Testing Software Modelling Tools Using Data Mutation

Lijun Shan
Dept. of Computer Sci, National Univ. of Defence Tech, 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
Modelling tools play a crucial role in model-driven software development methods. A particular difficulty in testing such software systems is the generation of adequate test cases because the test data are structurally complicated. This paper proposes an approach called data mutation to generating a large number of test data from a few seed test cases. It is inspired in mutation testing methods, but differs from them in the way that mutation operators are defined and used. In our approach, mutation operators transform the input data rather than the program under test or the specification of the software. It is not a test adequacy measurement. Instead, it generates test cases. The paper also reports a case study with the method on testing a modelling tool and illustrates the applicability of the proposed method. Experiment data clearly demonstrated that the method can achieve a high test adequacy. It has a high fault detecting ability and good cost effectiveness.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging – *Testing tools*

General Terms
Measurement, Experimentation, Verification

1. INTRODUCTION
With the rapid growth of research on model-driven software development, modelling tools are developed to support various model-based software development activities, such as model construction, model-based validation and verification, model-based testing, model transformation, etc. Such modelling tools and software development environments take diagrams of a graphical notation such as UML as input. A particular difficulty in testing such software systems is the generation of adequate test cases because the input data are complicated in their structures.

For example, a modelling tool that supports software engineers to create, edit and analyse models in UML requires input data in the form of a set of diagrams of various types, such as use case diagrams, activity diagrams, sequence diagrams, etc. Each diagram may contain a set of nodes of various types and a set of edges of different types that link the nodes. Specific types of text and/or numeric values can be associated to the specific types of nodes and edges. Consistency between the diagrams must also be maintained. Generating adequate set of test cases to test a modelling tool is very difficult, labour intensive and expensive. Tremendous pressure has been placed on software testers to test such systems adequately, yet few tool supports have been offered to automate the testing activities.

In this paper, we address the problem in the generation of test cases for testing modelling tools. We propose a method that automatically generates a large number of test cases with a reasonable effort to achieve high test adequacy.

1.1 Related Work
In the past a few years, a great amount of research has been reported in the literature in the area of automatic generation of test cases. Research on *program-based test generation* methods can be back dated to 1970's, e.g., [1, 2]. In addition to these static methods, dynamic test generation methods have also been advanced for both *path-oriented* test generation, which takes certain selected paths in the program as input [3, 4], and *goal-oriented*, which aims at achieving certain test goals, such as executing certain elements in the program [5, 6] or kill a certain mutant [7, 8]. *Specification-based methods* derive test cases from formal specifications in various formalisms, such as first order logic and set theory [9], logic programs [10], algebraic specifications [11, 12, 13], finite state machines [14, 15, 16], Petri nets [17, 18], etc. *Model-based test generation* derives test cases from semi-formal models in diagrammatic notations. In [19], a hierarchy of test criteria was defined on dataflow, entity relationship and state transition diagrams. A testing tool was developed to generate test cases to meet these adequacy criteria. More recently, advances have been made in the derivation of test cases from UML diagrams [20, 21], and extended finite state machine or statecharts [22, 23, 24], 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 inputs. Further generation of test data relies on heuristic search and constraint satisfaction techniques, constraint logic programming, deductive theorem proving and model checking, etc. [25], which are the same techniques that program-based methods rely on [26]. Although significant progress has been made in the past decades, the capability of the test generators are still very limited due to the expressiveness of constraints, computational complexity of constraint solving and the complexity of the software systems [27].

Random testing methods generate test cases through random sampling over the input space. A simple method is to sample over an existing software operation profile at random. More sophisticated methods use various types of stochastic models of software usages to represent the probabilistic distributions, such as Markov chain [28, 29], stochastic automata networks [30, 31],

AST'06, May 23, 2006, Shanghai, China.

Bayesian networks [32], and so on. However, it is unclear if random testing methods are capable of generating test cases that are structurally complex, such as software models.

In addition to the research on general test generation methods, works have been reported on test generation for specific types of software systems, such as database applications [33], spreadsheets [34], XML data schemas [35], XML-based web component interactions [36], etc. These methods address the specific requirements in the testing of such systems.

