SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2001; **31**:1211–1236 (DOI: 10.1002/spe.409)

Software testing using model programs

L. I. Manolache and D. G. Kourie*,†

Department of Computer Science, University of Pretoria, Pretoria 0002, South Africa

SUMMARY

A strategy described as 'testing using M model programs' (abbreviated to 'M-mp testing') is investigated as a practical alternative to software testing based on manual outcome prediction. A model program implements suitably selected parts of the functional specification of the software to be tested. The M-mp testing strategy requires that M ($M \geq 1$) model programs as well as the program under test, P, should be independently developed. P and the M model programs are then subjected to the same test data. Difference analysis is conducted on the outputs and appropriate corrective action is taken. P and the M model programs jointly constitute an approximate test oracle. Both M-mp testing and manual outcome prediction are subject to the possibility of correlated failure. In general, the suitability of M-mp testing in a given context will depend on whether building and maintaining model programs is likely to be more cost effective than manually pre-calculating P's expected outcomes for given test data. In many contexts, M-mp testing could also facilitate the attainment of higher test adequacy levels than would be possible with manual outcome prediction.

A rigorous experiment in an industrial context is described in which M-mp testing (with $M = 1$) was used to test algorithmically complex scheduling software. In this case, M-mp testing turned out to be significantly more cost effective than testing based on manual outcome prediction. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: test oracle; N-version programming; back-to-back testing; comparison testing; M-mp testing

INTRODUCTION

One of the most difficult tasks in software testing is to assess the correctness of the outcomes of a program that is subjected to particular test inputs. The problem of establishing an appropriate mechanism to adjudicate on whether or not an outcome is correct, somewhat neglected in testing theory, is referred to as 'the oracle problem' [1,2]. Hamlet [1] points out that: 'Testing theory, being concerned with the choice of tests and testing methods, usually ignores the oracle problem. It is typically assumed

*Correspondence to: D. G. Kourie, Department of Computer Science, University of Pretoria, Pretoria 0002, South Africa.
†E-mail: dkourie@cs.up.ac.za

Received 19 June 2000
Revised 5 February 2001 and 25 June 2001
Accepted 25 June 2001

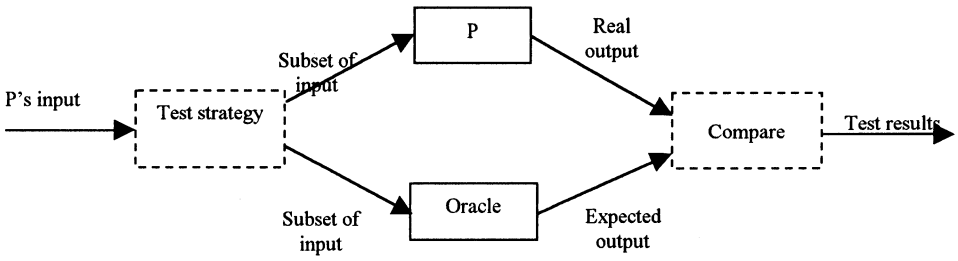

Figure 1. Global view of the test process.

that an oracle exists, and the theoretician then glibly talks about success and failure, while in practice there is no oracle but imperfect human judgment.'

A test oracle is a means of determining whether a program passed or failed a test[‡]. The test process can be depicted as in Figure 1 [3]. P denotes the program to be tested and a test strategy determines the set of test data to be used. In order to obtain a test result for a given test in this set, P's output has to be compared to the oracle's output.

The simplest conceptual form of an oracle is the comparison of actual and expected program outcomes. For a small number of simple test inputs, it might be feasible to work out the expected outputs manually. However, thorough testing in a practical context usually requires rather complex and large test-datasets. Therefore, as emphasized in the testing literature, manual outcome prediction might be time consuming and error prone (e.g. Beizer [4], Richardson *et al*. [5], Peters and Parnas [6,7]).

The research community has addressed the oracle problem especially in relation to formal methods. Stocks and Carrington [8] discuss the generation of oracle templates from formal specifications. Richardson *et al*. [5] propose an approach to deriving and using specification-based oracles in the testing process. Peters and Parnas [6,7] 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 [9]. 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. Both Weyuker [10] and Beizer [4] suggest several practical alternatives. One of them is to build a *pseudo-oracle* in the form of a detailed prototype or a dual program that is capable of providing the expected outputs. The approach examined here is similar and is based on N-version diversity, a principle that is applied especially in the well-known software fault-tolerant technique known as N-version programming. To highlight its similarity with N-version programming, the approach advocated below is called 'testing using M model programs' or, in short, 'M-mp testing'.

The next section surveys research into N-version diverse systems that has inspired and guided the proposal for M-mp testing. It is followed by a section outlining theoretical issues relating to

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

M-mp testing. Next described is the execution of an industrial experiment to investigate whether M-mp testing is likely to be more cost effective than testing based on manual outcome prediction. Thereafter, evidence resulting from the experiment is presented and interpreted. The penultimate section extrapolates more general conclusions and the final section overviews related research. A full account of the present research is available in Manolache [11].

N-VERSION DIVERSE SYSTEMS

An N-version (or multiversion) diverse system comprises N independently written versions of the software, all conforming to the same functional specification. At run-time, voting based on majority agreement is used to decide on a single, probable correct, output. An obvious disadvantage of an N-version system is the increased development cost, at least three versions being required for a voting system to work.

N-version experiments are fundamentally related to the work presented here. As a result, it is relevant to mention research pointing to various other advantages and disadvantages of N-version systems. Rather than providing a complete survey, only work that has been the most influential in the present study will be mentioned.

A central concern in N-version systems is the fact that correlated failures may limit the practical gain in reliability. Such failures, also known as 'common-mode failures' or 'coincident errors' are said to occur when versions erroneously agree on outputs. Eckhardt and Lee [12] provide a probabilistic model that, *inter alia*, indicates sufficient conditions under which a multi-version system may be expected to be more reliable than a single one. In addition, their analysis shows that should errors in the various versions occur in some dependent or correlated fashion, then more versions are required to attain the same level of reliability than is necessary when errors occur randomly. In a landmark experimental study, Knight and Leveson [13] demonstrated 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 merely shows that the reliability of an N-version system may not be as high as theory predicts under the assumption of independence. Further analysis of their data indicates that programmers are likely to make equivalent logical errors and that various regions of the input space may be prone to result in common mode failures [14].

Brilliant *et al*. [15] caution against another concern in N-version systems, the so-called consistent comparison problem, which may arise when decisions are based on comparisons involving finite-precision calculations. Because results could depend on such factors as the order of computation, the particular algorithms used, the hardware etc. consistency cannot be guaranteed, even though the various versions may be correct.

These disadvantages caution that N-version systems should not be casually assumed to be completely reliable. However, despite these limitations, Hatton [16] concludes that N-version design 'appears to offer more reliability than we can gain any other way.' Where the cost of failure is high, it would be more cost effective to build an N-version system rather than focusing on building 'one good version'. His argumentation includes an admittedly 'simplified and highly averaged' analysis of the data provided by the Knight and Leveson [13] study, suggesting that a three version system governed by majority voting would result in correlated failures on average in approximately eight out of a million trials. This is estimated to be about a 45-fold improvement over an 'average' 1-version system.

