widely adopted in industry yet. A major obstacle seems to be the lack of compelling evidence of their effectiveness for industrial-size applications (Pfleeger and Hatton (1997)). Specification-based oracles might be the answer for the future, but, in the meantime, the industry needs better approaches to the oracle problem than manual outcome prediction. Beizer (1995) suggests several practical alternatives. One of them is to build either a detailed prototype or a model program to provide the correct expected outcomes. The testing approach examined in this dissertation is similar. The perspective, however, is slightly different and it is based on N-version diversity, a principle that is applied especially in fault tolerance. N-version diversity and its application to testing are explained in more detail in the next section.

## 1.2 Testing Using M Model Programs

### 1.2.1 N-version diverse systems

N-version diversity is a principle that is applied primarily in software fault tolerance and the technique based on N-version diversity is known as N-version programming (or multi-version programming). The technique is an adaptation to software of modular redundancy (a hardware fault-tolerance technique). In hardware, the diversity is achieved mainly by physical means, whereas in software, the diversity is achieved at design level. A comprehensive overview of fault tolerance and its techniques can be found in Somani and Vaidya (1997). An N-version diverse system comprises N independently written versions of the software, all conforming to the same functional specification (Figure 1-1, Hatton (1997)). At run-time, a voting system that is usually based on majority agreement is used to decide on a single, probable correct, output.

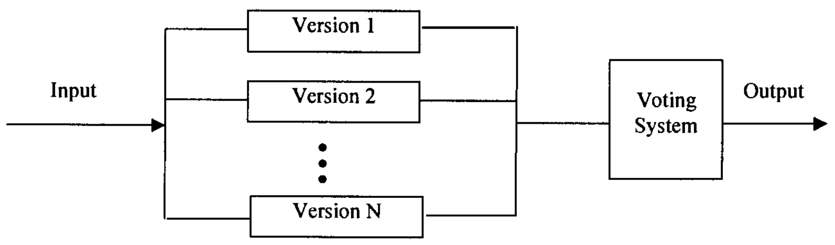


*Figure 1-1: A schematic of an N-version diverse system*

Some of the advantages of an N-version system are:

- It can be much more reliable than any of its individual channels. Hatton (1997) suggests that, given the current state-of-the-art in software development, N-version systems might be the only practical way to achieve high reliability.
- It has self-diagnostic capability. Disagreements can be logged and then used to assess and improve the reliability of individual versions, thus increasing the system reliability even further (Hatton (1997)).

- Because of its self-diagnostic capability, an N-version system plays the role of an oracle with respect to its individual versions and, therefore, there is no need for pre-calculated expected outputs. The testing of an N-version system can be automated to a large degree.

The main disadvantages of N-version systems are the increased development cost[2] and the fact that correlated failures (i.e., versions agreeing on erroneous outputs; also known as "common-mode failure") limit the practical gain in reliability.

### 1.2.2 The N-version diversity perspective on testing

How can N-version diversity be used in the context of testing? The test process can be thought of as depicted in Figure 1-2 (Van Vliet (1996), 320). The box labelled P denotes the object to be tested. The real output produced by P is compared to the expected output that is determined by means of an oracle.

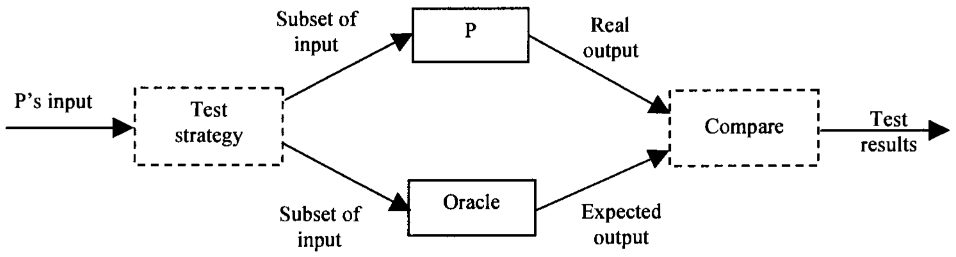


*Figure 1-2: Global view of the test process*

In theory, an N-version system can be completely reliable and it is, therefore, equivalent to an oracle. Consequently, an alternative view of the test process can be obtained by replacing the oracle from Figure 1-2 with M diverse programs that are equivalent to P. Since P and the programs form an (M+1)-version system, the equivalent oracle can be recursively extended to include P. Figure 1-3 depicts the test process from an N-version diversity perspective. V1-V*m* denote the diverse versions.

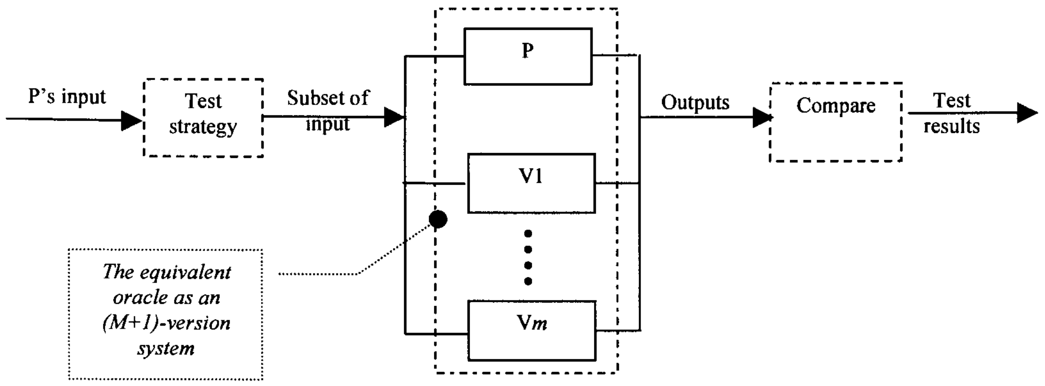


*Figure 1-3: The N-version diversity perspective on testing*

[2] For the voting system to work, an N-version system requires at least three versions.

### 1.2.3 Testing using M model programs

The view of the test process that is depicted in Figure 1-3 should be seen as a theoretical framework. A practical framework can be derived based on three approximations:

- The oracle problem is usually difficult only with respect to certain behavioural properties of P. Moreover, not all of P's properties are equally important. In practice, therefore, V1-Vm can be simplified to cover only the complex and/or critical areas of P's behaviour that would normally require a lot of human assistance for correctness checking. V1 – Vm become thus models with respect to P or, simply, model programs.
- Since in testing the programs do not need to run in parallel, P can be executed first and V1-Vm can be designed to indicate agreement or disagreement with P's output. V1-Vm need not re-compute the output in order to deliver a verdict of agreement or disagreement. For instance, they could implement a set of post-conditions.
- Since arbitrating disagreements essentially entails computing (manually) the correct outputs independently of any of the programs, the disagreement analysis system can be considered as an additional "version".

The practical framework is depicted in Figure 1-4. The primary program (P), the model programs (MP1-MPm), and the disagreement analysis "version" form an approximate (M+2)-version system that may constitute a reasonably accurate and automated oracle. In this dissertation, testing approaches that are based on such a framework will be called "testing using M model programs", or, in short, "M-mp testing".

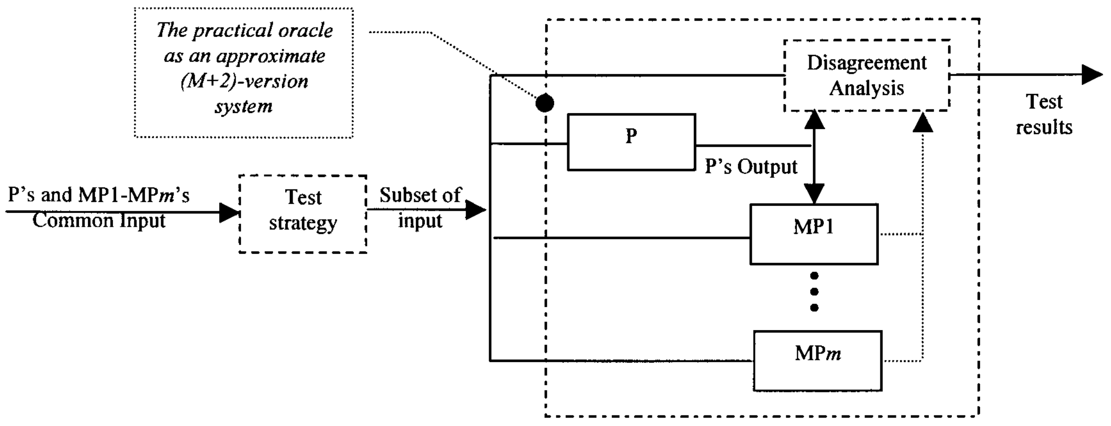

The practical oracle as an approximate (M+2)-version system
P's and MP1-MPm's Common Input
Test strategy
Subset of input
Disagreement Analysis
P
P's Output
MP1
MPm
Test results

*Figure 1-4: Testing framework based on M model programs*

Simply, M-mp testing entails subjecting the primary and the model programs to the same test inputs. If any of the model programs disagrees with the output of the primary program, that indicates the presence of specification defects and/or software faults (in at least one of the programs). Once defects are detected and removed, the programs are re-run. The cycle is repeated until all disagreements for a particular data set are resolved. If all programs reach agreement, the outputs of the primary program that correspond to the selected test inputs can be considered for all practical purposes correct. That is because, as indicated by research into N-version programming, the likelihood of all programs failing identically should be, in general, small. Although the assumption of independence of errors in N-version systems may not hold (Knight and Leveson (1986)), the probability of simultaneous failures of independently written versions seems to be reasonably low (Hatton (1997)). It is also important to note that in general any correctness checking mechanism (i.e., test oracle), whether M-mp based or not, may also fail to detect incorrect outputs. For instance, a tester who works out expected results manually might make the same wrong assumptions as the programmer. Similarly, an executable oracle and the program under test may both contain defects that could lead to correlated failures. In fact, using an executable oracle, which can be seen as an extremely reliable model program, is also a form of M-mp testing. The M-mp framework, however, emphasises that instead of developing an oracle, which by definition has to be highly reliable from the start, it might be more cost-effective to write one or more less reliable model programs. Their quality is likely to increase rapidly along with that of the primary program. Lucid arguments to this effect can be found in Hatton (1997).

### 1.2.4 Applicability of M-mp testing

The applicability of the framework depends in particular on the nature of the primary program. For instance, in the case of embedded applications that work on specialised hardware, because of the strong constraints, building a model program might mean duplicating the primary program. That implies that writing a model program might be as expensive as developing the primary one and the diversity may also be limited, thus increasing the risk of correlated failures. This suggests that M-mp testing might not be beneficial for such applications. On the other hand, as indicated by the case study, the M-mp approach is likely to be suitable for testing algorithmically complex software.

In general, deciding whether M-mp testing is suitable for a particular application should be based ona thorough cost-benefit analysis. The cost of developing and maintaining useful model programs can be high, but so is the cost of inadequate testing. Broadly, a cost-benefit analysis with respect to M-mp testing versus a manual approach should take into consideration the following aspects:

- the cost of building and maintaining model programs (-);
- increased test adequacy (+); and
- improved test automation (+).

## 1.3 Case Study Overview

As mentioned earlier, the core of this dissertation reports on a realistic and well-measured experiment that was conducted in an industrial environment. The purpose of the experiment was to evaluate the merits of M-mp testing versus testing that is based on manual outcome prediction. The experiment will be described in detail in *Chapter 3*, while *Chapter 4* will present the conclusions that were drawn from the case study.

The object of the experiment was a medium-size industrial program that implements a generic scheduling algorithm. The program is data-intensive and its behaviour is customisable to a relatively high degree. Its input domain has approximately 50 dimensions. In broad terms, the program can be characterised as algorithmically complex. By the time the experiment started, the program had already been in use for more than one year and it had gone through several maintenance and testing cycles. Its functional testing was based on a relatively small set of test cases derived from the informal specification of the program. The test inputs were selected mainly based on equivalence class partitioning and boundary value analysis and branch coverage was used as the main adequacy criterion[3]. The expected outcome for each test case was determined manually and that had severe drawbacks:

- Only a small number of test cases could be designed because the cost of selecting adequate test data that is also suitable for manual processing was very high.
- The test cases did not adapt very well to change. A small change in the specification could require that many test cases be re-designed.

[3] Test adequacy criteria will be addressed in more detail in Chapter 2.

- A test case usually captured only the input and the expected output, but not a detailed description of the steps that were used to calculate the expected output. That made the verification of the test cases very difficult.

The M-mp experiment was conducted in response to the above problems and it essentially entailed developing a model program that covered the core functionality of the scheduling application, testing the programs "back-to-back", and analysing their disagreements. The most important results of the experiment are summarised below:

- The development of the model program required only 24% of the total effort that was spent to design test cases manually and it was about ten times less expensive than the development of the primary program.
- Two specification defects and two primary program faults were detected by analysing the disagreements that resulted from 50 tests. Two model program faults and one defect in the test environment were also found.

To keep the cost within acceptable limits, certain pragmatic decisions had to be taken during the experiment. For instance, the model program was designed to cover only the core functionality of the scheduling application and the number of tests was limited. However, as discussed in *Chapter 4*, the *prima facie* evidence suggests that the M-mp approach may be more cost-effective than testing based on manually pre-calculated results. Of course, the validity of such a conclusion is dependent upon the specific context in which the experiment was carried out. However, there are good indications that the experiment is likely to be repeatable and its results may be applicable in general to testing algorithmically complex software.

## 1.4 Related Work

The author is not aware of any other experiment that compares an M-mp approach versus testing that is based on manual outcome prediction. The research community seems to rule out manual result checking (Richardson et al (1992), Beizer (1995), Peters and Parnas(1998)), but in industry, possibly because of too much emphasis on independent black-box testing (Beizer (1998)), manual approaches are still employed. The practitioner community seems to have more confidence in human judgement than in software and the idea of "software used to test software" is considered far-fetched. Moreover, M-mp testing may require good analytical and programming skills. An organisation would generally want to use those scarce skills for development rather than for testing.

Probably the closest work to the case study presented in this dissertation, is the approach to testing described by Buettner and Hayes (1998). The software test team successfully used Mathematica® for unit testing algorithmically complicated software. Mathematica® simulations uncovered numerous ambiguities and errors within the algorithm documents and the corresponding prototypes, despite the high documentation quality that was assumed as a result of several formal reviews. In the context of this dissertation, Mathematica® simulations are model programs and the approach is, therefore, a form of M-mp testing. Buettner and Hayes (1998) also mention improvements on the software process especially increased testers' motivation.

Another closely related work is that presented by Peters and Parnas (1994, 1998). The authors mention that manual result checking can be time consuming, tedious, and error-prone and they propose a method to generate software oracles from precise design documentation. One of the difficulties encountered in their experiment was the presence of specification and, implicitly, oracle faults. From a practical viewpoint, therefore, the generated oracles are equivalent to model programs (highly reliable though).

N-version experiments are fundamentally related to the work presented in this dissertation. The case study was actually triggered by an incidental application of N-version diversity to streamline the testing of an inventory management application. This episode will be described in more detail in*Chapter 3*. N-version experiments inherently address aspects as software diversity, automatic test data generation and disagreement analysis. All of those aspects are important for M-mp testing in general and for the case study in particular. Because it would be practically impossible to cover all the research and experimental work in N-version programming, the work that was most influential with respect to this dissertation will be briefly presented in the next paragraph.

Bishop et al (1986) carried out a research project that mainly evaluated the merits of using diverse software. The authors mention that back-to-back testing successfully detected all the residual seeded faults[4] and it could be carried out in an economic manner. Knight and Leveson (1986) conducted an experiment that addresses the assumption of independence in N-version systems. The main purpose of the experiment was to demonstrate that programs that are written independently do not always fail independently. The authors stress that the experiment result does not mean that N-version programming should never be used. It only shows that the reliability of an N-version system may not be as high as theory predicts under the assumption of independence. Hatton (1997) analyses the same experimental data from a different angle. He argues that even in the presence of common-mode failure an N-version system can still be more reliable than a "one good version" and even more cost-effective, especially in situations where the cost of failure is high. Another interesting N-version experiment was started by Hatton and Roberts (1994). At that stage, the experiment tested back-to-back nine seismic data processing packages written by different vendors. The very first comparisons showed that two packages were so deviant that they were initially excluded until they had been corrected. Overall, analysing the disagreements and feeding back the errors to the package designers led to significant reductions in disagreement. The experiment showed that N-version programming provides a way of detecting errors that is not achievable by other means and that the overall disagreement can be reduced relatively quickly.

It is also important to mention in the context of related work the function equivalence testing technique that is described by Kaner et al ((1996), 135). Function equivalence testing means comparing the results of two programs that implement the same mathematical function. If one of the programs has been in use for a long time, and it can be considered reliable, its actual function is called the "reference function". The actual function implemented by the program being tested is called the "test function". There is only a slight difference between M-mp testing and function equivalence testing. M-mp testing does not assume or require the existence of a reference program, but if such a program exists then, of course, it might be cheaper to buy it rather than develop an equivalent model program. In addition, M-mp testing stresses that the two programs, if diverse, can have close individual reliabilities. In a short time, both will become more reliable. Kaner et al (1996) advocate the use of function equivalence testing whenever possible because it lends itself to automation. The authors also give some guidelines on how to conduct a cost-benefit analysis that promotes function equivalence testing as a replacement to manual testing.

[4] Two faults out of seven were common between two programs and they would have caused the 3-version system to fail. As far as testing is concerned, however, the faults were detected because the third program disagreed on the erroneous outputs.

# 2. SOFTWARE TESTING OVERVIEW

To put M-mp testing and the case study into perspective, it is important to understand what testing entails and why solving the oracle problem effectively is essential for performing adequate testing. This is the purpose of this chapter.

## 2.1 Introduction to Testing

Fundamentally, testing can be defined as the process of executing software on selected test inputs and evaluating whether the software behaved as specified. Often in the literature, testing definitions have a strong bias towards particular goals of testing (e.g., defect detection, reliability assessment, requirements validation). In this dissertation, to make the discussion as objective as possible, testing is not constrained to a specific role or particular objectives. It is also important to note that some authors prefer to use the term "dynamic testing" (e.g., Osterweil et al (1996)), as opposed to "static testing"[1], to indicate explicitly that the software is exercised. In this regard, the term "testing" as used in this dissertation denotes "dynamic testing".

As exhaustive testing is infeasible for most useful software systems, many questions arise in both testing theory and practice. What criteria and source of information should be used to select the test inputs? Is revealing defects more important than confidence building? Can the software be trusted because it has been tested? How can different testing methods be compared to each other? In general, the answers to these kinds of questions are very diverse and sometimes controversial, mainly because testing is usually strongly context-dependent. The definition and choice of testing strategies and techniques are in general largely determined by the characteristics of the software to be tested and by its requirements. Other factors such as budget, time to market, people's skills, etc., may also influence to various degrees the way that testing is done. Consequently, testing approaches may differ considerably across software development projects and even within the same project.

The foremost subject of controversy in the testing literature seems to be the role of testing. It is acknowledged that testing is an effective defect-detection technique. The debate is whether general properties of the software (e.g., reliability) can be inferred from a finite number of tests. The role of testing in the software life cycle is discussed in more detail in the next section.

## 2.2 The Role of Testing

The role of testing in the software life cycle can be best understood from a software verification and validation (V&V) perspective. Testing as a V&V technique will be discussed next based on a simplified view of the software development process.

[1] The term "static testing" is sometimes used to denote V&V techniques that do not require the actual execution of the software (e.g., walkthroughs, inspections, formal verification).

## 2.2.1 Testing as a Dynamic V&V Technique

### *2.2.1.1 A Simplified View of the Software Development Process*

Essentially, any software development process involves the basic transformations as depicted in Figure 2-1: requirements identification, requirements engineering, analysis and design, construction, and source code translation[2].

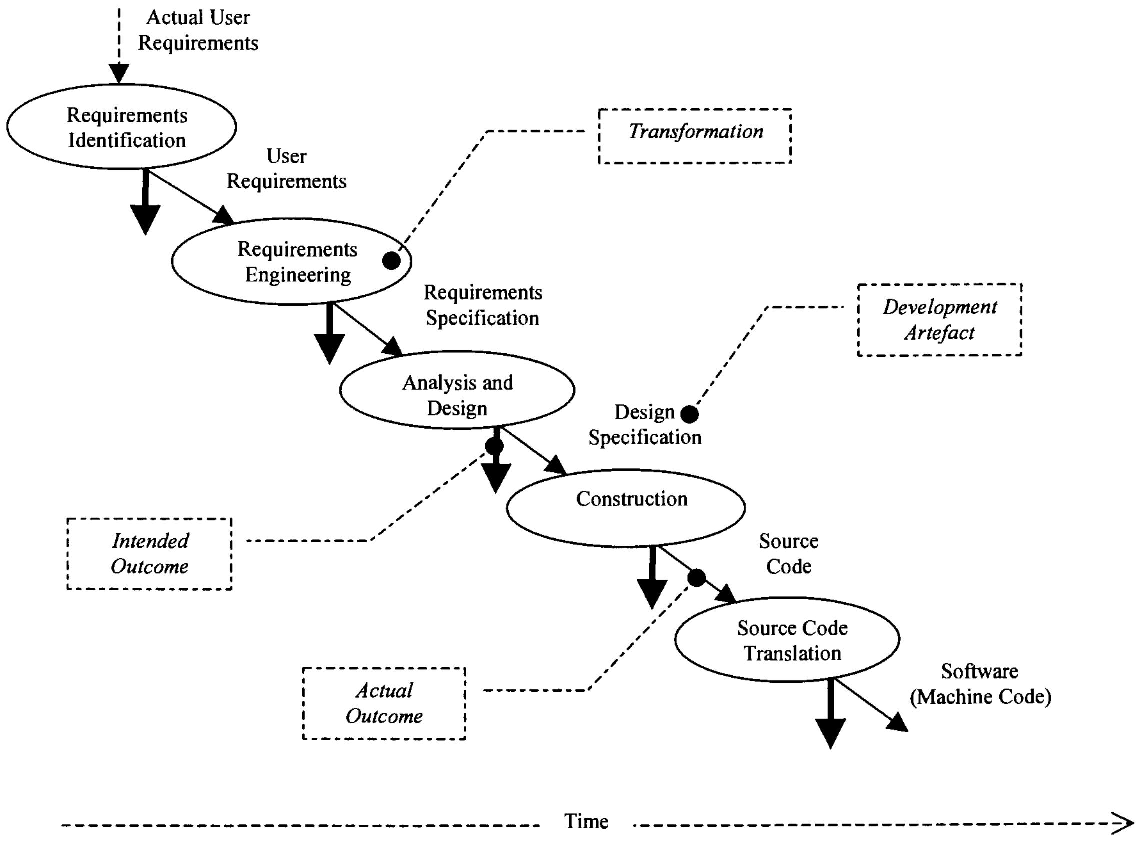


*Figure 2-1: A Simplified View of the Software Development Process*

The identified user requirements are structured into a requirements specification that constitutes the foundation for developing the software using various mechanisms such as refinement, enrichment, and translation. Each transformation may be erroneous, causing its actual outcome to deviate from the intended outcome. Since the output of a transformation is used as input to the succeeding transformations, errors may be propagated. The view of the development process depicted in Figure 2-1, although very simple, provides a good framework for understanding the objectives of V&V and one of its main techniques, software testing.

### *2.2.1.2 The V&V Process*

In principle, the aim of the V&V process is to check the correctness of the outcome of each development transformation. In the literature, a fine distinction is often made between validation and verification (e.g., Sommerville ((1992), 8, 374), Van Vliet ((1996), 319)). Succinctly, "validation" boils down to the question – "Are we building the right product?", whereas "verification" tries to answer the

[2] Various models of the software development process are widely spread in the software engineering literature. A similar view can be found, for instance, in Van Vliet ((1996), 10).

question – "Are we building the product right?". In other words, "validation" means checking the correctness of a development artefact against the actual user requirements, whereas "verification" focuses on checking the correctness of the outcome of a transformation with respect to the actual input to that transformation. The distinction is not that categorical, but it emphasises the importance of making sure that each development artefact satisfies the user's requirements and expectations. If the requirements specification, for instance, fails to reflect the user's needs then the software conforming to that specification will most probably be unusable.

The V&V process usually employs both static and dynamic techniques[3]. Prototyping and testing are dynamic V&V techniques, whereas techniques such as walkthroughs, reviews, inspections, and correctness proofs (or formal verification), are considered static V&V techniques (Sommerville (1992)).

Static techniques involve the systematic examination of software artefacts such as requirements specification, design specification, and source code. They are attractive mainly because defects can be detected early in the software life cycle and the checking results are general (i.e., they are not restricted to particular inputs), but static techniques usually lack validation power. The user requirements are drawn up based on the user's mental model of the software. Unless the software is exercised, it is hard to assess how well the user's needs are actually met. Moreover, the user requirements are usually detailed and unstructured. Their systematic examination can prove to be very difficult.

### *2.2.1.3 The Role of Testing from a V&V Perspective*

In general, prototyping and testing are better suited for validation than static techniques. Compared to testing, prototyping has the advantage that deviations from the actual user requirements can be detected early in the software life cycle. A prototype, however, is just a simplified implementation of the software specification. Its usefulness in validation depends on how well the prototype reflects the specification intent. Testing, on the other hand, exercises the software that will be used in operation and thus the validation is more accurate.

Testing also plays an important role in verification. In general, although normally cost-effective, the static V&V techniques cannot be fully relied upon to detect all specification and implementation faults. Clear arguments to this effect can be found, for instance, in Osterweil et al (1996). Moreover, as emphasised in Figure 2-1, the software is actually the machine code and not the source code. Compilers, linkers, interpreters, and standard and third party libraries may also be faulty or just misused. In addition, operating systems are also software-based and, therefore, they are prone to errors. Since static techniques cannot usually be used further than analysing the source code, testing is necessary to assess whether the software is operational in a particular environment. In this context, it is worth mentioning an experience that the author had. One of the standard library functions provided with C is *fmod(a, b)*, where $a$ and $b$ are real numbers[4]. The function is supposed to compute the remainder of dividing $a$ by $b$, so that $a = n * b + r$, where $n$ is a whole number, $r$ is the remainder, and the absolute value of $r$ is less than the absolute value of $b$. A program failed because *fmod(1.0, 0.1)* returned *0.1* instead of *0.0*. Subsequent tests of *fmod* on various platforms showed that the behaviour of the function is rather unpredictable for $b < 1.0$. The defect could not have been detected using static techniques because the standard library function was assumed to be correct. Of course, after uncovering this kind of error, static techniques may be employed to detect whether known unreliable functions or language constructs are used in the source code.

