Generating Structurally Complex Test Cases by Data Mutation: A Case Study of Testing an Automated Modelling Tool

Lijun Shan
Department of Computer Science, National University of Defence Technology, Changsha, 410073, China
Email: lijunshancn@yahoo.com

Hong Zhu
Department of Computing, Oxford Brookes University, Oxford OX33 1HX, UK
Email: hzhu@brookes.ac.uk

ABSTRACT

Generation of adequate test cases is difficult and expensive, especially for testing software systems whose input is structurally complex. This paper presents an approach called data mutation to generating a large number of test data from a few seed test cases. It is inspired by mutation testing methods, but differs from them in the aim and the way that mutation operators are defined and used. While mutation testing is a method for measuring test adequacy, data mutation is a method of test case generation. In traditional mutation testing, mutation operators are used to transform the program under test. In contrast, mutation operators in our approach are applied on input data to generate test cases, hence called data mutation operators*. The paper reports a case study with the method on testing an automated modelling tool to illustrate the applicability of the proposed method. Experiment data clearly demonstrate that the method is adequate and cost effective, and able to detect a large proportion of faults.*

1 INTRODUCTION

Generation of adequate test cases is one of the most difficult and expensive tasks in software testing, especially for software systems whose input has complex structure. A structurally complex input consists of a variable number of elements interrelated according to a specific set of rules. For example, the input to a modelling tool is a model following the syntax of the modelling language. The modelling tool processes a model according to not only the values of its individual elements, but also the way that the elements are interconnected. Other well-known software systems with structurally complex input include compilers of textual programming languages, XML document processor, CAD software, etc. To test such a software system adequately, test cases must be generated with all combinations of the values of the elements and all valid combinations of the ways they can be interrelated. Automatic generation of test cases of such complexity is still beyond the current state of art in software testing techniques. Manually preparing such test cases is tremendously labour-intensive, difficult and error-prone. This paper is concerned with the test case generation for such software systems.

In [1], we proposed a test case generation technique, called data mutation testing, in the context of testing a modelling tool. It involves the design of a set of data mutation operators and the application of them to a few seed test cases to generate a large number of test data. Our method is inspired by mutation testing methods, but differs from them in the way that mutation operators are defined and used. Our method is not for test adequacy measurement and analysis. Instead, it is a test case generation technique. In our approach, mutation operators transform the input data rather than the program under test or the specification of the software. This paper describes the method in a more general context and reports in more detail of a case with the method.

The paper is organised as follows. Section 2 briefly reviews existing work on test case generation. Section 3 describes the method and illustrates it with a simple example. Section 4 reports a case study with the method on testing a modelling tool. The fault detecting ability, test adequacy and test cost of the method are analysed through the case study. Section 5 concludes the paper and discusses further work.

2 RELATED WORK

In the past few years, a great amount of research has been reported in the literature in the area of automatic generation of test cases. Test generation methods can be classified into three main types: *program-based*, *requirements-based* and *random* test.

Research on program-based test generation methods can be backdated to 1970's, for example, [2, 3, 4, 5, 6, 7]. Such methods rely on either the analysis of the program code under test without actually executing the program or observations on the dynamic behaviour of the software during its execution on test cases. The former are called *static* test generation methods, such as those employing symbolic execution [2-7]; the latter are called *dynamic* methods, such as [8] and [9]. The methods reported in [2-8]

and their variants are *path-oriented* because the test case generation algorithms take certain selected paths in the program as input. In contrast, *goal-oriented* methods aims at executing certain selected statements or branches in the program. The algorithm can determine the paths that cause the statements or branches to be executed. Typical examples of such test generation methods include [10] and [11]. Another example of goal-oriented method is the test case generator Godzilla in Mothra [12], which aims at executing the location in a program where a mutation operator is applied and killing the mutant [13, 14]. Program-based test generation methods eventually rely on either solving constraints represented in the form of equations and inequalities on numerical variables, or heuristic searches in the input space, e.g. using genetic algorithms; see [15] for a recent survey. Although significant progress has been made in the past three decades in the improvement of the capability of the test case generators, the applicability of program-based test case generation methods is still very limited due to the following reasons. First, the expressiveness of constraints that can be solved by existing tools is limited. The constraints are equations or inequalities of numerical values. However, some software systems, e.g. modelling tools and compilers, require structurally complex input, which contains non-numerical elements with more complicated logic constraints such as type compatibility. Second, it is well known that the computational complexity of constraint solving, even for the satisfaction of propositional statements, is NP complete. The scale of the problem that is practically solvable is very limited. It can hardly reach the scale of test cases that are adequate for testing software systems like modelling tools and compilers. In particular, the state space for search-based and constraint-solving test case generation could explode as the scale of input increases [16]. Finally, the space of the input data for such software systems can be highly complicated. Search in such input space is computationally complex. It is impractical to apply simple enumerative techniques to find meaningful combinations of the elements in the input space. Therefore, they are still far away from meeting the requirements of testing software systems whose input data are non-numerical and have highly complex structure. Moreover, the test cases are generated aiming at covering the program code. They do not necessarily represent the inputs that are meaningful in the real operation of the software systems.

Requirements-based test generation methods, also called specification-based testing in the literature [17], derive test cases from either formal or semi-formal specifications of the required functions and/or designs of the software under test. They can be further classified into methods based on formal specifications and *model-based* methods. The former derives test cases from various formalisms of software specifications, which include first order logic [18], Z [19, 20], logic programs [21], algebraic specifications [22, 23, 24], finite state machines [25, 26, 27], Petri nets [28, 29], etc. The latter derives test cases from semi-formal models of software systems in diagrammatic notations. In [30, 31], a hierarchy of test adequacy criteria was defined and investigated on structured requirement definitions in dataflow diagrams, entity relationship diagrams and state transition diagrams. A testing tool for validation of requirement definitions was designed and implemented to generate test cases to meet these adequacy criteria. More recently, advances have been made in the derivation of test cases from various types of diagrams of UML [32, 33], and from extended finite state machines or statecharts [34, 35, 36, 37], etc. Both specification-based and model-based test generation methods derive high level descriptions of test suites, e.g. in the form of a set of constraints on the input. Further generation of test data on which the software can execute relies on heuristic search and constraint satisfaction techniques, including constraint logic programming, deductive theorem proving and model checking, etc. [38]. To be used for testing software that takes structurally complex inputs, requirements-based methods suffer from the same problem to program-based methods. The generation of highly complex test data remains an open problem [39, 40].

Random testing methods generate test cases based on certain probabilistic models of the operation of the software under test. They generate or select test cases through random sampling over the input space of the software according to certain probabilistic distribution. A simple random testing method is to sample over an existing software operation profile at random. More sophisticated random testing methods have been developed using various types of stochastic models of software usages to represent the probabilistic distributions over the sequences of activities in the operations of a software system, such as Markov chain [41, 42, 43, 44, 45], stochastic automata networks [46, 47], Bayesian networks [48], and so on. They are employed to generate test cases for load testing [41, 42, 49], reliability testing [44], as well as fault detection and functional verification and validation [43, 48], etc. However, it is unclear if random testing methods are capable of generating test cases that are structurally complex.

In addition to generic test generation methods, researches have been reported on test generation for specific types of software systems, such as database applications [50], spreadsheets [51], XML data schemas [52], XML-based web component interactions [53], etc. These methods address the specific requirements in the testing of such systems. However, the structural complexity of test data of such systems is far less than that of the systems this paper concerns, such as modelling tools and program compilers. As program compilers have a long history, a large amount of efforts have been spent on testing of compilers; see [54] for a survey. Existing work on compiler testing produces test cases from grammar of the programming language. However, due to the limitation of grammar-based automatic program generators, the approach is more appropriate for testing compilers in processing language's syntax of than in processing complex semantics. The current practical approach to testing a compiler's correctness in processing program semantics relies on benchmarks, which are collections of manually prepared test cases. As far as we know, there exists no special method for testing graphic modelling tools.

The method presented in this paper is inspired by mutation testing. Mutation analysis was originally proposed in [55, 56] for measuring test case adequacy. Since then, it has been intensively investigated; Cf. [57, 58, 12, 59, 60, 61]. The principle of mutation testing has also been applied to specification-based testing [62, 63]. In such an approach, the specification of a program is considered as a set of language elements that describe the input and output of the program. Replacing one element in the specification by another creates a mutant of the original specification. A test case detects the difference between a specification

mutant and the original specification if the output is correct according to one specification but incorrect according to the other. In such a case, the mutant is said to be dead, or killed by the test case. Test adequacy can, therefore, be measured according to the proportion of dead specification mutants. As for all test adequacy criteria, such adequacy measurement can also be used to generate test cases, for example, by setting the goal of test case generation as detecting the differences between the original and the mutant specifications. Murnane and Reed reported the case study for comparing the method with other popular black box testing methods, such as equivalence class testing and boundary value testing [64]. They showed the advantages of the specification mutation method. However, it suffers the same problem with other specification-based test case generation methods. In [65, 66], Offutt and Xu et al. proposed an approach to Web Services testing by modifying XML messages – the input data of the Web Services, and then analysing the response messages. A collection of schema perturbation operators was designed to modify XML schemas from which XML messages were produced as test data. Because XML schemas are definitions of the structures and types of data, schema perturbation operators play a similar role of mutation operators on formal specifications. In comparison with the mutation testing methods, our approach aims at generating test cases rather than measuring test adequacy. It offers a practical solution to testing software that requires complex input.

The idea of varying one test case to produce another has also been explored by other researchers. Meek and Siu used randomised error seeding to evaluate the ability of language processors in detecting and reporting errors in source programs [67]. This method is similar to our method that mutates graphic models for testing modelling tools.