Although N-version systems are typically deployed in the context of fault-tolerant computing, they have also been used (directly or indirectly) as approximate test oracles. Regression testing, widely used in industry, is an indirect form of 2-version testing whereby a new version of the software is tested against an older version on the basis of stored input and results. The two versions can be said to jointly constitute an approximate test oracle, and the newer version is the primary software to be tested. Similarly, during operational use of a fault-tolerant N-version system, testing could occur as a by-product. Disagreements could be logged and later used to improve the reliability of individual versions. Again, the set of N-versions jointly constitute an approximate test oracle, while each version is also a program under test.

Direct N-version testing experiments include the study of Hatton and Roberts [17]. They 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. Bishop *et al*. [19] also researched the merits of testing using diverse software. Back-to-back testing successfully detected all residual seeded faults in their programs. It was noted that two faults out of seven were common to two programs and would have caused a 3-version majority-voting system to fail. In the testing context, however, the faults were detected because the third program disagreed with the erroneous outputs. The authors also report that a large number of tests could be carried out in an economic manner.

THEORETICAL FOUNDATIONS OF M-mp TESTING

This section indicates how M-mp testing is related to N-version diversity. It also argues from first principles the advantages of M-mp testing over manual outcome prediction. Finally, it offers a partitioning of the input domain, and gives reasons why certain partitions need not be catered for explicitly by a model program.

A framework for M-mp testing

The research mentioned in the foregoing section raises the question: Could a testing strategy based on aspects of N-version diverse systems be devised as a cost effective alternative to the routinely used manual outcome prediction testing strategy? Figure 2 modifies Figure 1, depicting the framework for the variant of N-version testing that is proposed here as a theoretical answer to this question. P is the primary program (the program under test) and mp_1-mp_M are M so-called 'model' programs of P. M-mp testing entails subjecting P and mp_1-mp_M to the same test inputs. Any disagreement on the output 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 dataset are resolved.

The following distinguishes the framework in Figure 2 from N-version majority-voting systems as well as from testing P based on manual outcome prediction.

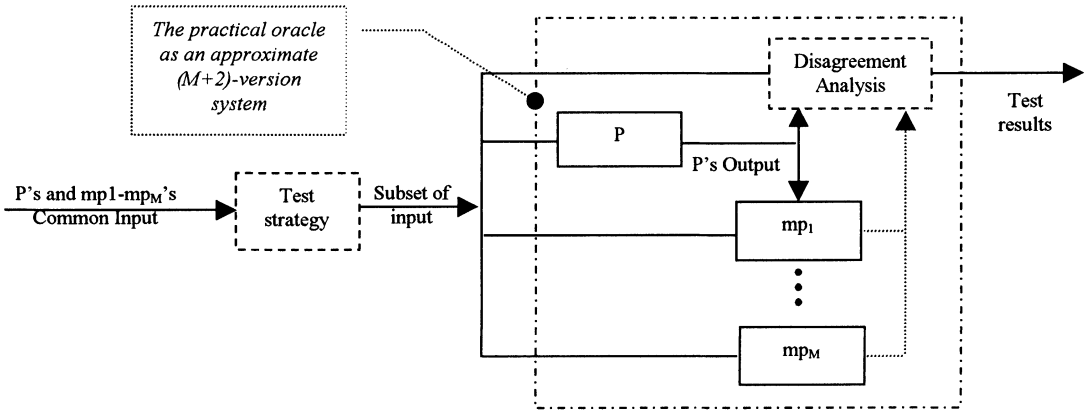

Figure 2. M-mp testing framework.

- In M-mp testing, a model program may be an abstraction of P in the sense that its properties (relevant to the task at hand) constitute a subset of P's properties. Generally, P has properties in addition to those of its model. (Thus, a model is a version of P only in the limiting case where P and the model have exactly the same set of properties.) P may be thought of as a refinement of a model in relation to its additional properties that logically follow from the properties of the model, and as an enrichment in relation to its additional properties that do not logically follow from the properties of the model. Precise definitions of these terms may be found in Kourie [18].
- From a testing perspective, for reasons of economy, the M abstractions of P, mp_1-mp_M, are designed to cover only those complex and/or critical areas of P's behaviour that would normally require a lot of human assistance for correctness checking.
- In N-version programming, each version independently and explicitly computes its rendition of the output. In contrast, Figure 2 leaves open the possibility that the output computed by P may serve as input to mp_1-mp_M, which then subsequently deliver their individual verdicts as to whether the P's output was correct or not. For instance, mp_1-mp_M could implement a set of post-conditions that are strong enough to indicate whether P has failed or not[§]. Note that, in such cases, the models mp_1-mp_M are not abstractions of P.
- Arbitrating disagreements essentially entails computing (manually) the correct outputs independently of any of the programs. In this loose sense, then, disagreement analysis can be considered as an additional 'model' that is invoked on an *ad hoc* basis. Thus, P, mp_1-mp_M

[§]In the present context, any software that does post-condition checking may be regarded as a model. Such a model need not be limited to software separate from P. In an attenuated sense, a set of post-condition checking routines and/or assertions embedded within P could also be seen, collectively, as a model of P. However, from a testing point of view there is something of a dilemma here, since to retain these routines during live operation is to incur a performance penalty, while to turn them off is to run a version of P that has been modified from the version that was tested—albeit in a seemingly innocuous fashion.

and the disagreement analysis 'model' form an approximate $(M + 2)$-model system that jointly constitute a reasonably accurate and automated oracle.

- The M-mp framework emphasizes that instead of attempting to attain a reliable primary program, P, by testing based on manual outcome prediction, it might be more cost effective to write M less reliable model programs. As the number of test cases increases, the quality of both the M model programs and of P is likely to increase [16]. Thus, the $(M+2)$ model system becomes a continuously perfecting test oracle.
- M-mp testing ends when *all* programs agree in output for *each item* in the given set of test data. Any remaining correlated failure is therefore a failure in each of the $M + 1$ programs. This is somewhat stricter than N-version systems in which mere *majority* agreement is required at each stage for progress to the next processing cycle. Nevertheless, the possibility of correlated failures remains a perennial encumbrance of any system based on N-version diversity. It should be noted, however, that the alternative—a manual outcome predication strategy—is itself not immune from correlated failures: a tester who works out expected results manually might well make the same wrong assumptions as the programmer.

The suitability of the M-mp approach for a particular context thus primarily depends on whether it is more viable in terms of cost and time to independently build and maintain model programs than to pre-calculate expected outcomes manually.

Asymptotic behaviour of M-mp testing

An important difference between the M-mp and manual approaches to testing is the cost of outcome verification; each approach incurs an initial cost of writing the model programs or manually computing expected outcomes, respectively. During testing, in both cases, the outputs are considered correct when they agree, so no cost is incurred; if outputs differ, a cost for the disagreement analysis is incurred. If the cause is sought in a suspected defect in the primary program, then the costs would be the same for the two approaches. However, if a model program is suspected to be faulty, it has to be analysed and possibly corrected at a certain cost. In the manual approach, the cost of recalculating the expected outputs is incurred.

Figure 3 suggests a model of how the cost of outcome verification in the two respective cases may vary on average as the number of tests increases. The *cost of M-mp outcome verification*, although initially high because of the *cost of model program development*, may be expected to increase only slightly. For simplicity, the cost growth is modelled as linear. The non-zero slope of the graph reflects the disagreement analysis and model-program maintenance effort. Of course, the actual rate of increase depends on how reliable and maintainable the model programs are. However, the reliability of a model program is likely to increase rapidly along with that of the primary program, and high maintainability can be promoted by simple model program design and implementation.