Testing can detect defects that were introduced at any stage in the software development process. However, the cost of removing a defect after the software is built can be extremely high. Testing should always be used in conjunction with other V&V techniques and, ideally, it should reveal only minor defects

[3] Here and usually in the literature, "dynamic" and "static" indicate whether the checking involves exercising the software or not.

[4] "C" is a programming language. Standard library functions are extensions to the language.

(i.e., the cost of removing a defect is low). Incremental development[5] may also be used to mitigate the risk of late defect detection (Sommerville ((1992), 109), Van Vliet ((1996), 38))

Removing defects only improves the software quality, but to manage effectively the V&V process the quality also needs to be assessed. In software engineering, the most widely used quality measure is the reliability of the software. Besides being a means for defect detection, testing may also be used as a means of assessing the software reliability. Depending on whether the main objective of a testing approach is fault detection or reliability estimation, two kinds of testing are often distinguished in the literature: defect testing and software-reliability-engineered testing[6].

## 2.2.2 Defect Testing vs. Reliability-engineered Testing

To understand software reliability and the difference between the two kinds of testing, it is important to make first a distinction between faults and failures.

### *2.2.2.1 Faults and Failures*

Failures are departures from the intended behaviour of the software that are observed when the software is exercised. Failures are dynamic in nature and they may be caused by one or more faults. In testing, a fault[7] is considered a defect of the software structure (or form). Every time it is exercised, a fault may cause the software to fail.

The notion of a fault is somewhat imprecise. For instance, Frankl et al (1997) show that it is extremely difficult to formally define faults because they have no unique characterisation. A fault is usually defined by the code change that is made to correct the software behaviour. Since the code may sometimes be changed in many different ways to produce the same behavioural effect, the definition does not assure uniqueness. Frankl et al (1997) suggest that one should avoid using the term "fault" in discussing testing and the dependability of software. Nevertheless, despite its limitations, the concept of a fault can still be useful in both theory and practice. For instance, in this dissertation and usually in the literature (Sommerville ((1992), 390), Van Vliet ((1996), 319), Musa (1996)), the difference between defect testing and reliability-engineered testing is made clearer by distinguishing faults from failures.

The notion of a failure (i.e., departure from the intended behaviour of the software) can be considered more precise than the notion of a fault. Still, it needs further refinement especially when used in the context of software reliability as shown next.

### *2.2.2.2 Software Reliability*

The software reliability can be defined as the probability of execution without failure for some specified interval, called the *mission time* (Musa (1996)). This definition is compatible with that used for hardware reliability, but it might be too general for software. In software reliability, the notions of "time" and "failure" are not as precise as in hardware reliability. Usually a hardware component is in continuous use, it fails because of physical ageing and the failures tend to be permanent (i.e., the component has to be repaired before it can be used again). On the other hand, software applications may be idle for long intervals, their failures are often transient, and the consequences of different failures might vary considerably (Sommerville ((1992), 394-395)). To be meaningful, software reliability and reliability metrics have to take into account the application domain and the expected usage of the software when defining "time" and "failures". In general, time is not always seen as calendar time, but also as processor time, or number of

[5] Incremental development is also referred to as "incremental and iterative" or "evolutionary" development.
[6] The term "defect testing" has been adopted in this dissertation from Sommerville (1992), while the term "software-reliability-engineered testing" from Musa (1996).
[7] In the literature, faults are also referred to as "bugs" or, simply, "defects" (actually meaning "software defects").

transactions, or number of runs[8], etc. In addition, the expected costs of failures are used either to partition failures into severity classes and to specialise the reliability metrics for each class (Musa (1996), Sommerville ((1992), 396)) or to adjust the reliability definition (Weyuker (1996)). One of the most used software reliability metrics is the failure intensity (or failure rate) which is defined as the number of "failures" per "time unit". The exact meanings of "failure" and "time unit" are usually determined by the software usage context. Ideally, measuring quality should avoid reliance on the expected usage of the software. Hamlet (1994) suggests a usage-independent quality measure, called "dependability"[9], and a theoretical foundation of software testing: "the dependability theory". "Dependability" captures the intuitive idea of "unlikely to fail". The dependability theory is refined further in Hamlet (1996).

Faults that hardly ever manifest themselves in normal operation or have no serious consequences have little impact on the reliability of the software. Research done in this area (mentioned in Sommerville ((1992), 390)) suggested that, for the software products that were studied, removing 60% of the faults would have achieved only a 3% improvement in reliability. Therefore, instead of trying to detect as many faults as possible (defect testing), it might be more cost-effective to focus on exposing only those faults that are likely to cause frequent or critical failures in operation (reliability-engineered testing).

#### *2.2.2.3 Reliability-engineered Testing*

The main objective of reliability-engineered testing is to estimate quantitatively the reliability of the software based on statistical inference, thus revealing faults that have a big impact on the reliability. In the literature, reliability-engineered testing is also called operational testing (Frankl et al (1997)) or statistical testing (Sommerville (1992))[10]. Essentially, the approach comprises the following steps:

- defining the expected operational profile of the software;
- selecting test inputs from the input domain based on the operational profile and on the anticipated failure costs;
- executing the software on the selected inputs and recording failure information; and
- estimating the reliability of the software using statistical inference.

Essentially, an operational profile reflects the expected usage of the software in operation. An operational profile can be defined for instance:

- as a set of possible input classes together with the probabilities of selecting input from those classes (Van Vliet ((1996), 364), or
- as a set of operations and their probabilities of occurrence for an operational mode or across operational modes (Musa (1996)), or
- as a probability distribution over the state space of a Markov chain (Weyuker (1996)), etc.

Operational profiles are discussed in more detail in Musa (1996), Frankl et al (1997), Hamlet (1994, 1996), and Weyuker (1996, 1998). In principle, the test inputs are selected so that the testing profile is representative for the expected usage of the software. As mentioned in the introduction to this section, the

[8] Simply, a run is an instance of executing a program function on a specific input.
[9] Dependability has also been referred to as "trustworthiness" or "probable correctness" in previous papers.
[10] The term "statistical testing" (or "random testing") is usually used in a wider sense than "reliability-engineered testing". The former term refers to testing approaches that involve random sampling according to any probability distribution over the input space (e.g., uniform, Poisson, binomial, etc.) and not only to the operational distribution. In this regard, reliability-engineered testing is rather a particular case of random testing (Frankl et al (1997), Ntafos (1998)).

cost of failure is an important factor in software reliability. Although certain inputs have low probability of occurrence, failures on those inputs might have serious consequences. Usually, the input selection procedure also takes into consideration the predicted failure costs (Musa (1996), Weyuker (1996)).

Reliability-engineered testing is applicable when it is possible to define realistic operational profiles and it is feasible to carry out a statistically significant number of tests. In such cases, the approach can be very cost-effective. Successes achieved in applying software-reliability-engineered testing have been reported by Musa (1996) and Weyuker (1998). The approach is not usually suitable when the future use of the software cannot be predicted accurately or the reliability requirements are too high. In the former case, the validity of any operational profile might be too uncertain. In the latter, the required number of tests might be prohibitive. For instance, estimating a failure rate in the region of "$10^{-6}$ failures/run" would require millions of tests.

*2.2.2.4 Defect Testing*

As the name suggests, the main goal of defect testing is to detect as many faults as possible by selecting test inputs on which the software is likely to fail. Human intuition may play an important role in selecting peculiar test inputs, but well-defined testing methods are methodical and generally do not rely on intuition. Adequacy criteria (e.g., structural coverage of the specification or the program, fault-detection ability of test data sets, etc.) play an important role in selecting test inputs and measuring the sufficiency of the tests. Defect testing is also called systematic testing (Podgurski (1991), Frankl et al (1997)[11]) or partition testing (Ntafos (1998)).

Reliability estimation is not exclusive to reliability-engineered testing. Although in general it is difficult to estimate quantitatively the reliability of a program that passed an adequate test (Zhu et al (1997)), reliability estimation is also possible in defect testing. For instance, the number of residual defects and even the future failure rates can be predicted based on reliability growth models (Wood (1996)). The confidence in the estimate, however, depends on how "typical" the software is with respect to some "population" of programs on which the growth model was based (Podgurski (1991)). In defect testing, precise reliability estimates are achievable if the input domain is partitioned into homogeneous sub-domains. In this case, partition testing is an application of stratified random sampling[12] and, thus, accurate reliability estimates can be made (Ntafos (1998)).

It is also important to note that the expected usage of the software may play an important role in defect testing. For instance, functionality that is known or perceived as very important to the user is usually tested more thoroughly than "trivial" functionality. In addition, the anticipated failure rates and the cost of failure are usual heuristics used in practice to prioritise fault removal.

### 2.2.3 Summary

The main roles of testing in the software life cycle are defect detection and reliability assessment. Although sometimes disputed, the validity of statistical reliability estimation through testing is supported by the successes achieved in applying reliability-engineered testing as mentioned, for instance, by Musa (1996).

The main aim of discussing defect testing vs. reliability-engineered testing was to exemplify how testing may fulfil its dual role in the software life cycle. The comparison of the two kinds of testing is not meant to be complete. In-depth analytical or empirical comparisons can be found for instance in Frankl et al (1997) and Ntafos (1998). The main differences between defect testing and reliability-engineered testing, as

[11] Frankl et al (1997) prefer to use the term "debug testing" as a collective name for systematic testing methods. The term might be misleading since it could be interpreted as "testing by means of stepping through the code (i.e., debugging)". Perhaps a better name for defect testing would be "systematic defect testing".

[12] Stratified random sampling is a statistical sampling technique that is generally used when the population can be divided into homogeneous, or similar, sub-groups (strata).

discussed in this section, are summarised in Table 2-1. The two kinds of testing should not be seen as rigid testing approaches. Ideally, they should be used jointly and, as the software matures, the emphasis should shift from fault detection to reliability estimation (Sommerville ((1992), 400), Osterweil et al (1996)).

| | **Defect Testing** | **Reliability-engineered Testing** |
|---|---|---|
| **Main Objective** | Fault detection | Quantitative reliability estimation |
| **Primary Source of Information for Selecting Test Inputs** | Structural, behavioural, or fault-based models of the software or its specification | The expected usage of the software |
| **Reliability Assessment Characteristics** | Based on statistical correlation across similar projects or statistical inference from stratified random sampling; usually qualitative | Based on statistical inference from simple random sampling; quantitative |

*Table 2-1: Differences between Defect Testing and Reliability-engineered Testing*

Software engineering usually deals with the development of large and complex software systems. To fulfil its role (or roles) efficiently, testing employs various and well-defined strategies and techniques throughout the software life cycle. The testing process is discussed from a generic viewpoint in the next section.

## 2.3 The Test Process

In general, large software systems are decomposed into smaller, manageable components that are integrated incrementally into the final system. Testing is usually done in conjunction with system implementation and proceeds in stages. Before testing the system as a whole, individual components and groups of components are tested as they are developed. At high-level, testing comprises a whole life-cycle process. At low-level, however, whether the software to be tested is a system or part of a system, testing is essentially an iterative process of test design, test execution, and test evaluation. The low-level, intrinsic process of testing is discussed next.

### 2.3.1 The Intrinsic Process of Testing

The view of the intrinsic process of testing that is depicted in Figure 2-2 revolves mainly around the notion of "test adequacy" that is comprehensively described in Zhu et al (1997). The view is intended to be generic enough to accommodate any testing approach and it will be used as the basis for presenting the case study in the next chapter.

Underlying to the view is the life cycle of a test: a test is designed, executed, and evaluated[13]. Since the software and its documentation may change because of defect removal, a test usually goes through the same stages repeatedly (i.e., a test may be re-designed, re-executed, and re-evaluated). The view depicts only one "spiral".

[13] "Executed" and "evaluated" actually mean that the software is exercised and the execution outcomes are evaluated according to the correctness checking criteria.

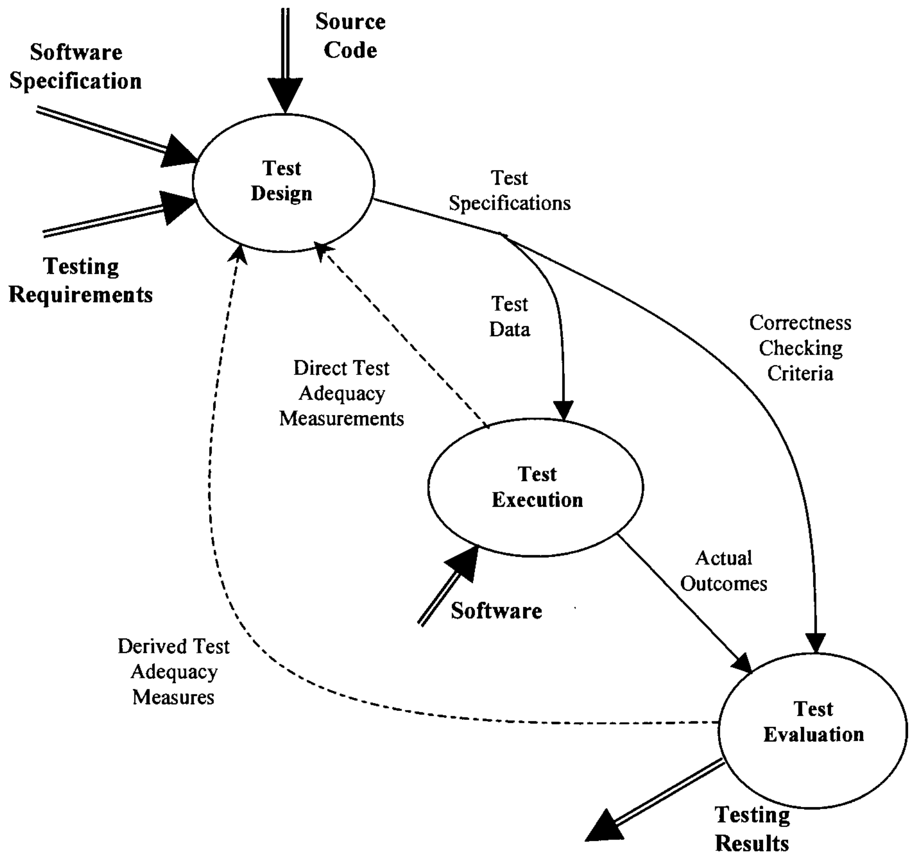


*Figure 2-2: The Intrinsic Process of Testing*

Because in general the testing terminology may vary greatly from author to author, the terms that pertain to understanding the intrinsic process of testing as depicted in Figure 2-2 are described, specific to this dissertation, in the table below:

| Term | | Description |
|---|---|---|
| Actual Outcomes | - | the software behaviour characteristics (e.g., displayed screens, logged messages, execution traces, memory usage, response times, computation results) that are identified (i.e., observed, traced, measured, or derived) for a test execution; the actual outcomes[14] are evaluated according to correctness checking criteria during the test evaluation activity |
| Correctness Checking Criteria | - | the rules that determine whether the software passed or failed a test; evaluating the actual outcomes according to correctness checking criteria is usually considered to be done by means of a "test oracle"; the simplest form of an oracle is the comparison of actual and expected outcomes |

[14] Although terms as "output", "result", and "outcome" do not have exactly the same meaning, they are usually used interchangeably. Probably the term that is most appropriate to describe the software behaviour is "outcome" (Beizer (1995)).

| **Term** | | **Description** |
|---|---|---|
| Derived Test Adequacy Measures | - | test adequacy measures that are derived during the test evaluation activity from the test records (e.g., mutation adequacy score) |
| Direct Test Adequacy Measurements | - | the test adequacy measurements taken during the test execution activity (e.g., branch coverage, definition-use coverage) |
| Test | - | the act of exercising a distinct piece of functionality of the software and evaluating the correctness of the actual outcomes according to well-defined criteria (i.e., correctness checking criteria); a test spans over both test execution and test evaluation activities |
| Test Adequacy Criterion | - | a rule (or criterion) specifying the way in which a set of test data is to exercise the code; the extent to which the test data complies with a criterion determines the test adequacy level (in terms of the particular criterion) |
| Test Case | - | generic term that is used in the literature to denote either a "test input" (e.g., Cohen et al (1997), Zhu et al (1997)), or a "test specification" (e.g., Somerville (1992)), or a "test record" (e.g., Richardson (1994)) |
| Test Data | - | the subset of the input domain that is used to test the software; the test data is the collection of all test inputs |
| Test Design | - | the testing activity that sets up test adequacy criteria and designs tests to meet those criteria at a given level of adequacy |
| Test Evaluation | - | the testing activity that establishes whether the software passed or failed each test and determines the overall testing results |
| Test Execution | - | the testing activity that sets up the test execution environment, exercises the software according to the test specifications, performs the necessary measurements, and records the actual execution outcomes |
| Test Execution Environment | - | a particular software and hardware infrastructure that enables the execution of the software according to the test specifications |
| Test Input | - | the data that is used to exercise the software during a test; it comprises both control and computation data |
| Test Record | - | the information that characterises a test from inception to completion (e.g., the versions of the test specification, of the execution environment, and of the software, actual outcomes, failure analysis information) |

| Term | | Description |
|---|---|---|
| Test Specifications | - | the specifications of test input, along with the correctness checking criteria |
| Testing Requirements | - | the expression of the particular role that testing is expected to play in the software life cycle; testing requirements can be general statements (e.g., "detect as many faults as possible") or they can impose specific test adequacy criteria (e.g., a minimum branch coverage level) |
| Testing Results | - | broadly, test records, test adequacy measures, and software quality estimates (e.g., reliability) |

*Table 2-2: Brief Glossary of Testing Terms*

Before discussing the test design activity in more detail, it is important to look first at test adequacy criteria.

### *2.3.1.1 Test Adequacy Criteria*

Since exhaustive testing is unfeasible, one of the most important problems to be addressed in testing is that of selecting a subset of the input domain that will exercise the software "thoroughly enough". Whether the main role of testing is fault detection or reliability estimation, in the end "everybody wants to believe that software can be trusted for use because it has been tested" (Hamlet (1994)). The objective rules that try to ensure the sufficiency of testing with respect to particular testing requirements are called test adequacy criteria. They can be defined either as guidelines for test data selection (test data selection criteria) or as targets for test quality measurements (test data adequacy criteria). For instance, a data selection criterion could be an algorithm that ensures proper coverage of linear domain boundaries (domain testing). A data adequacy criterion could be stated, for example, as "the tests shall exercise at least 80% of all branches of the control-flow graph of a program". A comprehensive survey on test adequacy criteria can be found in Zhu et al (1997).

### *2.3.1.2 Test Design*

The test design activity sets up test adequacy criteria and designs tests to satisfy those criteria[15]. Broadly, a test specification states:

- the initial states of the software and of the test execution environment (including the computation data),
- the stimuli that cause the software to execute and their sequence (i.e., control data), and
- the criteria for determining the correctness of the actual outcomes.

A test is usually designed to cover a distinct, relatively small, piece of functionality of the software. In the simple case of a "batch" program (i.e., a program that starts, processes its input data, produces results, and terminates) a test involves in general a single execution of the program. Although a performance test may

[15] Setting up test adequacy criteria and designing tests could be seen as separate activities. They have been combined together mainly for simplicity and to emphasise the strong dependency of test design on adequacy criteria.

require multiple executions of the program, it could be thought of as an aggregation of single execution tests. In the case of interactive software, a test may be delimited, for example, by two "idle state" screens (i.e., user input is required to change the state of the software). In general, a test exercises the software between two well-defined, idle states of the world (which is made up by the software and the test execution environment).

Test specifications inherit their initial level of detail from the source of information used in test design. For instance, test specifications that are drawn from the requirements specification are inherently high-level. As the design specification, the source code, and the software become available, the high-level test specifications have to be refined in terms of executing and evaluating the actual software. Ultimately, the concrete forms of test specifications are embodied into test procedures that are used during test execution and evaluation.

#### *2.3.1.3 Test Execution and Evaluation*

Theoretically, test execution and test evaluation could be seen as a single activity. A test is complete only after determining the correctness of the execution outcomes. In practice, however, the methods and tools employed during test execution and test evaluation have a different nature. Executing the software is usually done by a test driver, whereas outcome verification by an oracle (a comparison tool in the simplest case). In addition, it is in general more cost-effective to group tests into test suites, exercise first the software for the entire suite and then evaluate the execution outcomes.

### 2.3.2 Test Automation

Automation is imperative in testing for two reasons. Firstly, the most desirable adequacy criteria usually require an overwhelming number of tests. Secondly, mainly because of software evolution, testing may involve frequent iterations. The testing literature provides numerous arguments and examples of why manual testing is infeasible (e.g., Beizer ((1995), 232), Richardson (1994), Eickelmann and Richardson (1996)). Test automation, however, may also have drawbacks. According to Kaner et al ((1996), 282), documenting and automating tests may take ten times longer than creating and running them once. It follows that test automation is justified only if more than say, ten testing cycles are anticipated. By test automation, the authors actually mean automating the test execution and evaluation activities. In this case, if the software is subject to significant and frequent changes, test automation may prove very expensive because some of the test specifications and procedures may need to be changed manually. Automation in testing, however, is possible beyond test execution and evaluation. Test design and setting up the test execution environment can be automated as well. For instance, Eickelmann and Richardson (1996) describe and compare three Software Test Environments (STEs) that share the goal of achieving fully automated testing. The case study that is presented in the next chapter is also a good example of how testing can be automated in a cost-effective manner.

Automating the test design activity is not easy. In general, it is possible to generate test data efficiently. However, providing a cost-effective mechanism for correctness checking might not be as straightforward. Working out the expected outcomes manually is usually infeasible for adequate test data. In the literature, the most discussed alternative to manual outcome prediction is generating executable oracles from formal specifications. However, as suggested in the first chapter, even if the software is informally specified, it may still be possible to eliminate the need for calculating expected outcomes manually by writing one or more model programs (i.e., M-mp testing). This dissertation's case study evaluates the merits of such an approach. The *prima facie* evidence indicates that M-mp testing may be a practical alternative to manual outcome prediction in the case of algorithmically complex software. The case study is described in *Chapter 3*.

# 3. CASE STUDY

Broadly, the experiment entailed writing a model program that conformed to the functional and interface specifications of a scheduling application and testing the two programs "back-to-back". The scheduling application is part of a family of third-generation language (3GL) programs that implement complex operations-management algorithms. The programs are similar in nature (i.e., non-interactive data-processing applications) and they have been developed using the same development and testing methodologies. Although conducted only on the scheduling program, the experiment targeted the entire family. The case study was actually motivated by the achievements of an incidental application of M-mp testing during the development of another program, an inventory management application. The general context and this "pilot study" are briefly described in the next section.

## 3.1 Background and Motivation

### 3.1.1 The General Context

The programs that the experiment targeted have been developed at Paradigm Systems Technology, South Africa. They are non-interactive (i.e., "batch") 3GL applications that implement complex operations-management algorithms and they are integrated into a large logistics management system, EPMS®. Essentially, as depicted in Figure 3-1, the operations-management programs read the *Input Data* from the *User Database* and convert it into *Generic Input* according to the *User-defined Mapping Rules*. The computed *Generic Output* is converted into user-specific *Output Data* and written to the database. The computation algorithm can be customised by means of *Control Parameters*.

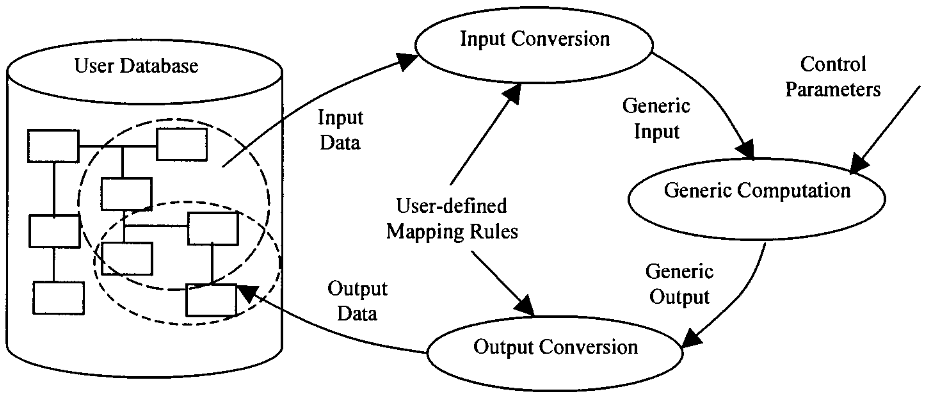


*Figure 3-1: Generic View of the Complex Data-Processing Programs*

The programs have the following general characteristics:

- They deal with complex input-output (I/O) data structures.
- They solve complex problems (e.g., maintenance-task scheduling, resource balancing, inventory management).
- They provide mechanisms for customising both the I/O interface and the computation algorithms.

Designing test cases manually for such programs is a very difficult task because the input spaces are fairly large (20 to 100 dimensions) and the algorithms are complex (in some cases they are based on heuristic search). The testing process, however, although well defined and supported by standards and procedures, practically imposes manual outcome prediction and, consequently, it has the following drawbacks:

- Working out the expected results manually is a tedious and time-consuming job. Testing, therefore, is rendered as an unattractive and demoralising job, thus leading to staffing problems.
- The tendency is to keep the number of tests as small as possible and to select test inputs that are simple enough to be "manually processed". The test adequacy is questionable as the (only) data adequacy criterion used, "65% branch coverage", is probably insufficient itself, given the complexity of the programs.
- Maintaining the test specifications tends to be very expensive since a small change in the specification could require the manual re-design of many tests.
- To keep the volume of test documentation within reasonable limits, test specifications usually capture only the inputs and the expected outputs, but not a detailed description of the steps that were followed to determine the expected results. Therefore, verifying a test specification can be as time-consuming as creating it.

All the above deficiencies of manual test design are well known. Yet, no alternatives have been considered worth exploring, mainly because of budget limitations and lack of hard evidence that other approaches could be more cost-effective. The first time M-mp testing showed its potential as a cost-effective alternative to manual outcome prediction was during the testing of an inventory management application. This circumstance is presented next.

### 3.1.2 The Pilot Study

To understand the pilot study it is important to give first a brief description of the inventory management program. Essentially, the aim of (time-phased) inventory management is to make sure that there will be always sufficient stock to satisfy all planned material requirements and to keep the stock level "as low as possible". The simplest form of inventory management that is usually applied to low-cost materials is to consider the planned orders (i.e., due-in quantities) and the material requirements fixed and to generate procurement advice notices in order to balance out projected shortages. In the more general case, since orders and requirements may also be modified (e.g. moved or cancelled), inventory management systems can produce one or more combinations of different recommendations such as procurement, re-scheduling, or cancellation. The inventory management program implements such an algorithm. The application domain of the program is exemplified in Figure 3-2, where the arrows labelled A, B, and C are planned orders, while M and N represent material requirements. The lengths of the arrows reflect the due-in or due-out quantities. In the figure, the procurement advice, labelled D, caters for the projected negative balance caused by the requirement N. Delaying N would have the same effect (i.e., preventing the shortage). The procurement and re-scheduling recommendations are mutually exclusive. The cancellation advice tries to prevent the excess stock that will be caused the planned order B.