The method proposed in this paper is inspired in mutation test methods, which were originally proposed in [37], investigated intensively in [38, 39, 40, 41]. It is also extended to specification-based testing in [42, 43], and more recently applied to test XML schemas [35], etc. Our approach differs from these existing works in the way that our mutation operators are defined to transform the input data, rather than the program, or specifications, or XML schemes which define the format of the input data. The existing works do not actually generate test data. Instead, they are used to measure the adequacy of an existing set of test data. In contrast, our method is a test data generation method.

A closely related work is proposed in [44]. They use randomised error seeding to evaluate the ability of language processors to detect and report errors in source programs. Our method is for testing diagrammatic modelling tools rather than textual languages. The main idea of generating test data are quite similar, namely to artificially insert errors into a *seed* test data so as to obtain a set of *mutant* test data. Therefore, the methods have similar advantages and disadvantages as stated in [44].

1.2 Organization of the Paper

The paper is organized as follows. Section 2 describes the proposed method in detail. Section 3 presents a case study with the proposed method in testing a modelling tool called CAMLE, which stands for Caste-centric Agent-oriented Modelling Language and Environment [45]. Section 4 concludes the paper with a brief discussion of the advantages of the method and directions for future work.

2. DATA MUTATION

The test method proposed in this paper is call *data mutation*, because test data are generated through mutating input data of the program under test. It is developed for testing modelling tools that take graphic models as input.

2.1 Modelling Language

Modelling languages play a central role in software development methodologies, especially in model-driven methods. A typical modelling language like UML [46], CAMLE [45] and structured modelling notations [47] usually has the following features.

- *Multiple views*. A model M consists of a set $\{D_1, \ldots, D_n\}$ of diagrams of types $T_1, \ldots, T_k$. 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, version numbers, etc.
- *Typed nodes and edges*. Each diagram D_i of type T_j may consists 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.
- *Typed annotations on nodes and edges*. Each node $n_{i,x}$ (and edge $e_{i,x}$) can be annotated with a certain set of text and/or numeric values, such as the name of the node or edge and its multiplicity.
- *Typed annotations on diagrams*. Each diagram can be annotated with a certain set of text and/or numeric values, such as the title and author of the diagram.
- *Consistency constraints.* A set of consistency constraints $\boldsymbol{C}=\{C_1, C_2, \ldots, C_w\}$ may be defined on the models. A set of diagrams must satisfy these constraints to be considered valid and meaningful. A consistency constraint $C \in \boldsymbol{C}$ is usually a predicate such that $C(M) = true$ means that the model M is consistent with respect to the constraint. There are several commonly used taxonomies of consistency constraints, which include:
  - *Intra-diagram vs. inter-diagram constraints*. A consistency constraint is said to be inter-diagrams, if it is defined on two or more diagrams. Otherwise, it is said to be intra-diagram.
  - *Intra-model vs. inter-model constraints*. A consistency constraint is said to be inter-model, if it is defined on diagrams of more than one type; otherwise, it is said to be intra-model.
  - *Global vs. local consistency constraints*. A consistency constraint is called a global constraint, if it is defined on the whole set of diagrams of a model. Otherwise, it is called a *local constraint*.
  - *Horizontal vs. vertical constraints*. For modelling languages that supports hierarchical decomposition and refinement, a consistency constraint is a horizontal constraint if it is defined between diagrams of the same abstraction level. In contrast, a vertical consistency constraint is defined on diagrams that have refinement relationships between them.

The readers are referred to [48] for more detailed discussion of the structure of modelling languages and consistency and completeness constraints.

Consistency checkers are often implemented to ensure models are well-formed and consistent before they are further processed, say, to generate code. A consistency checker is correctly implemented with respect to consistency constraints $\boldsymbol{C}$, if for all models M the checker reports an error if and only if there is a consistency constraint C in $\boldsymbol{C}$ such that $C(M) = false$.

2.2 Data Mutation Operators

The crucial step to generate test data in our method is to design data mutation operators, which are then applied on graphic models to generate test data. A data mutation operator φ is a transformation defined on the models so that when it is applied to a model M on a particular location l, a new model $M'=\varphi(M, l)$ is generated, which is called a mutant of the original model M. The following types of data mutation operators are identified.