In the manual approach, the growth of the outcome verification cost may be 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 each test. For simplicity, the growth of the *cost of manual outcome verification* is also modelled as linear in Figure 3.

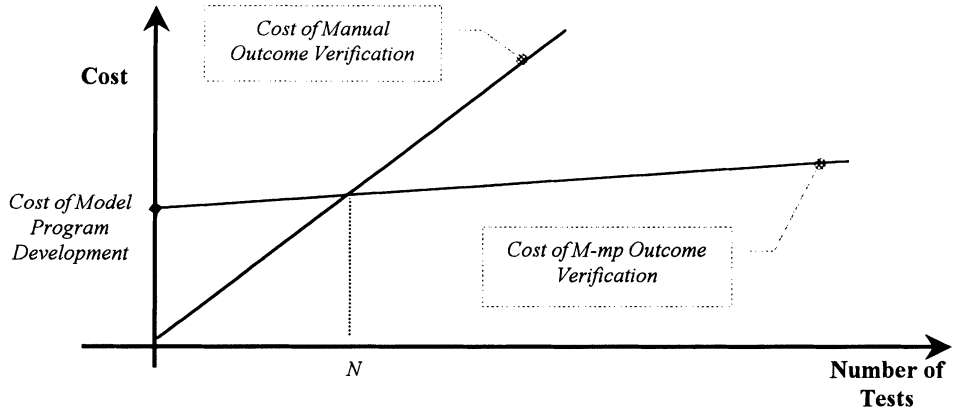

Figure 3. Predicted manual against M-mp outcome verification costs.

In Figure 3, each point on the horizontal axis refers to the number of tests used to test some hypothetical program using either M-mp testing or the manual approach. Assume a point refers to a common set of test data that is used for both approaches. Such a set of test data (or, equivalently, a test strategy) is characterized by its adequacy in reference to a stated adequacy criterion. A large variety of such adequacy criteria have been proposed and studied. The percentage of elements in the test dataset that meet the criterion when applied to the program under test, is the test set's or test strategy's adequacy level with respect to that criterion. A comprehensive survey on these matters is provided by Zhu *et al.* [2].

Thus, each point on the horizontal axis in Figure 3 can be associated with a certain adequacy level with respect to some prespecified criterion (e.g. branch coverage). Note, however, that while the number of tests change linearly along the axis, this need not apply to the adequacy level—e.g. doubling the number of tests does not necessarily result in a doubling of the adequacy level. Figure 3 suggests that M-mp testing would be more cost effective in attaining the associated adequacy level than the manual approach once the number of tests is greater than N. The comparison model, however, does not emphasize sufficiently a major advantage of M-mp testing: the possibility of achieving the same (or higher) adequacy levels at lower cost of data selection.

Intuitively, the size of a test dataset that achieves a particular adequacy level with respect to some criterion is determined by the precision with which test data is selected to meet that criterion. Suppose, for example, that we require a set of test data that is at an 80% level with respect to the 'branch coverage' criterion. Clearly, there will be a minimum size test dataset that achieves this adequacy level. (Each test in such a set must exercise at least one branch in the code that is not exercised by any other test in the same way and, in total, 80% of all paths exiting from branches in the code must be exercised.) In a manual approach the test data-selection process should aim for such a minimum set in order to minimize the effort and expense involved. This is especially relevant if high adequacy levels are to be attained and/or the input space is large. In such cases, the selection procedure tends to become complex

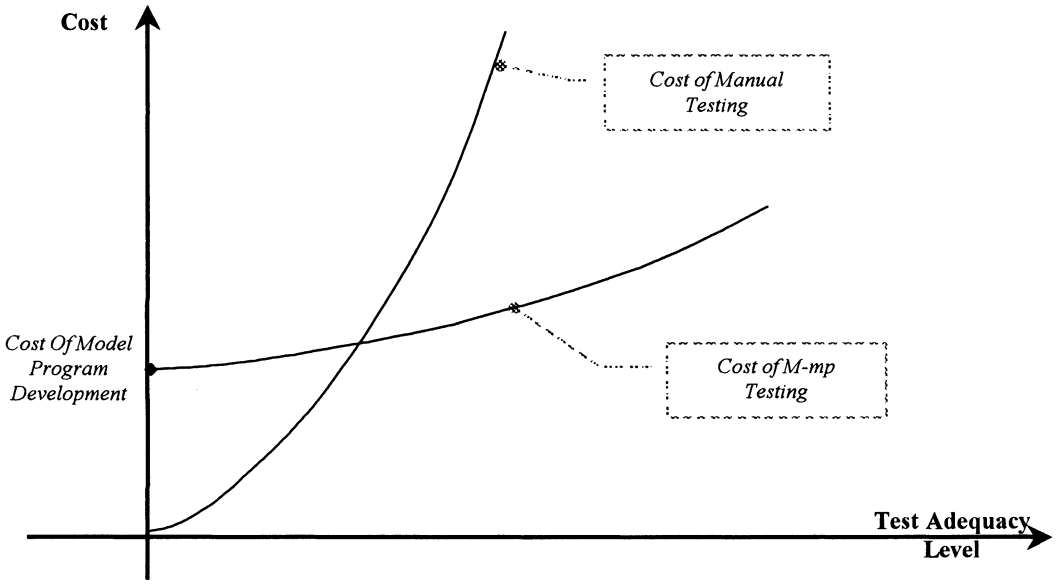

Figure 4. Predicted manual against M-mp testing costs.

and, therefore, expensive. On the other hand, if it is feasible to carry out large numbers of tests cheaply, as is expected in the M-mp testing case, then the accuracy required of the test data selection procedure can be relaxed, thus reducing the cost.

In other words, M-mp testing can attain a given test adequacy level more cheaply than the manual approach. It simply relies on larger datasets, generated less expensively than would be appropriate for the manual approach. A more suitable comparison model than Figure 3, which takes into account the cost of test data selection, is depicted in Figure 4. The *cost of manual testing* and *cost of M-mp testing* plots have superlinear shapes to reflect the fact that, in order to attain higher adequacy levels, not only do we need to select a larger number of tests, but we also need to expend an increasing amount of effort (and thus cost) to ensure that the data selected will indeed attain the hoped-for adequacy level.

Input domain partitioning

The input domain of a program may be divided into a standard domain and an exception domain [3]. In turn, the exception domain may be split into an incomplete-data domain and an invalid-data domain, as depicted in Figure 5. 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.