After hard work on both development and test design sides, the project reached a crisis. The computation results differed from the manually predicted outputs for a large number of tests and there was practically no guarantee that the test specifications were more reliable than the program itself. The disagreement resolution was progressing very slowly, putting at risk the planned date for the first release.

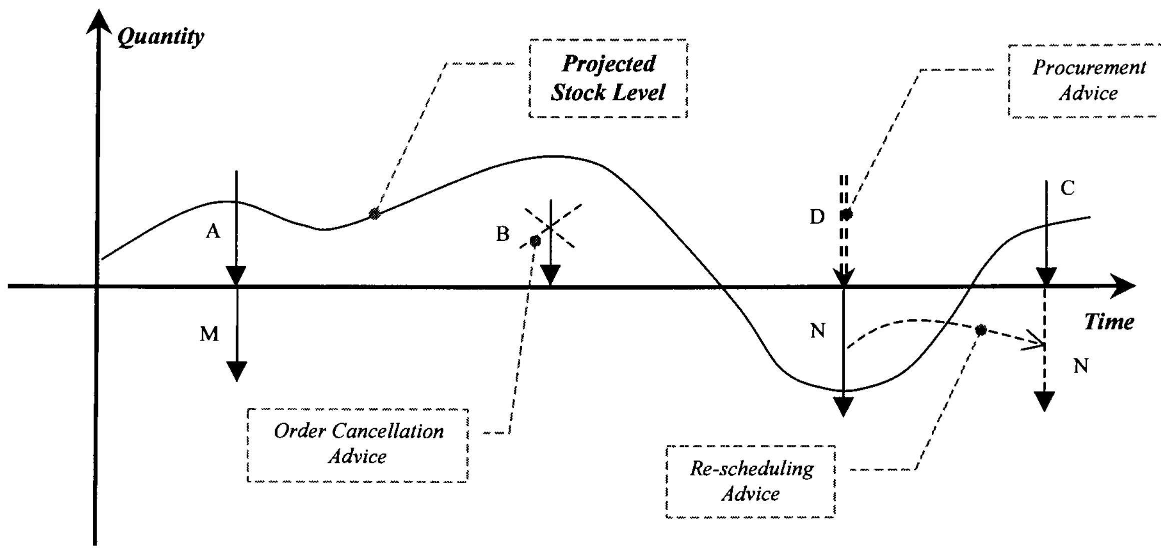


*Figure 3-2: Overview of Inventory Management*

After becoming aware of N-version programming as a test strategy option, the author resolved to write a simplified version of the program. This model program is the subject of the pilot study. It implemented only the basic algorithm (i.e., it generated only procurement advice notices) mainly because of time constraints and uncertainty around M-mp testing. Even with this limited functionality, the model program still proved very useful. It provided a welcome "third opinion" (the other two opinions being the original program output, and the manually predicted test output) and, consequently, the disagreement resolution became more efficient:

- if the two programs reached agreement, the corresponding test specification had to be re-visited;
- on the other hand, if only the model program "passed" a test, the primary program was assumed incorrect;
- finally, when the primary program and the test specifications agreed, this provided good checking points for the model program.

The primary program, the model program, and the test specifications mimicked a 3-version system. Although only the basic functionality was tested in this way, some of the defects that were found pertained to the general case as well. In the end, the reliability of the primary program increased rapidly enough and the crisis was successfully passed. Even now, after two years, the model program is still used sometimes when new tests are added or previous tests have to be re-designed because of specification changes.

Unfortunately, no accurate records have been kept and it is difficult, therefore, to make a proper cost-benefit analysis. It is important to note, however, that writing the simplified algorithm took around 20 person-hours. That represents about 2.5% of the manual test design cost and around 1% of the primary program development effort! The general feeling of the project team was that the benefits definitely outweighed the cost.

In this experiment, the model program was written to arbitrate the disagreements between the primary program and the manually pre-calculated results. It was not intended to be, and it could not have been, an alternative to manual outcome prediction. Besides the fact that the model program had limited computational scope, it also used the same software architecture and I/O mechanisms as the primary program (refer also to

i 15787710
b15225197

Figure 3-1). Consequently, the probability of common failures that could have been caused by architectural and I/O interface faults was high. The limited implementation diversity was compensated for in this case by the manually pre-calculated results, which are inherently implementation-independent[1].

The experiment raised two fundamental questions:

- Could the necessity for manually pre-calculated results be eliminated by using diverse model programs that implement the same function as the primary program (i.e., M-mp testing)?
- Would it be feasible to build such programs?

The answer to the first question is rather obvious. As also implied in the first chapter and the pilot study, there is no guarantee that exercising the specification manually (i.e., "playing computer" as Beizer (1995) calls it) is more reliable than a program that does it automatically. Moreover, a model program can be a more accurate, objective and verifiable reflection of the tester's mental model of the software than a small set of manually produced test specifications. The answer to the second question is not that obvious and it could depend largely on the testing context. However, the pilot study indicated that the M-mp approach might be feasible for testing algorithmically complex software such as the inventory management program. Therefore, a larger scale experiment was conducted to evaluate whether M-mp testing could be more cost-effective than the manual approach in the case of testing algorithmically complex software. The case study is presented in the remainder of this chapter.

## 3.2 Experimental Strategy

### 3.2.1 Method Comparison Strategy

The main difference between M-mp testing and the manual approach is the use of different correctness checking mechanisms. To recapitulate, in M-mp testing, outcome verification is done by means of one or more model programs that give verdicts of agreement or disagreement with the outputs of the primary program. If agreement is reached, the outputs are considered correct. Otherwise, the disagreements are analysed in order to detect defects. If a model program is faulty, it has to be corrected. In the manual approach, the outputs of the (primary) program are compared against pre-calculated expected outcomes. As in the case of M-mp testing, disagreements are analysed in order to detect defects. That might require re-calculating the expected outputs.

As suggested in Figure 3-3, the *Cost of M-mp Outcome Verification*, although initially high because of the *Cost of Model Program Development*, is expected to increase only slightly as the number of tests grows. Of course, that depends on how reliable and maintainable the model programs are. (The non-zero slope of the *Cost of M-mp Outcome Verification* graph reflects the disagreement analysis and model program maintenance effort. For simplicity, the cost growth is modelled as linear.) However, the reliability of a model program, as highlighted in the first chapter, is likely to increase rapidly along with that of the primary program. Moreover, high maintainability could be ensured by simple model program design and implementation. That was the case in the experiment.

In the manual approach, the growth of the outcome verification cost is expected to be much steeper than in M-mp testing. That is because, besides disagreement analysis and correcting expected results (the equivalent of model program maintenance), manual outcome verification requires pre-calculating outputs for

[1] In this dissertation, "implementation diversity" and "design diversity" are treated as distinct notions. The former implies using different development environments and third-party components, whereas the latter implies different design methods. In the pilot study, the algorithm designs were different, but the implementation forms were the same.

each test. In Figure 3-3, for simplicity, the growth of the *Cost of Manual Outcome Verification* is modelled as linear.

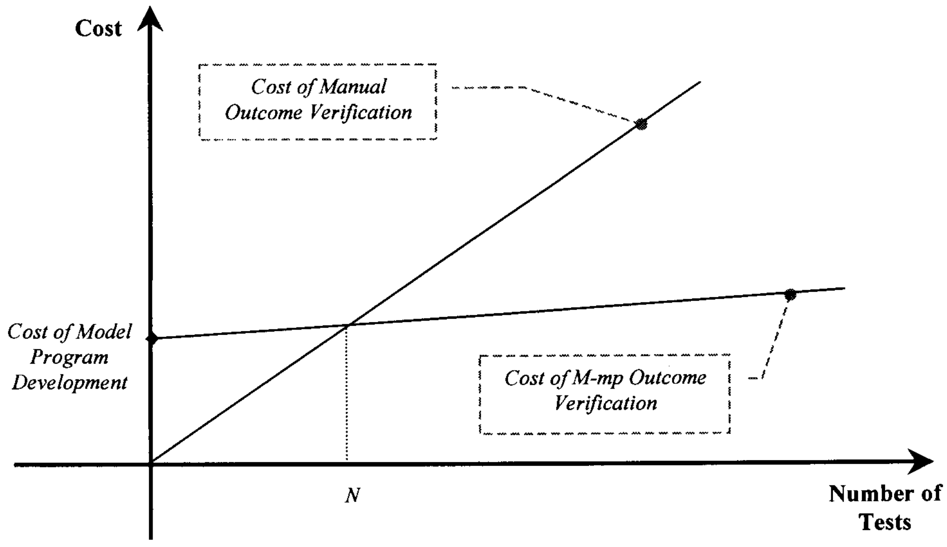


*Figure 3-3: Predicted Manual vs. M-mp Outcome Verification Costs*

Figure 3-3 could serve as a reasonable comparison model. If the same test data selection method is used in both approaches, M-mp testing would be a better option than the manual approach if the number of tests that achieve a particular adequacy level is greater than *N*. The model, however, does not sufficiently emphasise a major advantage of M-mp testing over the manual approach: the possibility of achieving the same (or higher) adequacy levels at lower cost of data selection.

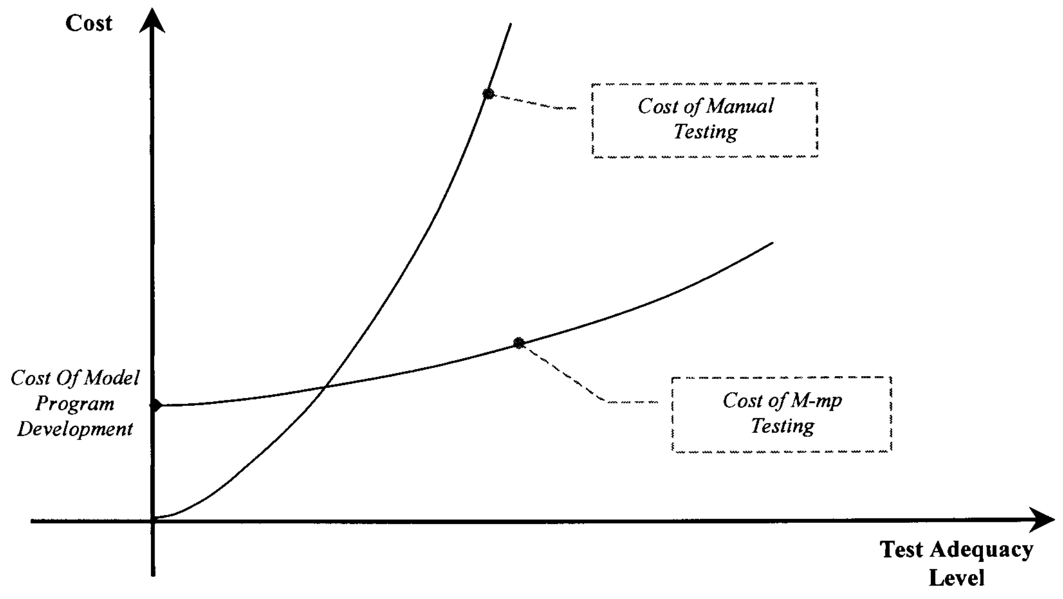


*Figure 3-4: Predicted Manual vs. M-mp Testing Costs*

Intuitively, the size of a test data set that satisfies a particular adequacy criterion is determined by the precision of the data selection procedure with respect to that criterion. If the intention is to keep the number of test inputs as small as possible, as it is the case in the manual approach, then the selection procedure could turn very complex and, therefore, expensive. On the other hand, if it is feasible to carry out large numbers of tests, as is expected to be the case in M-mp testing, then the accuracy of the selection procedure can be relaxed, thus reducing the cost of test data selection. In other words, given a particular test adequacy requirement, M-mp testing can use larger data sets, but less expensive, than the manual approach. This aspect is discussed in more detail in *Chapter 4*. A more suitable comparison model, which takes into account the cost of test data selection, is depicted in Figure 3-4. Because both the number of tests and the cost of selecting data may increase largely as the adequacy level increases, the *Cost of Manual Testing* and *Cost of M-mp Testing* graphs have non-linear shapes.

Simply, the main objective of the experiment was to provide *prima facie* evidence that M-mp testing, given the particular context, could achieve higher adequacy levels, at lower cost, than the manual approach. The implementation strategy that was devised to achieve this goal is presented in the next sub-section.

### 3.2.2 Implementation Strategy

To make the experiment feasible, the main challenge was to develop a diverse, reliable, and maintainable model program in a very short time. Therefore, the strategy of model program development has been carefully addressed from the start.

As also discussed in the first chapter, a model program should cover only those behavioural properties of the primary program that are difficult or tedious to verify by other means (including manual outcome prediction). For reasons that will become clearer in *Chapter 4*, in the experiment, the model program targeted from the start only the positive testing (i.e., subjecting the software to standard input data) of the primary program. Briefly, that is because verifying whether a program handles non-standard data correctly could be automated efficiently by other means than using model programs. Moreover, the complexity of a program may be reduced largely by limiting its exception-handling capability.

As shown in Figure 3-1, the data-processing applications that are targeted by this case study provide generic mechanisms that allow the customisation of the I/O interface. Moreover, the applications can access different databases (e.g., Oracle, Informix) and they are portable across various operating systems (e.g. Windows NT, UNIX). A model program, however, does not need to be as generic as the primary program. That is because most of the primary program's functionality is non-I/O and non-platform related. Therefore, it is sufficient for a model program to cater only for one platform and a fixed I/O model. The test data and the corresponding program outputs that are verified by means of M-mp testing can be easily migrated from the particular test environment to other platforms and/or different I/O models.

Besides the above simplifications (i.e., valid input data, single-platform, and fixed I/O model), the development strategy also included the following guidelines:

- To ensure a high degree of diversity, a model program should avoid using the same design methods, third-party components, programming language, or software architecture as the primary program.
- To facilitate efficient automation of test execution and evaluation, the interface of the model program should be fully compatible with that of the primary program (i.e., the two programs should be interchangeable in the test execution environment).
- High reliability (i.e., correctness of computation results) and maintainability should be given a higher priority than other desirable properties such as high performance, low memory usage, and small size. Even if a model program is computationally inefficient in a normal execution environment, it might still be possible to execute it efficiently in an enhanced environment that is specifically available for testing (i.e., faster processor, more memory, and more storage space).

The above simplifications and guidelines ensured cost-effective design and implementation of the model program. Its development and the other activities of the experiment will be described in the next section.

## 3.3 Experiment Description

To put the experiment into perspective and to facilitate the correct interpretation of the results, it is important to give first an overview of the primary program.

### 3.3.1 Primary Program Overview

Although the pilot study could have been extended for the purpose of this experiment, another application, a scheduling program, was chosen for the following reasons:

- The author was not involved in the development of the scheduling application. Consequently, the cost of model program development can more authentically provide evidence for the general case of "developing from scratch". Moreover, the risk of correlated failures that could be caused by common design errors was reduced.
- The scheduling application is representative of the family of complex data-processing programs.
- By the time the experiment started, the program had been in use for more than one year. This seemed a good opportunity to test the fault-detection ability of M-mp testing, as the likelihood of still finding defects was expected to be low.

Essentially, the application schedules recurrent maintenance tasks for machines (e.g., car, aircraft) and/or their parts (e.g., engine, fuel pump) at variable intervals based on rules that depend mainly on:

- the forecast utilisation rates (e.g., kilometres per month, flying hours per day) and
- the anticipated utilisation conditions (e.g., altitude, humidity level).

The application domain of the primary program is depicted in Figure 3-5. The upper half of the diagram, labelled *Machine/Part Utilisation*, reflects at high-level that the utilisation rates and conditions are usually variable. The bottom part of the figure contains a very simple schedule. Mainly because of complex machine structures, there can be strong dependencies between tasks. In this particular implementation, the dependencies are expressed as "suppression" and "trigger" relationships. The program schedules each "non-triggered" task individually and then it applies the "suppression" and "trigger" rules. For example, inFigure 3-5, task *A* "triggers" occurrences of task *B* and "suppresses" some occurrences of task *C*. Past task occurrences, unless suppressed, are depicted by greyed rectangles.

To facilitate the correct interpretation of the case study results, it is important to note that, mainly because of time constraints, only the core functionality (i.e. scheduling non-triggered tasks individually) was implemented by the model program. Having presented the strategy of the experiment and the application domain of the primary program the next logical step is to describe how the experiment was carried out.

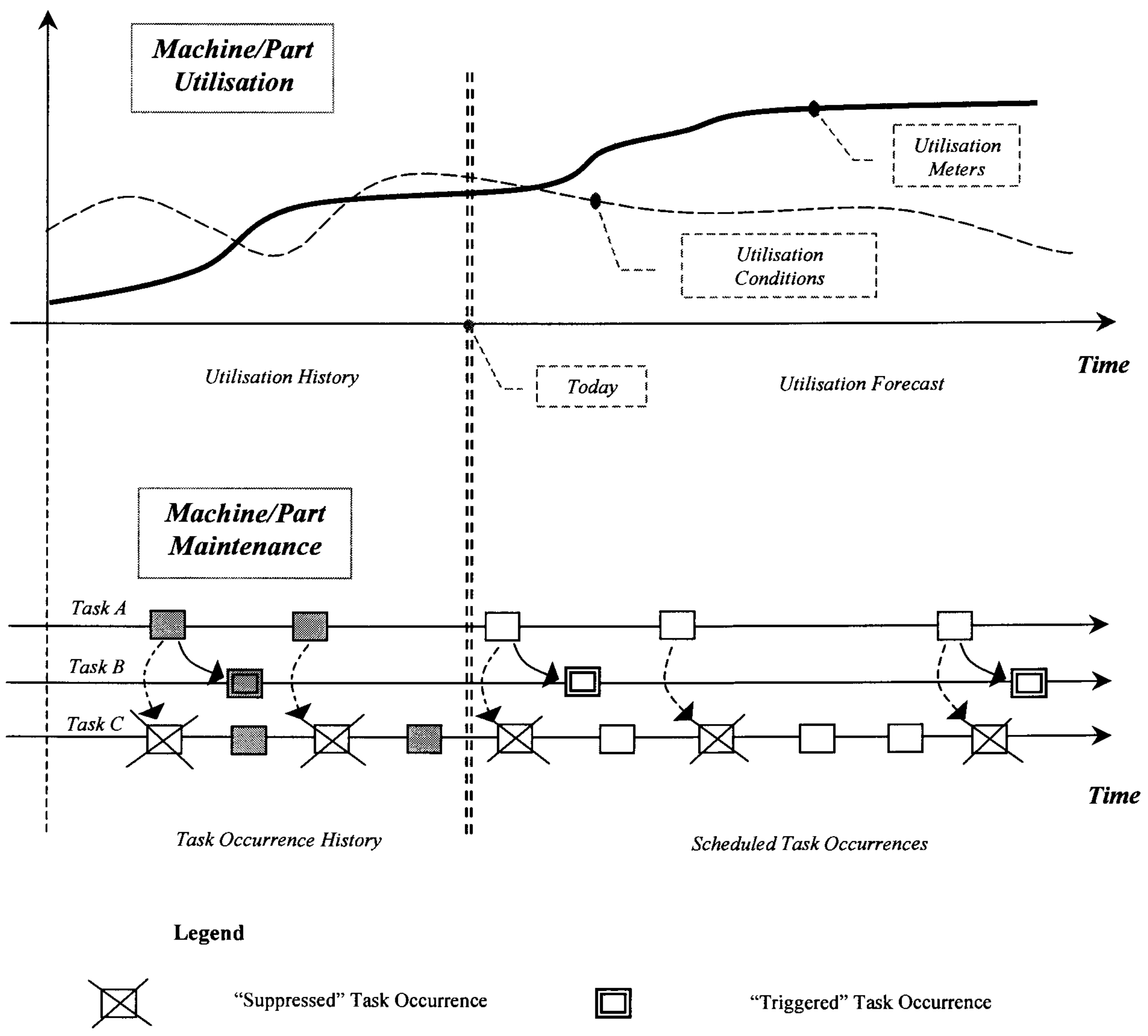


*Figure 3-5: Scheduling Application Domain*

## 3.3.2 Experiment Execution Overview

Essentially, the experiment entailed developing a model program from the (informal) requirements specification, subjecting the primary and model programs to the same input test data, and analysing the disagreements among the programs. The information flow model of the experiment's process and its relationship with the generic view of testing that has been presented in *Chapter 2* are depicted in Figure 3-6.

*Test Design* comprised the creation of test data (*Test Data Selection*) and the development of the model program (*Model Program Design* and *Model Program Implementation*). The *Design Specification* is depicted in bold in Figure 3-6 to emphasise its vital role in the experiment. Besides setting a common foundation for all subsequent activities, the design specification served as the basis for automatically generating:

- most of the model program code (around 80%),

- the configuration file template for the test data generator[2], and
- the database set-up scripts.

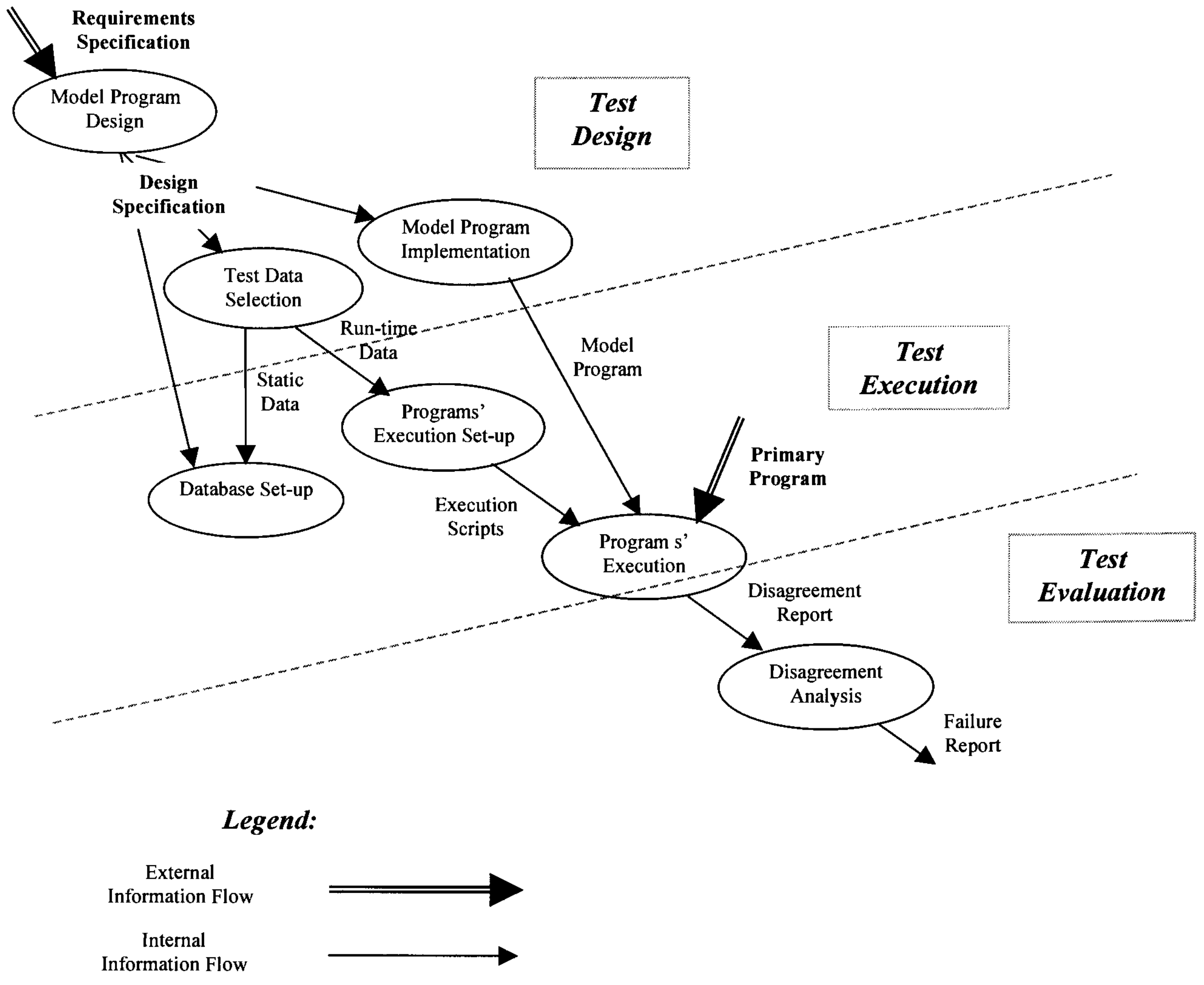


*Figure 3-6: The Experiment's Process*

*Test Execution* included setting up the test environment (*Database Set-up* and *Programs' Execution Set-up*) and executing the primary program (*Programs' Execution*). *Programs' Execution* crosses the boundary between *Test Execution* and *Test Evaluation* to indicate that running the model program is a test evaluation activity. That is because, by either agreeing or disagreeing with the outputs of the primary program, the model program performs the initial correctness checking. For all practical purposes, the primary program can be provisionally regarded as producing correct results for test inputs that agree in output.

*Test Evaluation*, in addition to model program execution (*Programs' Execution*), comprised arbitrating the disagreements between the programs (*Disagreement Analysis*).

"Large-scale, cost-effective test automation" is probably the best way to characterise the experiment. Except for *Model Program Design*, all other activities have been efficiently automated to various degrees. Each activity will be described in more detail in the remainder of this section. Because *Test Data Selection,*

[2] A test data generator was also developed in this experiment.

*Database Set-up, Programs' Execution Set-up, Programs' Execution,* and *Disagreement Analysis* are largely interdependent, they are discussed together under *Back-to-back Testing.*

### 3.3.3 Model Program Design

Conducted in a top-down fashion, the design combined in a practical manner various methods such as relational modelling, functional decomposition, object orientation, and model-based formal specification. It comprised the following main activities:

- partitioning the application into modules,
- producing a relational model and defining the interface (or public) methods for each module, and
- creating object models as graphically depicted in Figure 3-7 (i.e., decomposing the interface methods of a module and assigning the lower-level methods to appropriate relational model entities[3]).