- *Add node*. The operator adds a new node n of type tn into a diagram $D \in M$, where tn is a node type of $Type(D)$, when it is applied to diagram D. This may also require values annotated on the node to be added, and sometimes to add edges to link the node to the diagram.
- *Delete node*. The operator deletes a node n of type tn in diagram $D \in M$ when applied to the node n in diagram D.
- *Change node type*. The operator changes the node type tn of node n into another node type tn' when applied to the node n in diagram D.
- *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*.

- *Add edge*. The operator adds a new edge *e* of type *te* between existing nodes into a diagram $D \in M$, when it is applied to *D*.
- *Delete edge*. The operator deletes an edge *e* in a diagram $D \in M$. It will also delete the values annotated on the edge *e*, if any.
- *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*.
- *Change edge annotation*. The operator simply changes the value annotated to an existing edge *e* in a diagram $D \in M$.
- *Change edge direction*. The operator simply reverses the direction of an existing edge *e* in a diagram $D \in M$, if the type of the edge allows directions.
- *Change edge association*. The operator changes the node(s) in a diagram that the edge links.
- *Delete diagram*. The operator simply deletes a diagram *D* from the model *M* when applied to the diagram *D*.
- *Add diagram*. The operator adds a diagram *D* of some type *T* to the model *M*, when it is applied to *M*.
- *Change diagram annotation*. The operator simply changes a field of the values annotated on the diagram according to its data type.

Each application of a mutation operator on an original model generates a mutant model. It is required that a mutation operator will only generate mutants that are syntactically valid, provided that the original model is syntactically valid. Therefore, with one model as seed, a number of mutants can be generated. Such mutants are appropriate to be taken as test data of modelling tools because they represent varied forms of input data. As in all mutation testing methods, it is assumed of the competent developers hypothesis and coupling hypothesis [37, 39]. Each mutation operator aims at simulating a typical type of errors that competent software developers may make. The set of mutation operators should provide an adequate coverage of all possible changes at syntax level. Each mutation operator should be implemented in the way so that when it is applied to a model, it automatically recognizes the locations in the model that it can be applied and then performs the transformations to the original model on each location one by one. In general, the design of mutation operators depends on the syntax and semantics of the modelling language under investigation.

2.3 Test Process

The process of data mutation testing consists of an iterative sequence of the following activities.

(1) Prepare seed test data.
A set of initial test cases must be prepared as the seed for the generation of more test cases. Given the fact that the test cases have complex structures, it is not easy to obtain a large set of such original test cases. Fortunately, the method does not require the existence of a large number of such test cases. A small number of seeds that contains all possible types of elements of the input data will be enough.

(2) Generate mutant test data.
Given a set of seed test data, the set of specifically designed mutation operators are applied to each seed to generate a set of mutants as described in the previous section. This step can be automated by a software tool that implements the data mutation operations. 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. Given a seed, the tool will generate all possible mutants. There is no additional stopping criterion for the mutant generation process, since each mutant contains a distinct defect in the model.

(3) Execute the software system on the seeds and their mutants.
The software under test is executed on the seeds and mutant test data. The behaviours and outputs of the software on each test data are observed and recorded for further analysis. The proposed method does not depend on any specific method that the behaviour of the software is observed and recorded. In our experiments, the input/outputs of the software are observed and recorded.

(4) Classify mutants.
The mutants are classified into either *dead* or *alive* according to the recorded behaviours 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 observed different from the execution on the seed test data. Otherwise, it is classified as *alive*.

For example, when testing a model consistency checker,if the original model passes the consistency check, a mutant is dead if and only if the checker reports inconsistency of the mutant. Otherwise, it is alive.