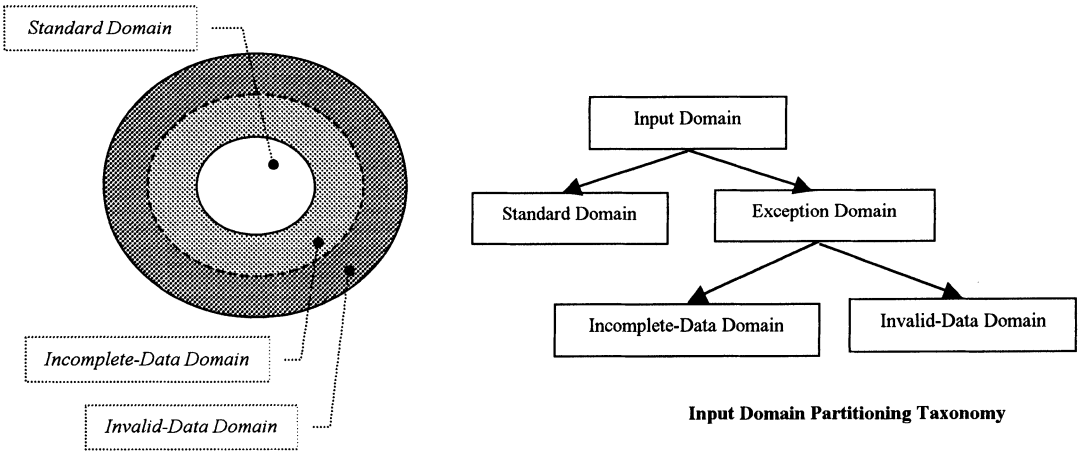

Figure 5. Input domain partitioning into standard and exception domains.

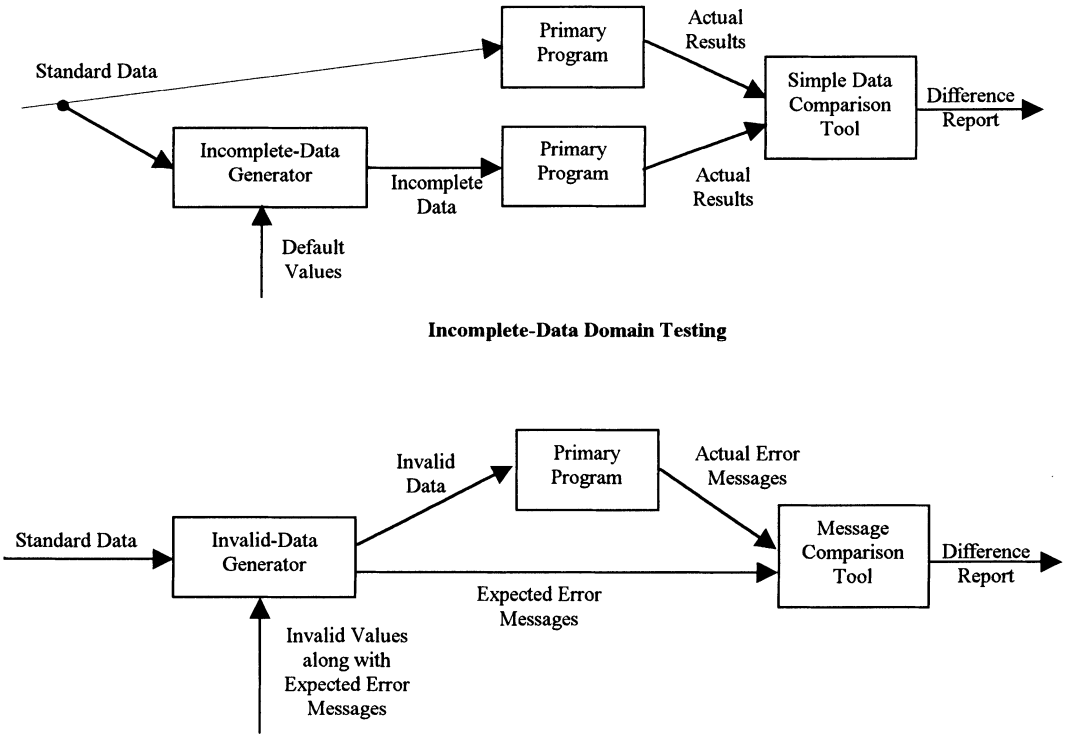

Figure 6. Envisaged exception-domain testing approach.

As a general rule, a model program should be designed to handle only standard input data of the primary program. Verifying whether a program handles non-standard data correctly can be automated efficiently by other means. For instance, Figure 6 suggests that 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 running the primary program on both datasets. 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 (see, for example, Bishop *et al*. [19] and Cohen *et al*. [20]). If the program handles invalid data correctly, it should terminate without producing results and the *actual error messages* should be the same as the *expected error messages*.

Adopting such an approach for exception-domain testing in the M-mp context allows one to reduce the complexity of the model programs which then no longer require exception-handling capability.

AN M-mp EXPERIMENT

Background and motivation

The M-mp experiment described below took place in an IT industrial setting. The primary program is a scheduling application, a non-interactive data-processing subsystem of a large logistics management system (EPMS®). The various such subsystems deal with complex input–output (I/O) data structures, their input spaces are fairly large (20 to 100 dimensions) and they solve complex operations-management problems (e.g. maintenance-task scheduling, resource balancing, inventory management).

Thus, in broad terms, the programs can be characterized as algorithmically complex. In practice, the testing process in the company was well defined and supported by standards and procedures, but had several drawbacks. Designing test cases based on manual outcome prediction was not only difficult, error-prone and time-consuming, but consequently also unattractive and demoralizing. There was a tendency to limit the number and complexity of test inputs. The adequacy criterion used, '65% branch coverage', seemed insufficient in relation to the complexity of the programs. Test-specification maintenance was expensive, since small specification changes could require the manual redesign of many tests. Finally, to economize on documentation, test specifications did not provide a description of steps followed to determine the expected results. Therefore, verifying a test case could be as time-consuming as creating it.

All the above deficiencies of manual test design were well known. Yet no alternatives were initially considered, mainly because of budget limitations and lack of hard evidence that other approaches could be cost effective. An incidental *ad hoc* but successful pilot exercise in M-mp testing within the company, conducted to meet a software-delivery crisis [11], motivated a more structured and focused experiment. The objective was to provide evidence that M-mp testing, in the particular context, could achieve higher adequacy levels, at lower cost, than the manual approach.

Selecting the application

Instead of extending the pilot experiment, the scheduling application was chosen as the primary program, for the following reasons.