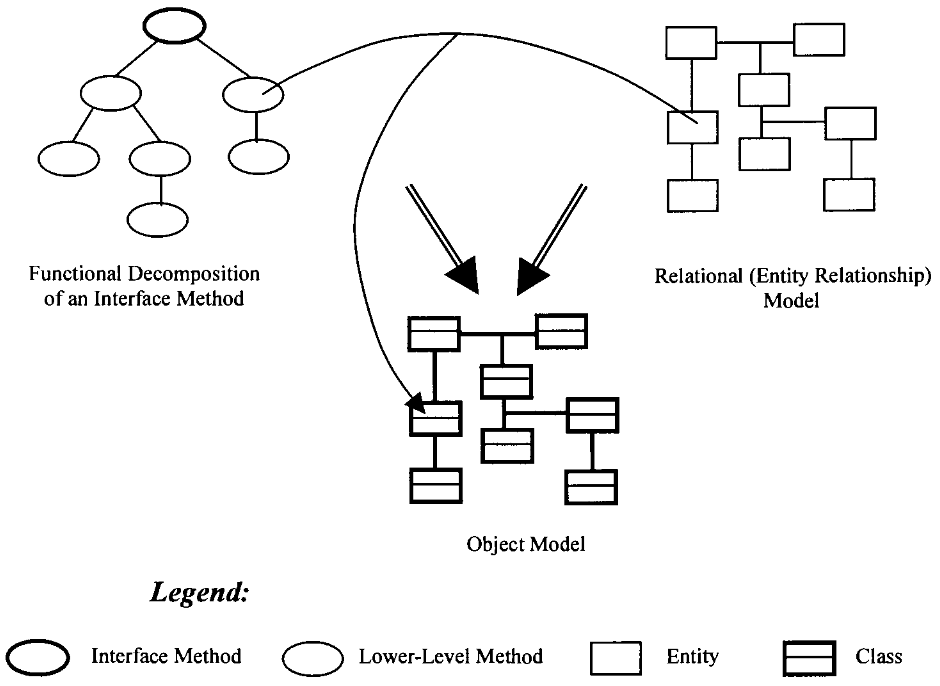

Functional Decomposition of an Interface Method
Relational (Entity Relationship) Model
Object Model
Legend:
Interface Method
Lower-Level Method
Entity
Class

*Figure 3-7: Deriving an Object Model for a Module*

To facilitate efficient test environment set-up and simple output comparison, the aim from the beginning was to use the same I/O data model for both the primary program and the model program[4]. To accommodate that, the model program was split into two main modules: the scheduling engine and the I/O bridge. The former does the scheduling based on a generic computation model, while the latter, as simply depicted in Figure 3-8, is responsible for the data transfer between the scheduling engine and the database. The I/O bridge was necessary to cater for differences between the generic computation model and the I/O data model. For convenience, the test data model was chosen so that most of the test cases that were previously designed for testing the primary program could be reused. The intention was to validate the model program and the test cases in a 3-version system in the same manner as discussed in the *Pilot Study.*

[3] An entity (structure) and its methods (behaviour) form an object that belongs to an object class. An entity in the relational model is thus transformed into an object in the object model.

[4] The programs could then share the same input data tables, while a simple SQL (Structured Query Language) script could be written to compare the results.

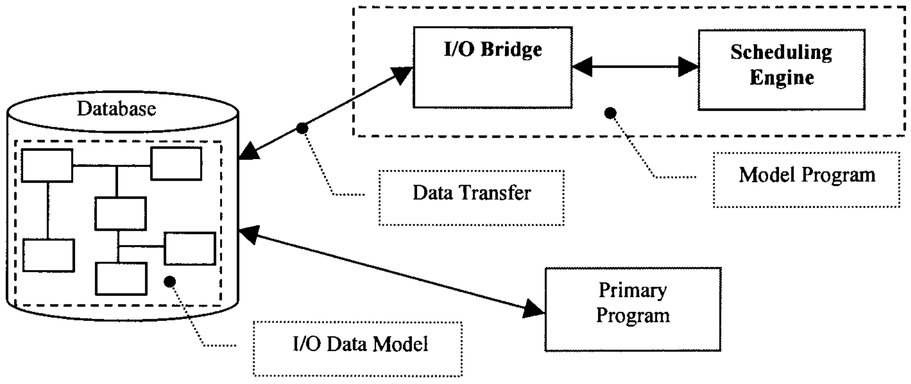


*Figure 3-8: Model Program Modules*

Test environment set-up efficiency, simple output comparison, and test case reuse were not the only reasons for splitting the program into an I/O bridge and a scheduling engine. In general, in the case of data-processing applications, such an approach can be very beneficial for both program development and maintenance. Firstly, the algorithm, which is the complex part of the application, can be designed in isolation from data accessibility and storage concerns. Secondly, I/O data model changes are likely to affect primarily the I/O bridge, which is much less complex than the computation engine.

As a side activity, the design also entailed defining a simple specification language that has been derived from Shlaer-Mellor Action Language[5] (SMALL, WWW Resources (2)). Appendix A contains its definition along with an abbreviated specification example. The definition is imprecise since the language was created specifically for this experiment and it was refined only to be sufficiently formal and expressive for specifying the scheduling algorithm in a simple, concise and unambiguous manner. The language resembles model-based formal specification languages[6]. Although an attractive alternative, using an acknowledged formal specification language (such as Z or VDM) was not pursued mainly because the author had no experience in applying formal methods. It was felt that the learning curve could be too steep for this experiment. SMALL, on the other hand, was well known by the author and although specific to Shlaer-Mellor OOA/RD, it was easily adaptable to specify the scheduling algorithm. It is also important to mention in this context that the specification language could be kept simple as the algorithm specification tried to capture the steps that are normally followed by a person to schedule tasks by hand. The aim was to produce a "natural" algorithm specification, thus ensuring high readability and, consequently, high maintainability of the specification and its implementation.

The high quality of the specification was expected to lead to cost-effective implementation. Yet, the level of automation that was achieved in the end was not anticipated. The implementation of the model program, including the code generation approach, is described next.

## 3.3.4 Model Program Implementation

The first step in model program implementation was to choose a development environment (i.e., a programming language and a development tool). Since a model program is not constrained to specific

[5] SMALL is a specification language that is designed to specify the actions associated with Moore states in Shlaer-Mellor OOA/RD (Object-Oriented Analysis/Recursive Design). Shlaer-Mellor OOA/RD is the main design method that is used for 3GL development in the organisation.

[6] The language supports mathematical notions such as sets, relations, functions, and sequences that are characteristic to model-based formal specification languages (e.g. Z, VDM). More information regarding formal methods can be found at WWW Resources (1).

programming languages, operating systems, or databases, M-mp testing offers the opportunity of selecting practically any programming environment. Choosing one, however, is not a trivial task as there might be too many attractive options. It is worth mentioning in this context that 26 different languages were used in 38 rapid prototyping studies that are analysed by Gordon and Bieman (1995)!

### *3.3.4.1 Development Environment Selection*

In this experiment, the following three environments were the main candidates:

- UnifAce – the 4GL development environment of EPMS® (the logistics management system that integrates the 3GL applications that are targeted by this case study),
- C++ / Microsoft Developer Studio 97, and
- C / Microsoft Developer Studio 97.

Other promising development environments such as LISP, Prolog, C++ Builder, Visual J++, Oracle PL/SQL, which were also considered in the beginning, had to be excluded primarily because of the envisaged steep learning curves. Given the time and resource constraints, the experiment could not afford the extra cost.

Although the author was less familiar with UnifAce than with C/C++, the implementation started in UnifAce for the following reasons:

- As a 4GL is more expressive than a 3GL, the implementation in UnifAce was likely to result in significantly less lines of code than in C/C++. In addition, in UnifAce the database I/O is practically transparent to the developer, whereas in C/C++ extra (embedded SQL) code is required for database access. Because at that stage the implementation was envisaged to be done manually, less lines of code would have meant lower implementation cost.
- The UnifAce 4GL and the specification language have similar levels of abstraction and semantics. That would have ensured an effortless implementation and the resulting code would have been easily traceable to the design specification.
- The decision to implement the operations-management applications in C and not UnifAce on the grounds of superior performance achievable in 3GL was sometimes disputed. Implementing the model program in UnifAce would have allowed accurate 4GL vs. 3GL performance comparison.

The learning curve, however, was much steeper than expected and the development in UnifAce was abandoned. The implementation was thus re-started in C++. Although C would have been a slightly more cost-effective choice, the personal gain of the author was the deciding factor. The author wanted to consolidate his C++ knowledge. In general, in M-mp testing, to ensure a high motivation level for model program development, the personal gain of the developer should be always given a high weight.

As mentioned above, one of the advantages of initially choosing UnifAce was that its 4GL and the specification language have similar levels of abstraction and semantics. Consequently, writing the algorithm would have been effortless and the code easily traceable to the specification. The implementation in C++ aimed at achieving the same benefits through appropriate layering.

### *3.3.4.2 The Implementation Strategy*

The implementation strategy will be discussed around the scheduling engine, which is the core of the model program. The implementation of the I/O bridge followed the same strategy.

As illustrated in Figure 3-9, the aim was to create a supporting layer so that the algorithm specification could be "mirrored" (i.e., translated at a similar level of abstraction) into code, thus producing an equivalent (executable) specification of the algorithm.

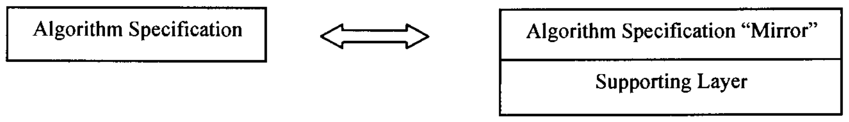


*Figure 3-9: Implementation Strategy*

Below is a simple example that shows how the algorithm specification, its "mirror", and the supporting layer, which are depicted in Figure 3-9, relate to each other. The statement from the algorithm specification means "select the latest *Task_Occurrence* instance whose *StartDate* is earlier than *i_StartDate*", where *Task_Occurrence* is an entity/class and *StartDate* is one of its attributes/data members. The subset selection statement from the specification maps to an equivalent C++ statement in the specification "mirror". The supporting layer contains the lower-level C++ code (i.e., the implementation of the *Select* method).

```
Algorithm Specification
task_occ = Task_Occurrence(last, StartDate < i_StartDate, /StartDate)

Algorithm Specification "Mirror"
task_occ = Task_Occurrence.Select(LAST, "/StartDate", "StartDate < %ld", i_StartDate);

Supporting Layer
Task_Occurrence*
Task_Occurrence::Select(int i_WhichInstance, char* i_OrderBy, char* i_Where, ...)
{// Method implementation}
```

In the above example, the correspondence between the specification and its "mirror" is very exact and that represented the ideal that guided the implementation. The actual kind of mapping that was obtained in the end is shown in the example below:

```
Algorithm Specification
task_occ = Task_Occurrence(last, StartDate < i_StartDate, /StartDate)

Algorithm Specification "Mirror"
selection_id = Task_Occurrence.SelectOnStartDate("<", i_StartDate);
task_occ = Task_Occurrence.Last(selection_id);
Task_Occurrence.ReleaseSelection(selection_id);

Supporting Layer
long Task_Occurrence::SelectOnStartDate(char* i_relation, long i_StartDate)
{// Method implementation}
Task_Occurrence* Task_Occurrence::Last(long i_selection_id)
{// Method implementation}
void Task_Occurrence::ReleaseSelection(long i_selection_id)
{// Method implementation}
```

It would have been technically possible to achieve a better correspondence between the specification and its "mirror". It was felt, however, that additional refinements to the supporting layer, although likely to reduce the cost of writing the "mirror", would have been too expensive for this experiment.

Even if not ideal, the approach still had significant benefits. Firstly, the level of correspondence between the specification and the "mirror" was sufficient to allow an easy translation of the algorithm into code. Secondly, the supporting layer practically defined a domain-dependent executable specification language[7]. Two additional programs, which were necessary to overcome some difficulties encountered in output comparison and test data generation[8], could be written effortlessly. Thirdly, because the supporting layer was very generic, the approach led to large-scale code generation.

### *3.3.4.3 The Code Generation Approach*

The possibility of large-scale code generation was not envisaged from the beginning. As the implementation advanced, however, automation became increasingly appealing for reasons that will be discussed next.

While writing the first module by hand, it was noticed that all supporting layer classes provided essentially the same capabilities such as creation, deletion, set ordering, and subset selection. Moreover, the implementation differences between methods with common semantics were largely determined by distinct class structures. For instance, a basic creation method of an arbitrary class with N data members looks in principle as shown below:

```
class <Class>
{
   <type1> <name1>;
   <type2> <name2>;
   ...
   <typeN> <nameN>;
};

<Class>::Create(<type1> i_<name1>, <type2> i_<name2>, ..., i_<nameN>)
{
   <name1> = i_<name1>;
   <name2> = i_<name2>;
   ...
   <name3> = i_<nameN>;
}
```

If the same naming conventions are used across classes, it is obvious that the creation methods will be different from class to class only because of different class structures. This kind of commonality between class methods was exploited from the beginning but only to derive a new class from a previously implemented one by using text search and replace. It was soon realised that it would be possible and much

[7] The approach is actually a form of domain analysis as discussed in Mili et al (1995). In this regard, the supporting layer is the realisation of the domain model, while the algorithm specification "mirror" embodies the application (i.e., scheduling).

[8] The two additional programs will be discussed later in this section in the context of model program calibration and test data generation.

more beneficial to generalise existing class implementations into a code generator. All the supporting layer classes could then be created automatically.

Automation is usually considered a long-term investment, as writing or setting up automation tools can be very expensive. In this experiment, however, code generation was very likely to pay off even within the short time scale for the following main reasons:

- As also mentioned before, model program performance, memory usage, and size were not a primary concern. The code generator could be kept thus very simple.
- The supporting layer, which was the target of code generation, was by far the largest portion of the implementation (in the end it accounted for around 80% of the total program size).

The code generator was implemented using Microsoft Excel spreadsheets and Visual Basic (VB) macros. Simply, the code generator works as follows: a VB macro reads the structure and behavioural parameters of a class from an MS Excel spreadsheet and generates the appropriate C++ and embedded SQL code. Appendix B shows a simple example of an input to the generator and the resulting code.

The code generation approach is actually a particular case of a reuse technique that is known as "application generation". A comprehensive overview of reuse techniques can be found in Mili at al (1995). As opposed to most application generators that are designed to produce a family of similar programs, the code generator was written to create a family of classes that have similar (structure-based) methods. The scope, commonality, and variability (SCV) analysis that is described in Coplien et al (1998) is also relevant for the way in which code generation was approached in this experiment. From an SCV analysis perspective, the code generator was based on parametric polymorphism[9].

Model program development ended with a calibration phase, which will be described next.

### *3.3.4.4 Calibrating the Model Program*

While writing the model program, it became evident that floating-point accuracy could cause slight output differences between the model program and the primary program. Eliminating those differences was important because in scheduling a task occurrence is created with respect to the previous one. Consequently, the first schedule difference is propagated and likely to be amplified for all subsequent scheduled task occurrences. Besides the first one, all other differences are practically meaningless from a disagreement analysis viewpoint, thus reducing the failure detection ability of back-to-back tests[10].

Ideally, floating-point accuracy should be clarified in co-operation with the development team of the primary program. That, however, was not feasible mainly because the experiment was conducted after-hours. The alternative was to provide the model program with a set of internal parameters to allow fine-tuning. The calibration tried to adjust those parameters until the model program ideally performed floating-point computations using the same precision as the primary program. Although it contributed to reduce substantially the number of disagreements, calibration started to become increasingly expensive. The internal parameters did not seem to be sufficient to eliminate all small differences that were likely to be caused by floating-point accuracy.

To be able to perform effective output comparison even in the presence of tolerable small differences, a new model program was written on top of the first one. The new program will be called for convenience the Evaluator, while the first program will be called the Scheduler. Simply, after scheduling a task occurrence, the Evaluator compares it to the corresponding occurrence that was scheduled by the primary program. If they are different, the Evaluator logs the disagreement and then it resets its state to be in accord

[9] C++ method and class templates also fall under parametric polymorphism.
[10] It is difficult to determine whether an output difference is encountered only because of a previous one or because of other failures.

with the output of the primary program. In Figure 3-10 for instance, for the first and the third occurrences, the Evaluator resets first its state and then it schedules the corresponding next occurrences as indicated by the dashed arrows.

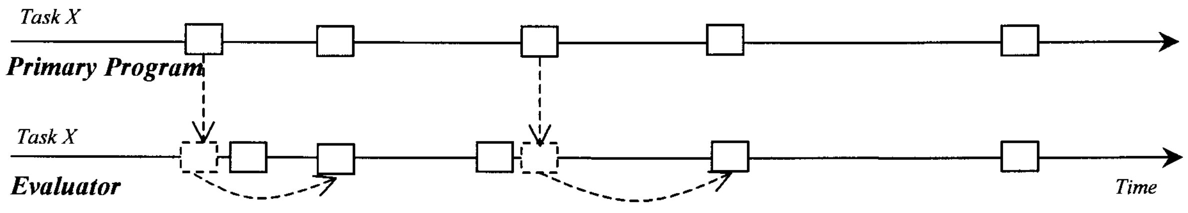


*Figure 3-10: The Evaluator's Technique*

Besides allowing effective output comparison in the presence of tolerable small differences, the Evaluator also increased the failure detection ability of tests in general. Because differences are not propagated, all disagreements encountered during a single test run can be analysed separately. The Evaluator provided the necessary means to perform effective back-to-back testing.

## 3.3.5 Back-to-back Testing

Essentially, back-to-back testing entailed generating test data, setting up the test execution environment from the generated data, running the programs, and analysing the output disagreements. Except for the disagreement analysis, all other activities were practically fully automated. Before going into more detail about the actual back-to-back testing, the means of automation will be presented first.

### *3.3.5.1 Test Data Generation*

As highlighted throughout this dissertation, one major advantage of M-mp testing is the possibility of carrying out a large number of tests in a cost-effective manner. Emphasising and exploiting this advantage was a goal from the start and automating the creation of test data was one of the first challenges of this experiment.

Although the initial intention was to use a commercially available test data generator or one produced by research, only a few data generators could be evaluated given the time constraints and even that was done on a superficial level. However, none of the evaluated test data generators seemed to accommodate both referential integrity and arbitrary mathematical attribute dependencies across a relational model. Support for statistical testing[11] was another desirable property that was considered when evaluating test data generators. In the end, a test data generator, which is described in *Appendix C*, was developed especially for this experiment.

Although the test data generator is quite powerful, it could be used effectively only to generate data for the particular case of scheduling "from scratch", that is, the case when no previous task schedule exists for a particular machine. In Figure 3-11 for instance, test data could be generated for the first use of the scheduler, but not for subsequent uses. Generating a realistic updated schedule was too difficult as it would have implied meshing the scheduler logic into the input to the test data generator. This problem is likely to be encountered with any generic data generator. The solution was to develop a program that used the same domain knowledge as the scheduler, but with the purpose of acting as the user. In other words, the program, which will be called the Modifier, was designed to alter the output of the primary program to be used as input for new tests.

[11] Reliability-engineered testing, which is a particular case of statistical testing, has been comprehensively addressed in Chapter 2.

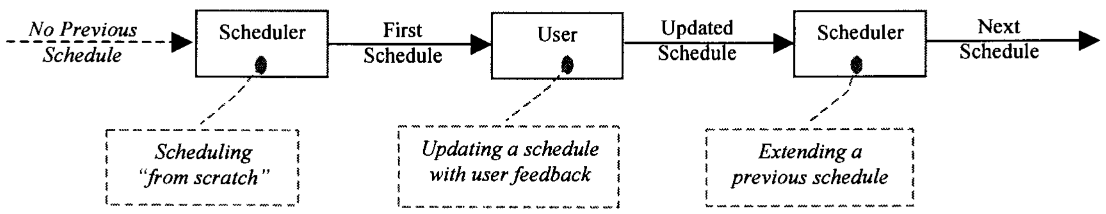

No Previous Schedule
Scheduler
First Schedule
User
Updated Schedule
Scheduler
Next Schedule
Scheduling "from scratch"
Updating a schedule with user feedback
Extending a previous schedule

*Figure 3-11: Typical Use Scenario for a Scheduler*

Together, the test data generator and the Modifier provided a cost-effective means of generating practically any kind of test data: simple or complex, positive or negative (i.e., valid or invalid), uniform or distributed according to other statistical profiles. This flexibility was in particular useful for adjusting the complexity and the specificity of the test data to increase the efficiency of disagreement analysis. This aspect will be discussed in more detail later in this section.

Once a test data set was generated, the test execution environment had to be set up from the generated data. That was one of the most highly automated activities in this experiment and it will be briefly described next.

### *3.3.5.2 Test Execution Environment Set-up*

As illustrated in Figure 3-12, loading the static data into the database was done by means of *SQL Scripts*, which were in turn generated from the *I/O Bridge Class Structures* that were captured in Microsoft Excel spreadsheets. The same spreadsheets were used to generate the *C++ and embedded SQL code* for the I/O module of the model program (i.e., the I/O bridge) and the *Input Layout* for the *Test Data Generator*[12]. That ensured exceptional naming and structural consistency across development and testing artefacts and one important consequence was that the format of the *Static Data* could be automatically synchronised with that set in the *SQL Scripts*. The *Static Data* could thus be loaded into the database without necessitating any adjustment. Generating the *Execution Scripts* was also very direct. The *Runtime Data* needed only to be inserted into a Microsoft Excel spreadsheet.

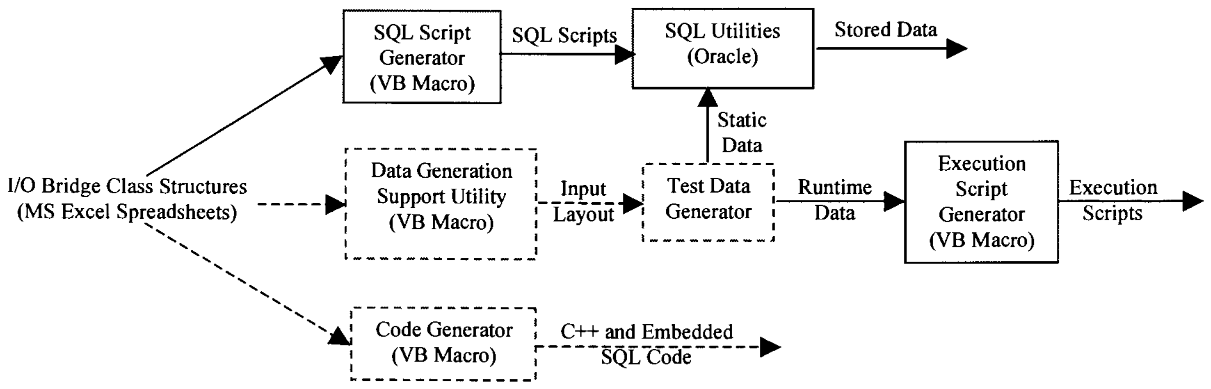

SQL Script Generator (VB Macro)
SQL Scripts
SQL Utilities (Oracle)
Stored Data
Static Data
I/O Bridge Class Structures (MS Excel Spreadsheets)
Data Generation Support Utility (VB Macro)
Input Layout
Test Data Generator
Runtime Data
Execution Script Generator (VB Macro)
Execution Scripts
Code Generator (VB Macro)
C++ and Embedded SQL Code

*Figure 3-12: Test Execution Environment Set-up*

[12] The dashed lines in Figure 3-12 indicate that generating code and test data are not part of setting up the test environment. They have been included, however, to emphasise the common root of otherwise distinct activities and the extent of automation.

Test data generation and the automation of the execution environment set-up provided the means for running the programs back-to-back in a cost-effective manner on any feasible test data set. The main aspects of the way testing has been conducted in this experiment are presented next.

### *3.3.5.3 Conducting the Tests*

Back-to-back testing actually started during the model program calibration and it was based on the test data that was created manually for formally testing the primary program. Reusing the manually created test data, however, was not as beneficial as expected mainly because many test cases involved "positive" exception handling[13]. The model program, which was designed to process only standard data, rejected the inputs for those test cases. Although it would have been possible to complete the test data to make it suitable for the model program, that was not pursued mainly because it was much more cost-effective to generate data. Moreover, there were no disagreements between the primary program and the manually designed test cases. Therefore, the 3-version system made up by the model program, the primary program, and the test cases degenerated into a 2-version system. In this regard, using the manually created test data would not have been different from using any data set.

Besides model program development (and maintenance), intrinsic to M-mp testing is the disagreement analysis activity. Assessing and investigating ways of improving its cost-effectiveness was the focus of back-to-back testing in this experiment. Simply, the analysis entails determining which programs failed on a test input that caused a disagreement[14]. The most direct way of disagreement analysis is to work out the correct results by hand and compare them with the results produced by each program. Because the outputs need to be manually calculated only in cases of disagreement, this is already an improvement over manual test design.

In this experiment, the cost of disagreement analysis could be reduced even further through data mutation and correlation. Data mutation entailed changing test inputs slightly with the purpose of drawing general conclusions about the corresponding disagreement cases. For instance, it was useful to know that turning off a flag brings the programs back into agreement. Similarly, it could be observed that changing the relationship between two input parameters influenced whether the programs agreed or disagreed. General conclusions about disagreements could also be drawn by correlating the test inputs and the results of different tests. Generalising from data allowed the formulation of high-level behavioural hypotheses that could be first confirmed and then validated against the requirements specification. Individual disagreements could then be analysed to determine whether they are caused by known failure types[15]. Alternatively, if a disagreement disappeared after removing a defect, it was likely that the disagreement was caused by that defect (or by the corresponding failure type). In these cases, there was no need for manual computation.

To understand how the disagreement analysis was performed in this experiment, a sample of the disagreement analysis sheet is shown in Table 3-1 (the full analysis sheet is included in Appendix D). A row corresponds to a distinct disagreement. Each column is briefly explained below:

- **Test Data Set** – The unique identifier of a test data set. *0* is the manually created test data set (before the experiment started), while *1, 1A, 1B, 1B.1, 2,* and *2A* are the data sets that were generated in the experiment.
- **PP** – The versions of the primary program. Only one version was used because the detected faults were not corrected within the scope of this experiment.

[13] "Positive" exception handling refers to cases when a program supplies default values to complete its input or it rejects only a part of that input. As opposed to "negative" exception handling, the program still produces results.
[14] It is important to note that if a disagreement is caused by a specification defect (e.g., conflicting requirements, omission), none of the programs failed.
[15] A failure type means a class of failures that are caused by the same, not necessarily localised, defect.