More recently, Chen et al. proposed Adaptive Random Testing (ART), a failure pattern-based random testing technique, targeting an even spread of randomly selected test cases within the input space [68]. A number of versions of ART have been developed since then, e.g. Mirror ART [69], Restricted Random Testing [70], Probabilistic ART [71], etc. In comparison with our method, the similarity of ART is it generates test cases according to previous test cases. However, the two test case generation methods differ in the rationale, in the assumption, and in the aim of considering relations between test cases. First, ART is based on the observation that failure-causing inputs form certain kinds of patterns [72]. The rationale of ART is, given the number of test cases can be used, widely spread test cases in the input space are expected to have a greater chance of hitting failure-causing regions. The applicability of the method replies on the notion of distances between test cases. However, for software with complexly structured input, the notion of distance between test cases is not so obvious. Consider modelling tools as an example. It is difficult to demarcate an input space, not to mention to further plot failure pattern in the input domain or to measure the distance between two input models. The rationale of our method is that a test set containing all possible constituents of input data and their combinations is expected to have an adequate coverage of the functions of the program under test. Second, ART focuses on the selection of test cases from an input domain at random, while implicitly assuming that a large set of candidate test cases is already available. For programs with numerical input, the assumption holds and test case generation equals to test case selection. However, for software with structurally complex input, the assumption cannot be easily satisfied. Instead, test case construction is one of the most labour intensive tasks in test case generation. Third, in ART, the relations between test cases are utilised in order that each new test case is as far as possible away from previous test cases. In our method, the relations between data are employed to obtain as many as possible variations from one test case. In spite of the differences between ART and the method proposed in this paper, the research results of ART can be used in our approach. For example, according to ART's research results, it could be a good guideline to select seed test cases that are different as much as possible, or in ART's terminology, as far away as possible. Evaluation of this guideline could be a topic for further research.

Another testing method that involves the variation of test cases is the Metamorphic Testing method proposed by Chen et al. [73]. The main purpose of the testing method is to alleviate the test oracle problem by using metamorphic relations rather than full strength formal specifications. A metamorphic relation is an expected relation among the inputs and outputs of *multiple* executions of the program under test. For example, if the program under test implements a monotonic increasing function on an integer input, then the output on input x_1 should be greater than the output on input x_2 when x_1 is greater than x_2. Such a metamorphic relation can be used, at least, to detect errors. To enable such relations to be checked, test cases must be generated to satisfy the condition of the metamorphic relations used in the metamorphic testing. This can be achieved by varying an existing test case, for example, by deriving x_2 as x_1-1. In the sense of producing test cases by varying existing test cases, metamorphic testing has similarity to our method. The difference is that their variations are derived from metamorphic relations while we do not rely on the existence of metamorphic relations. If test cases are to be automatically generated from metamorphic relations, theoretically speaking, it will suffer from the same problems for specification-based test case generation although metamorphic relations could be much simpler than full strength formal specification. The applications of metamorphic testing on non-numeric computing software such as program compilers and interactive software are demonstrated in [17]. The applications make use of metamorphic relations on specific types of test cases rather than over the whole input space. For example, in testing a compiler, a program that contains two statements independent of each other can be used to construct a follow-up test case by changing the sequence of the two statements. This highlights the difference of Metamorphic Testing from our method. We aim to generate a test set that covers the whole input space. Thus, our data mutation operators are designed to change the test case in as many different ways as possible. A conceivable combination of our data mutation testing and metamorphic testing is that if the difference in the outputs can be predicted from a data mutation operator, the predicted difference could be expressed as a metamorphic relation, and checked for the correctness of the software under test. In fact, in our case study, we used such predictions to check the correctness of software's output on test cases. Since data mutation operators are easier to develop than metamorphic relations, data mutation operators can be used to help construct metamorphic relations.

3 THE DATA MUTATION METHOD

We use a small paedagogic example to illustrate the basic ideas and the process of data mutation technique. The advantages of the testing method are shown by a case study of testing more complicated software in section 4.

Suppose we are to test a Triangle Classification program whose input consists of three natural numbers *x*, *y*, and *z* as the lengths of the sides of a triangle. Its function is to classify the triangle into *equilateral* (all sides the same length), or *isosceles* (two the same), or *scalene* (none the same), or to determine that the input does not represent an actual triangle when the summary of two parameters is not greater than the third.

As shown in Figure 1, the data mutation testing process comprises an iterative sequence of the following activities.

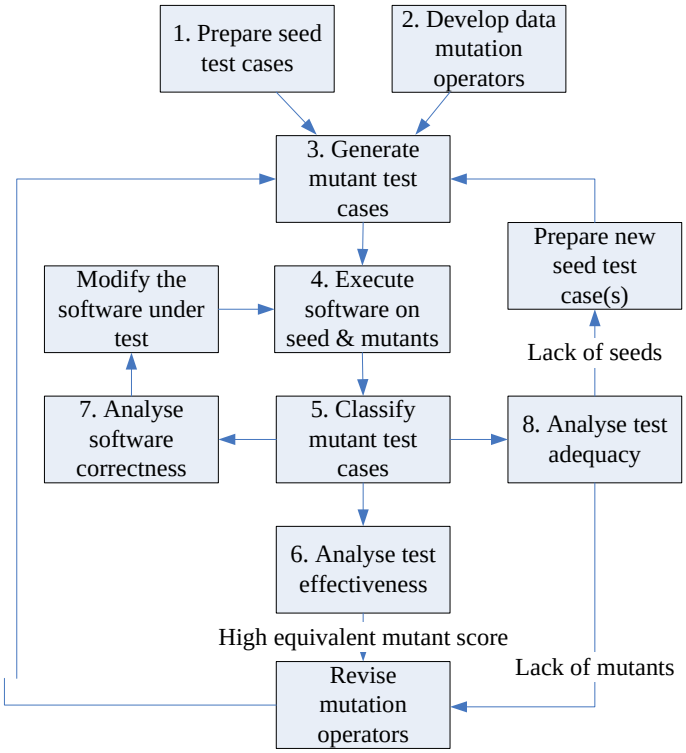

Figure 1. The process of data mutation software testing

Step 1. Prepare seeds.
The data mutation testing starts with preparing some initial test cases either manually or with some automatic test case generation method. These test data are called the seeds because more test cases will be generated from them. When the test cases have complex structures, it is not easy to obtain a large and adequate set of such initial test cases. Fortunately, our experiments have shown the method does not require a large number of seed test cases. A small number of seeds that contain all possible types of elements of the input data will be enough.

For example, one may design four test cases for the Triangle Classification program as follows to cover each type of triangle with one test case.

- *Test case* t_1: Input: (x=5, y=5, z=5), Expected output: *Equilateral*.
- *Test case* t_2: Input: (x=5, y=5, z=7), Expected output: *Isosceles*.
- *Test case* t_3: Input: (x=5, y=7, z=9), Expected output: *Scalene*.
- *Test case* t_4: Input: (x=3, y=5, z=9), Expected output: *Not a triangle*.

Step 2. Design and implement data mutation operators.
Data mutation operators are simple transformations on the input data. They are required to preserve the syntactic correctness of the input data with respect to the structure and constraints on the input, if the testing is only concerned with valid input. Otherwise, invalid input can be generated as well by breaking the structure or constraints. Manual application of the mutations can be carried out. Alternatively, the data mutation operators can be implemented in a software tool so that the generation of mutants of the

seeds can be automated. Since the mutation operators are usually simple transformations of the input data, their implementation should not be too demanding. However, the design of the mutation operators requires insight into the functions of the software under test as well as the syntax and semantics of the input data.

For testing the triangle classification program, the structure of the input data consists of three parameters. Therefore, in order to generate test cases as various kinds of instances of the structure, data mutation operators should be designed to mutate the seeds on the parameters, such as to change one parameter's value by a small amount, to change the relationships between the parameters by swapping their values, etc. A constraint on the valid input data is that the parameters must be natural numbers. Data mutation operators can also be designed to test the program on both valid and invalid input data by introducing test cases that violate the constraint. For example, the following data mutation operators can be defined on the input data for the Triangle Classification program.

- IVP: Increase the value of a parameter by 1;
- DVP: Decrease the value of a parameter by 1;
- SPL: Set the value of a parameter to a very large number, say 1000000;
- SPZ: Set the value of a parameter to 0;
- SPN: Set the value of a parameter to a negative number, say -2;
- WXY: Swap the values of parameters x and y;
- WXZ: Swap the values of parameters x and z;
- WYZ: Swap the values of parameters y and z;
- RPL: Rotate the values of parameters towards left;
- RPR: Rotate the values of parameters towards right.

Step 3. Generate mutant test data.
Given a set of seeds, the set of specifically designed mutation operators are applied to each seed to generate a set of mutants. In many cases, a mutation operator can be applied to a seed on a number of different locations, which generates a number of different mutants. The software tool that implements the data mutation operations should also automatically locate applicable elements in the input data for each mutation operator. The number of mutants generated from a seed is decided by two factors: the types and numbers of elements in the seed and the designed data mutation operators. It is worth noting that some mutation operators can be applied to a mutant to produce other mutants, which are called the second generation mutants of the original seed. Similarly, a mutant of the second generation can also be used to generate the third generation mutants, and so on. Whether high generation mutants should be developed and used depends on the specific requirement of the test.

For instance, by applying the mutation operator IVP to test case t_1 on parameter x, we can obtain the following test case t_5.

- *Test case* t_5: Input: (x=6, y=5, z=5).

Among the above 10 data mutation operators, the first 5 can be applied on each of the 3 parameters of a seed, so totally (5*3 +5)*4 = 80 test cases covering all sorts of combinations of data elements can be systematically produced from the four seeds.

Step 4. Execute the software system on the seeds and their mutants.
The software under test is executed on the seeds and their mutants. The outputs and/or other aspects of dynamic behaviour of the software on each test case are observed and recorded for further analysis. Our method does not depend on any specific method that the behaviour of the software is observed and recorded. Instead, whether the observation is sufficient can be analysed using the mutation scores; see step (6).