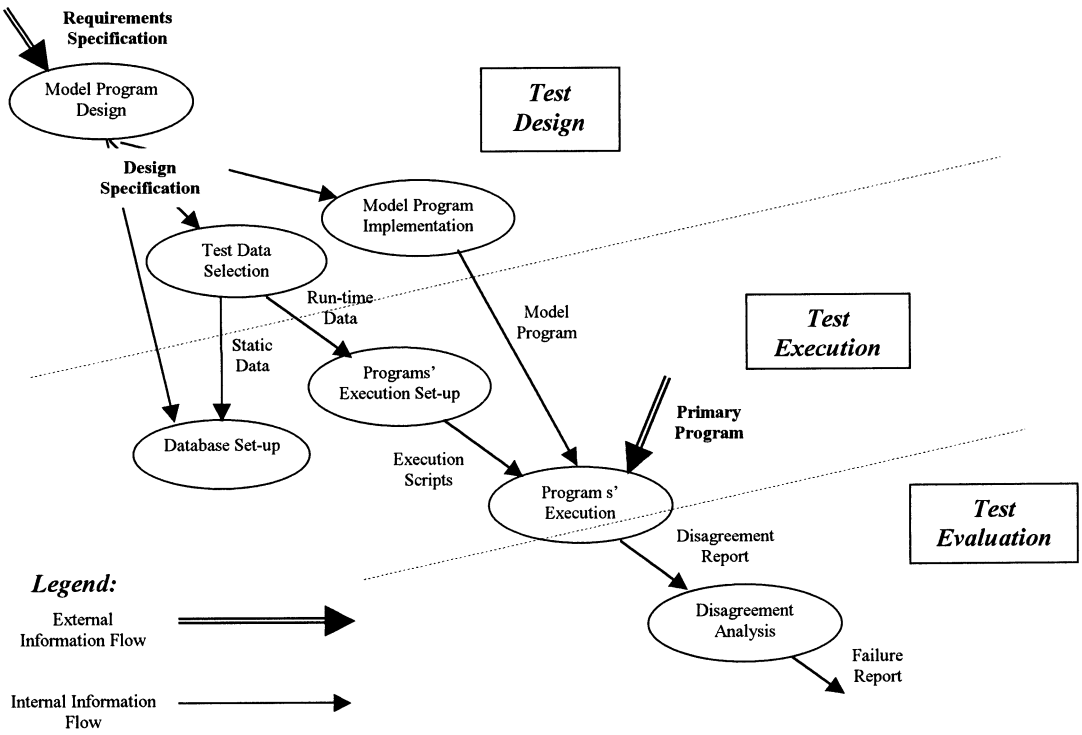

Figure 7. The experiment's process.

- Since the developer was not familiar with the scheduling domain and application, the cost of 'developing from scratch' could be assessed more authentically.
- On the same grounds, the risk of correlated failures caused by common design errors was reduced.
- The scheduling application is representative of the family of operations-management programs.
- By the time the experiment started, the scheduling subsystem had been in use for more than a year. This seemed a good opportunity to test the fault-detection ability of M-mp testing, as the likelihood of finding defects was expected to be low.

Phases of the experiment

Essentially, the experiment entailed a *test design*, a *test execution* and *test evaluation* phase as depicted in Figure 7. During the experiment, the costs associated with these phases were noted as carefully as possible.

The *Test design* phase comprised *test data selection* (which includes activities related to test data generation), *model program design* and *model program implementation*. The *design specification* is depicted in bold because it set a common foundation for all subsequent activities. It served as the basis for automatically generating most of the model program code (around 80%), a configuration-file template for a test data generator (also developed in this experiment), and the database set-up scripts.

The *test execution* phase includes activities relating to setting up the test environment (*database set-up* and *programs' execution set-up*) and executing the programs (*programs' execution*). *Programs' execution* crosses the boundary between *test execution* and *test evaluation* phases 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 initial correctness checking. The *test evaluation* phase also includes the *disagreement analysis* procedures whereby disagreements between the programs need to be arbitrated.

'Large-scale, cost-effective test automation' is probably the best way to characterize 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, database set-up, programs' execution set-up, programs' execution* and *disagreement analysis* are largely interdependent, they are discussed together under the single heading of 'back-to-back testing'.

Model program design

Several design decisions were taken to simplify the model program as much as possible, while providing the core functionality of the primary program. Firstly, the model was designed to handle standard input data only. Secondly, because most of the scheduling program's functionality is non-I/O and non-platform related, the model program was designed to cater only for one platform and for a fixed I/O model. The development strategy also included the following guidelines.

- To mitigate the risk of correlated errors, the model program developer avoided 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 was designed to be interchangeable with the primary program in the test execution environment.
- High reliability (i.e. correctness of computation results) and maintainability were given a higher priority than other desirable properties such as high performance, low memory usage and small size.

To facilitate efficient test environment set-up and simple output comparison, the I/O data model for the primary program was also used for the model program. The programs could then share the same input data tables, while a simple specification query language (SQL) script could be written to compare the results. To this end, 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 is responsible for the data transfer between the scheduling engine and the database. For convenience, the I/O data model was chosen so that most of the test cases that were previously designed for testing the primary program could be reused.

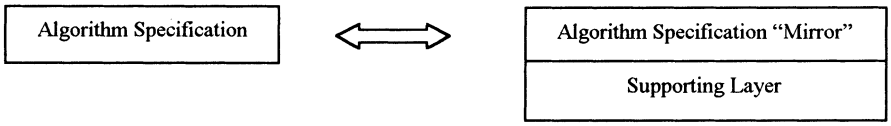

Figure 8. Implementation strategy.

As a side activity, a simple specification language was derived from SMALL [21] (Shlaer–Mellor Action Language, one 'L' being gratuitous)—a specification language designed to specify the actions associated with Moore states in Shlaer–Mellor Object-oriented Analysis/Recursive Design. The language supports mathematical notions such as sets, relations, functions and sequences that characterize model-based formal specification languages such as Z and VDM. It is sufficiently formal and expressive for specifying the scheduling algorithm in a simple, concise and unambiguous manner. It 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 a smooth, cost-effective implementation. Yet, the level of automation that was achieved in the end was not anticipated.

Model-program implementation

The first step in model-program implementation was to choose a programming language and a development tool as an environment. Since a model program is not constrained to specific 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 many attractive options. It is worth mentioning that 26 different languages were used in 38 rapid prototyping studies analysed by Gordon and Bieman [22].

Development environments considered for the experiment included UnifAce (a 4GL), C++/Microsoft Developer Studio 97, C/Microsoft Developer Studio 97, LISP, Prolog, C++ Builder, Visual J++, Oracle PL/SQL. For a variety of reasons, C++/Microsoft Developer Studio 97 was selected. By appropriate layering of C++ code, it was possible to achieve a close correspondence between the specification language statements and the calls made to C++ methods. As illustrated in Figure 8, a supporting layer was created so that the algorithm specification could be 'mirrored' into a higher layer of code, thus producing an executable specification of the algorithm.

The ideal that guided the implementation was that there should be an almost exact correspondence between each specification statement and its 'mirror' C++ method call. Although it would have been possible to achieve this ideal, compromises were made in an effort to reduce the overall costs of the experiment. Even though not ideal, the approach still had significant benefits. Firstly, the supporting

layer defined a domain-dependent executable specification language[¶] which could be used very easily write two important additional programs. (These programs will be discussed later in the context of model program calibration and test data generation.) Secondly, because the supporting layer was generic and because the level of correspondence between the specification and the 'mirror' was sufficient to allow a smooth translation of the algorithm into code, the large-scale C++ code generation was possible.

Generating C++ code

The possibility of large-scale C++ code generation was not initially envisaged. While writing the first module by hand, however, it was noticed that all classes in the supporting layer provided essentially the same capabilities such as creation, deletion, set ordering and subset selection. It was soon realized that it would be possible and much more beneficial to generalize 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 become 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.

- Model program performance, memory usage and size were not a primary concern. The code generator could thus be kept very simple.
- The supporting layer, which was the target of code generation, was by far the largest portion of the implementation, eventually accounting for about 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 a spreadsheet and generates the appropriate C++ and embedded SQL 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 *et al.* [23]. 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 described in Coplien *et al.* [24] is also relevant to the way in which code generation was approached in this experiment. From an SCV analysis perspective, the code generator was based on parametric polymorphism.

Calibrating the model program

During the design phase, it was observed that certain behavioural aspects, which were addressed in the requirements specification at very high level, could lead to programs that produce slightly different, but correct, outputs. Moreover, while writing the model program it became evident that floating-point

[¶]The approach is actually a form of domain analysis as discussed in Mili *et al.* [23]. In this regard, the supporting layer is the realization of the domain model, while the algorithm specification 'mirror' embodies the application (i.e. scheduling).