- **MP** – The versions of the model program. As the model program was corrected twice, there are three versions: *A, B,* and *C.* Each of the 3 columns is assigned a distinct version. An empty cell indicates that the disagreement was not encountered for the corresponding version. For instance, the disagreement for *Test Data Set 0* happened for *MP A* and *B*, but not for *C*. An "x" indicates that the version is not relevant for that particular disagreement. For example, the disagreement for *Test Data Set 2A* is relevant only for *MP C.* (In this particular case, *MP A* and *B* are not relevant because they were not tested on *Test Data Set 2A.*)
- **Env.** – The versions of the test execution environment. The test environment was also versioned because a primary program failure (*PP-Fail-2*) was caused by an environment defect. The same conventions that have been described for the model program versions apply.
- **Test Id** – The unique identifier of a test within a particular data set.
- **Task Occ.** – The task occurrence for which the disagreement occurred.
- **Occ. Attribute** – The task occurrence attribute that caused the disagreement.
- **PP Output** – The value of *Occ. Attribute* computed by the primary program. An empty cell indicates that no output was produced or the output is irrelevant in that case.
- **MP Output** – The value of *Occ. Attribute* computed by the model program. The same conventions as for *PP Output* apply.
- **Correct Output** – The correct value of *Occ. Attribute*. An empty cell indicates that the correct output was not determined, either because it was not necessary or because there was a specification defect.
- **PP Failure** – Primary program failure type. An empty cell indicates that it is not known whether the primary program failed the corresponding test, while an "N" marks that the output of the primary program was correct for *Occ. Attribute.*
- **MP Failure** – Model program failure type. The same conventions as for *PP Failure* apply.
- **Spec. Defect** – Specification defect identifier. With one exception, if the specification was faulty, whether any program failed could not actually be determined. The exception was *MP-Fail-1* and *Spec-2 (e.g., Test Data Set 2A).* The disagreement was too big to be caused by the specification defect alone.
- **Remarks** – General characteristics of the disagreements. This column was the main means for data correlation. The best examples in Table 3-1 are *Test Data Set 2* and *2A*.

| Test Data Set | PP | MP | | | Env. | | Test Id | Task Occ. (machine \| task \| occ. number) | Occ. Attribute | PP Output | MP Output | Correct Output | PP Failure | MP Failure | Spec. Defect | Remarks |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | A | A | B | | A | x | 21 | (21\|26\|0) | Start Date | 01/02/1996 | 21/03/1996 | 01/02/1996 | N | MP-Fail-2 | | |
| 1 | A | A | B | C | A | B | 1000 | (1000\|1000\|0) | Start Date | 15/03/2000 | 17/01/2000 | | | | Spec-2 | |
| 1 | A | A | B | C | A | | 1000 | (1000\|1001) | | none | | | PP-Fail-2 | | | Factored meters |
| 1 | A | A | B | C | A | B | 1100 | (1100\|1101\|0) | Start Date | 12/01/2000 | 07/10/1999 | 07/10/1999 | PP-Fail-3 | N | | |
| 1 | A | A | B | C | A | B | 1200 | (1200\|1201\|18) | Start Date | 30/05/2007 | 06/06/2007 | | | | Spec-1 | |
| 1A | A | x | x | C | x | B | 1000 | | | none | | | PP-Fail-4 | | | PP does not terminate |

| Test Data Set | PP | MP | | | Env. | | Test Id | Task Occ. (machine \| task \| occ. number) | Occ. Attribute | PP Output | MP Output | Correct Output | PP Failure | MP Failure | Spec. Defect | Remarks |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1A | A | A | B | C | A | B | 1200 | (1200\|1202\|0) | Start Date | 27/03/2000 | 29/11/1999 | 29/11/1999 | PP-Fail-3 | N | | |
| 1A | A | A | | | A | x | 1300 | (1300\|1303\|0) | Start Date | 24/12/1999 | 04/08/1999 | | | MP-Fail-1 | Spec-2 | |
| 1A | A | | B | C | A | B | 1300 | (1300\|1303\|0) | Start Date | 24/12/1999 | 03/09/1999 | | | | Spec-2 | |
| 1A | A | A | B | C | A | B | 1700 | (1700\|1704\|17) | Start Date | 27/07/2006 | 20/07/2006 | | | | Spec-1 | |
| 1B | A | x | B | C | A | B | 1000 | (1000\|1003\|0) | Start Date | 18/11/1999 | 30/08/1999 | | | | Spec-2 | |
| 1B | A | x | B | C | A | B | 1100 | (1100\|1103\|0) | Start Date | 30/03/2000 | 08/11/1999 | 08/11/1999 | PP-Fail-3 | N | | |
| 1B.1 | A | x | B | C | A | B | 1100 | (1100\|1103\|0) | Start Date | 30/03/2000 | 08/11/1999 | 08/11/1999 | PP-Fail-3 | N | | |
| 1B.1 | A | x | B | C | A | B | 1300 | (1300\|1303\|0) | Start Date | 21/02/2000 | 12/10/1999 | | | | Spec-2 | |
| 2 | A | x | x | C | A | B | 2000 | (2000\|2001\|0) | Start Date | 27/09/1999 | 07/06/1999 | | | | Spec-2 | |
| 2 | A | x | x | C | A | B | 2000 | (2000\|2002\|0) | Start Date | 25/01/2000 | 07/06/1999 | 07/06/1999 | PP-Fail-3 | N | | IntvType = 6 |
| 2 | A | x | x | C | A | B | 2000 | (2000\|2003\|3) | Start Date | 20/03/2000 | 13/03/2000 | | | | Spec-1 | |
| 2 | A | x | x | C | A | B | 2000 | (2000\|2004\|0) | Start Date | 19/11/1999 | 20/07/1999 | 20/07/1999 | PP-Fail-3 | N | | ForceInd = ON, First Date = Non-work |
| 2A | A | x | x | C | x | B | 2600 | (2600\|2600\|0) | Start Date | 21/10/1999 | 31/05/1999 | 31/05/1999 | PP-Fail-3 | N | | IntvType = 6 |

*Table 3-1: Disagreement Analysis Sheet Sample*

The automated tests started with a large set of random data. That gave a rough estimate of the level of disagreement between programs, but it was not too useful for disagreement analysis because the test inputs were too complex and diverse. The experiment, therefore, proceeded incrementally with small and simple data sets (*1, 1A, 1B, 1B.1, 2,* and *2A* in Table 3-1), taking full advantage of the data generation capability.

Even if performed on a small scale, the tests led to the detection of two specification defects, two primary program faults, two model program faults, and one test environment defect. The full results of the experiment are presented in the last section of this chapter.

## 3.4 Results

### 3.4.1 Raw Experimental Results

| | Total Lines Of Code | % | Lines of Code without Comments | % | Executable Statements | % |
|---|---|---|---|---|---|---|
| **Generated Code** | 72730 | **86.1** | 43770 | **84.9** | 24072 | **86.1** |
| **Manually Written Code** | 11790 | 13.9 | 7783 | 15.1 | 3876 | 13.9 |
| **Total** | 84520 | 100 | 51553 | 100 | 27948 | 100 |

*Table 3-2: Code Generation Effectiveness*

| Activity | Cost (Person-Hours) |
|---|---|
| **Test Design** | **664** |
| **Model Program Development** | **458** |
| Specification Language Definition | 30 |
| Model Program Design | 122 |
| Code Generation[16] | 137 |
| Scheduler Implementation[17] | 90 |
| Scheduler Calibration | 51 |
| Evaluator Implementation | 28 |
| **Test Data Selection** | **206** |
| Test Data Generator Implementation | 150 |
| Modifier Implementation | 15 |
| Test Data Generation | 41 |
| **Test Execution and Evaluation** | **73** |
| Test Environment Set-up | 42 |
| Programs' Execution and Disagreement Analysis | 31 |
| **Total** | **737** |

*Table 3-3: Experiment's Cost Breakdown*

| Artefact | Number of Defects |
|---|---|
| Requirements Specification | 2 |
| Primary Program | 2 |
| Model Program | 2 |
| Test Environment | 1 |

*Table 3-4: Detected Defects*

### 3.4.2 Adjusted Test Design Effort

To keep the cost within reasonable limits, the experiment was carried out gradually until it was felt that sufficient evidence had been gathered to draw a valid conclusion. It is important to note, therefore, that the above results pertain only to testing the core functionality of the scheduling application using standard data (i.e., "perfect data"). Although the interpretation of the results is done in full in*Chapter 4*, it is convenient to give here the cost estimates of an M-mp test design covering the full scheduling functionality (Table3-5). The adjustment factor that is used for most activities (i.e., 1.5) is based on the realistic assumption that the core functionality represents at least two thirds of the function/feature point size of the scheduling application. Because the specification language, the code generator, and the test data generator are reusable artefacts, their costs are practically constant. The cost of *Code Generation* is adjusted by a factor of 1.25

[16] This includes the cost of developing the code generator. It is not known exactly how much effort was spent to write the code generator, but it is estimated in the region of 100 person-hours.

[17] Only the manually written code is taken into account.

because it is not known exactly h[illegible] write the code generator (it is estimated, however, in the region of 100 person-hours).

| Activity | Actual Cost (Person-Hours) | Adjustment Factor | Adjusted Cost (Person-Hours) |
|---|---|---|---|
| **Model Program Development** | **458** | **1.39** | **638** |
| Specification Language Definition | 30 | 1 | 30 |
| Model Program Design | 122 | 1.5 | 183 |
| Code Generation | 137 | 1.25 | 171 |
| Scheduler Implementation | 90 | 1.5 | 135 |
| Scheduler Calibration | 51 | 1.5 | 77 |
| Evaluator Implementation | 28 | 1.5 | 42 |
| **Test Data Selection** | **206** | **1.14** | **234** |
| Test Data Generator Implementation | 150 | 1 | 150 |
| Modifier Implementation | 15 | 1.5 | 23 |
| Test Data Generation | 41 | 1.5 | 61 |
| **Total** | **664** | **1.31** | **872** |

*Table 3-5: Adjusted Test Design Costs*

The adjusted cost of M-mp test design from Table 3-5 represents around 45% of the manual test design cost (i.e., the effort that was spent on manually designing the test cases for the first release of the primary program). If the reusable artefacts had been available at the beginning of the experiment, the ratio would have decreased to 30%. It is also interesting to note that according to Table 3-5 the model program development would have been 6 – 7 times cheaper than the development of the primary program. The full interpretation of the results is done in *Chapter 4*.

# 4. CONCLUSIONS

To increase the clarity of the discussions in this chapter, it is important first to make the following remarks:

- Cost, unless otherwise stated, represents the number of person-hours required to carry out a task. In this respect, the terms "cost" and "effort" are sometimes used interchangeably.
- To allow meaningful and easy comparisons, costs are in general expressed as percentages of the manual test design cost (i.e., the effort that was spent on designing manually the test cases for the first release of the primary program).
- Very similar to the input domain partitioning described in Van Vliet ((1996), 355-356), in this dissertation, the input domain of a program is divided into a standard domain and an exception domain. In turn, the exception domain is split into an incomplete-data domain and an invalid-data domain, as depicted in Figure 4-1. The standard domain consists of inputs that a program can process "as is" (i.e., perfect data). At the other extreme, the invalid domain contains the inputs that a program cannot process and, therefore, they should be rejected with appropriate error messages. The incomplete-data domain is "between" the standard and invalid-data domains and it is made up of inputs that the program has to complete with default values before processing them. For reasons that will be explained later in this chapter, the model program that was developed in the experiment was especially designed to cover in particular the standard domain of the primary program.

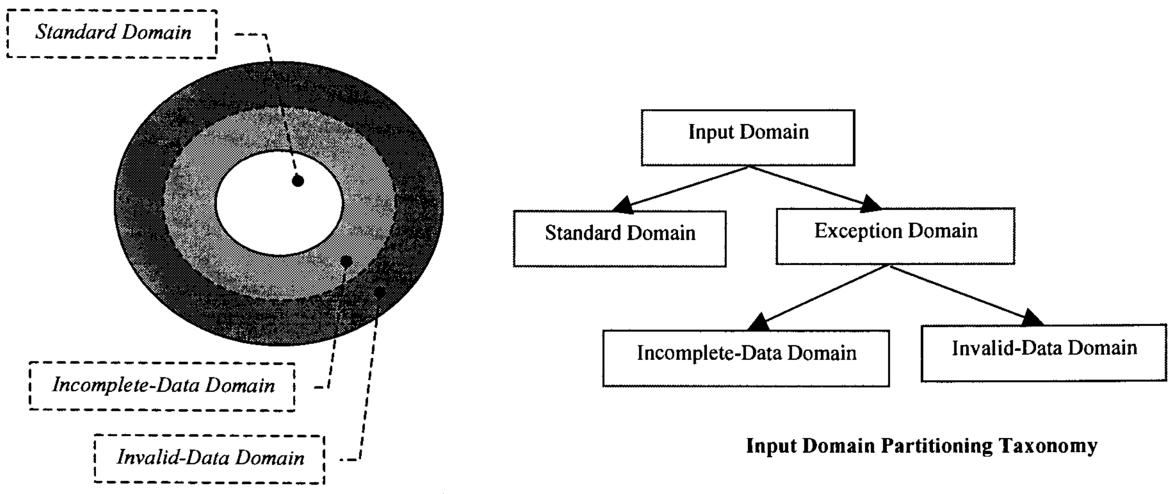


*Figure 4-1: Input Domain Partitioning into Standard and Exception Domains*

## 4.1 Results Summary

The case study that has been presented in the previous chapter provides *prima facie* evidence to suggest that the M-mp approach may be more cost-effective than testing based on manually pre-calculated results. Of course, the validity of such a conclusion is dependent upon the specific context in which the experiment was carried out. The most salient features of the present context are summarised below:

- The experiment developed, at low cost, a reliable and maintainable model program that provided the correctness checking means to test the core functionality of the primary program on standard test data without necessitating pre-calculation of expected results. As shown in Table 4-1, developing the model program required only 24% of the cost of manual test design. By excluding the effort that was spent on creating reusable artefacts (i.e., the specification language and the code generator) the ratio reduces to 17%.
- Because it was feasible to carry out a large number of arbitrary tests, a data generator, which is described in *Appendix C*, was developed especially for the experiment. As indicated in Table 4-1, test data generation was very economical. Its total cost represents only 11% of that of the manual test design. Moreover, if the data generator had been available at the start of the experiment, the ratio would have decreased to 3%!
- Two specification defects and two primary program faults were detected at low cost (around 2% of the manual test design effort) by analysing the disagreements that resulted from 50 tests. The test inputs were especially generated to facilitate cost-effective failure analysis and defect detection. Two model program faults and one defect in the test environment were also found. It is important to mention in this context that the experiment dealt with a mature version of the primary program – by the time the experiment started, the primary program had been in use for more than one year. Consequently, the likelihood of detecting specification and primary program defects was rather low.
- Because of the low cost of correctness checking, the experiment could easily have been extended to carry out tens of thousands of tests in a cost-effective manner. The testing of the primary program, on the other hand, was based on less than 150 test inputs.

| Activity | Total Cost | Net Cost (excluding reusable artefacts) |
|---|---|---|
| **Standard-Domain Test Design** | **35%** | **20%** |
| Model Program Development | 24% | 17% |
| Test Data Generation | 11% | 3% |

**Note**: All values represent percentages of the manual test design cost.

*Table 4-1: Actual Test Design Cost in the Experiment*

The direct interpretation of the results is that, by using the M-mp approach, 20 – 35% of the manual test design cost would have been sufficient to provide the capability of testing the core functionality of the primary program on its standard domain, using large data sets. However, it was not possible to determine accurately the actual percentage of the manual test design cost corresponding to the same testing scope as the M-mp tests (i.e., where the scope is limited to the core functionality). Therefore, the next discussion will show, based on sound estimates, that the 65 – 80% cost savings margin implied above could have been more than sufficient to make up the costs of an M-mp test design covering the system's full functionality. In addition, it will be shown that the M-mp approach could have also achieved higher test adequacy levels than the manual approach.

## 4.2 Discussion

### 4.2.1 Test Design Cost Comparison

To keep the cost within reasonable limits, the experiment proceeded gradually until sufficient evidence had been gathered to draw a valid conclusion. Because implementing the core functionality, which is the most complex part of the scheduling algorithm, proved very economical in comparison with the manual test design cost, completing the model program was not considered necessary in the end. As shown in Table 4-2, the estimated cost of implementing the complete functionality is reasonably low (25 – 33%)[1]. Consequently, although the extra effort would have provided better accuracy for the cost estimates, it is unlikely that the significance of the results would have been influenced.

| Activity | Adjusted Total Cost | Adjusted Net Cost (excluding reusable artefacts) |
|---|---|---|
| **Standard-Domain Test Design** | **45%** | **30%** |
| Model Program Development | 33% | 25% |
| Test Data Selection | 12% | 4% |

**Note**: All values represent percentages of the manual test design cost.

*Table 4-2: Adjusted M-mp Test Design Cost (1)*

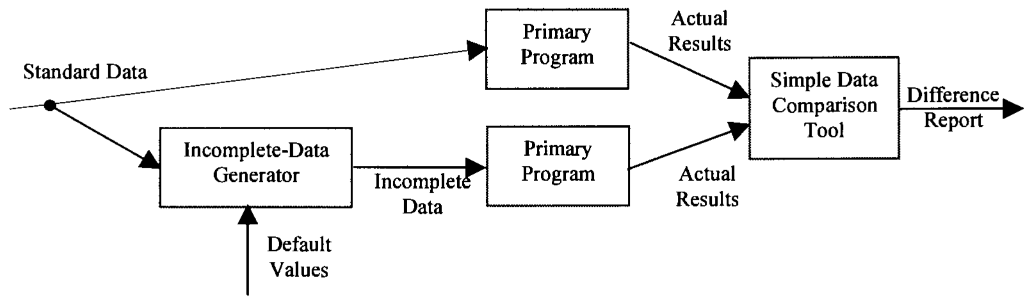
The adjusted results from Table 4-2 suggest that, by using the M-mp approach, 45% of the manual test design cost would have been sufficient to provide the capability to test the primary program on its standard input domain. If the reusable artefacts had been available at the start of the experiment, the ratio would have decreased to 30%.

Because the manually designed tests cover the entire input domain of the primary program, the next issue that has to be addressed is the cost of exception-domain test design. One way to cater for exception-domain testing would be to complete the model program to match the exception-handling capability of the primary program. That, however, might not be the most cost-effective option for the following reasons:

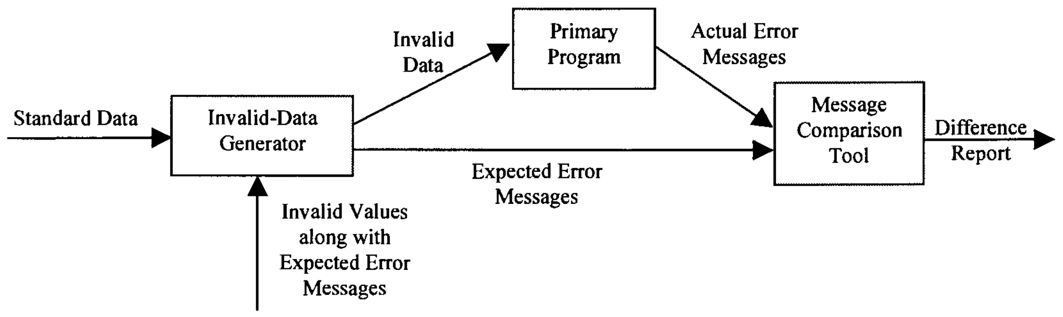
- Firstly, based primarily on experience, fully-fledged exception handling could increase significantly the complexity of a program and, consequently, its development cost. Besides other factors such as design simplicity and cost-effective code generation, limited exception handling could also explain why developing the model program (100% functionality) would have been 6 times cheaper than the primary program development.
- Secondly, as suggested in Figure 4-2, exception-domain testing can be automated efficiently by other means than using model programs. To test whether the *Primary Program* supplies correct default values, it is sufficient to generate *Incomplete Data* by removing explicit default values from (arbitrary) *Standard Data* and to run the program on both data sets. If the program handles incomplete data correctly, the *Actual Results* should be the same in both cases. Similarly, to test whether the *Primary Program* rejects invalid inputs with appropriate error messages, *Invalid Data* can be generated from *Standard Data* by replacing valid values with invalid ones. (This method is also mentioned for instance by Bishop et al (1986) and Cohen et al (1997).) If the program handles invalid data correctly, it should terminate without producing results and the

[1] In the previous chapter, to cater for the additional functionality, most of the test design costs were increased by a factor of 1.5.

*Actual Error Messages* should be the same as the *Expected Error Messages*. It is worth mentioning in this context that it is advisable to select only one invalid value per test to avoid masking effects (i.e., an invalid value might cause the program to "miss" other invalid values).



**Incomplete-Data Domain Testing**



**Invalid-Data Domain Testing**

*Figure 4-2: Envisaged Exception-Domain Testing Approach*

The test design for the exception-domain testing approach that is depicted in Figure 4-2 comprises writing the exception-data generators (if necessary) and setting up their inputs. Generating exception data, as discussed above, entails simple value substitutions and it is expected to be (much) less complex than generating standard data as it was done in the experiment. Therefore, the test data selection cost in Table 4-2 can safely be assumed to be an upper cost estimate for generating incomplete data and also for generating invalid data. Consequently, the exception-domain test design costs can be roughly estimated to be twice the costs of generating standard data. The cost estimates for a complete M-mp test design are shown in Table 4-3.

| Activity | Adjusted Total Cost | Adjusted Net Cost (excluding reusable artefacts) |
|---|---|---|
| **Test Design** | **69 %** | **38%** |
| **Standard-Domain Test Design** | **45%** | **30%** |
| **Exception-Domain Test Design** | **24%** | **8%** |

**Note**: All values represent percentages of the manual test design cost.

*Table 4-3: Adjusted M-mp Test Design Cost (2)*

Based on sound estimates, it has been shown that 38 – 69% of the manual test design cost could have been sufficient for an M-mp test design covering the system's full functionality. Besides the potential cost savings, the M-mp test design has another important advantage over the manual one: because there is no need for pre-calculated results, its cost is practically independent of the number of tests. The cost of manual test design, on the other hand, increases with the number of tests. This aspect is emphasised in Figure 4-3, where the vertical axis represents test design costs as a percentage of the manual test design cost in the current study. Accordingly, the *Manual Test Design Cost* graph passes through the point (150,100) and has a constant non-zero slope[2]. It intersects the y-axis at a point marked *Fixed Cost of Manual Test Design*. This point represents the cost of manual test design that is independent of the number of tests. It mainly involves equivalence class partitioning and boundary value analysis and it has been roughly estimated to account for 40% of the manual test design cost.

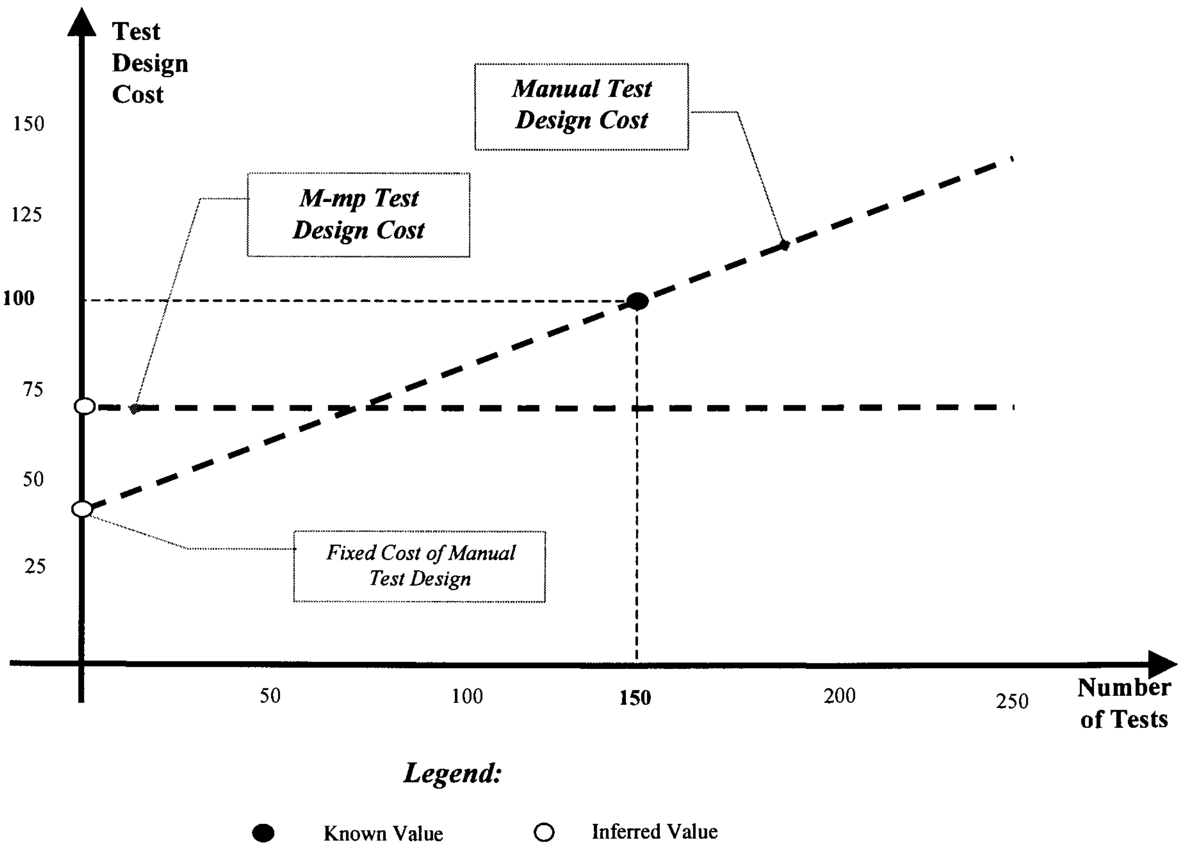


*Figure 4-3: Test Design Cost function of Number of Tests*

As suggested in Figure 4-3, designing tests manually is not feasible for large numbers of tests. As it will be shown next, this might limit the test adequacy levels or it might increase the cost of data selection.

## 4.2.2 Test Adequacy Comparison[3]

As mentioned in the previous chapter, it would have been ideal to conclude the experiment by comparing precisely the M-mp and manual approaches in terms of their cost-effectiveness with respect to various test adequacy criteria. Performing test adequacy comparisons, however, would have increased the

[2] The manual test design produced around 150 tests.