Step 5. Classify mutants.
Similar to traditional mutation testing, the mutants can be classified into either *dead* or *alive* according to the recorded behaviour and outputs of the program under test. A mutant is classified as *dead*, if the execution of the software under test on the mutant is different from the execution on the seed test case. Otherwise, the mutant is classified as *alive*. Depending on the functionality of the software under test, it varies that what exactly means by two executions of the software on two test data are different.

For example, for a correctly implemented Triangle Classification program, the execution on the mutant test case t_5 will output *isosceles* while the execution on its seed t_1 will output *equilateral*. Therefore, the test case t_5 will be dead after the test execution. In testing other software, classification of mutants may be less simple as the Triangle Classification program. For example, if the functionality of the software under test is to transform a model into executable code, the analysis of an execution of the software may involve the actual execution of the code generated from the original model as well as the execution of the code generated from the mutants if no error message reported by the code generator. This may require further analysis and testing on the generated code to distinguish them. In our case study on testing a model consistency checker reported later in this paper, the difference is in the checker's reports on the consistency of the models. In particular, if the original model passes the consistency

checking, a mutant is dead if and only if the checker reports inconsistency of the mutant.

The notion of *dead* and *alive* mutant is different to the notion in traditional mutation testing. In traditional mutation testing, program mutants are classified into *dead* or *alive* according to whether their behaviour on a given test set is different from the original program. The classification of mutants into dead and live plays a significant role in that the percentage of dead mutants indicates the fault detecting ability of the test set. However, it is less important in data mutation testing. Neither the aliveness nor the death of a mutant test case means the program is correct on the test case. A live mutant should be further analysed to find the reason why the mutation of the input does not affect the output of the program under test. A dead mutant also needs further analysis because a difference in the behaviour of the program does not necessarily imply that the program behaves correctly on the mutant. Nevertheless, mutation scores for data mutation testing can serve as useful indicators to guide further analysis of the test effectiveness.

Step 6. Analyse test effectiveness.
A mutant can be alive due to several reasons. First, the mutant is equivalent to the original with respect to the functionality or property of the software under test. For example, in the testing of Triangle Classification program, applying the RPL (rotate the value of parameters towards left) mutation operator to test case t_1 will produce a test case identical to t_1. Therefore, such mutants cannot be distinguished from its original by the software. Hence, the test case will remain alive in the testing. In such cases, the mutant can be regarded as duplicates of the original test case. Since mutation operators are supposed to achieve variety in the test cases to discover different behaviour of the software under test, a high equivalent mutant score *EMS*, which is defined in formula (1) below, indicates that the mutation operators have not been well designed to achieve their purposes.

$$EMS = \frac{EM}{TM}, \qquad (1)$$

where *TM* is the total number of mutants and *EM* is the number of equivalent mutants.

Second, a mutant that is not equivalent to the original can also be alive because the observation on the behaviour and output of the software under test is not sufficient to detect the difference. For example, apply the RPL operator to test case t_2 will produce the following test case.

- *Test Case*: Input: t_6:(x=5, y=7, z=5).

A correct implementation of the Triangle Classification program will output *isosceles* on both the seed t_2 and the mutant t_6. If the tester only observes the output, no difference can be detected on the test cases t_2 and t_6. However, if the execution path is observed, there will probably be differences in the behaviour of the program on these test cases. Thus, t_2 and t_6 should not be considered as equivalent. Formulas (2) and (3) below define live mutant score *LMS* and mutation score *MS*, respectively. If the software under test is correctly implemented and the observation is exactly on the program's function or property concerned in the test, the mutants that cannot be differentiated from their seeds are exactly those equivalent mutants, i.e. the number of live mutants *LM* equals *EM* and *MS* equals 1. A high *LMS* or equivalently a low *MS* indicates that the observation on the behaviour and output of the software under test is insufficient. Hence, a better test method needs to be applied in order to observe the differences.

$$LMS = \frac{LM - EM}{TM - EM}, \qquad (2)$$

$$MS = \frac{DM}{TM - EM}, \qquad (3)$$

where *LM* is the number of live mutants; *DM* is the number of dead mutants. Because *DM*=*TM*–*LM*, we have that *MS*=1–*LMS*.

Third, a mutant can remain alive because the software is incorrectly designed and/or implemented so that it is unable to differentiate the mutants from the original even though they are not equivalent. Therefore, every live mutant deserves an investigation of the true reason of its aliveness. If the live mutant score of a particular type of mutation operators is unusually high, it reveals that the program under test is not sensitive to the type of mutation, which may be due to particular faults in the design or implementation of the software. Hence, we have the following measurements LMS_Φ and MS_Φ defined in equation (4) and (5), respectively.

$$LMS_\Phi = \frac{LM_\Phi - EM_\Phi}{TM_\Phi - EM_\Phi}, \qquad (4)$$

$$MS_\Phi = \frac{DM_\Phi}{TM_\Phi - EM_\Phi} = 1 - LMS_\Phi, \qquad (5)$$

where Φ is a specific type of mutation operators and Φ-mutants are mutants that are generated by applying a mutation operator in Φ. LM_Φ, DM_Φ, EM_Φ, and TM_Φ are the numbers of live Φ-mutants, dead Φ-mutants, equivalent Φ-mutants and total number of Φ-

mutants, respectively.

In case that the mutation scores defined above are unsatisfactory, revisions on the seed test cases and the mutation operators are needed. The exact number of equivalent mutants may be difficult to count due to the high complexity and the large number of the mutants. The following can be used as an approximation to replace the above measurements.

Raw Mutation Score: $RMS = \frac{DM}{TM}$. (6)

Raw Survival Rate: $RLMS = 1 - RMS$. (7)

Raw Φ Type Mutation Score: $RMS_{\Phi} = \frac{DM_{\Phi}}{TM_{\Phi}}$. (8)

Raw Φ type Survival Rate: $RLMS_{\Phi} = 1 - RMS_{\Phi}$. (9)

The measurements defined above can provide empirical data on the effectiveness of the testing and can be used to decide whether refinements on the design and implementation of the mutation operators should be made.

Step 7. Analyse the correctness of the software under test.
Test data mutation is not for the evaluation or measurement of the test adequacy of an existing set of test data. It is a method of test case generation intended to discover faults in the software under test or gain confidence in the software if it passes the test. Therefore, in addition to the classification of mutants into dead or alive, further analysis of the observed behaviour and output of the software on each mutant is important for the purpose of validation and verification of the software under test.

Like metamorphic and other mutation testing methods, data mutation testing has the feature of the existence of the behaviour and outputs of the software on both the original test data and the mutants. In our case study we find that the knowledge about how a mutant is generated, including the applied mutation operator and the location of the application, greatly helps the tester to focus on the difference between the behaviour of the software on the seed and its mutant. With such knowledge, it becomes much easier to identify whether the behaviour of the software on the seed and the mutant are correct or not. To some extent, the program's behaviour on the mutant can be predicted.

Take the Triangle Classification program as an example. For the test case generated by applying the IVP or DVP to test case t_1, we can expect the output to be *isosceles.* For the RPL, RPR, WXY, WYZ, and WYZ mutation operators for the Triangle Classification program, we can expect that the program should output the same classification on a seed and its mutant test cases. If the software's behaviour on a mutant is not as expected, an error in the software under test can be detected.

Step 8. Analyse test adequacy.
Since data mutation is a test case generation method, test adequacy analysis must be performed to decide whether the testing is thorough enough. As discussed in [74], an adequacy analysis can be based on the coverage of the program code of the software under test, or based on the coverage of the functionality of the software according to its specification and/or design, or based on the coverage of the input/output data space according to the usage of the software, or a combination of them. A great number of test adequacy criteria has been proposed and investigated in the literature; see [74] for a survey.

The adequacy of data mutation testing is decided by the coverage of a test set on executing the program, while the test set generation depends on two main factors: the set of seed test cases and the set of data mutation operators. Inadequacy of data mutation testing could be due to the lack of certain elements or their combinations in the seed test cases so that certain segments in the program code or functionality of the software or subset of the input/output data space cannot be exercised by the generated mutants. Inadequacy may also be caused by weakness in the set of mutation operators if they are incapable of generating certain types of test cases. Therefore, if the test adequacy is unsatisfactory, either new seeds should be used or new mutation operators need to be developed.

4 A CASE STUDY

This section presents a case study of the data mutation method with a real software system to demonstrate that the method is feasible, effective and efficient.

4.1 The Software System under Test

The software tested in the case study is the consistency checker in the agent-oriented modelling environment CAMLE [75]. Modelling languages play a central role in current software development methodologies, especially in model-driven methods. Typical examples of modelling languages include UML [76] used in object-oriented software development and Yourdon notation [77] used in structured analysis and design methods. CAMLE is a modelling language and environment we have designed to support agent-oriented software development. CAMLE environment comprises a number of tools, including a model constructor, a consistency checker, and a specification generator. A well-formedness checker is embedded in the model constructor to prevent ill-formed model elements from being input. In the development of the modelling environment, especially its automated tools such as consistency checker, there is a tremendous difficulty in the testing due to the complexity of the input data i.e.

diagrammatic models of multi-agent systems. To manually prepare test data is time consuming while the time and resources available for testing the system are very limited. The data mutation testing method described in section 3 was applied to test the system successfully.

4.1.1 The Modelling Language

To set the background for the software under test, we characterise modelling languages in general and CAMLE language in particular. As many modern graphic modelling languages, CAMLE modelling language has the following structural features.

- *Multiple views.*

In general, a modelling language ***ML*** supports multiple views, if a model M in the language ***ML*** consists of a set $\{D_1, D_2, \ldots, D_n\}$ of diagrams of types $T_1, T_2, \ldots, T_k$, $k>1$. We write $Type(D_x)$ to denote the type of diagram D_x in a model M. Each diagram may have a number of annotated values of various data types, such as the title of the diagram in text string, an informal description of the diagram in text format, the names of the creator(s) of the diagram, version numbers, the dates of creation and modifications, and the password to protect the security of the diagram, etc.