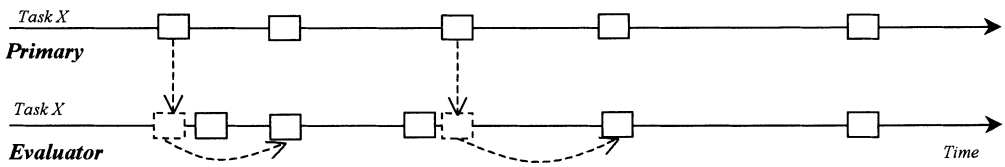

Figure 9. The Evaluator's technique.

inaccuracies could also 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, therefore, all other differences become practically meaningless from a disagreement analysis viewpoint because it is difficult to determine whether an output difference is encountered only because of a previous one or because of other failures.

To be able to perform effective output comparison in the presence of unresolved floating point and specification non-determinism, the model program, henceforth referred to as the Scheduler, was enhanced with a module called the Evaluator. After the Scheduler schedules a task occurrence, the Evaluator compares it to the corresponding occurrence scheduled by the primary program. If they are different, the Evaluator logs the disagreement and then, differentiating between minor and major disagreements and based on specified tolerances, it may reset the model program's state to be in accord with the output of the primary program. In Figure 9 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.

Besides allowing effective output comparison in the presence of unresolved non-determinism, the Evaluator also increased the failure-detection ability of tests in general. Because differences are no longer propagated, *all* disagreements encountered during a single test run can be analysed separately.

Ideally, to complete the calibration process, the model program itself ought to have undergone some initial testing. Test data that had previously been created manually for testing the primary program was available; however, since a large number of these test cases involved exception-domain (i.e. non-standard) data, and since the model had been designed to process only standard data, it was decided not to rely on this data. Instead, the model itself was tested as part of the overall back-to-back testing.

Back-to-back testing

Essentially, back-to-back testing entailed (1) test data selection, (2) setting up the test execution environment from the generated data, (3) executing the programs and (4) disagreements analysis.

[‖]Although this 'consistent comparison problem' had been identified and discussed by Brilliant *et al*. [15] in earlier literature, we were unaware of it at the time. We are indebted to a referee for pointing us to this research.

All except the last of these activities were significantly automated. Each activity is briefly discussed below.

Test data selection

A major advantage of M-mp testing is the cost effectiveness of carrying out a large number of tests. Emphasizing and exploiting this advantage was a goal from the start. As a result, it was decided to make use of a test data generator. Although test data generators such as DGL [25], DataShark [26] and Datatect™[27] were considered, developing a tailor-made data generator for the application at hand was considered the most cost-effective solution.

The resulting generator was quite powerful, but it could not be used for updating a schedule. Generating a realistic updated schedule was too difficult, as it would have implied meshing the scheduling 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 an additional program with the purpose of acting as the 'user'. The program, called the Modifier, was designed to alter the output of the primary program to be used as input for new tests.

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 some other statistical profile.

Clearly, the adequacy level, in terms of some coverage criterion, of a set of test data thus generated (with respect to either the primary or the model program) can only be established *ex post facto*. However, it was beyond the scope of the present experiment to make precise measurements of adequacy levels as a result of data selection based on the test generator. Instead, it is argued later that the adequacy levels with respect to commonly used coverage criteria (such as statement, branch, path, data-flow, etc.) are likely to increase, on average, as a result of generating large numbers of test cases.

Setting up the test execution environment

Once a test dataset was generated, the test-execution environment had to be set up from the generated data (i.e. creating the test execution scripts and populating the database). Although this was one of the most automated activities in the M-mp experiment, the supporting software is rather specific to the present application. In the interests of conciseness, the interested reader is referred to Manolache [11] for full details. Note, however, that the provision of the associated software represents a cost that was included in the overall cost-benefit evaluation of the experiment, as discussed later.

Executing the programs

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 dataset. The automated tests started with a large set of random data. Although this gave a rough estimate of the level of disagreement between programs, it was too complex and diverse to be useful for disagreement analysis. The experiment, therefore, proceeded incrementally with small and simple datasets, taking full advantage of the data-generation flexibility.

Disagreements analysis

The aim of disagreement analysis is to determine the cause of differing output for a given test-case input, whether it be due to a defect in one or more programs, or perhaps even a specification defect such as an ambiguity that leads to different implementations. The most direct approach is to work out the outputs by hand and compare them with results produced by each program. In M-mp testing this is the only point at which manual calculation might be needed—an important advantage over manual test design. In the experiment, however, the cost of disagreement analysis and the need for manual computation was reduced even further by relying on data mutation and correlation.

Data mutation entailed changing test inputs slightly with a view to drawing general conclusions about the causes of disagreements. For instance, it was useful to know that turning off a flag brings the programs back into agreement. Similarly, changing the relationship between two input parameters could sometimes influence 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. Generalizing 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**. Alternatively, if a disagreement disappeared after removing a defect, it was likely that the disagreement had been caused by that defect (or by the corresponding failure type).

EXPERIMENTAL RESULTS AND INTERPRETATION

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

- Cost, unless otherwise stated, represents the amount of person-hours required to carry out a task. In this respect, the terms 'cost' and 'effort' are used interchangeably.
- To allow meaningful and easy comparisons, costs are sometimes 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).

Observed and estimated costs

Tables I and II provide raw data indicating the effectiveness of code generation and the experiment's cost breakdown in person-hours respectively. Table I shows that some 85% of the code was eventually generated automatically. Table II gives a breakdown of person-hours needed to complete the model program development (458), test data selection (208) and test execution and evaluation (73). Thus a total of 737 person-hours were taken to run the complete experiment. It is important to emphasize that the above results pertain to testing only the *core functionality* of the scheduling application using *standard data.*

**A failure type means a class of failures that are caused by the same, not necessarily localized, defect.

Table I. Code generation.

	Total lines of code		Lines of code without comments		Executable statements	
	Number	%	Number	%	Number	%
Generated code	72 730	86.1	43 770	84.9	24 072	86.1
Manually written code	11 790	13.9	7783	15.1	3876	13.9
Total	84 520	100	51 553	100	27 948	100

Table II. Experiment's cost breakdown and estimated test design costs.

Activity	Person-hours		Adjustment factor		Estimated person-hours	
Model program development		458		1.39		638
Specification language definition	30		1		30	
Model program design	122		1.5		183	
Code generation[a]	137		1.25		171	
Scheduler implementation[b]	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	
Test design		664		1.31		872
Test execution and evaluation		73				
Test environment set-up	42					
Programs' execution and disagreement analysis	31					
Total		737				

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

Table III. Test design costs. All values represent percentages of the manual test design cost.

	Model costs			
	Complete development		Reuse-based development	
Functionality implemented	Core	Full	Core	Full
Model program development	24	33	17	25
Test data generation	11	12	3	4
Standard domain test design	35	45	20	30
Exception domain test design	—	24	—	8
Complete domain test design	—	69	—	38

The last column of table II gives person-hour estimates of an M-mp test design covering the *full* scheduling functionality, based on an adjustment factor for the various activities. The adjustment factor 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 because the effort spent writing the code generator is not known exactly. (It is estimated to be in the region of 100 person-hours.) The estimates suggest a total test design time of 872 person-hours compared to the observed 664 person-hours.