[3] The discussion pertains primarily to standard-domain testing.

cost and scale of the experiment beyond the scope of the present study. Moreover, while the testing of the primary program was based on a small data set (around 150 test inputs), the experiment could perform tens of thousands of tests in a cost-effective manner. Intuitively, for a 50-dimensional input domain, it is very likely that 15000 tests will achieve higher adequacy levels than 150, even if the test data accuracy requirements are relaxed. The term "data accuracy" is used in respect of a test generation procedure for a given application where a specified adequacy level is to be attained in terms of a specified adequacy criterion. The proportion of test inputs that the test generation procedure has to generate in order to achieve the stated adequacy level is then referred to as the data accuracy. For example, in Figure 4-4, 100% data accuracy corresponds to the minimum number of test inputs that achieve a specific test adequacy level (e.g., 65% branch coverage), while 0% means that none of the generated test inputs exercises the adequacy criterion that is being used. In general, the lower the data accuracy of a test data generation procedure, the more the numbers of tests that have to be generated to achieve a given adequacy level.

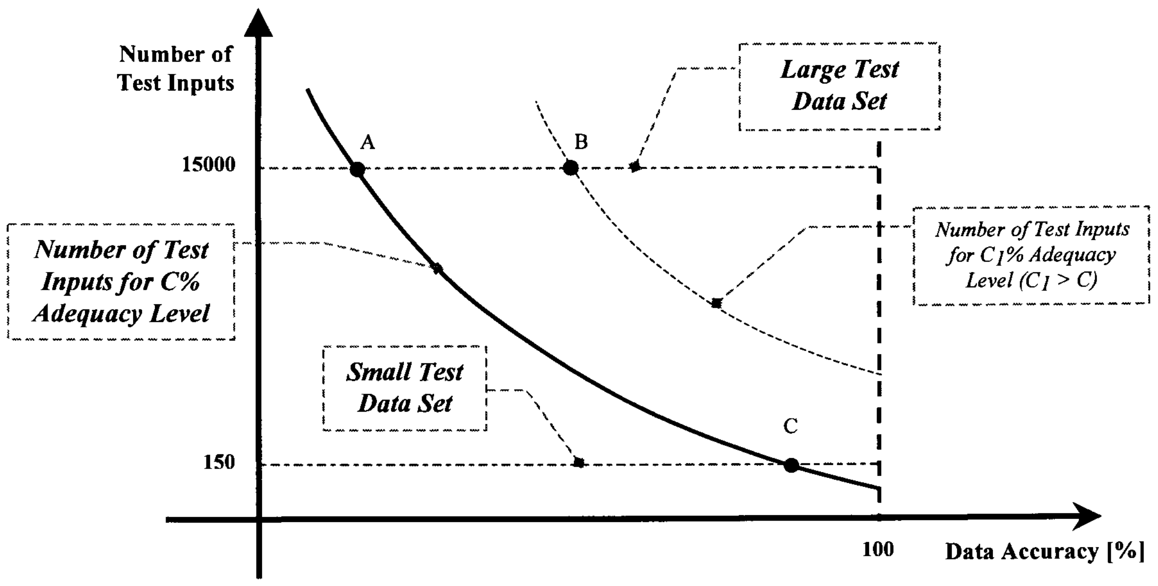


*Figure 4-4: Adequate Test Data Size function of Data Accuracy*

As suggested in Figure 4-4, even if the 150 test inputs derived from one test generation method are highly accurate (point C), 15000 test inputs derived by another test generation method could achieve the same adequacy level (point A) albeit with significantly less accuracy. In fact, the 150000 test inputs might even achieve a higher adequacy level should its accuracy turn out to be better (Point B). Moreover, selecting highly accurate data is likely to be very expensive, while an automated procedure to generate less accurate data could cost much less. The test data generator that was developed in the experiment provided the capability of selecting high quality test data based on accurate input domain analysis. Since domain analysis was the basis for selecting the test inputs in the manual test design as well, the accuracy difference between the generated data and the manually selected data is expected to be relatively low. Consequently, as indicated in Figure 4-4 (point B), the experiment has the potential to achieve adequacy levels that are perhaps not even attainable by 150 test inputs, no matter how accurate.

If one wished to extend the experiment, a large number of tests could have been carried out in a cost-effective manner mainly because of the low cost of test evaluation. Firstly, program disagreements could be recorded automatically for around 8640 test inputs in 24 hours, for 60480 in one week and so forth[4]. Secondly, disagreement analysis proved extremely cost-effective because it was possible to reduce the number of disagreements that had to be analysed explicitly. It is obvious that once a defect is removed, all its related disagreements, whether explicitly analysed or not, would disappear. In other words, it is practically sufficient to analyse thoroughly only one disagreement per defect. It is logical to conclude, therefore, that the overall cost of disagreement analysis would have been largely determined by the number of exercised residual defects and not by the number of tests (or by the number of recorded disagreements). Moreover, the cost of disagreement analysis per defect may be assumed to be much lower on average than in the manual approach. That is because the data mutation and correlation techniques that have been mentioned in*Chapter 3* are not feasible in the manual approach because they may require additional tests and, therefore, more manually pre-calculated results. Consequently, as also highlighted in the*Pilot Study* that has been presented in the previous chapter, resolving differences between the actual and the expected outputs usually requires re-calculating the expected results by hand and, therefore, it can become very expensive.

Besides the potential high cost of test evaluation, the number of tests that could feasibly be carried out would also have been limited if the model program maintenance turned expensive. Within the scope of the experiment, however, the model program proved as reliable as the primary program and the cost of localising and removing its two defects was rather insignificant (only a few hours!).

To summarise, it has been argued that, because of the low cost of test evaluation and model program maintenance, the experiment could relatively easily have been extended to carry out a much larger number of tests than the manual approach. Because the input to the test data generator was based on domain analysis, which was also the method for selecting the test inputs in the manual test design, it is almost certain that the larger number of tests could achieve higher adequacy levels. This hypothesis is also confirmed by the fact that two specification defects and two primary program faults were detected with relative ease even though the primary program had been in use for more than one year. Those defects went undetected by the manually designed tests.

Based on *prima facie* evidence, sound estimates, and theoretical considerations, it has been shown that the M-mp approach could test the scheduling program more adequately than the manually designed tests and at lower cost. Of course, the validity of such a conclusion is dependent upon the specific conditions of the experiment. The immediate question that arises is whether similar results are likely to have been obtained if a different team had implemented the M-mp approach to test the same application. In other words, is the experiment repeatable?

## 4.3 Repeatability of the Experiment

The low cost of developing a reliable and maintainable model program in the experiment was a direct consequence of an M-mp testing principle that has been presented in the first chapter: a model program does not need to be equivalent to the primary program and it should cover primarily functionality whose correctness checking is normally expensive. In the case of the scheduling application, as discussed earlier, to eliminate the need for manually pre-calculated results, it is sufficient for a model program to cater only for standard-domain scheduling. Consequently, a model program does not need to implement fully-fledged exception handling and in addition, because it could be run on special hardware if necessary, its speed and resource usage are not very important[5]. Mainly because of these simplifications, which are applicable irrespective of particular design methods and programming environments, the development of the model program was straightforward. Except for defining the specification language and writing the code generator,

[4] In the experiment, executing the primary program and then the model program took on average 10 seconds per test input.

[5] In the experiment, the speed of the model program was comparable with that of the primary program, while the size and the resource usage were bigger, but reasonably low.

developing the model program did not require advanced analytical and programming skills. Most probably, if a high-level development environment is available, a person who is able to schedule tasks by hand is also sufficiently skilled to write a reliable and maintainable model program with minimum training. Alternatively, developing the model program could be a joint effort of people having complementary knowledge and skills.

For practical reasons, a specification language was defined and a code generator was developed. In the general case, however, such reusable artefacts might be already available or they could be created through a separate project. Moreover, a code generator as the one built in the experiment might not be needed if the implementation is done, for instance, in 4GL or in a rapid application development (RAD) environment.

Given the above considerations, it is reasonable to expect that developing a model program for standard-domain scheduling will be in general as cost-effective as suggested by the case study. Because test data selection can usually be automated cost-effectively[6], it is likely that if the experiment as a whole were to be repeated then similar results would be obtained. This suggests that the M-mp approach could be regarded as more cost-effective than the manual approach for adequately testing the scheduling application. As will be discussed next, this conclusion might apply in general to testing algorithmically complex software.

## 4.4 Testing Algorithmically Complex Software

Intuitively, the simpler the algorithm, the easier the calculation of expected results and the simpler the development of the model program. Because the same applies to the selection of test data, this suggests that, given a fixed number of manually designed test cases, the ratio between the M-mp and manual test design costs should not vary much from application to application. Therefore, the cost graphs that are depicted in Figure 4-3 are likely to be relevant for a wide range of algorithms, whether simpler or more complex than the scheduling one. As suggested in Figure 4-3, the M-mp approach may be a better option than the manual one if the number of tests that are required to achieve particular adequacy levels is greater than 100. That is usually the case when testing algorithmically complex software.

[6] Test data selection is probably the most widely spread subject in the testing literature and it is usually supported by good tools.

# APPENDIX A: THE SPECIFICATION LANGUAGE

## Language Definition

The specification language is an adaptation of Shlaer-Mellor Action Language (SMALL, *WWW Resources (2)*). SMALL is primarily used to specify the actions that are associated with Moore states in Shlaer-Mellor OOA/RD (Object-Oriented Analysis/Recursive Design). The adaptation involves mostly the mixing in of certain C++ notation and it also introduces the author's own naming conventions. The definition of the specification language as used in the M-mp experiment is given in the table below:

| **Language Syntax and Conventions** | **Examples** |
|---|---|
| Comment | // Comment |
| Data types: STRING, BOOLEAN,<br>INTEGER, REAL, DATE, TIME,<br>DATETIME | |
| System limits:<br>STRING: [0, SYS_MAXSTRLENGTH]<br>INTEGER:[SYS_MININT, SYS_MAXINT]<br>REAL: [SYS_MINREAL, SYS_MAXREAL]<br>DATE: [SYS_MINDATE, SYS_MAXDATE]<br>TIME: [SYS_MINTIME = 00:00:00.00,<br>SYS_MAXTIME = 23:59:59:99]<br>DATETIME: (DATE, TIME) | |
| Data types conventions:<br>A variable of any data type can<br>be NULL (i.e., not specified)<br>By default the constraint on any<br>variable is (!= NULL and within<br>the system limits) | |
| Number literal | 3, 4.25, 3E-6 |
| String literal | "Therapy", "11:30:00" |
| Boolean literal | true, false |
| Relational operators: '!', '!=',<br>'<', '<=', '==', '>=', '>' | |
| Logical operators: 'and', 'or' | |
| Arithmetic operators: '+', '-',<br>'*', '/', '**'(power) | |
| 'Write/Assign From' operator: '=' | l_temperature = 21; |
| Composition operator: , | l_OvenProperties = (l_Temperature, l_Pressure);<br>// l_OvenProperties is (implicitly) defined as<br>(numeric, numeric) the composition of 2 numbers |
| Prefix for local variables<br>(except reference variables):<br>'l_' | l_Temperature // non-reference local variable<br>i_Temperature // input argument<br>o_Temperature // output argument |

| Language Syntax and Conventions | Examples |
|---|---|
| Prefix for input arguments: 'i_'<br>Prefix for output arguments: 'o_' | |
| Reference variable - contains one or more references to instances of the same object/class | `myOven, myOvens` |
| Access to non-referential attributes (.) | `myOven.(Temperature, Pressure) = l_OvenProperties;` |
| Selecting an arbitrary instance and writing a value to an attribute | `myOven = Oven(one);`<br>`Oven(one).Temperature = 20;`<br>`myOven.Temperature = 20;` |
| Selecting a specific instance | `myOven = Oven(one,(Temperature == 23) and (Pressure == 4));`<br>`myOven = Oven(one, Temperature >= 23, /Temperature);`<br>`myOven = Oven(first, Temperature >= 23, /Temperature);`<br>`myOven = Oven(last, Temperature >= 23, /Temperature);` |
| Selecting and using multiple instances | `myOvens = Oven(all);`<br>`myOvens = Oven(all,(Temperature >= 23) or (Pressure <= 4));`<br>`myOvens.Temperature = 25; // Writes to ALL selected instances`<br>`l_BagOfTemperatures = myOvens.Temperature;` |
| Selecting in Order<br>Ascend: '/', Descend: '\' | `l_OrderedBag = Oven(all,Temperature > 21, /Temperature \Pressure).Temperature;` |
| Abstract syntax of an instance Selector | `([<qualifier>][<where clause>][<'order by' clause>])` |
| Creating instances<br>Create operator: '>>'<br>• It is illegal to initialise a referential attribute.<br>• An identifying, non-referential attribute must be always initialised. | `("X", 24, 4) >> Oven.(Name, Temperature, Pressure);`<br>`>> Oven(Name = "X", Temperature = 24, Pressure = 4);`<br>`myOven = (("X", 24, 4) >> Oven.(Name, Temperature, Pressure));`<br>`l_Name = "X"; l_a~Temperature = 24;`<br>`>> Oven(l_Name, l_a~Temperature, Pressure = 4);` |
| Deleting instances<br>Delete operator: '<<' | `<< myOven;`<br>`<< Oven(Temperature < 20);` |
| Data flow<br>Pipe (flow) operator: '\|' | `(myOven.(Temperature, Pressure), l_x, l_y) \| (l_z = ComputeSomething);` |
| shuffle - language-defined process | `(myOven.(Temperature, Pressure), l_x, l_y) \| shuffle(2,4,1,3) \| (l_z = A);`<br>`l_z = (myOven.(Temperature, Pressure), l_x, l_y) \| shuffle(4,1,3) \| B; // data element 2 is dropped` |
| Calling external component services | `(l_myRobot:i_RobotId, 3:i_distance) \| ROBOT_RetractHand;`<br>`ROBOT_RetractHand(l_distance, l_my~RobotId);`<br>`(l_my~RobotId, 3:distance) \| ROBOT_RetractHand;`<br>`l_Current = MAGNET_GetCurrent(l_myMagnet:i_MagnetId);` |

| Language Syntax and Conventions | Examples |
|---|---|
| | (l_my~MagnetId) \| (l_Cur:o_Current = MAGNET_GetCurrent);<br>(l_my~MagnetId) \| (l_Cur = MAGNET_GetCurrent); // l_Cur:o_Current is implied<br>(l_x, l_y, l_th:o_theta) = (l_aRobot:i_RobotId) \| ROBOT_WheresTheRobot); |
| Calling class specific methods | ("X":OvenName) \| Oven.Create;<br>Oven.Create(l_my~OvenName); |
| Calling instance specific methods | l_aTemperature \| Oven(all).SetTemperature;<br>23:Temperature \| aOven.SetTemperature;<br>aOven.SetTemperature(23); // 23:Temperature is implied |
| Calling component private services | ("33-Mar-1998":i_date) \| ValidDate ?<br>"33-Mar-1998" \| ValidDate ? |
| Return - language-defined process | (val1:<outp1>, val2:<outp2>,...) \| Return; |
| Traversing a single relationship | aTherm = myOven->[R2.IsEquippedWith]Thermometer(one);<br>aTherm = myOven->[R2.'Is equipped with']Thermometer(one);<br>aTherm = myOven->[R2]Thermometer(one);<br>allTherms = myOven->[.IsEquippedWith]Thermometer(all);<br>aTherm = myOven->Thermometer(one); // One direct relationship |
| Traversing multiple relationships | l_Volume = myBench->[R2]GasLine(one)->[R4]GasBottle(one).Volume;<br>l_Volume = myBench->[R2->R4]GasBottle(one).Volume;<br>l_Volume = myBench->GasLine(one)->GasBottle(one).Volume; // the best |
| Reaching multiple instances | l_AllTemperatures = myOvens->Thermometer(all).Temperature;<br>l_OneTemperaturePerOven = myOvens->Thermometer(one).Temperature; |
| Unique - language-defined process | l_BagOfOvenNames = Thermometer(all)->Oven(all).Name;<br>Thermometer(all)->Oven(all).Name \| (l_SetOfOvenNames = unique); |
| Link - language-defined process | (myOven, aTherm) \| link [R5];<br>(myOven, aTherm) \| link [R6] >> ThermPerOven;<br>(aEqp, aMet) \| link [R7] >> DURP.(27/12/1998:FromDate);<br>(myOven, aTherm) \| link;<br>(myOven, aTherm) \| link >> ThermPerOven;<br>(aEqp, aMET) \| link >> DURP.(27/12/1998:FromDate); |
| Unlink - language-defined process | (myOven, aTherm) \| unlink [R5];<br>(myOven, aTherm) \| unlink [R6] << ThermPerOven;<br>(aEqp, aMET) \| unlink [R7] << DURP.(FromDate == 27/12/1998);<br>(myOven, aTherm) \| unlink;<br>(myOven, aTherm) \| unlink << ThermPerOven; |

| **Language Syntax and Conventions** | **Examples** |
|---|---|
| | (aEqp, aMET) \| unlink << DURP.(FromDate ==<br>27/12/1998); |
| Guards | l_Temperature = myOven.Temperature @TempKnown; //<br>Setting a guard<br>  @TempKnown: << myOven; // Guard handling<br><br>@A or @B: statement; // execute 'statement' if @A or<br>@B have been set<br><br>@A and @B: statement; // execute 'statement' if @A and<br>@B have been set<br><br>@A: [statement1;<br>    statement2;] // Guarded block of statements<br><br>myOven->[R5]Thermometer(Name == "X").Temperature \|<br>IsOk ? @OK, @NotOK;<br>  @OK: ; // this guard handling is optional<br>  @NotOK: statement;<br><br>myOven->Thermometer(== l_Name).Temperature \| IsOk ?<br>  @NotOK: statement;<br><br>(l_temperature >= 34) ?<br>  @True: statement;<br>  @False: statement; |
| Preserving previous selections | Crate(all).(l_Height, l_Width, l_Depth) \|<br>(Crate().Volume = ComputeVolume)<br>// Crate() means Crate(all)<br>Crate(all).(l_Height, l_Width, l_Depth) \|<br>ComputeVolume \| Size ? @Big, @Small;<br>  @Big: Crate().Transport = "Slow Boat";<br>          // Crate() means Crate(all, Size ? == @Big)<br>  @Small: Crate().Transport = "Plane";<br>          // Crate() means Crate(all, Size ? ==<br>@Small) |
| None? @IsNone, @IsOne, @IsMany;<br>language-defined test process | Oven(all, Temperature > 23) \| None ?<br>  @IsNone: "no ovens?!" \| SendErrorMessage;<br>  @IsOne: "one oven" \| SendMessage;<br>  @IsMany: (Oven() \| Cardinal, " ovens") \|<br>SendMessage;<br><br>myOven->Thermometer(one) \| None ?<br>  @IsMany: // can never be set |
| CurrentTime(o_year, o_month,<br>o_day, o_hour, o_minute, | (l_year, l_month, l_day, l_hour, l_minute, l_second,<br>l_secondFraction) = CurrentTime; |

| Language Syntax and Conventions | Examples |
|---|---|
| `o_second, o_secondFraction)` | `l_date = (CurrentTime \| shuffle(1,2,3));`<br>`l_time = (CurrentTime \| shuffle(4,5));` |
| Loops<br><br>`LOOP [[WHILE <cond>] \| [<elem> IN <set>]]`<br>`    statement;`<br>`    BREAK;`<br>`    CONTINUE;`<br>`ENDLOOP` | `l_temperature = 0;`<br>`LOOP`<br>`  (l_temperature == i_target) ?`<br>`     @True: @TargetReached BREAK;`<br>`     @False: (l_temperature > i_target) ?`<br>`         @True: 2:delta \| Heater.ReduceT;`<br>`         @False: l_plus~delta \| Heater.IncreaseT;`<br>`ENDLOOP`<br><br>`LOOP WHILE (l_temperature != i_target) @TargetReached`<br>`  (l_temperature > target) ?`<br>`      @True: l_minus~delta \| Heater.ReduceT;`<br>`      @False: l_plus~delta~1 \| Heater.IncreaseT;`<br>`ENDLOOP`<br><br>`LOOP oneHeater IN allHeaters`<br>`  (oneHeater.temperature > i_target) ?`<br>`      @True: 2 \| oneHeater.ReduceT;`<br>`      @False: l_D \| oneHeater.IncreaseT;`<br>`ENDLOOP` |

## Method Specification Example

The purpose of the method specified below, *SchedFirstFromCOCC*, is to schedule the first occurrence of a recurring task (*RT*) relative to the previous one, which is 100% completed. *COCC* stands for "Completed (Task) Occurrence".

The method takes one argument of type *DATE*, *i_endDate*, and it may set a guard, *@ExitCriterionMet*, if, for instance, the newly scheduled occurrence falls outside the planning horizon. In that case, the scheduling for the specific *RT* ends. The caller of the method is responsible for handling the guard.

The pre-condition verifies that the previous occurrence is indeed a completed one. *PCOCC* (i.e., "Partially Completed Occurrence") and *COCC* are subtypes of *TOCC* (i.e., "Task Occurrence"). *thisRT* identifies the specific *RT* instance (e.g., car service A). The pre-condition states that there must be at least one *COCC* instance and no *PCOCC* instances linked to *thisRT*. The method specification has no post-conditions.

The implementation starts by selecting the last completed occurrence, *cocc*. The start of the next occurrence is determined by taking the earliest scheduling date that results from applying either calendar-based (e.g., every 6 months) or meter-based scheduling rules (i.e., every 10000 Km). The first loop iterates through all *CALMET* instances (i.e., Calendar "Meters") to determine the earliest start date according to the calendar-based rules. Similarly, the second loop, which is not included in the example, determines the earliest start date according to the meter-based rules. The earlier of the two dates is used as the start date for the next occurrence.

```
RT.SchedFirstFromCOCC
   (i_endDate:DATE) ? @ExitCriterionMet
@Pre: // Pre-condition
   thisRT->TOCC:PCOCC(one) | None ? == @IsNone and
   thisRT->TOCC:COCC(one) | None ? == @IsOne;
@Post: // Post-condition
@Implementation:
   cocc = thisRT->TOCC:COCC(last, /OccNumber);

   // Determine the Calendar-based start dates
   earliestCalmet = NULL;
   earliestCalmetOrigSchedStart = UNDEFINED;
   earliestCalmetSchedStart = UNDEFINED;

   LOOP thisCALMET in thisRT->CALMET(all)
      (thisCalmetOrigSchedStart, thisCalmetSchedStart) =
                                        thisRT.DetermineCalmetStartForNextUocc(
                                              thisCALMET,
                                              cocc.schedStart:i_prevSchedStart,
                                              cocc.schedEnd:i_prevSchedEnd,
                                              cocc.actualStart:i_prevActualStart,
                                              cocc.actualEnd:i_prevActualEnd);

      (earliestCalmet == NULL) ?
         @True:
            earliestCalmet = thisCALMET;
            earliestCalmetOrigSchedStart = thisCalmetOrigSchedStart;
            earliestCalmetSchedStart = thisCalmetSchedStart;
         @False: (thisCalmetOrigSchedStart < earliestCalmetOrigSchedStart) ?
            @True:
               earliestCalmet = thisCALMET;
               earliestCalmetOrigSchedStart = thisCalmetOrigSchedStart;
               earliestCalmetSchedStart = thisCalmetSchedStart;
   ENDLOOP

   // Determine the Meter-based start dates
   earliestMet = NULL;
   earliestMetOrigSchedStart = UNDEFINED;
   earliestMetSchedStart = UNDEFINED;

   LOOP thisMET in thisRT->MET(all)

      // Similar to the previous loop

   ENDLOOP
   // Choose the earliest between Calendar-based and Meter-based start dates
   earliestCalmetOrigSchedStart < earliestMetOrigSchedStart ?
      @True:
         // Calmet
         l_origSchedStart = earliestCalmetOrigSchedStart;
         l_schedStart = earliestCalmetSchedStart;
         earliestMet = NULL;
      @False:
         // Met
         l_origSchedStart = earliestMetOrigSchedStart;
         l_schedStart = earliestMetSchedStart;
         earliestCalmet = NULL;

   // Create the first UOCC
   newUocc = RT.CreateFirstUocc(
                 cocc.OccNumber + 1:i_OccNumber,
                 l_origSchedStart,
                 l_schedStart);

   // Link CALMET or MET
   (earliestMet != NULL) => (newUocc, earliestMet) | link;
   (earliestCalmet != NULL) => (newUocc, earliestCalmet) | link;

   // Verify the exit criteria
   thisRT.CheckExitCriteriaForLastUocc(i_endDate) ?
      @ExitCriterionMet: return;
@End
```

# APPENDIX B: THE CODE GENERATOR

The code generator was implemented using Microsoft Excel spreadsheets and Visual Basic (VB) macros. Simply, the code generator works as follows: a VB macro reads the code generation parameters of a class from an MS Excel spreadsheet and produces generic C++ and embedded SQL (Structured Query Language) code. The table below shows an example of the input to the code generator for a class called *UTIL*:

| | ASSET_ID | DATE_FROM | UTIL_RATE | UTIL_LMU | EQP_UTL_CODE | | |
|---|---|---|---|---|---|---|---|
| | INTEGER | DATE | REAL | STRING | STRING | | |
| | | | | 32 | 32 | | |
| IN | NOTNULL | NOTNULL | NOTNULL | NOTNULL | NULL | | |
| OUT | | | | | | | |
| ERROR | | | < 0 | | | | |
| ALL | | | | | | | |
| | EQ | | | | | asset_id | |
| UtilLmu | | | | 1 | | NONUNQ | |
| DateFrom | | 1 | | | | NONUNQ | |
| Unq | | 2 | | 1 | | UNQ | NAV |

The first section, which is made up by the first three rows, specifies the structure of the class. The top row contains the names of the data members, while the second specifies their types. The third row provides the maximum lengths of *STRING* data members. The corresponding C++ and embedded SQL code is shown below.

```
class cUTIL // C++
{
// generic class methods (not included in this example)

private:
    BASE_id             m_util_id;
    long                m_asset_id;
    CAL_linDate         m_date_from;
    double              m_util_rate;
    char*               m_util_lmu;
    char*               m_eqp_utl_code;

    // more data members (not included in this example)
};
```

```
struct UTIL_sREC // embedded SQL
    {
    long                asset_id;
    short               asset_id_ind;
    char                date_from[11];
    short               date_from_ind;
    double              util_rate;
    short               util_rate_ind;
    char                util_lmu[33]; // 32 + 1
    short               util_lmu_ind;
    char                eqp_utl_code[33];
    short               eqp_utl_code_ind;
    };
```