In particular, a CAMLE model comprises a caste diagram that describes the static structure of a multi-agent system, a set of collaboration diagrams that describe how agents collaborate with each other, a set of scenario diagrams that describe typical scenarios namely situations in the operation of the system, and a set of behaviour diagrams that define the behaviour rules of the agents in the context of various scenarios.

- *Typed nodes and edges*.

In a modern modelling language, nodes and edges are often typed and represented in different graphic notations. That is, each diagram D_i of type T_j may consist of a set N_i of nodes $\{n_{i,1}, n_{i,2}, \ldots, n_{i,ui}\}$ classified into several node types $tn_{j,1}, tn_{j,2}, \ldots, tn_{j,kj}$, and a set E_i of edges $\{e_{i,1}, e_{i,2}, \ldots, e_{i,vi}\}$ classified into edge types $te_{j,1}, t_{j,2}, \ldots, te_{j,sj}$. An edge can be directed, bi-directed or undirected. An edge is usually associated with two nodes in N_i, but sometimes associated with another edge in E_i.

In CAMLE, a caste diagram contains one type of nodes that denote the castes of agents that constitute the system, and six types of edges that represent various kinds of relationships between castes including inheritance, whole-part relations and migration relations. A collaboration diagram consists of two types of nodes that denote specific agent and all agents in a caste, respectively, and one type of edges with optional annotations that represent the communication links between agents. Behaviour diagrams contain eight types of nodes and four types of edges to specify the behaviour rules. Elements of scenario diagrams are a subset of the elements of behaviour diagrams.

- *Typed and formatted annotations on nodes and edges*.

Similar to many other graphic modelling languages, the text annotations on various types of nodes and edges in CAMLE diagrams must be in certain format or data type. For example, the data type of node/edge name is text of certain syntax. The data type of multiplicity is a numeric value or a formula of certain syntax.

- *Explicitly specified consistency constraints.*

Because different diagrams or views in a model may have overlapped or inter-related contents, a set of consistency constraints $\boldsymbol{C}=\{C_1, C_2, \ldots, C_w\}$ are explicitly defined on the modelling language. A set of diagrams must satisfy these constraints to be considered as a valid and meaningful model. In general, a consistency constraint $C(x)\in\boldsymbol{C}$ is a predicate with variable x ranges over models. For any given model M, $C(M) = true$ means that the model M is consistent with respect to the constraint. There are several types of consistency constraints, which include:

- *Intra-diagram constraints*. A consistency constraint C is said to be intra-diagram, if there is a predicate C' defined on a specific type T of diagrams in the model such that

  $C(M) \Leftrightarrow \forall D_x \in M.[Type(D_x)=T \Rightarrow C'(D_x)]$.

- *Inter-diagram constraints*. A consistency constraint C is said to be inter-diagram on type T, if there is predicate C' defined on two or more diagrams of the type T such that

  $C(M) \Leftrightarrow \forall D_x, D_y \in M.[Type(D_x)=Type(D_y)=T \Rightarrow C'(D_x, D_y)]$.

- *Inter-model constraints*. A consistency constraint C is said to be inter-model, if there is a predicate C' defined on diagrams of more than one type, say between diagrams of types T_1 and T_2, such that

  $C(M) \Leftrightarrow \forall D_x, D_y \in M.[Type(D_x)=T_1 \wedge Type(D_y)=T_2 \Rightarrow C'(D_x, D_y)]$.

In a multi-agent system, an agent can be composed from a number of other agents. To describe how a composite agent behaves, CAMLE allows the agent to be associated with a collaboration diagram to specify the communications between its component agents. This results in a hierarchical structure of collaboration diagrams that describe the system at various abstraction levels and various granularities. At the top of the hierarchy, the collaboration diagram describes the whole system as an agent and its interaction with the outside if any. Therefore, consistency between different levels of abstraction as well as on the overall

structure of the hierarchy must be maintained.

- *Global/local consistency constraints.* A consistency constraint C is called a global constraint, if it is defined on the whole hierarchical structure of diagrams. Otherwise, it is called a *local constraint.* The above three types of constrains are local constraints.

Table 1 summarises the number of CAMLE's consistency constraints. A consistency checker is implemented as an automatic tool in the CAMLE modelling environment to ensure models are consistent before they are further processed, say, to generate formal specifications. More details about the definition of consistency constraints and implementation of consistency checker of CAMLE can be found in [78, 75].

Table 1. Summary of CAMLE's Consistency Constraints

		Horizontal Consistency	Vertical Consistency	
			Local	Global
Intra-model	Intra-diagram	10	–	–
	Inter-diagram	8	8	–
Inter-model		4	1	4

The case study is to test if the consistency checker is correctly implemented with respect to the consistency constraints $\boldsymbol{C}$. In other words, it is to test for any model M the checker reports an error if and only if there is a consistency constraint C in $\boldsymbol{C}$ such that $C(M) = false$.

4.1.2 Data Mutation Operators

As discussed in section 2, a crucial step to generate test data via data mutation is to design data mutation operators. In the case study, a data mutation operator φ is a transformation defined on the models so that when it is applied to a model M at a particular location l, a new model $M'=\varphi(M, l)$ is generated as a mutant of M. Each mutation operator aims at simulating a typical type of inconsistency errors that competent modellers may make. Therefore, the consistency checker will be tested of whether it can detect such errors. Note that the data mutation operators are designed to only generate well-formed models, i.e. models that are syntactically valid with respect to the syntax and well-formedness rules of CAMLE language. This is because a well-formedness checker imbedded in the CAMLE model constructor prevents ill-formed models from being input to the consistency checker. Thus, we exclude the ill-formed models from the mutants to be generated by the data mutation operators. Based on the features of the CAMLE modelling language, the following types of data mutation operators are identified.

- *Add node*. The operator adds a new node n of type tn into a diagram $D \in M$ when it is applied to diagram D, where tn is a node type of $Type(D)$. This may require values annotated on the node to be added, and sometimes edges to be added to link the node to the diagram.
- *Delete node*. The operator deletes a node n of type tn from diagram $D \in M$ when applied to the node n in diagram D. This may result in the deletion of associated edges and annotations on the node.
- *Change node type*. The operator changes the node type tn of node n into another node type tn', when it is applied to the node n in diagram D. This is usually realised by replacing node n with a new node of type tn', and naming the new node after n. To ensure the result diagram is still syntactically valid, the mutation may require changing/adding/deleting the associated edges and/or annotations on the node, depending on the differences between the two node types.
- *Change node annotation*. The operator changes the value v annotated on a node n to another value v' of the same type, when it is applied to the node n in diagram D. Subtypes of this mutation operator include: change a field of annotated value to a default value, increase or decrease a numeric value by a small deviation, change a field of annotated value to another value of the same type that occurs in the diagram or model, change a field of annotated value to another value not occurred in the diagram at random, etc. In particular, when it changes the name of a node, it may have the effect of merging two nodes into one.
- *Add edge*. The operator adds a new edge e of type te between existing nodes in a diagram $D \in M$, when it is applied to D. This may require values annotated on the edge to be added.
- *Delete edge*. The operator deletes an edge e in a diagram $D \in M$ when applied to the edge e. It will also delete the values annotated on the edge e, if any. This may result in disconnected nodes in the diagram, and if necessary such disconnected nodes are also deleted.
- *Change edge type*. The operator changes the type te of an existing edge e in a diagram $D \in M$ to another type te', when it is applied to the edge e. The realization and effect is similarly to those of "change node type".
- *Change edge annotation*. The operator changes the value annotated on an existing edge e in a diagram $D \in M$, when it is applied to the edge e. Subtypes of this mutation operator is same to those of "change node annotation".

- *Change edge direction*. The operator reverses the direction of an existing edge *e* in a diagram $D \in M$ when applied to the edge *e*, if the type of the edge allows directions.
- *Change edge association.* The operator changes the node(s) in a diagram that an edge *e* links, when it is applied to *e*.
- *Delete diagram*. The operator deletes a diagram *D* from the model *M* when applied to the diagram *D*. Meanwhile, all nodes and edges in the diagram are also deleted.
- *Add diagram*. The operator adds a diagram *D* of some type *T* to the model *M*, when it is applied to *M*. This may require adding annotations or nodes or edges to the diagram.
- *Change diagram annotation.* The operator changes a field of the values annotated on diagram *M* according to the data type of the field, when it is applied to *M*. Subtypes of this operator are similar to those of "change node annotation".

The data mutation operators are implemented in a *mutation analysis tool* that generates mutants of CAMLE models. Considering the types of language elements represented in the CAMLE modelling constructor, totally 24 data mutation operators are designed and implemented in the mutation analysis tool, as summarised in Table 2. The mutation analysis tool also classifies mutants and reports statistic data for evaluating the program under test and analysing test adequacy.

Table 2. Data mutation operators for CAMLE

No.	Operator type	Description
1	Add diagram	Add a collaboration or behaviour or scenario diagram
2	Delete diagram	Delete an existing diagram
3	Rename diagram	Change the title of an existing diagram
4	Add node	Add a node of some type to a diagram
5	Add node with edge	Add a node and link it to an existing node with an edge
6	Add edge	Add an edge of some type to a diagram
7	Replicate node	Replicate an existing node in a diagram
8	Delete node	Delete an existing node in a diagram
9	Rename node	Rename an existing node in a diagram
10	Change node type	Replace an existing node in a diagram with a new node of another type
11	Add sub diagram	Generate a sub-collaboration diagram for an existing node in a diagram
12	Delete environment node	Delete an existing environment node in a sub-collaboration diagram
13	Rename environment node	Rename an existing environment node in a sub-collaboration diagram
14	Delete node annotation	Remove an annotation on an existing node in a diagram
15	Replicate edge	Replicate an existing non-interaction edge in a diagram
16	Delete edge	Delete an existing edge in a diagram
17	Change edge association	Change the Start or End node of an existing edge in a diagram
18	Change edge direction	Reverse the direction of an existing edge in a diagram
19	Change edge type	Replace an existing edge in a diagram with a new edge of another type
20	Replicate interaction edge	Replicate an existing interaction edge in a diagram without Action List
21	Replicate interaction	Replicate an existing interaction edge in a diagram with Action List
22	Change edge annotation	Change the Action List annotated to an existing interaction edge in a diagram
23	Delete edge annotation	Delete the Action List of an existing interaction edge in a diagram
24	Change edge end to environment	Change the Start or End node of an existing edge in a diagram to an environment node

4.2 The Seed Test Cases and Their Mutants

In a number of case studies of agent-oriented software development methodology, we have constructed a number of CAMLE models of various types of agent-based systems. Three of these models were used as the seeds in the case study of data mutation testing. These models are the evolutionary multi-agent Internet information retrieval system Amalthaea, online auction web service, and the agent-oriented model of the United Nation's Security Council. Amalthaea model describes the Amalthaea system, which is an evolutionary multi-agent system developed at MIT Media Lab to help the users to retrieve information from the Internet [79]. The Amalthaea model we constructed with CAMLE was presented in [80, 81]. The model of online auction studied in [82] describes the architecture of web services and an application of web services to online auctions. The model of United Nations' Security Council (UNSC) describes the organisational structure and the work procedure to pass resolutions at UNSC.