Table III gives the experiment's test design costs for testing core functionality, as well as estimated costs for testing full functionality as percentages of the manual test design cost. The implications will be analysed in the remainder of this section.

Core functionality cost analysis

The most important results of the experiment, based on costs for developing core functionality, may be summarized as follows.

- Table III shows that developing the model program cost only 24% of the manual test-design cost. By excluding the effort that was spent on creating reusable artefacts (i.e. the specification language and the code generator) the percentage reduces to 17%. The experiment thus 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.
- The test data generation costs in Table III includes the effort required to construct the test input conditions. As indicated in the table, test data generation was very economical. Its total cost was only 11% of that of the manual test design. Moreover, had the data generator been available at

Table IV. Detected defects.

Artefact	Number of defects
Requirements specification	2
Primary program	2
Model program	2
Test environment	1

the start of the experiment, the percentage would have decreased to 3%! Because the test data generator was developed, it became feasible to carry out a large number of arbitrary tests.

- Thus, 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 datasets.
- 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, had to be limited to less than 150 test inputs.
- Table IV summarizes the defects detected. 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 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 the implementation of 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. This decision, which was reached by proceeding incrementally with the experiment until sufficient evidence had been gathered to draw a valid conclusion, kept experimentation cost within reasonable limits.

Full functionality cost analysis

Ideally, notwithstanding the foregoing, core functionality M-mp test costs should have been compared to the manual test design cost corresponding to the same testing scope (i.e. where manual test design is limited to core functionality testing). Unfortunately, it was not possible to determine these latter costs accurately. Instead, as previously explained, informed and reasonable projections were made of what the M-mp test design costs would have been, had the model covered the system's full functionality.

As shown in Table III, the estimated cost of implementing the complete functionality in the model program is reasonably low (25–33%). Consequently, although the extra effort would have provided

better accuracy of the cost estimates, it is unlikely that the overall significance of the results would have been influenced. (It is, however, interesting to note in passing that the estimate of the time to develop a model program providing full functionality is six to seven times cheaper than the development of the primary program.)

The estimated full functionality results in Table III 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 percentage 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. This would comprise writing the exception-data generators (if necessary) and setting up their inputs. As previously discussed, generating exception data entails simple value substitutions and it is expected to be much less complex than generating standard data. Therefore, the test data-generation costs of 12% in the second row of Table III can safely be assumed to be upper bounds for the generation of incomplete and invalid data. Consequently, the exception-domain test design costs can be pessimistically estimated to be double the costs of generating standard data, shown in the fourth row of Table III.

Thus, based on sound estimates, Table III indicates that 38–69% of the manual test design cost would be sufficient for an M-mp test design covering the system's full functionality across the complete input domain.

Test adequacy comparison

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 almost independent of the number of tests. The cost of manual test design, on the other hand, increases with the number of tests. This is emphasized in Figure 10, 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* plot passes through the point (150, 100), since approximately 150 tests consumed 100% of the cost. The graph has a constant non-zero slope and 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.

As suggested in Figure 10, designing tests manually is not feasible for large numbers of tests. As will be shown next, this might limit the test adequacy levels or it might increase the cost of data selection.

The term 'data accuracy' is used to indicate how well a test-generation procedure generates a test dataset that ensures a specified adequacy level with respect to 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 referred to as the data accuracy. For example, in Figure 11, 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, low data-accuracy of a test-data generation procedure means that a large number of tests have to be generated in order to achieve a given adequacy level.

As suggested in Figure 11, even if the 150 test inputs derived from one test generation method are highly accurate (point C), 15 000 test inputs derived by another test generation method could, on

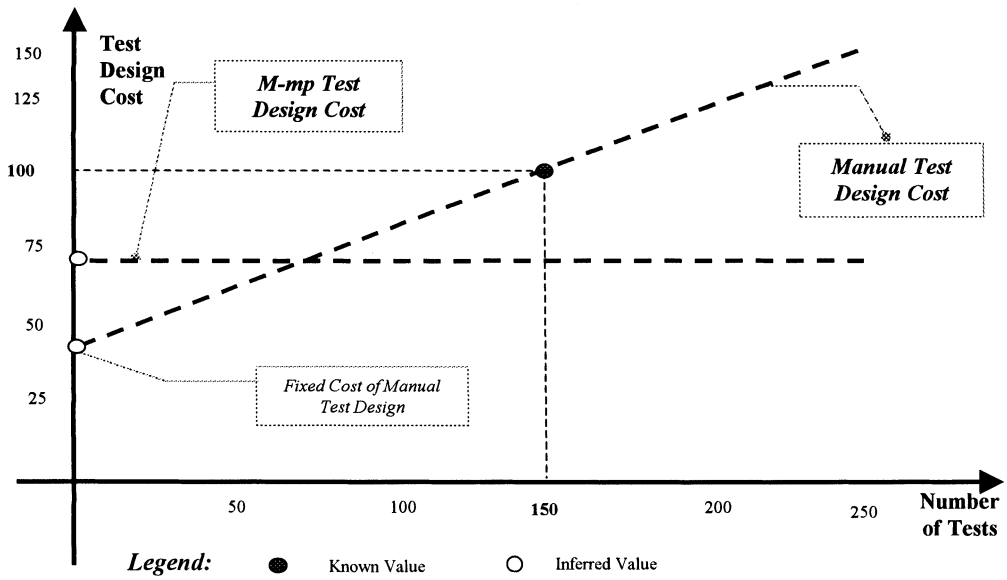

Figure 10. Test design cost function of number of tests.

average, achieve the same adequacy level (point A) albeit with significantly less accuracy. In fact, the 15 000 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 generating less accurate data could cost much less. Clearly, however, these are probabilistic claims whose validity is contingent on the precise nature of the randomly-generated sample of test data, as well as on the nature of the input domain.

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 11 (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.

In the experiment, executing the primary program and then the model program took on average 10 s per test input. If one wished to extend the experiment, therefore, 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 h, for 60 480 in one week and so on. 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 in practice sufficient to analyse thoroughly only one

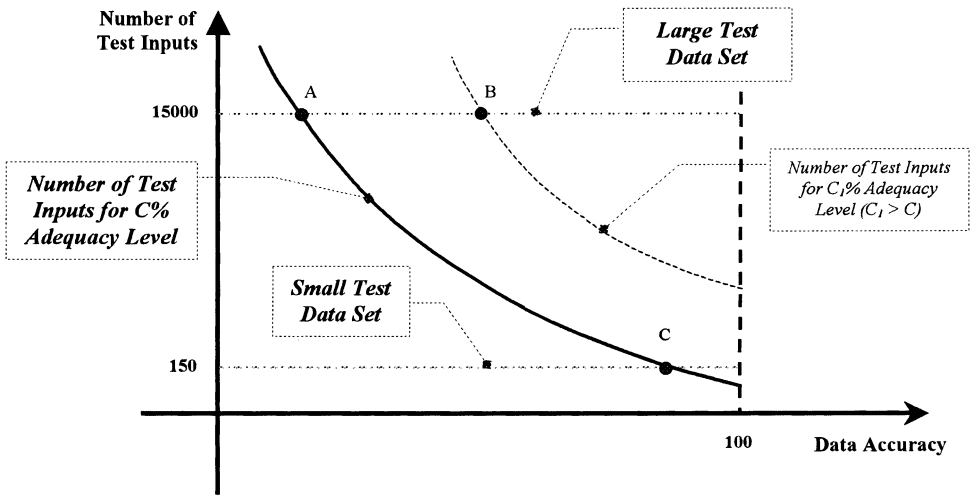

Figure 11. Adequate test data size function of data accuracy.