The second section, which contains the *IN*, *OUT*, and *ERROR* rows, specifies the input argument constraints upon creating an instance of a class (*IN* and *ERROR*) and the data members whose values are to be written to the database (*OUT*). The latter does not apply in this example (i.e., data is "read-only"). The *IN* row specifies whether the input argument that corresponds to a data member is compulsory (*NOTNULL*) or optional (*NULL*). The *ERROR* row contains constraints on the input domains of the data members. In this example, the constraint means that supplying a negative value for *UTIL_RATE* is an error. Below is the listing of the creation method that was produced by the code generator. ("-11" is used as general "null" indicator for numerical data types. To cater for cases when "-11" is a valid value, the input arguments have to be set to *NOTNULL* even if they are compulsory and the validation code has to be written manually.)

```
cUTIL*
cUTIL::Create(
    long                    i_asset_id,
    CAL_linDate             i_date_from,
    double                  i_util_rate,
    char*                   i_util_lmu,
    char*                   i_eqp_utl_code)
{
    cUTIL*                  util;

    if (i_asset_id == -11)
        BASE_EXCEPTION(ErrExit, "Create: i_asset_id must not be NULL")

    if (i_date_from == -11)
        BASE_EXCEPTION(ErrExit, "Create: i_date_from must not be NULL")

    if (i_util_rate == -11)
        BASE_EXCEPTION(ErrExit, "Create: i_util_rate must not be NULL")

    if (i_util_lmu == NULL)
        BASE_EXCEPTION(ErrExit, "Create: i_util_lmu must not be NULL")

    if (i_util_rate != -11 && i_util_rate < 0)
        BASE_EXCEPTION(ErrExit, "Create: i_util_rate is invalid")

    util = UnqGetInst(i_util_lmu, i_date_from);
    if (util != NULL)
        BASE_EXCEPTION(ErrExit, "Create: Duplicate instance(Unq)")

    util = new cUTIL;

    util->m_util_id = GenId();
    util->m_asset_id = i_asset_id;
    util->m_date_from = i_date_from;
    util->m_util_rate = i_util_rate;
    util->m_util_lmu = strdup(i_util_lmu);

    if (i_eqp_utl_code)
        util->m_eqp_utl_code = strdup(i_eqp_utl_code);
    else
        util->m_eqp_utl_code = NULL;

    return util;

ErrExit:
    return NULL;
}
```

The third section of the input to the code generator provides the necessary information for setting up the embedded SQL where clause. In this example, it means "read *ALL* records from the *UTIL* table where the value of the *ASSET_ID* column is equal to that of the *asset_id* parameter". The embedded SQL function that reads the *UTIL* records from the database is shown below.

```
BASE_ret
UTIL_Load(
    long                i_asset_id)
{
    cUTIL*              util = NULL;

    EXEC SQL BEGIN DECLARE SECTION;
    struct UTIL_sREC    rec;
    long                asset_id = -11;
    EXEC SQL END DECLARE SECTION;

    asset_id = i_asset_id;

    EXEC SQL WHENEVER SQLERROR
        DO BASE_EXCEPTION(ErrExit,"SqlError");

    // Declare and open the cursor(s)
    {
    EXEC SQL DECLARE c1_util CURSOR FOR
        SELECT
            asset_id,
            TO_CHAR(date_from,'YYYYMMDD'),
            util_rate,
            util_lmu,
            eqp_utl_code
        FROM UTIL
        WHERE
            asset_id = :asset_id  // Where Clause
        ;

    EXEC SQL OPEN c1_util;
    }

    // Retrieve the db record(s) and create the corresponding instance(s);
    // close the cursor(s)
    {
    for (;;)
        {
        EXEC SQL FETCH c1_util INTO
            :rec.asset_id:rec.asset_id_ind,
            :rec.date_from:rec.date_from_ind,
            :rec.util_rate:rec.util_rate_ind,
            :rec.util_lmu:rec.util_lmu_ind,
            :rec.eqp_utl_code:rec.eqp_utl_code_ind;

        if (SQLCODE == ESQL_NOT_FOUND)
             break;

        CreateUtil(&rec); // Create the cUTIL instance from UTIL_sREC
        }

    EXEC SQL CLOSE c1_util;
    }

    return BASE_SUCCESS;

ErrExit:
    if (SQLCODE < 0)
        cBASE::Log("\tSQLCODE = %ld", SQLCODE);

    return BASE_FAILURE;
}
```

The last section of the input to the generator specifies "order by" criteria ("indexes"). The third "index" (*Unq*), which is an identifier/key of the class (*UNQ*), is used by default to "navigate" from one instance to the next (*NAV*). "1" and "2" mean "order by *UTIL_LMU* first and then by *DATE_FROM*". The part of the class interface that reflects best the purpose of the last section is shown below.

```
class cUTIL
{

// Other class elements (not included in this example)

// Instance identification
// ------------------------
static BASE_id
UnqGetId(
    char*                 i_util_lmu,
    CAL_linDate           i_date_from);

// Selection filters
// ------------------
static BASE_id
SelectAll(
    BASE_id               i_selId);    // it uses the UNQ index

static BASE_id
UtilLmuSelect(
    BASE_id               i_selId,
    BASE_eRelation        i_relation,
    char*                 i_util_lmu);

static BASE_id
DateFromSelect(
    BASE_id               i_selId,
    BASE_eRelation        i_relation,
    CAL_linDate           i_date_from);

static void
ReleaseSelection(
    BASE_id               i_selId);

// Selection navigation
// --------------------
static cUTIL*
SFirst(
    BASE_id               i_selId);

static cUTIL*
SLast(
    BASE_id               i_selId);

static cUTIL*
SNext(
    BASE_id               i_selId);

static cUTIL*
SPrev(
    BASE_id               i_selId);

// Other class elements (not included in this example)
};
```

# APPENDIX C: THE TEST DATA GENERATOR

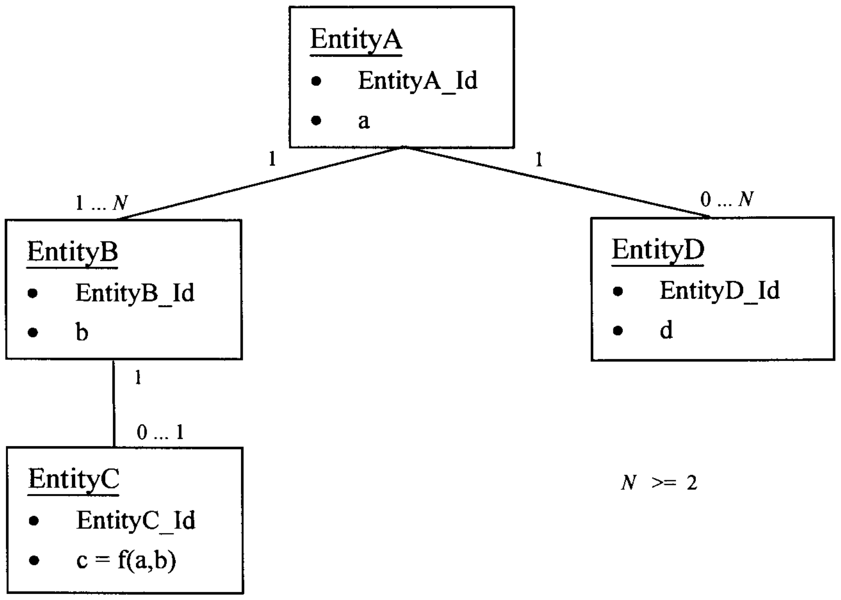
Although the original intention was to acquire a test data generator, the time scale of the experiment was too short to allow proper evaluation and integration of available tools. Moreover, the data generators that were considered (i.e., DGL (*WWW Resources (3)*, DataShark (*WWW Resources (4)*), and Datatect™ (*WWW Resources (5)*), did not seem to have all of the following desirable properties:

1) support for generation of relational data;
2) support for selecting test inputs according to probability distributions over attribute and cardinality[1] domains; and
3) support for evaluating expressions defined over attributes and cardinalities.

The first of the above features was desirable because the operations-management programs process relational data. The second one supports statistical testing, which is definitely worth considering in the M-mp context. That is because in M-mp testing it is usually feasible to carry out large numbers of tests. The third property makes provision for expressing equivalence class boundaries that involve more than one attribute and/or cardinality, thus accommodating domain testing.

Perhaps the tools that have been mentioned in the first paragraph could have been integrated or extended to achieve the desired data generation capability in the experiment. However, that capability was more conveniently obtained by writing a data generator that reused most of the components available in the organisation for C development (including an *Expression Evaluator*).

Essentially, the test data generator traverses a user-defined hierarchy of entities in a depth-first fashion and generates records for each entity. A simple example of such a hierarchy is depicted below.



***Hierarchical Entity Relationship Diagram***

[1] In data modelling, a relationship between two entities/tables is described in terms of "cardinality". Simply, it is a pair of domains that correspond to the number of records in one table that are linked to a single record in the other and *vice versa*. Its meaning will become clearer in the example that is used to describe how the test data generator works.

In the above diagram, each entity has two attributes: its identifier (e.g.,*EntityA_Id*) and an abstract attribute (e.g., *a*). The non-identifier attribute of *EntityC* is a function of the non-identifier attributes of *EntityA* and *EntityB*. The cardinality of the relationship between *EntityA* and *EntityB* is one-to-many (1:N), that is, each instance/record of *EntityA* can be associated with more than one *EntityB* record, whereas each instance of *EntityB* is always linked to a single *EntityA* record. The relationship between *EntityA* and *EntityD* is also one-to-many. However, there might be situations when an *EntityA* instance has no linked *EntityD* records and the relationship (from *A* to *D*) is usually characterised as being "optional" (or "conditional"). The relationship between *EntityB* and *EntityC* is one-to-one (optional).

An example of the input to the test data generator that corresponds to the above entity relationship diagram is shown below using an abstract syntax.

```
Cardinalities
EntityA_EntityB = 80:{random(1,5)} | 20:{random(6,300)}
EntityA_EntityD = 40:0 | 40:1 | 20:{random(2,1000)}
EntityB_EntityC = 30:0 | 70:1

EntityA Attributes
EntityA.EntityA_Id = 100:{EntityA.EntityA_Id + 1}
EntityA.a = 70:a1 | 30:{random(a2,a3,a4)}

EntityB Atrributes
EntityB.EntityB_Id = 100:{EntityB.EntityB_Id + 1}
EntityB.EntityA_Id = 100:{EntityA.EntityA_Id}; referential attribute[2]
EntityB.b = 25:b1 | 75:b2

EntityC Atrributes
EntityC.EntityC_Id = 100:{EntityC.EntityC_Id + 1}
EntityC.EntityB_Id = 100:{EntityB.EntityA_Id}; referential attribute
EntityC.c = 25:c1 | 75:{f(EntityA.a, EntityB.b)}

EntityD Attributes
EntityD.EntityD_Id = 100:{EntityD.EntityD_Id + 1}
EntityD.EntityA_Id = 100:{EntityA.EntityA_Id}; referential attribute
EntityD.d = 100:{random(d1,d2)}
```

Probability distributions over the attribute and cardinality domains are defined by means of weights (i.e., the numbers in bold that are followed by colons). For instance, the value of *EntityA.a* will be *a1* for around 70% of all generated *EntityA* records, and *a2, a3, or a4* for the rest. In this particular case, a probability distribution such as "EntityA.a = **70**:a1 | **10**:a2 | **10**:a3 | **10**:a4" is equivalent to the original one as the ***random*** function implements a pseudo-uniform distribution over the specified domain. For convenience, the weights have been chosen to add up to 100, but that is not a requirement.

Distinct test values/choices are separated by vertical bars and they can be expressions (e.g., {*EntityA.EntityA_Id* + 1}). All cardinalities and attributes can be used as expression variables and they retain their last generated values. Besides cardinalities and attributes, extra variables can be defined and used as needed (not shown in this example).

[2] The referential attributes are not depicted in the entity relationship diagram.

Given the above input, the test data generator will produce its output as follows (depth-first, left-to-right iterations):

**First Iteration**

```
    EntityA - record 1
    EntityA_Id = 1          ; {0 + 1}
    EntityA.a = a1          ; 70% for a1

    EntityA_EntityB = 2        ; just to keep the example simple

    EntityB - record 1
    EntityB.EntityB_Id = 1      ; {0 + 1}
    EntityB.EntityA_Id = 1   ; {EntityA.EntityA_Id}
    EntityB.b = b2          ; 75% for b2

    EntityB_EntityC = 1        ; 70% for 1

    EntityC - record 1
    EntityC.EntityC_Id = 1      ; {0 + 1}
    EntityC.EntityB_Id = 1   ; {EntityA.EntityB_Id}
    EntityC.c = f(a1,b2)     ; 75% for f(EntityA.a, EntityB.b)}

    EntityB - record 2
    EntityB.EntityB_Id = 2      ; {1 + 1}
    EntityB.EntityA_Id = 1   ; {EntityA.EntityA_Id}
    EntityB.b = b2          ; 75% for b2

    EntityB_EntityC = 0        ; just to keep the example simple

    EntityA_EntityD = 1        ; just to keep the example simple

    EntityD - record 1
    EntityD.EntityD_Id = 1      ; {0 + 1}
    EntityD.EntityA_Id = 1      ; {EntityA.EntityA_Id}
    EntityD.d = (d1 + d2)/2  ; {random(d1,d2)}
```

**Second Iteration**

```
    EntityA - record 2
    EntityA_Id = 2          ; {1 + 1}
    EntityA.a = a1          ; 70% for a1

    EntityA_EntityB = 150
    .
    .
    .
```

The test data generator has two main limitations:

- Its input has to be defined as a hierarchy. Therefore, accommodating many-to-many relationships or one-to-one and one-to-many relationships that are optional on both sides might be tricky. In the experiment, this difficulty was overcome by transforming the test data model into a hierarchy and using appropriate variables and/or constants to cater for relationships that could not be explicitly defined in the data generator's input. For instance, the identifiers of all entities were obtained by concatenating a prefix (defined as a constant) and a number that was increased by one for each new generated record. It was possible, therefore, to infer values for referential attributes without an explicit relationship between two entities.
- Only the most recent generated values for attributes and cardinalities are accessible in expressions. There might be cases, however, when knowing all previously generated values could be useful. For instance, in a one-to-many relationship, an attribute of the "one" entity could represent the sum of the values of an attribute of the "many" entity. Therefore, it would be useful to access all generated records for the "many" entity in order to compute the sum. In general, it is possible to accommodate computational dependencies amongst attributes by using appropriate variables and programming logic. Complex dependencies, however, could clutter the input to the test data generator and this might be a problem even if all previously generated data would be accessible in expressions. In the experiment, therefore, to avoid embedding the scheduler logic into the input to the test data generator this kind of difficulty (i.e., complex computational dependencies amongst attributes) was eliminated by writing an additional program, the*Modifier*. The purpose of the program is described in more detail in *Chapter 3*.

Despite its limitations, the test data generator provided a cost-effective means of generating practically any kind of test data in the experiment: simple or complex, positive or negative, uniform or distributed according to other statistical profiles. Whether such a (simple) tool is suitable for other contexts depends on the complexity of the data models.

# APPENDIX D: TEST RECORDS

## Tests

### Test Data Sets

| Test Data Set | Description |
|---|---|
| 0 | Manually selected test data[1] |
| 1 | Generated simple test data.<br>All assets will be utilised in the near future.<br>For each asset-meter pair, the "current" meter is set to 0 on the commissioning date of the asset.<br>"fixed" indicator = "OFF" |
| 1A | Generated simple test data – similar to Data Set 1.<br>The "current" meters are set to 0 on today's date.<br>For each asset-meter pair, the utilisation rate is set to 0 from today's date until the commissioning date of the asset. |
| 1B | Generated simple test data – similar to Data Set 1A.<br>"duration" indicator = "OFF"<br>"force" indicator = "OFF" |
| 1B.1 | Generated from Data Set 1B and primary program' output (that corresponds to Data Set 1B). |
| 2 | Generated simple test data.<br>Today's date is greater than, but close to, the assets' commissioning dates.<br>For each asset-meter pair, the "current" meter is set to 0 on the commissioning date of the asset.<br>"fixed" indicator = "OFF" |
| 2A | Generated simple test data.<br>Today's date is greater than, but close to, the assets' commissioning dates.<br>For each asset-meter pair, the "current" meter is set to 0 on the commissioning date of the asset.<br>Tasks' first dates are greater than, but close to, the assets' commissioning dates (to avoid Spec-2)<br>"fixed" indicator = "OFF"<br>"force" indicator = "OFF" (to avoid PP-Fail-3)<br>"duration" indicator = "OFF" (to avoid Spec-1) |

[1] The test data was created as part of the manual test design before this experiment started.

## Version Combinations

| Test Data Set | Primary Program (PP) Version | Model Program (MP) Version | Env. Version |
|---|---|---|---|
| 0 | A | A | A |
| 0 | A | B | A |
| 0 | A | C | A |
| 1 | A | A | A |
| 1 | A | B | A |
| 1 | A | C | A |
| 1 | A | C | B |
| 1A | A | A | A |
| 1A | A | B | A |
| 1A | A | C | A |
| 1A | A | C | B |
| 1B | A | B | A |
| 1B | A | C | A |
| 1B | A | C | B |
| 1B.1 | A | B | A |
| 1B.1 | A | C | A |
| 1B.1 | A | C | B |
| 2 | A | C | A |
| 2 | A | C | B |
| 2A | A | C | B |

## Failures and Defects

### Specification Defects

| Defect Id | Versions | Description |
|---|---|---|
| Spec-1 | A | The case when the duration indicator is ON and the task spans over non-work days is non-deterministic. What is the starting point of the next interval? Is the meter utilisation rate considered 0 for the duration of the task? |
| Spec-2 | A | The case when the meter-based scheduled date of the first occurrence is earlier than the first_date is non-deterministic. The spec states: "the occurrence is not scheduled". When is the first occurrence going to be scheduled? Version A of the primary program schedules it on (first_meter + interval), but it would be more logical to schedule it on the first_date. |

## Primary Program Failures and Defects

| Failure Type | Versions | Description | Faults/Defects |
|---|---|---|---|
| ~~PP-Fail-1~~ | ~~A~~ | ~~The target meter value for the first occurrence is (first_meter + interval) instead of first_meter.~~ | Spec-2 |
| PP-Fail-2 | A | The utilisation rate for a factored meter is calculated by multiplying its factor with the utilisation rate of the natural meter. Therefore, a factored meter does not need an utilisation rate. The program, however, raises an error if the factored meter does not have one. | Env-1 |
| PP-Fail-3 | A | The scheduled start date for the first occurrence is (first_date + interval) instead of first_date. This happens when the interval is "daily with holidays" and the force indicator is ON and the first_date is a non-work day. | PP-Fault-1 |
| PP-Fail-4 | A | The program does not terminate (it seems that it hangs while determining the remaining life). | PP-Fault-2 |

| Fault Id | Versions | Description |
|---|---|---|
| PP-Fault-1 | A | To be investigated |
| PP-Fault-2 | A | To be investigated |

## Model Program Failures and Defects

| Failure Type | Versions | | | Description | Faults/Defects |
|---|---|---|---|---|---|
| MP-Fail-1 | A | | | The target meter value for the first occurrence is 0 instead of the first_meter. | MP-Fault-1 |
| MP-Fail-2 | A | B | | The start date of the first occurrence is calculated according to the interval and not to the first_date. | MP-Fault-2 |

| Fault Id | Versions | | | Description |
|---|---|---|---|---|
| MP-Fault-1 | A | | | The utilisation rate of the first utilisation period is incorrectly used for all subsequent utilisation periods. (Corrected in B) |
| MP-Fault-2 | A | B | | The algorithm was based on the incorrect assumption that the first_date is strictly greater than the asset's commissioning date (it did not cater for equality). (Corrected in C) |

## Test Environment Defects

| Defect Id | Versions | | Description |
|---|---|---|---|
| Env-1 | A | | In the input group definition (sch97.sch) table "fact" belongs to the same subgroup as table "mout". The two tables must be in separate subgroups. (Corrected in B) |

## Disagreement Analysis Sheet

The format of the disagreement analysis sheet is described in detail in *Chapter 3*.

| Test Data Set | PP | MP | | | Env. | | Test Id | Task Occ. (machine \| task \| occ. number \| meter) | Occ. Attribute | PP Output | MP Output | Correct Output | PP Failure | MP Failure | Spec. Defect | Remarks |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | A | A | B | C | A | x | 12 | (12\|15\|2) | start_date | 15/02/1996 | 29/02/1996 | | | | | |
| 0 | A | A | B | | A | x | 13 | (13\|16\|0) | start_date | 01/02/1996 | 12/02/1996 | | | | | |
| 0 | A | A | B | | A | x | 21 | (21\|26\|0) | start_date | 01/02/1996 | 21/03/1996 | 01/02/1996 | N | MP-Fail-2 | | |
| 0 | A | A | B | C | A | x | 21 | (21\|26\|1) | start_date | 07/03/1996 | 21/03/1996 | | | | | |
| 0 | A | A | B | | A | x | 24 | (24\|29\|0) | start_date | 01/02/1996 | 09/02/1996 | | | | | |
| 0 | A | A | B | | A | x | 25 | (25\|30\|0) | start_date | 01/02/1996 | 22/02/1996 | | | | | |
| 0 | A | A | B | | A | x | 27 | (27\|32\|0) | start_date | 01/02/1996 | 04/05/1996 | | | | | |
| 0 | A | A | B | C | A | x | 27 | (27\|32\|1) | start_date | 03/05/1996 | 06/05/1996 | | | | | |
| 0 | A | A | B | | A | x | 35 | (35\|40\|0) | start_date | 01/02/1996 | 07/03/1996 | | | | | |
| 0 | A | A | B | | A | x | 36 | (36\|41\|0) | start_date | 01/02/1996 | 06/05/1996 | | | | | |
| 0 | A | A | B | C | A | x | 58 | (58\|64\|1) | occ_no | 1 | 0 | | | | | |
| 0 | A | A | B | C | A | x | 58 | (58\|64\|1) | start_date | 15/02/1996 | 08/02/1996 | | | | | |
| 0 | A | A | B | C | A | x | 59 | (59\|65\|1) | occ_no | 1 | 0 | | | | | |
| 0 | A | A | B | C | A | x | 59 | (59\|65\|1) | start_date | 15/02/1996 | 08/02/1996 | | | | | |
| 0 | A | A | B | C | A | x | 61 | (61\|67\|2) | start_date | 15/02/1996 | 11/03/1996 | | | | | |
| 0 | A | A | B | C | A | x | 62 | (62\|68\|1) | occ_no | 1 | 0 | | | | | |
| 0 | A | A | B | | A | x | 62 | (62\|68\|1) | start_date | 01/02/1996 | 01/03/1996 | | | | | |
| 0 | A | A | B | | A | x | 68 | (68\|158\|0) | start_date | 01/01/1996 | 15/01/1996 | | | | | |
| 0 | A | A | B | | A | x | 69 | (69\|79\|0) | start_date | 01/02/1996 | 08/02/1996 | | | | | |
| 0 | A | A | B | C | A | x | 70 | (70\|80\|1) | occ_no | 1 | 0 | | | | | |
| 0 | A | A | B | | A | x | 70 | (70\|80\|1) | start_date | 01/02/1996 | 08/02/1996 | | | | | |
| 0 | A | A | B | | A | x | 73 | (73\|83\|0) | start_date | 01/02/1996 | 07/03/1996 | | | | | |
| 0 | A | A | B | | A | x | 75 | (75\|88\|0) | start_date | 01/02/1996 | 09/02/1996 | | | | | |
| 0 | A | A | B | C | A | x | 76 | (76\|91\|1) | occ_no | 1 | 0 | | | | | |
| 0 | A | A | B | | A | x | 76 | (76\|91\|1) | start_date | 01/02/1996 | 08/02/1996 | | | | | |
| 0 | A | A | B | | A | x | 79 | (79\|95\|0) | start_date | 01/02/1996 | 06/05/1996 | | | | | |
| 0 | A | A | B | C | A | x | 82 | (82\|99\|2\|km) | meter | 395 | 345.9 | | | | | |
| 0 | A | A | B | C | A | x | 82 | (82\|99\|4\|km) | meter | 952 | 902.9 | | | | | |
| 0 | A | A | B | C | A | x | 82 | (82\|99\|6\|km) | meter | 1377 | 1459.9 | | | | | |
| 0 | A | A | B | C | A | x | 106 | (106\|125\|1) | occ_no | 1 | 2 | | | | | |

| Test Data Set | PP | MP | | | Env. | | Test Id | Task Occ. (machine \| task \| occ. number \| meter) | Occ. Attribute | PP Output | MP Output | Correct Output | PP Failure | MP Failure | Spec. Defect | Remarks |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | A | A | B | C | A | x | 106 | (106\|125\|1) | start_date | 27/03/1997 | 30/03/1997 | | | | | |
| 0 | A | A | B | C | A | x | 116 | (116\|135\|0) | occ_no | 0 | 1 | | | | | |
| 0 | A | A | B | C | A | x | 116 | (116\|135\|0) | start_date | 10/02/1997 | 22/02/1997 | | | | | |
| 0 | A | A | B | C | A | x | 117 | (117\|136\|0) | occ_no | 0 | 1 | | | | | |
| 0 | A | A | B | C | A | x | 117 | (117\|136\|0) | start_date | 11/04/1997 | 11/03/1997 | | | | | |
| 0 | A | A | B | C | A | x | 119 | (119\|138\|0) | occ_no | 0 | 1 | | | | | |
| 0 | A | A | B | C | A | x | 119 | (119\|138\|0) | start_date | 31/01/1997 | 25/01/1997 | | | | | |
| 0 | A | A | B | C | A | x | 122 | (122\|141\|2) | start_date | 21/01/1997 | 19/02/1997 | | | | | |
| 0 | A | A | B | C | A | x | 127 | (127\|146\|0) | occ_no | 0 | 1 | | | | | |
| 0 | A | A | B | C | A | x | 127 | (127\|146\|0) | start_date | 01/04/1996 | 01/02/1996 | | | | | |
| 0 | A | A | B | C | A | x | 131 | (131\|150\|2) | start_date | 04/07/1996 | 03/08/1996 | | | | | |
| 0 | A | A | B | | A | x | 132 | (132\|160-259\|0) | start_date | 01/02/1996 | 08/02/1996 | 01/02/1996 | N | MP-Fail-2 | | |
| 1 | A | A | B | C | A | B | 1000 | (1000\|1000\|0) | start_date | 15/03/2000 | 17/01/2000 | | | | Spec-2 | |
| 1 | A | A | B | C | A | | 1000 | (1000\|1001) | | none | | | PP-Fail-2 | | | |
| 1 | A | x | x | C | x | B | 1000 | (1000\|1001\|0) | start_date | 08/11/1999 | 20/09/1999 | | | | Spec-2 | |
| 1 | A | A | B | C | A | B | 1000 | (1000\|1004\|0) | start_date | 14/06/2000 | 02/11/1999 | 02/11/1999 | PP-Fail-3 | N | | |
| 1 | A | A | B | C | A | B | 1100 | (1100\|1101\|0) | start_date | 12/01/2000 | 07/10/1999 | 07/10/1999 | PP-Fail-3 | N | | |
| 1 | A | A | B | C | A | B | 1100 | (1100\|1103\|0) | start_date | 01/12/1999 | 21/10/1999 | | | | Spec-2 | |
| 1 | A | A | B | C | A | B | 1200 | (1200\|1201\|18) | start_date | 30/05/2007 | 06/06/2007 | | | | Spec-1 | 04/01/2007 + (5*30.5) ~ 05-06/06/2007 DurInd = ON, ForceInd = ON |
| 1 | A | A | B | C | A | B | 1200 | (1200\|1201\|6) | start_date | 17/04/2002 | 24/04/2002 | | | | Spec-1 | 22/11/2001 + (5*30.5) ~ 23-24/04/2002 DurInd = ON, ForceInd = ON |
| 1 | A | A | B | C | A | B | 1300 | (1300\|1301\|3) | start_date | 12/12/2000 | 05/12/2000 | | | | Spec-1 | 18/07/2000 + (20*7) = 05/12/2000 DurInd = ON |
| 1 | A | A | B | C | A | B | 1300 | (1300\|1301\|4) | start_date | 14/05/2001 | 07/05/2001 | | | | Spec-1 | 18/12/2000 + (20*7) = 07/05/2001 DurInd = ON |
| 1 | A | A | B | C | A | B | 1300 | (1300\|1303\|0) | start_date | 18/10/1999 | 13/09/1999 | | | | Spec-2 | |
| 1 | A | A | B | C | A | B | 1300 | (1300\|1303\|2) | start_date | 05/04/2000 | 29/03/2000 | | | | Spec-1 | Meters DurInd = ON, ForceInd = |