Details of our UNSC model as well as other researchers' models in other agent-oriented modelling notations can be found on AUML's website at http://www.auml.org/. All of these three models have passed consistency check of the CAMLE modelling environment. Table 3 summarises the scales of the seed models in terms of the number of nodes, edges and diagrams they contain.

Three sets of mutants are generated from the seeds by the mutation analysis tool. The numbers of mutants generated from each seed are also given in Table 3. In the sequel, we will use Amalthaea suite, Auction suite and UNSC suite to refer to their mutant sets, respectively.

Table 3. Scales of the seed test cases

		Amalthaea	Auction	UNSC	Total
Caste Diagram	#Diagrams	1	1	1	3
	#Nodes	9	7	3	19
	# Edges	7	6	4	17
Collaboration Diagram	#Diagrams	3	5	4	12
	#Nodes	15	14	8	37
	#Edges	26	17	6	49
Behaviour Diagram	#Diagrams	8	6	2	16
	#Nodes	112	115	43	270
	#Edges	67	75	28	170
Scenario Diagram	#Diagrams	2	1	0	3
	#Nodes	22	4	0	26
	#Edges	10	1	0	11
Total	#Diagrams	14	13	7	34
	#Nodes	158	140	54	352
	#Edges	110	99	38	257
Number of Mutants		3466	3260	1082	7808

4.3 Fault Detecting Ability

This section reports the experiment that assesses the fault detecting ability of the test method. We will first present the experimental data on the method's ability of detecting implementation errors and then the data on detecting design errors.

4.3.1 Detecting implementation errors

To investigate the ability of the proposed test method on detecting faults in the implementation of the program under test, we conducted an experiment using error seeding. A total number of 118 faults of six types, following the classification scheme of program faults presented in [83], were manually inserted into the consistency checker. The checker with inserted faults (called faulty checker in the sequel) was tested on both the seed models and the mutants. Its output on the test suites was compared with the original checker's output. A difference between the two outputs indicates there is a fault in the faulty checker (inserted fault) or in the original checker (indigenous fault). Then, the faulty checker was examined to identify the fault. The results of the experiment are given in Table 4.

The experiment demonstrates that the set of test data generated by data mutation method, namely the mutant models, is capable of revealing various types of program faults. As the statistics in Table 4 shows, mutant models as test data to detect faults are much more effective than the original seed models although each mutant is of similar size as its seed. The remarkable advantage of the mutants over the seeds is due to: (a) the large number of mutants, and (b) the diversity of the mutants that contain all possible inconsistencies. All faults detected by the original models are also detected by the mutant models. While the faults detected by the original models are just the inserted ones, testing on the mutants revealed 5 indigenous faults in the program. Besides the error seeding experiment, manual analysis on part of the checker's output on the mutants detected 5 other faults in the program; see section 4.5.

Table 4. Results of the Error Seeding Experiment

Fault Type		No. of Inserted Faults	No. of Detected Faults		
			By seeds	By mutants	
				Inserted	Indigenous
Domain	Missing path	12	5 (42%)	12 (100%)	2

	Path selection	17	8 (47%)	17 (100%)	2
Computation	Incorrect variable	24	14 (58%)	21 (88%)	0
	Omission of statements	31	13 (42%)	31 (100%)	0
	Incorrect expression	15	9 (60%)	14 (93%)	1
	Transposition of statements	19	12 (63%)	19 (100%)	0
Total		118	61 (52%)	114 (97%)	5

4.3.2 Detecting design errors

In the case study, we found that a large number of mutants remain alive, i.e. passed consistency checking. Table 5 shows the statistics on the numbers of dead and live mutants in the three mutant sets. As shown in Table 5, the raw dead mutation scores are consistently low in all sets of mutants generated from three seeds.

Table 5. The numbers of alive and dead mutants

Seed	#Mutant	#Dead	#Alive	%Dead
Amalthaea	3466	697	2769	20.11%
Auction	3260	422	2838	12.94%
UNSC	1082	167	915	15.43%
Total	7808	1286	6522	16.47%

As discussed in section 3, there are three possible reasons or their combinations for low mutation scores. They are: (a) improper design of data mutation operators, (b) insufficient observation on the behaviour and output of the software under test, or (c) defects in the software under test. To find out which of these reasons actually caused the low mutation score, we examined the mutation scores of various data mutation operators in each test suite. For example, Table 6 shows the detailed statistics of Amalthaea suite about the mutation scores with respect to each mutation operator type. It reveals that the mutation scores of different operator types vary greatly.

Table 6. Statistics on Amalthaea test suite

Operator type	#Total	#Dead	#Live	%Dead
Add diagram	3	2	1	67%
Delete diagram	9	2	7	22%
Rename diagram	9	9	0	100%
Add node	88	14	74	16%
Combine node	61	39	22	64%
Add edge	1378	173	1205	13%
Replicate node	130	0	130	0%
Delete node	147	37	110	25%
Rename node	123	77	46	63%
Change node type	61	24	37	39%
Add sub diagram	8	8	0	100%
Delete environment node	4	4	0	100%
Rename environment node	4	4	0	100%
Delete annotation on node	39	0	39	0%
Replicate edge	22	0	22	0%
Delete edge	102	24	78	24%
Change edge association	1078	139	939	13%
Change edge direction	26	26	0	100%
Change edge type	6	6	0	100%
Replicate interaction edge	24	0	24	0%
Replicate interaction	24	0	24	0%
Change edge annotation	24	24	0	100%
Delete edge annotation	24	16	8	67%

Change link to environment	72	69	3	96%

In a further analysis of the reasons why a large proportion of mutants generated by applying some types of mutation operations remain alive, we found that the consistency checker is correctly implemented with respect to the defined consistency constraints. The problem is within the definitions of consistency constraints. They are too weak to expose certain types of problems in CAMLE models. In particular, two types of weakness in the CAMLE modelling tools can be identified.

(a) Weakness in dealing with replicated information. As shown in Table 6, the mutation scores of the following types of mutation operators are very low: "replicate node", "replicate edge", "replicate interaction edge", "replicate interaction", etc. These types of mutation operators introduce replicated information into models. However, the consistency constraints do not regard such replicated information as inconsistency and, therefore, the correctly implemented consistency checker does not report the errors in the model. Indeed, a model that contains replicated elements should be regarded as ill-formed rather than inconsistent. However, the well-formedness constraints on the CAMLE language lack the definition of such type of well-formedness. So the well-formedness checker imbedded in the CAMLE model constructor is too weak to detect such ill-formedness in input models. To overcome this weakness, the well-formedness constraints were strengthened and the well-formedness checker was modified accordingly.

(b) Dealing with lack of information. Also as shown in Table 6, the mutation scores of the following types of mutation operators are very low: "delete diagram", "delete node", "delete annotation on node", "delete edge", etc. These types of mutation operators remove certain elements from the model. Therefore, the mutants are in lack of certain information. However, the consistency constraints do not regard such lack of information as inconsistency, and again, the correctly implemented consistency checker does not report error in the model. The absence of element in a model is indeed a problem of incompleteness. To overcome this weakness, a set of completeness constraints were introduced and implemented in the CAMLE consistency checker. Note that the completeness constraints are less compulsory than consistency constraints. A model may violate a completeness constraint if the model is still under incremental construction, or is intentionally left open. Therefore, a violation of a completeness constraint is reported as a warning message by the checker.

The result of the testing leads to a revision of the CAMLE modelling tools. We strengthened the well-formedness checker with new well-formedness constraints, and improved the consistency checker by modifying 3 consistency constraints and introducing 13 new completeness constraints.

The revised consistency checker was then tested again on the same set of seeds and mutants. The revised checker significantly improved the mutation scores, as shown in Table 7. Note that the numbers of generated mutants in each suite decrease slightly. It is because the well-formedness rules of the modelling language are strengthened thus some formerly generated mutants are now syntactically invalid. The remaining live mutants were analysed and testified to be consistent.

Table 7. The statistics of alive and dead mutants with strengthened checker

Seed	#Mutant	#Dead	#Alive	%Dead
Amalthaea	3065	2692	373	87.83%
Auction	3095	2579	516	83.33%
UNSC	992	821	171	82.76%
Total	7152	6092	1060	85.18%

The experiment clearly demonstrated that the data mutation test method is capable of detecting not only faults in the implementation of the program, but also the faults due to errors made in early stages of software development such as requirement analysis and design. In the case study, it resulted in the revision on the definition of consistency constraints and thus a better performance of the checker.

4.4 Test Adequacy

The proposed approach is based on the observation that it is difficult and expensive to produce an adequate set of test cases that are of high structural complexity. The mutants are therefore generated to improve the test adequacy. Our experiments show this goal can be achieved through data mutation. The following uses two criteria to evaluate the test adequacy.