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 above are not feasible in the manual approach. (Since they generally require additional tests, they would therefore imply additional expensive manually pre-calculated results.)

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 had turned out to be expensive. Within the scope of the experiment, however, the model program proved as reliable as the primary program, and the cost of localizing and removing its two defects was fairly insignificant (only a few hours!).

SUMMARY AND CONCLUSIONS

Based on *prima facie* evidence, sound estimates and theoretical considerations, it has been shown that the M-mp approach could test a scheduling program more adequately than manually designed tests and at lower cost. Of course, generalizing the validity of such a conclusion is dependent upon the specific conditions of the domain. An 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?

Repeatability of the experiment

The low cost of developing a reliable and maintainable model program in the experiment was a direct consequence of M-mp testing principles that have been presented in the second section: a model program does not need to be equivalent to the primary program; it should, primarily, cover functionality whose correctness checking is normally expensive; and optimizing the model program's speed and resource usage is unimportant. In the case of the scheduling application, it was sufficient for the model program to cater only for standard-domain scheduling. As a result, the development of the model program was straightforward. Except for defining the specification language and writing the code generator, developing the model program did not require advanced analytical or programming 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 such 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 environment. Given the appropriate development environment, a person able to schedule tasks by hand would also be 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.

Given the above considerations, it is reasonable to expect that developing a model program for standard-domain scheduling will in general be as cost effective, as broadly suggested by the experiment. Because test-data selection can often be automated cost effectively, 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 adequate testing of the scheduling application. As will be discussed next, this conclusion might apply more generally to testing algorithmically-complex software.

Wider applicability of M-mp testing

The applicability of M-mp testing should be judged on a case-by-case basis. 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 depicted in Figure 10 are likely to characterize a wide range of applications, both simpler and more complex than scheduling. As suggested by Figure 11, the M-mp approach will tend to be a better option than the manual one when the number of tests required to achieve a particular adequacy level becomes large, say greater than about 100.

In judging whether M-mp testing is likely to be cost effective, it is also important to assess whether sufficient diversity between model and primary programs is possible. If the domain is such that it forces model programs to be very close to the primary one in functionality and structure, then the limited diversity would increase the risk of correlated failures. In addition, such model programs would tend to be as expensive the primary one.

RELATED WORK

The authors are not aware of any other experiment that compares an M-mp approach to testing that is based on manual outcome prediction. The research community seems to rule out manual result checking [4,5,7], but in industry, possibly because of too much emphasis on independent black-box testing [28], manual approaches are still employed. The practitioner community seems to have more confidence in human judgment 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 organization would generally want to use those scarce skills for development rather than for testing.

The closest work to the presented experiment is perhaps the testing approach described by Buettner and Hayes [29]. The software test team successfully used Mathematica® simulations to unit test algorithmically-complicated software, uncovering numerous specification ambiguities and software errors. In the context of this paper, Mathematica® simulations are model programs and the approach is, therefore, a form of M-mp testing. Buettner and Hayes [29] also mention that improvements in the software process significantly increased testers' motivation.

Another closely related work is that presented by Peters and Parnas [6,7]. The authors 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 (highly reliable) model programs.

It is also important to mention the function equivalence testing technique described by Kaner *et al.* [30]. 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 to 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.* [30] advocate the use of function equivalence testing whenever possible because it lends itself to automation. They also give guidelines on how to conduct a cost-benefit analysis that promotes function equivalence testing as a replacement to manual testing.

REFERENCES

1. Hamlet D. Predicting dependability by testing. *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA)*, San Diego, California. IEEE, 1996; 84–91.
2. Zhu H, Hall PAV, May JHR. Software unit test coverage and adequacy. *ACM Computing Surveys* 1997; **29**(4):366–427.
3. Van Vliet H. *Software Engineering: Principles and Practice*. Wiley, 1996.
4. Beizer B. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. Wiley: New York, 1995.
5. Richardson DJ, Aha SL, O'Malley TO. Specification-based test oracles for reactive systems. *Proceedings of the 14th International Conference on Software Engineering (ICSE)*. IEEE, 1992; 105–118.
6. Peters KD, Parnas DL. Generating a test oracle from program documentation: Work in progress. *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, Seattle, Washington. IEEE, 1994; 58–65.
7. Peters KD, Parnas DL. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering* 1998; **24**(3):161–173.

8. Stocks P, Carrington D. A framework for specification-based testing. *IEEE Transactions on Software Engineering* 1996; **22**(11):777–793.
9. Pfleeger SL, Hatton L. Investigating the influence of formal methods. *Computer* 1997; **30**(2):33–43.
10. Weyuker EJ. On testing non-testable programs. *The Computer Journal* 1982; **25**(4):465–470.
11. Manolache LI. Testing algorithmically complex software using model programs. *MSc Dissertation*, University of Pretoria, 2000.
12. Eckhardt DE, Lee LD. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Transactions on Software Engineering* 1985; **SE-11**(12):1511–1517.
13. Knight JC, Leveson NG. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering* 1986; **SE-12**(1):96–109.
14. Brilliant S, Knight JC, Leveson NG. Analysis of faults in an N-version software experiment. *IEEE Transactions on Software Engineering* 1990; **SE-16**(2):238–247.
15. Brilliant S, Knight JC, Leveson NG. The consistent comparison problem in N-version software. *IEEE Transactions on Software Engineering* 1989; **SE-15**(5):1481–1485.
16. Hatton L. N-version design versus one good version. *IEEE Software* 1997; **14**(6):71–76.
17. Hatton L, Roberts A. How accurate is scientific software? *IEEE Transactions on Software Engineering* 1994; **20**(10):785–797.
18. Kourie DG. An approach to defining abstractions, refinements and enrichments. *Quaestiones Informaticae* 1989; **6**(4):174–178.
19. Bishop PG, Esp DG, Barnes M, Humphreys P, Dahll G, Lahti J. PODS—a project on diverse software. *IEEE Transactions on Software Engineering* 1986; **SE-12**(9):929–940.
20. Cohen DM, Dalal SR, Fredman ML, Patton GC. The AETG system, an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering* 1997; **23**(7).
21. SMALL: Shlaer–Mellor action language. http://www.projtech.com/info/small.html [1999].
22. Gordon VS, Bieman JM. Rapid prototyping: Lessons learned. *IEEE Software* 1995; **12**(1):85–95.
23. Mili H, Mili F, Mili A. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering* 1995; **21**(6):528–562.
24. Coplien J, Hoffman D, Weiss D. Commonality and variability in software engineering. *IEEE Software* 1998; **15**(6):37–45.
25. Data Generation Language (DGL). http://www.csee.usf.edu/~maurer/dgl.html [1999].
26. DataShark. http://www.hardballsw.com [1999].
27. Datatect™. http://www.datatect.com [1999].
28. 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, DC, USA, June 1998; 1.7.1–1.7.11.
29. Buettner DJ, 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, DC, USA, June 1998; 21–24.
30. Kaner C, Falk J, Nguyen HQ. *Testing Computer Software*. International Thomson Computer Press, 1996.