| Test Data Set | PP | MP | | | Env. | | Test Id | Task Occ. (machine \| task \| occ. number \| meter) | Occ. Attribute | PP Output | MP Output | Correct Output | PP Failure | MP Failure | Spec. Defect | Remarks |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | ON |
| 1 | A | A | B | C | A | B | 1400 | (1400\|1400\|0) | start_date | 22/03/2000 | 22/12/1999 | 22/12/1999 | PP-Fail-3 | N | | |
| 1 | A | A | B | C | A | B | 1400 | (1400\|1404\|0) | start_date | 05/01/2000 | 11/11/1999 | | | | Spec-2 | |
| 1 | A | A | B | C | A | B | 1400 | (1400\|1404\|1) | start_date | 17/05/2000 | 10/05/2000 | | | | Spec-1 | Meters DurInd = ON, ForceInd = ON |
| 1 | A | A | B | C | A | B | 1500 | (1500\|1503\|0) | start_date | 20/12/1999 | 23/09/1999 | | | | Spec-2 | |
| 1 | A | A | B | C | A | B | 1500 | (1500\|1504\|2) | start_date | 30/08/2000 | 23/08/2000 | | | | Spec-1 | DurInd = ON, ForceInd = ON |
| 1 | A | A | B | C | A | | 1600 | (1600\|1601) | | none | | | PP-Fail-2 | | | |
| 1 | A | x | x | C | x | B | 1600 | (1600\|1601\|0) | start_date | 11/10/1999 | 01/09/1999 | | | | Spec-2 | |
| 1 | A | A | B | C | A | B | 1600 | (1600\|1604\|0) | start_date | 26/01/2000 | 21/10/1999 | 21/10/1999 | PP-Fail-3 | N | | |
| 1 | A | A | B | C | A | B | 1700 | (1700\|1702\|0) | start_date | 10/05/2000 | 20/10/1999 | 20/10/1999 | PP-Fail-3 | N | | |
| 1 | A | A | B | C | A | B | 1700 | (1700\|1703\|0) | start_date | 24/01/2000 | 14/09/1999 | | | | Spec-2 | |
| 1 | A | A | B | C | A | B | 1800 | (1800\|1800\|0) | start_date | 29/03/2000 | 15/12/1999 | 15/12/1999 | PP-Fail-3 | N | | |
| 1 | A | A | B | C | A | B | 1800 | (1800\|1801\|0) | start_date | 28/06/2000 | 25/10/1999 | 25/10/1999 | PP-Fail-3 | N | | |
| 1 | A | A | B | C | A | B | 1800 | (1800\|1802\|5) | start_date | 13/08/2001 | 06/08/2001 | | | | Spec-1 | DurInd = ON |
| 1 | A | A | B | C | A | B | 1800 | (1800\|1802\|8) | start_date | 16/09/2002 | 09/09/2002 | | | | Spec-1 | DurInd = ON |
| 1 | A | A | B | C | A | B | 1800 | (1800\|1803\|0) | start_date | 15/12/1999 | 07/10/1999 | | | | Spec-2 | |
| 1 | A | A | B | C | A | B | 1800 | (1800\|1803\|1) | start_date | 16/02/2000 | 09/02/2000 | | | | Spec-1 | Meters DurInd = ON, ForceInd = ON |
| 1 | A | A | B | C | A | B | 1900 | (1900\|1900\|2) | start_date | 07/08/2000 | 31/07/2000 | | | | Spec-1 | DurInd = ON |
| 1 | A | A | B | C | A | B | 1900 | (1900\|1900\|5) | start_date | 10/12/2001 | 03/12/2001 | | | | Spec-1 | DurInd = ON |
| 1 | A | A | B | C | A | B | 1900 | (1900\|1903\|0) | start_date | 05/07/2000 | 11/11/1999 | 11/11/1999 | PP-Fail-3 | N | | |
| 1 | A | A | B | C | A | | 1900 | (1900\|1904) | | none | | | PP-Fail-2 | | | |
| 1 | A | x | x | C | x | B | 1900 | (1900\|1904\|0) | start_date | 25/10/1999 | 20/09/1999 | | | | Spec-2 | |
| 1A | A | A | B | C | A | | 1000 | (1000\|1003) | | none | | | PP-Fail-2 | | | |
| 1A | A | x | x | C | x | B | 1000 | | | none | | | PP-Fail-4 | | | |
| 1A | A | A | B | C | A | B | 1100 | | | none | | | PP-Fail-4 | | | |
| 1A | A | A | B | C | A | B | 1200 | (1200\|1202\|0) | start_date | 27/03/2000 | 29/11/1999 | 29/11/1999 | PP-Fail-3 | N | | |
| 1A | A | A | | | A | x | 1200 | (1200\|1203\|0) | start_date | 29/11/1999 | 02/09/1999 | | | MP-Fail-1 | | |
| 1A | A | | B | C | A | B | 1200 | (1200\|1203\|0) | start_date | 29/11/1999 | 01/10/1999 | | | | Spec-2 | |
| 1A | A | A | | | A | x | 1300 | (1300\|1303\|0) | start_date | 24/12/1999 | 04/08/1999 | | | MP-Fail-1 | | |
| 1A | A | | B | C | A | B | 1300 | (1300\|1303\|0) | start_date | 24/12/1999 | 03/09/1999 | | | | Spec-2 | |
| 1A | A | A | B | C | A | B | 1400 | | | none | | | PP-Fail-4 | | | |
| 1A | A | A | B | C | A | B | 1500 | | | none | | | PP-Fail-4 | | | |

| Test Data Set | PP | MP | | | Env. | | Test Id | Task Occ. (machine \| task \| occ. number \| meter) | Occ. Attribute | PP Output | MP Output | Correct Output | PP Failure | MP Failure | Spec. Defect | Remarks |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1A | A | A | B | C | A | B | 1600 | (1600\|1602\|0) | start_date | 17/03/2000 | 05/11/1999 | 05/11/1999 | PP-Fail-3 | N | | |
| 1A | A | A | | | A | x | 1600 | (1600\|1603\|0) | start_date | 07/01/2000 | 27/08/1999 | | | MP-Fail-1 | | |
| 1A | A | | B | C | A | B | 1600 | (1600\|1603\|0) | start_date | 07/01/2000 | 19/11/1999 | | | | Spec-2 | |
| 1A | A | A | B | C | A | B | 1700 | (1700\|1701\|0) | start_date | 04/04/2000 | 01/11/1999 | 01/11/1999 | PP-Fail-3 | N | | |
| 1A | A | A | B | C | A | | 1700 | (1700\|1703) | | none | | | PP-Fail-2 | | | |
| 1A | A | x | x | C | x | B | 1700 | (1700\|1703\|0) | start_date | 06/12/1999 | 02/11/1999 | | | | Spec-2 | |
| 1A | A | A | B | C | A | B | 1700 | (1700\|1704\|17) | start_date | 27/07/2006 | 20/07/2006 | | | | Spec-1 | DurInd = ON |
| 1A | A | A | B | C | A | B | 1700 | (1700\|1704\|20) | start_date | 04/10/2007 | 27/09/2007 | | | | Spec-1 | DurInd = ON |
| 1A | A | A | | | A | x | 1800 | (1800\|1803\|0) | start_date | 17/11/1999 | 08/09/1999 | | | MP-Fail-1 | | |
| 1A | A | | B | C | A | B | 1800 | (1800\|1803\|0) | start_date | 17/11/1999 | 05/10/1999 | | | | Spec-2 | |
| 1A | A | A | B | C | A | B | 1900 | (1900\|1900\|0) | start_date | 19/01/2000 | 27/09/1999 | 27/09/1999 | PP-Fail-3 | N | | |
| 1A | A | A | | | A | x | 1900 | (1900\|1904\|0) | start_date | 02/11/1999 | 20/08/1999 | | | MP-Fail-1 | | |
| 1A | A | | B | C | A | B | 1900 | (1900\|1904\|0) | start_date | 02/11/1999 | 14/09/1999 | | | | Spec-2 | |
| 1B | A | x | B | C | A | B | 1000 | (1000\|1003\|0) | start_date | 18/11/1999 | 30/08/1999 | | | | Spec-2 | |
| 1B | A | x | B | C | A | B | 1100 | (1100\|1101\|0) | start_date | 16/11/1999 | 14/09/1999 | | | | Spec-2 | |
| 1B | A | x | B | C | A | B | 1100 | (1100\|1103\|0) | start_date | 30/03/2000 | 08/11/1999 | 08/11/1999 | PP-Fail-3 | N | | |
| 1B | A | x | B | C | A | B | 1200 | (1200\|1200\|0) | start_date | 10/12/1999 | 13/09/1999 | | | | Spec-2 | |
| 1B | A | x | B | C | A | B | 1200 | (1200\|1202\|0) | start_date | 29/05/2000 | 06/12/1999 | 06/12/1999 | PP-Fail-3 | N | | |
| 1B | A | x | B | C | A | B | 1300 | (1300\|1303\|0) | start_date | 06/01/2000 | 14/10/1999 | | | | Spec-2 | |
| 1B | A | x | B | C | A | B | 1300 | (1300\|1304\|0) | start_date | 07/02/2000 | 25/11/1999 | | | | Spec-2 | |
| 1B | A | x | B | C | A | B | 1400 | (1400\|1402\|0) | start_date | 08/02/2000 | 20/09/1999 | 20/09/1999 | PP-Fail-3 | N | | |
| 1B | A | x | B | C | A | B | 1500 | (1500\|1503\|0) | start_date | 04/10/1999 | 09/08/1999 | | | | Spec-2 | |
| 1B | A | x | B | C | A | B | 1600 | (1600\|1603\|0) | start_date | 29/11/1999 | 20/09/1999 | | | | Spec-2 | |
| 1B | A | x | B | C | A | B | 1700 | (1700\|1703\|0) | start_date | 20/09/1999 | 12/08/1999 | | | | Spec-2 | |
| 1B | A | x | B | C | A | B | 1800 | (1800\|1802\|0) | start_date | 18/04/2000 | 11/11/1999 | 11/11/1999 | PP-Fail-3 | N | | |
| 1B | A | x | B | C | A | B | 1800 | (1800\|1803\|0) | start_date | 03/02/2000 | 11/10/1999 | | | | Spec-2 | |
| 1B | A | x | B | C | A | B | 1900 | (1900\|1904\|0) | start_date | 06/03/2000 | 11/10/1999 | | | | Spec-2 | |
| 1B.1 | A | x | B | C | A | B | 1100 | (1100\|1103\|0) | start_date | 30/03/2000 | 08/11/1999 | 08/11/1999 | PP-Fail-3 | N | | |
| 1B.1 | A | x | B | C | A | B | 1200 | (1200\|1202\|0) | start_date | 29/05/2000 | 06/12/1999 | 06/12/1999 | PP-Fail-3 | N | | |
| 1B.1 | A | x | B | C | A | B | 1300 | (1300\|1303\|0) | start_date | 21/02/2000 | 12/10/1999 | | | | Spec-2 | |
| 1B.1 | A | x | B | C | A | B | 1400 | (1400\|1402\|0) | start_date | 08/02/2000 | 20/09/1999 | 20/09/1999 | PP-Fail-3 | N | | |
| 1B.1 | A | x | B | C | A | B | 1800 | (1800\|1802\|0) | start_date | 18/04/2000 | 11/11/1999 | 11/11/1999 | PP-Fail-3 | N | | |
| 1B.1 | A | x | B | C | A | B | 1800 | (1800\|1803\|0) | start_date | 04/01/2000 | 08/10/1999 | | | | Spec-2 | |
| 1B.1 | A | x | B | C | A | B | 1900 | (1900\|1904\|0) | start_date | 11/05/2000 | 11/10/1999 | | | | Spec-2 | |
| 2 | A | x | x | C | A | B | 2000 | (2000\|2001\|0) | start_date | 27/09/1999 | 07/06/1999 | | | | Spec-2 | |
| 2 | A | x | x | C | A | B | 2000 | (2000\|2002\|0) | start_date | 25/01/2000 | 07/06/1999 | 07/06/1999 | PP-Fail-3 | N | | IntvType = 6 |
| 2 | A | x | x | C | A | B | 2000 | (2000\|2003\|0) | start_date | 19/07/1999 | 21/06/1999 | | | | Spec-2 | |
| 2 | A | x | x | C | A | B | 2000 | (2000\|2003\|3) | start_date | 20/03/2000 | 13/03/2000 | | | | Spec-1 | DurInd = ON |
| 2 | A | x | x | C | A | B | 2000 | (2000\|2004\|0) | start_date | 19/11/1999 | 20/07/1999 | 20/07/1999 | PP-Fail-3 | N | | ForceInd = ON, First |

| Test Data Set | PP | MP | | | Env. | | Test Id | Task Occ. (machine \| task \| occ. number \| meter) | Occ. Attribute | PP Output | MP Output | Correct Output | PP Failure | MP Failure | Spec. Defect | Remarks |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | Date = Non-work |
| 2 | A | x | x | C | A | B | 2100 | (2100\|2102\|0) | start_date | 16/11/1999 | 13/07/1999 | 13/07/1999 | PP-Fail-3 | N | | ForceInd = ON, First Date = Non-work |
| 2 | A | x | x | C | A | B | 2100 | (2100\|2102\|1) | start_date | 31/03/2000 | 24/03/2000 | | | | Spec-1 | DurInd = ON, ForceInd = ON |
| 2 | A | x | x | C | A | B | 2200 | (2200\|2200\|0) | start_date | 08/02/2000 | 09/07/1999 | 09/07/1999 | PP-Fail-3 | N | | IntvType = 6 |
| 2 | A | x | x | C | A | B | 2200 | (2200\|2200\|1) | start_date | 19/09/2000 | 12/09/2000 | | | | Spec-1 | DurInd = ON, ForceInd = ON IntvType = 6 |
| 2 | A | x | x | C | A | B | 2200 | (2200\|2201\|0) | start_date | 23/11/1999 | 23/07/1999 | 23/07/1999 | PP-Fail-3 | N | | ForceInd = ON, First Date = Non-work |
| 2 | A | x | x | C | A | B | 2200 | (2200\|2202\|5) | start_date | 27/08/2004 | 20/08/2004 | | | | Spec-1 | DurInd = ON |
| 2 | A | x | x | C | A | B | 2300 | (2300\|2301\|0) | start_date | 17/03/2000 | 03/08/1999 | 03/08/1999 | PP-Fail-3 | N | | IntvType = 6 |
| 2 | A | x | x | C | A | B | 2400 | (2400\|2401\|2) | start_date | 24/04/2000 | 17/04/2000 | | | | Spec-1 | DurInd = ON |
| 2 | A | x | x | C | A | B | 2400 | (2400\|2401\|5) | start_date | 09/04/2001 | 02/04/2001 | | | | Spec-1 | DurInd = ON |
| 2 | A | x | x | C | A | B | 2400 | (2400\|2402\|0) | start_date | 05/11/1999 | 18/06/1999 | 18/06/1999 | PP-Fail-3 | N | | ForceInd = ON, First Date = Non-work |
| 2 | A | x | x | C | A | B | 2400 | (2400\|2403\|0) | start_date | 25/10/1999 | 14/06/1999 | | | | Spec-2 | |
| 2 | A | x | x | C | A | B | 2500 | (2500\|2501\|0) | start_date | 29/02/2000 | 09/07/1999 | 09/07/1999 | PP-Fail-3 | N | | IntvType = 6 |
| 2 | A | x | x | C | A | B | 2500 | (2500\|2502\|1) | start_date | 15/11/1999 | 08/11/1999 | | | | Spec-1 | DurInd = ON |
| 2 | A | x | x | C | A | B | 2500 | (2500\|2502\|2) | start_date | 13/03/2000 | 06/03/2000 | | | | Spec-1 | DurInd = ON |
| 2 | A | x | x | C | A | B | 2500 | (2500\|2504\|0) | start_date | 01/02/2000 | 03/09/1999 | 03/09/1999 | PP-Fail-3 | N | | ForceInd = ON, First Date = Non-work |
| 2 | A | x | x | C | A | B | 2600 | (2600\|2600\|0) | start_date | 23/06/2000 | 20/09/1999 | 20/09/1999 | PP-Fail-3 | N | | IntvType = 6 |
| 2 | A | x | x | C | A | B | 2600 | (2600\|2600\|3) | start_date | 08/11/2002 | 01/11/2002 | | | | Spec-1 | DurInd = ON IntvType = 6 |
| 2 | A | x | x | C | A | B | 2600 | (2600\|2600\|6) | start_date | 25/03/2005 | 18/03/2005 | | | | Spec-1 | DurInd = ON IntvType = 6 |
| 2 | A | x | x | C | A | B | 2600 | (2600\|2603\|0) | start_date | 05/11/1999 | 29/06/1999 | 29/06/1999 | PP-Fail-3 | N | | ForceInd = ON, First Date = Non- |

| Test Data Set | PP | MP | | | Env. | | Test Id | Task Occ. (machine \| task \| occ. number \| meter) | Occ. Attribute | PP Output | MP Output | Correct Output | PP Failure | MP Failure | Spec. Defect | Remarks |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | work |
| 2 | A | x | x | C | A | B | 2700 | (2700\|2700\|0) | start_date | 28/03/2000 | 29/06/1999 | 29/06/1999 | PP-Fail-3 | N | | IntvType = 6 |
| 2 | A | x | x | C | A | B | 2700 | (2700\|2704\|1) | start_date | 12/11/1999 | 05/11/1999 | | | | Spec-1 | DurInd = ON |
| 2 | A | x | x | C | A | B | 2700 | (2700\|2704\|2) | start_date | 03/03/2000 | 25/02/2000 | | | | Spec-1 | DurInd = ON |
| 2 | A | x | x | C | A | B | 2800 | (2800\|2800\|0) | start_date | 23/11/1999 | 16/07/1999 | 16/07/1999 | PP-Fail-3 | N | | ForceInd = ON, First Date = Non-work |
| 2 | A | x | x | C | A | B | 2800 | (2800\|2801\|0) | start_date | 23/08/1999 | 21/06/1999 | | | | Spec-2 | |
| 2 | A | x | x | C | A | B | 2800 | (2800\|2802\|0) | start_date | 11/04/2000 | 27/08/1999 | 27/08/1999 | PP-Fail-3 | N | | IntvType = 6 |
| 2 | A | x | x | C | A | B | 2800 | (2800\|2804\|0) | start_date | 14/07/2000 | 16/07/1999 | 16/07/1999 | PP-Fail-3 | N | | ForceInd = ON, First Date = Non-work |
| 2 | A | x | x | C | A | B | 2800 | (2800\|2804\|1) | start_date | 27/07/2001 | 20/07/2001 | | | | Spec-1 | DurInd = ON, ForceInd = ON |
| 2 | A | x | x | C | A | B | 2900 | (2900\|2900\|2) | start_date | 03/07/2000 | 26/06/2000 | | | | Spec-1 | DurInd = ON |
| 2 | A | x | x | C | A | B | 2900 | (2900\|2902\|0) | start_date | 07/03/2000 | 02/07/1999 | 02/07/1999 | PP-Fail-3 | N | | ForceInd = ON, First Date = Non-work IntvType = 6 |
| 2 | A | x | x | C | A | B | 2900 | (2900\|2904\|0) | start_date | 31/08/1999 | 07/06/1999 | | | | Spec-2 | |
| 2 | A | x | x | C | A | | 2900 | (2900\|2904\|1) | start_date | 17/12/1999 | 01/11/1999 | | PP-Fail-2 | | | |
| 2 | A | x | x | C | A | | 2900 | (2900\|2904\|2) | start_date | 03/04/2000 | 18/02/2000 | | PP-Fail-2 | | | |
| 2A | A | x | x | C | x | B | 2600 | (2600\|2600\|0) | start_date | 21/10/1999 | 31/05/1999 | 31/05/1999 | PP-Fail-3 | N | | IntvType = 6 |
| 2A | A | x | x | C | x | B | 2600 | (2600\|2604\|0) | start_date | 09/09/1999 | 17/05/1999 | 17/05/1999 | PP-Fail-3 | N | | IntvType = 6 |
| 2A | A | x | x | C | x | B | 2700 | (2700\|2700\|0) | start_date | 16/09/1999 | 24/05/1999 | 24/05/1999 | PP-Fail-3 | N | | IntvType = 6 |

# REFERENCES

## Publications

Beizer B, *Black-Box Testing: Techniques for Functional Testing of Software and Systems*, John Wiley & Sons, New York, 1995.

Beizer B, The Black Box Vampire or Testing out of the Box, *Presentations of the 15th International Conference and Exposition on Testing Computer Software (ICTCS)*, Washington D.C., USA, 1.7.1-1.7.11, June 8-12 1998.

Bishop PG, Esp DG, Barnes M, Humphreys P, Dahll G, and Lahti J, PODS - A Project on Diverse Software, *IEEE Transactions on Software Engineering*, **SE-12**(9), 929-940, September 1986.

Buettner DJ and Hayes CK, A Software-Development Risk-Mitigation Technique for Algorithmically Complicated Software, *Proceeding of the 15th International Conference and Exposition on Testing Computer Software (ICTCS)*, Washington D.C., USA, 21-24, June 8-12 1998.

Coplien J, Hoffman D, and Weiss D, Commonality and Variability in Software Engineering, *IEEE Software*, **15**(6), 37-45, November/December 1998.

Eickelmann NS and Richardson DJ, An Evaluation of Software Test Environment Architectures, *Proceedings of the 18th International Conference on Software Engineering (ICSE)*, 353-364, 1996.

Frankl P, Hamlet D, Littlewood B, and Strigini L, Choosing a Testing Method to Deliver Reliability, *Proceedings of the 1997 International Conference on Software Engineering (ICSE)*, 68-78, 1997.

Gordon VS and Bieman JM, Rapid Prototyping: Lessons Learned, *IEEE Software*, **12**(1), 85-95, January 1995.

Hamlet D, Foundations of Software Testing: Dependability Theory, *ACM SIGSOFT Software Engineering Notes (Proceedings of the second ACM SIGSOFT symposium on Foundations of software engineering)*, **19**(5), 128-139, December 1994.

Hamlet D, Predicting Dependability by Testing, *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA)*, 84-91, 1996.

Hatton L, and Roberts A, How Accurate Is Scientific Software?, *IEEE Transactions on Software Engineering*, **20**(10), 785-797, October 1994.

Hatton L, N-Version Design Versus One Good Version, *IEEE Software*, **14**(6), 71-76, November/December 1997.

Kaner C, Falk J, and Nguyen HQ, *Testing Computer Software*, International Thomson Computer Press, 1996.

Knight JC, and Leveson NG, An Experimental Evaluation of the Assumption of Independence in Multiversion Programming, *IEEE Transactions on Software Engineering*, **SE-12**(1), 96-109, January 1986.

Musa JD, Software-Reliability-Engineered Testing, *Computer*, **29**(11), 61-68, November 1996.

Osterweil L et al, Strategic Directions in Software Quality, *ACM Computing Surveys*, **28**(4), 738-750, December 1996.

Peters KD and Parnas DL, Generating a Test Oracle from Program Documentation: work in progress, *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, 58-65, 1994.

Peters KD and Parnas DL, Using Test Oracles Generated from Program Documentation, *IEEE Transactions on Software Engineering*, **24**(3), 161-173, March 1998.

Pfleeger SL and Hatton L, Investigating the Influence of Formal Methods, *Computer*, **30**(2), 33-43, February 1997.

Richardson DJ, Aha SL, and O'Malley TO, Specification-based Test Oracles for Reactive Systems, *Proceedings of the 14th International Conference on Software Engineering (ICSE)*, 105-118, 1992.

Richardson DJ, TAOS: Testing with Analysis and Oracle Support, *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, 138-153, 1994.

Somani AK and Vaidya NH, Understanding Fault Tolerance and Reliability, *Computer*, **30**(4), 45-50, April 1997.

Sommerville I, *Software Engineering*, Addison-Wesley Publishing Company, USA, 4th edition, 1992.

Van Vliet H, *Software Engineering: Principles and Practice*, John Wiley & Sons, 1996.

Weyuker EJ, Experience Testing Very Large Software Systems, *Proceeding of the 15th International Conference and Exposition on Testing Computer Software (ICTCS)*, Washington D.C., USA, 127-133, June 8-12 1998.

Weyuker EJ, Using Failure Cost Information for Testing and Reliability Assessment, *ACM Transactions on Software Engineering and Methodology*, **5**(2), 87-98, April 1996.

Zhu H, Hall PAV, and May JHR, Software Unit Test Coverage and Adequacy, *ACM Computing Surveys*, **29**(4), 366-427, December 1997.

## WWW Resources

1. Formal Methods, http://archive.comlab.ox.ac.uk/formal-methods.html
2. SMALL: Shlaer-Mellor Action Language (one L is gratuitous), http://www.projtech.com/info/small.html
3. Data Generation Language (DGL), http://www.csee.usf.edu/~maurer/dgl.html
4. DataShark, http://www.hardballsw.com/
5. Datatect™, http://www.datatect.com/