Because each type of mutants represents one type of errors that a modeller may make, a quantitative measurement of a test suite's adequacy is the coverage of the types of mutants generated from the seeds. For example, the number of mutants generated by applying the data mutation operators on scenario diagrams in the three original models are summarised in Table 8. Note that the mutation operator types listed in the left most column is a subset of the 24 mutation operator types defined in Table 2, because they are only the mutation operators applicable to scenario diagrams. From the statistics in Table 8, it can be inferred that a test set just consisting of Auction suite and UNSC suite is inadequate because operators of type 6 and type 17 generate no mutants from the scenario diagrams. Even though the original Auction model does contain a scenario diagram, using such a test set will miss some of the program's functions of checking the consistency involving scenario diagrams. When a set of original models is found inadequate due to the lack of specific elements, supplement of test data targeting at the absent elements can be made, until

the total numbers of mutants (the right most column in Table 8) are all non-zero.

Table 8. Number of mutants of scenario diagrams

Mutation operator type No.	Amalthaea	Auction	UNSC	Total
1	1	1	1	3
2	2	1	0	3
3	2	1	0	3
4	14	7	0	21
5	14	3	0	17
6	24	0	0	24
7	20	4	0	24
8	20	4	0	24
9	20	4	0	24
10	8	2	0	10
14	8	2	0	10
16	10	1	0	11
17	40	0	0	40

The second adequacy criterion used in the case study measures the coverage of the functions of the software under test. In the case study, each consistency and completeness constraint is a function that the checker must implement. A thorough testing of the checking tools must exercise all of these functions otherwise it is inadequate. The checker reports different types of error message or warning messages to indicate violation of different constraints. There are totally 19 different error messages that the early version of the checker can produce before new constraints are introduced. The revised version of the checker can report 21 error messages and 15 warning messages. If a consistency constraint is never violated in a test, there are two possible reasons. First, the test cases are not adequate so that the part of code in the checker that detects such inconsistency is not exercised in testing on these test cases. Second, the consistency constraint is incorrectly defined or implemented so that it can never be violated. The second situation did not occur in our case study. It is worth noting that whether a set of mutant test cases is adequate with regard to the function coverage criterion depending on two factors: the set of mutation operators and the set of seed test cases.

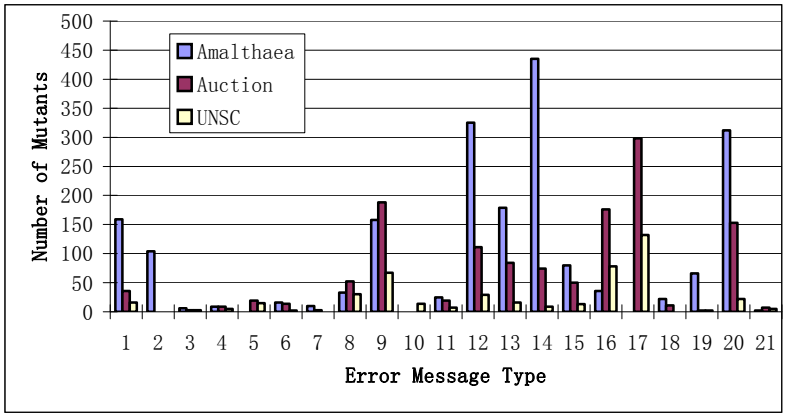

Figure 2. Distribution of Errors in Mutant Sets

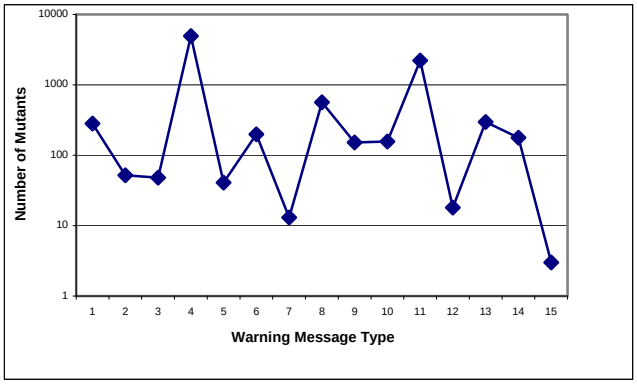

Figure 3. Logarithmic Distribution of Warnings in Mutants

Figure 2 gives the distribution of error messages produced by the checking tool on each mutant set. Figure 3 shows the logarithmic distribution of warning messages the checker produced on all mutants. Note that Figure 3 uses a log scale instead of a linear scale for visual clarity. The results shown in Figure 2 and Figure 3 demonstrate that for every type of error or warning messages, there are mutants that cause the message to be reported. Therefore, the test data have achieved 100% coverage of the functions of the consistency checker. The case study clearly demonstrates that the test method can satisfy the test adequacy.

4.5 Test Cost

The main cost of applying the test method comes from (a) the design and implementation of data mutation operators, (b) the development of seed test cases, and (c) the analysis of the correctness of the checker's output on each test case. In our case study, the seed test cases were readily available from previous case studies. The development of data mutation operators and implementation of mutation analysis tool took about 1.5 man-month work. Due to the large number of mutants, the analysis of test results on the mutants is probably the most costly part of the testing method. An experiment was conducted to find out the cost of this task. Based on the experiment, it was estimated that manually checking the correctness of the software on all mutants would take about 2 man-month efforts to complete. The following gives more details about the manual analysis of test results in the experiment.

In the experiment, we take a black-box approach to the observation of the behaviour of the software. In other words, we only observe the output of the consistency checker. When the checker discovers inconsistency or incompleteness, it gives a sequence of error or warning messages. An error/warning message describes the type of the inconsistency/incompleteness and the element(s) that causes the error. If a model passes the consistency checking, the checker will simply report that the model is consistent. Recall that, as defined in section 2, a mutant test case is dead if the behaviour of the original program on the mutant is different from the behaviour on the seed test case. In our case study, the seeds have already passed the consistency checking. Therefore, it is relatively easy to distinguish live from dead mutants because a mutant is *alive* if and only if it passes the consistency checking. The correctness of the checker on a mutant can be easily determined if the mutant is predicted to be equivalent to the seed (i.e. the checker should produce the same output on the seed and the mutant) and it remains alive in the testing. Otherwise, if a mutant is predicted to be equivalent but killed in the test, we can conclude that the checker is incorrect on the test case. Similarly, if a mutant is predicted to be non-equivalent but remains alive in test, the checker is also incorrect. However, the death of a non-equivalent mutant needs further investigation to determine if the checker is correct or not. It is because when a non-equivalent mutant is killed, the tool may be correct on the conclusion that the mutant is inconsistent, but it could still be incorrect on the decision about the consistency constraint that the mutant violates and the elements that cause the inconsistency. To determine whether the consistency checker produced a correct report on a dead mutant, we predict an expected error report and then compare it with the error messages actually output by the checker. The process of making prediction on the consistency status of a mutant model is as follows. First, we manually judge what error or warning messages should be produced by the checker by considering which consistency constraints would be violated by the mutant according to the applied mutation operator. Then, we work out how many errors and warnings would be reported according to the model elements affected by the mutation. Table 9 below shows an example of such a prediction. If the actual checking report does not meet the prediction, the implementation of the consistency checker is further examined to find the reason of the problem.

Table 9. Example of expected output of consistency checker

Mutant No.	Operator /Location	Expected Output		
		Violated Constraint	Message	#Error Messages
1	Add a new Collaboration diagram / Top of model	1	E003	5= #(Agent nodes in the main collaboration diagram)
		2	E004	6= #(Caste nodes in the main collaboration diagram)

		5	E016	14= #(Interaction edges in the main collaboration diagram)

In the experiment, we analysed two groups of mutants to examine the consistency checker's output. The first group includes 24 mutants selected at random from the Amalthaea suite, one from each set of the mutants generated by one type of mutation operator. The test with the set of 24 mutants in the Amalthaea suite exposed one fault in the implementation of the consistency checker and two faults in the other parts of the CAMLE modelling environment. The second group consists of another 22 live mutants from the Amalthaea suite. They are used to analyse whether the live mutants are actually equivalent to the seed. Two more faults were detected in the other parts of the CAMLE modelling environment. Table 10 below summarises the result of the experiment. The experiment shows that the analysis of dead mutants takes longer time, which is about 3 minutes per mutant, than the analysis of live mutants, which is about 1 minute per mutant.

Table 10. Summary of experiment results

Type of Mutant	Aliveness	#Mutants	#Detected Faults
Equivalent	Alive	34	2
Equivalent	Dead	0	0
Non-equivalent	Alive	1	1
Non-equivalent	Dead	11	2

Ideally, the output on all mutants, either dead or alive, should be analysed to decide whether the checker behaves correctly in every case. However, it is not compulsory to do so if it is too costly. A subset of the mutants can be used if the test data achieve high error detecting rate and satisfy the test adequacy.

Our experiment shows that the mutation operators and the applied locations can provide useful information on the expected output. This can significantly simplify the task of fault detecting and hence improve the efficiency of testing. The test efficiency and effectiveness as measured by the achieved test adequacy (see section 4.4) and fault detecting ability (see section 4.3) over the cost are relatively high.

5 CONCLUSION

In this paper, we present the data mutation testing method as an automatic test case generation method for testing software systems that require structurally complex input. A case study of the method is reported. The experiment reported in the paper was a real case. There was no aspect of the program under test abstracted off. No modification of the program was made to enable the case study to be conducted. The data were collected without any artificial adjustment. In the error seeding study, a non-trivial number of errors were manually inserted into the source code of the program under test following the well-established classification of program faults [83] in order to insert faults that represent real faults. The tester who conducted the experiments was also the developer of the modelling tool, thus the tester was familiar with the software under test. However, the familiarity with the software under test does not affect the fault detecting ability and test adequacy of the automatically generated test cases. It only helped in debugging the software after faults were detected. Debugging was not taken into account in the experiment in order to avoid bias. The case study on the testing method demonstrates that the method has good fault detecting ability and satisfies the test adequacy. Generally speaking, the cost efficiency of the test method is satisfactory though there is space for further improvement.

The main idea of generating test data of our method is quite similar to Meek and Siu's work on producing test cases for language processors by randomised inserting errors into an original program [67]. Our method has advantages and disadvantages similar to those stated in [67]. The advantages include labour saving because of automation of the process of generating and processing test data and collecting the results, good coverage with all likely kinds of test data, and elimination of bias compared with human-directed seeding. There are disadvantages yet, among which the main is analysing the results of test with each test case is labour-intensive. Our case study confirms the advantages and proves that the cost of analysing test results is acceptable. In our method, the knowledge about each mutant, including its seed, the applied mutation operator and the application location, helps to reduce the cost of analysing test results.

Compared with the existing mutation-inspired test case generation methods, our method is more general for testing software with structured input data. Our method emphasises the systematic design of mutation operators. It is applicable to all software systems, especially whose inputs have complex structure.

A direction of further work is to apply the method to other software systems and other application domains that have structurally complex inputs. To further improve the cost effectiveness of the technique, the test oracle problem is the key issue to be investigated. The test effectiveness is also to be improved, say, by considering the selection of seed test cases.

REFERENCES

[1] Shan, L. and Zhu, H. (2006) Testing software modelling tools using data mutation. *Proceedings of International Workshop on Automation of Software Test (AST'06) at ICSE'06*, Shanghai, China, May 23, pp. 43-49. ACM, New York, NY, USA.
[2] Howden, W. E. (1975) Methodology for the generation of program test data. *IEEE Transactions on Computers*, **24**, 554-560.
[3] Ramamoorthy, C., Ho, S. and Chen, W. (1976) On the automated generation of program test data. *IEEE Transactions on Software Engineering*, **SE-2**, 293-300.
[4] King, J. (1975) A new approach to program testing. *Proceedings of International Conference on Reliable Software*, Los Angeles, California, USA, 21-23 April, pp. 228-233. ACM, New York, NY, USA.
[5] Clarke, L. (1976) A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, **SE-2**, 215-222.
[6] Howden, W. E. (1977) Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, **SE-3**, 266-278.
[7] Howden, W. E. (1978) An evaluation of the effectiveness of symbolic testing. *Software- Practice and Experience*, **8**, 381-397.
[8] Korel, B. (1990) Automated software test data generation. *IEEE Transactions on Software Engineering*, **SE-16**, 870-879.
[9] Beydeda, S. and Gruhn, V. (2003) BINTEST - binary search-based test case generation. *Proceedings of 27th International Computer Software and Applications Conference (COMPSAC'03)*, Dallas, TX, USA, 3-6 November, pp. 28-33. IEEE Computer Society, Los Alamitos, CA, USA.
[10] Gupta, N., Mathur, A. P. and Soffa, M. L. (2000) Generating Test Data for Branch Coverage. *Proceedings of 15th IEEE International Conference on Automated Software Engineering (ASE'00)*, Grenoble, France, 11-15 September, pp. 219-228. IEEE Computer Society Press, Los Alamitos, CA, USA.
[11] Pargas, R. P., Harrold, M. J. and Peck, R. R. (1999) Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, **9**, 263-282.
[12] DeMillo, R. A., Guindi, D. S., McCracken, W. M., Offutt, A. J. and King, K. N. (1988) An extended overview of the Mothra software testing environment. *Proceedings of 2nd Workshop on Software Testing, Verification and Analysis*, Banff, Canada, 19-21 July, pp. 142-151. IEEE Computer Society, Washington, DC, USA.
[13] DeMillo, R. A. and Offutt, A. J. (1991) Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, **SE-17**, 900-909.
[14] DeMillo, R. A. and Offutt, A. J. (1993) Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering and Methodology*, **2**, 109-127.
[15] McMinn, P. (2004) Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, **14**, 105-156.
[16] McMinn, P. and Holcombe, M. (2003) The state problem for evolutionary testing. *Proceedings of Genetic and Evolutionary Computation Conference (GECCO'03)*, Chicago, USA, 12-16 July, pp. 2488-2497. Springer-Verlag, Berlin, Heidelberg.
[17] Zhu, H., Hall, P. and May, J. (1997) Software unit test coverage and adequacy. *ACM Computing Survey*, **29**, 366-427.
[18] Tai, K.-C. (1993) Predicate-based test generation for computer programs. *Proceedings of 15th International Conference on Software Engineering (ICSE'93)*, Baltimore, Maryland, USA, 17-21 May, pp. 267-276. IEEE Computer Society, Los Alamitos, CA, USA.
[19] Stocks, P. A. and Carrington, D. A. (1993) Test templates: A specification-based testing framework. *Proceedings of 15th International Conference on Software Engineering (ICSE'93)*, Baltimore, Maryland, USA, 17-21 May pp. 405-414. IEEE Computer Society, Los Alamitos, CA, USA.
[20] Ammann, P. and Offutt, J. (1994) Using formal methods to derive test frames in category-partition testing. *Proceedings of 9th IEEE Annual Conference on Computer Assurance (COMPASS'94)*, Gaithersburg, Maryland, USA, 27 June -1 July, pp. 69-80. IEEE Computer Society, Los Alamitos, CA, USA.
[21] Denney, R. (1991) Test-case generation from Prolog-based specifications. *IEEE Software*, **8**, 49-57.
[22] Bouge, L., Choquet, N., Fribourg, L. and Gaudel, M.-C. (1986) Test set generation from algebraic specifications using logic programming. *Journal of Systems and Software*, **6**, 343-360.
[23] Doong, R. K. and Frankl, P. G. (1994) The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, **3**, 101-130.
[24] Chen, H. Y., Tse, T. H. and Chen, T. Y. (2001) TACCLE: a methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering and Methodology*, **10**, 56-109.
[25] Fujiwara, S., Bochmann, G., Khendek, F., Amalou, M. and Ghedamsi, A. (1991) Test selection based on finite state models. *IEEE Transactions on Software Engineering*, **SE-17**, 591-603.
[26] Lee, D. and Yannakakis, M. (1996) Principles and methods of testing finite state machines -- a survey. *Proceedings of the IEEE*, **84**, 1090-1123.
[27] Hierons, R. M. (2001) Checking states and transitions of a set of communicating finite state machines. *Microprocessors and Microsystems*, **24**, 443-452.
[28] Morasca, S. and Pezze, M. (eds). (1990) Using high-level Petri nets for testing concurrent and real-time systems, North Holland, Amsterdam.
[29] Zhu, H. and He, X. (2002) A methodology of testing high-level Petri nets. *Information and Software Technology*, **44**, 473-489.
[30] Zhu, H., Jin, L. and Diaper, D. (1999) Application of task analysis to the validation of software requirements. *Proceedings of 11th International Conference on Software Engineering and Knowledge Engineering (SEKE '99)*, Kaiserslautern, Germany, 16-19 June, pp. 239-245. Knowledge Systems Institute, Skokie, IL, USA.

[31] Zhu, H., Jin, L. and Diaper, D. (2002) Software requirements validation via task analysis. *Journal of System and Software*, **61**, 145-169.
[32] Hartman, A. and Nagin, K. (2004) The AGEDIS tools for model based testing. *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'04)*, Boston, Massachusetts, USA, 11-14 July, pp. 129-132. ACM, New York, NY, USA.
[33] Offutt, J. and Abdurazik, A. (2000) Using UML collaboration diagrams for static checking and test generation. *Proceedings of 3rd International Conference on the Unified Modelling Language (UML '00)*, York, UK, 2-6 October, pp. 383-395. Springer-Verlag, Berlin, Heidelberg.
[34] Tahat, L. H., Bader, A. J., Vaysburg, B. and Korel, B. (2001) Requirement-based automated black-box test generation. *Proceedings of 25th International Computer Software and Applications Conference (COMPSAC'01)*, Chicago, IL, USA, 8-12 October, pp. 489-495. IEEE Computer Society, Los Alamitos, CA, USA.
[35] Vaysburg, B., Tahat, L. H. and Korel, B. (2002) Dependence analysis in reduction of requirement based test suites. *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '02)*, Roma, Italy, 22-24 July, pp. 107-111. ACM New York, NY, USA.
[36] Li, S., Wang, J. and Qi, Z.-C. (2004) Property-oriented test generation from UML statecharts. *Proceedings of 19th International Conference on Automated Software Engineering (ASE'04)*, Linz, Austria, 20-25 September, pp. 122-131. IEEE Computer Society, Los Alamitos, CA, USA.
[37] Li, S., Wang, J., Wang, X. and Qi, Z.-C. (2005) Configuration-oriented symbolic test sequence construction method for EFSM. *Proceedings of 29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, Edinburgh, Scotland, UK, 25-28 July, pp. 13-18. IEEE Computer Society, Los Alamitos, CA, USA.
[38] Pretschner, A. (2005) Model-based testing (Tutorial Abstract). *Proceedings of 27th international conference on Software engineering (ICSE'05)*, St. Louis, Missouri, USA, 15-21 May, pp. 722-723. ACM, New York, NY, USA.
[39] Pretschner, A., Prenninger, W., Wagner, S., Kühnel, C., Baumgartner, M., Sostawa, B., Zölch, R. and Stauner, T. (2005) One evaluation of model-based testing and its automation. *Proceedings of 27th International Conference on Software Engineering (ICSE'05)*, St. Louis, Missouri, USA, 15-21 May, pp. 392-401, New York, NY, USA.
[40] Heimdahl, M. P. E. (2005) Model-based testing: challenges ahead (Panel Discussion Position paper). *Proceedings of 29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, Edinburgh, Scotland, UK, 25-28 July, p330. IEEE Computer Society, Los Alamitos, CA, USA.
[41] Avritzer, A. and Larson, B. (1993) Load testing software using deterministic state testing. *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'93)*, Cambridge, Massachusetts, USA, June 28-30, pp. 82-88. ACM, New York, NY, USA.
[42] Avritzer, A. and Weyuker, E. J. (1994) Generating test suites for software load testing. *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'94)*, Seattle, WA, USA, 17-19 August, pp. 44-57. ACM, New York, NY, USA.
[43] Whittaker, J. A. and Poore, J. H. (1993) Markov analysis of software specifications. *ACM Transactions on Software Engineering and Methodology*, **2**, 93-106.
[44] Guen, H. L., Marie, R. and Thelin, T. (2004) Reliability estimation for statistical usage testing using Markov chains. *Proceedings of 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, Rennes and Saint-Malo, France, 02 - 05 November, pp. 54-65. IEEE Computer Society, Los Alamitos, CA, USA.
[45] Prowell, S. J. (2005) Using Markov chain usage models to test complex systems. *Proceedings of 38th Annual Hawaii International Conference on System Sciences (HICSS'05)*, Big Island, Hawaii, 3-6 January, pp. 318. IEEE Computer Society Los Alamitos, CA, USA.
[46] Bertolini, C., Farina, A. G., Fernandes, P. and Oliveira, F. M. (2004) Test case generation using stochastic automata networks: quantitative analysis. *Proceedings of 2nd International Conference on Software Engineering and Formal Methods (SEFM'04)*, Beijing, China, 28-30 September, pp. 251-260. IEEE Computer Society, Los Alamitos, CA, USA.
[47] Farina, A. G., Fernandes, P. and Oliveira, F. M. (2002) Measurement and empirical software engineering: Representing software usage models with stochastic automata networks. *Proceedings of 14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02)*, Ischia, Italy, 15-19 July, pp. 401-406. ACM, New York, NY, USA.
[48] Fine, S. and Ziv, A. (2003) Coverage directed test generation for functional verification using Bayesian networks. *Proceedings of 40th ACM IEEE Annual Conference on Design Automation (DAC'03)*, Anaheim, CA, USA, 2-6 June, pp. 286-291. ACM, New York, NY, USA.
[49] Yang, C. S. and Pollock, L. (1996) Towards a structural load testing tool. *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'96)*, San Diego, California, USA, 8-10 January pp. 201-208. ACM, New York, NY, USA.
[50] Zhang, J., Xu, C. and Cheung, S. C. (2001) Automatic generation of database instances for white-box testing. *Proceedings of 25th International Computer Software and Applications Conference (COMPSAC'01)*, Chicago, IL, USA, 8-12 October. IEEE Computer Society, pp. 161-165.
[51] Fisher, M., Cao, M., Rothermel, G., Cook, C. and Burnett, M. (2002) Automated test case generation for spreadsheets. *Proceedings of 22rd International Conference on Software Engineering (ICSE'02)*, Orlando, Florida, USA, 9-25 May, pp. 141-151. ACM, New York, NY, USA.
[52] Li, J. B. and Miller, J. (2005) Testing the Semantics of W3C XML Schema. *Proceedings of 29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, Edinburgh, Scotland, UK, 25-28 July, pp. 443-448. IEEE

Computer Society, Los Alamitos, CA, USA.
[53] Lee, S. C. and Offutt, J. (2001) Generating Test Cases for XML-Based Web Component Interactions Using Mutation Analysis. *Proceedings of 12th International Symposium on Software Reliability Engineering (ISSRE'01)*, Hong Kong, China, 27-30 November, pp. 200-209. IEEE Computer Society, Los Alamitos, CA, USA.
[54] Boujarwah, A. S. and Saleh, K. (1997) Compiler test case generation methods: a survey and assessment. *Information and Software Technology*, **39**, 617-625.
[55] DeMillo, R. A., Lipton, R. J. and Sayward, F. G. (1978) Hints on test data selection: Help for the practising programmer. *Computer*, **11**, 34-41.
[56] Hamlet, R. G. (1977) Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, **SE-3**, 279-290.
[57] King, K. N. and Offutt, A. J. (1991) A FORTRAN language system for mutation-based software testing. *Software--Practice and Experience*, **21**, 685-718.
[58] Budd, T. A. (1981) Mutation analysis: Ideas, examples, problems and prospects. In Chandrasekaran, B. & Radicchi, S. (eds), *Computer Program Testing*. North-Holland, Amsterdam.
[59] Howden, W. E. (1982) Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, **SE-8**, 371-379.
[60] Woodward, M. R. and Halewood, K. (1988) From weak to strong -- Dead or alive? An analysis of some mutation testing issues. *Proceedings of 2nd Workshop Software Testing, Verification and Analysis*, Banff, Canada, 19-21 July, pp. 152-158. IEEE Computer Society, Washington, DC, USA.
[61] Mathur, A. P. (1991) Performance, effectiveness, and reliability issues in software testing. *Proceedings of 15th Annual International Computer Software and Applications Conference*, Tokyo, Japan, 11-13 September, pp. 604-605. IEEE Computer Society, Los Alamitos, CA, USA.
[62] Gopal, A. and Budd, T. (1983) *Program testing by specification mutation*. TR 83-17. University of Arizona, Tucson, Arizona, USA.
[63] Woodward, M. R. (1993) Errors in algebraic specifications and an experimental mutation testing tool. *Software Engineering Journal*, **8**, 211-224.
[64] Murnane, T. and Reed, K. (2001) On the effectiveness of mutation analysis as a black box testing technique. *Proceedings of 13th Australian Software Engineering Conference (ASWEC'01)*, Canberra, Australia, 27-28 August, p12. IEEE Computer Society, Los Alamitos, CA, USA.
[65] Offutt, J. and Xu, W. (2004) Generating test cases for Web Services using data perturbation. *Proceedings of Workshop on testing, analysis and verification of web services (TAV-WEB)*, Boston, MA, USA, July, pp. 1-10. ACM, New York, NY, USA.
[66] Xu, W., Offutt, J. and Luo, J. (2005) Testing Web Services by XML perturbation. *Proceedings of 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, Chicago, Illinois, USA, 8-11 November, pp. 257-266. IEEE Computer Society, Los Alamitos, CA, USA.
[67] Meek, B. and Siu, K. K. (1989) The effective of error seeding. *SIGPLAN Notices*, **24**, 81-89.
[68] Chen, T. Y., Leung, H. and Mak, I. K. (2004) Adaptive random testing. *Proceedings of 9th Asian Computing Science Conference (ASIAN'04). LNCS 3321*, Chiang Mai, Thailand, 8-10 December, pp. 320-329. Springer-Verlag, Berlin, Heidelberg.
[69] Chen, T. Y., Kuo, F. C., Merkel, R. G. and Ng, S. P. (2003) Mirror adaptive random testing. *Proceedings of 3rd International Conference On Quality Software (QSIC'03)*, Dallas, Texas, USA, 6-7 November, pp. 4-11. IEEE Computer Society, Los Alamitos, CA, USA.
[70] Chan, K. P., Chen, T. Y. and Towey, D. (2006) Restricted random testing: adaptive random testing by exclusion. *International Journal of Software Engineering and Knowledge Engineering*, **16**, 553-584.
[71] Chan, K. P., Chen, T. Y. and Towey, D. (2006) Probabilistic adaptive random testing. *Proceedings of 6th International Conference on Quality Software (QSIC'06)*, Beijing, China, 26-28 October, pp. 274-280. IEEE Computer Society, Los Alamitos, CA, USA.
[72] Chan, F. T., Chen, T. Y., Mak, I. K. and Yu, Y. T. (1996) Proportional sampling strategy: Guidelines for software testing practitioners. *Information and Software Technology*, **38**, 775-782.
[73] Chen, T. Y., Kuo, F.-C., Tse, T. H. and Zhou, Z. Q. (2003) Metamorphic testing and beyond. *Proceedings of 11th Annual International Workshop on Software Technology and Engineering Practice (STEP'03)*, Amsterdam, The Netherlands, 19-21 September, pp. 94-100. IEEE Computer Society, Los Alamitos, CA, USA.
[74]
[75] Zhu, H. and Shan, L. (2005) Caste-centric modelling of multi-agent systems: The CAMLE modelling language and automated tools. In Beydeda, S. & Gruhn, V. (eds), *Model-driven Software Development, Research and Practice in Software Engineering, Vol. II*. Springer-Verlag, Berlin, Heidelberg.
[76] Fowler, M. and Scott, K. (2003) *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3 ed.). Addison Wesley, Boston, MA.
[77] Yourdon, E. (1988) *Modern structured analysis*. Prentice Hall, Englewood Cliffs, NJ.
[78] Shan, L. and Zhu, H. (2004) Consistency check in modeling multi-agent systems. *Proceedings of 26th International Computer Software and Applications Conference (COMPSAC'04)*, Hong Kong, China, 28-30 September, pp. 114-121. IEEE Computer Society, Los Alamitos, CA, USA.

[79] Moukas, A. (1997) Amalthaea: Information discovery and filtering using a multi-agent evolving ecosystem. *Journal of Applied AI*, **11**, 437-457.
[80] Zhu, H. (2002) Formal specification of evolutionary software agents. *Proceedings of 4th International Conference on Formal Engineering Methods (ICFEM'02)*, Shanghai, China, 21-25 October, pp. 249-261. Springer-Verlag, Berlin, Heidelberg.
[81] Shan, L. and Zhu, H. (2003) Modelling and specification of scenarios and agent behaviour. *Proceedings of IEEE/WIC conference on Intelligent Agent Technology (IAT'03)*, Halifax, Canada, 13-16 October, pp. 32-38. IEEE Computer Society, Los Alamitos, CA, USA.
[82] Zhu, H. and Shan, L. (2005) Agent-oriented modelling and specification of web services. *Proceedings of 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2005)*, Sedona, Arizona, USA, 2-4 February, pp. 152-159. IEEE Computer Society, Los Alamitos, CA, USA.
[83] Harrold, M. J., Offutt, J. A. and Tewary, K. (1997) An approach to fault modelling and fault seeding using the program dependence graph. *Journal of Systems and Software*, **36**, 273-296.