(5) Analyse test effectiveness.
The effectiveness of a data mutation test can be analysed through various mutation scores. A mutant can be alive due to several reasons. First, the mutant can be equivalent to the original with respect to the functionality or property of the software under test. Therefore, such mutants cannot be distinguished from its original seed by the software, hence will remain alive in the testing. Because mutation operators is supposed to simulate the errors that software developers may make, a high equivalent mutant score *EMS* indicates that the mutation operators have not been well-designed to achieve their purposes. Second, a mutant that is not equivalent to the original can also be alive because of the observations on the behaviour and output of the software under test is not sufficient to detect the differences. A high live mutant score *LMS* indicates that the behaviour observation is insufficient. Hence, a better testing method needs to be applied in order to observe the differences. Third, a mutant can remain alive because the software is incorrectly designed and/or implemented so that it is incapable to differentiate the mutants from the original though they are not equivalent. If the live mutant score LMS_Φ of a particular type Φ of mutation operator is unusually high, it reveals that the program under test is not sensitive to the type of mutation, which may due to particular faults in the design or implementation of the software. The measurements are defined as follows.

$$EMS = \frac{EM}{TM}, \quad LMS = \frac{LM - EM}{TM - EM}, \quad MS = \frac{DM}{TM - EM}$$

where *TM* is the total number of mutants; *EM*, *LM* and *DM* are the numbers of equivalent, live and dead mutants, respectively.

If the effectiveness of the testing is unsatisfactory, revisions on the design and implementation of the mutation operators or the software under test should be made.

(6) Analyse the correctness of the software under test.
Test data mutation is a method of test case generation intended to discover faults in the software under test and gain confidence in

the software. Therefore, for both dead and live mutants, software's behaviour and output on each mutant should be analysed for validation and verification of the software under test.

In comparison with other test case generation methods, one of the main advantages of data mutation testing is the existence of the behaviours and outputs of the software on both the original test data and the mutants. In our case studies, we find that the knowledge of the mutation operator applied on the seed and the location of the application greatly helps the tester to focus on the difference between the behaviour and outputs of the software on the seed and its mutant. With such knowledge, it becomes much easier to identify whether the behaviours of the software on the seed and the mutant are correct or not.

(7) Analyse test adequacy.
Test adequacy analysis intends to decide whether the testing itself is well performed and adequate. As discussed in [49], such an adequacy analysis can be based on the coverage of the program code of the software under test, 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 system under test, or a combination of them. A great number of test adequacy criteria has been proposed and investigated in the literature; see e.g. [49] for a survey.

It is worth noting that the adequacy of data mutation testing depends on two main factors: the set of seed test cases and the set of data mutation operators. The inadequacy of a data mutation testing could be due to the lack of certain elements or their combinations in the seed test cases so that certain elements in the program code or functionality or subset of the input/output data space cannot be exercised by the generated mutants. It could also be because the set of mutation operators is incapable of generating a certain type of mutants that are required to achieve adequate testing. Therefore, if the test adequacy is unsatisfactory, either new seed(s) should be used or new mutation operator(s) need to be developed.

3. CASE STUDY

A case study of data mutation is carried out with the testing of the consistency checker in our agent-oriented modelling environment CAMLE [45].

3.1 The System under Test

This subsection briefly reviews the modelling language and environment CAMLE to give the background of the case study.

The CAMLE modelling language employs the multiple-view principle to model agent-based systems. It satisfies the features discussed in section 2. In particular, a model in CAMLE consists of three views: a *caste model*, a *collaboration model* and a *behaviour model*. Each view is composed of one or more types of diagrams, while each diagram may contain some types of nodes and edges, which are annotated with text and/or numeric values.

A caste model, consisting of a *caste diagram*, specifies the architecture of the system. 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, such as the inheritance and whole-part relations. A collaboration model, consisting of a set of *collaboration diagrams* at various abstraction levels and hence various granularities, describes the communications between the agents in the system. If an agent is composed of a set of component agents, a collaboration diagram is associated to the agent to specify the communications between its component agents. This results in a hierarchical structure of collaboration diagrams. At the top of the hierarchy, the collaboration diagram describes the whole system as an agent and its interaction with the outside if any. A collaboration diagram consists of two types of nodes that denote specific agent and all agents in a caste, respectively, and one type of edge with optional annotation that represents the communication link between agents. A behaviour model, consisting of a set of *behaviour diagrams* and *scenario diagrams*, defines the behaviour rules for each caste of agents. Behaviour diagrams contain 8 types of nodes and 4 types of edges to specify the behaviour rules. Scenario diagrams define the scenarios that are referred to in behaviour diagrams.

To ensure model's quality, the CAMLE language defines a set of consistency constraints so that the diagrams that represent a system from various perspectives and different abstraction levels constitute a meaningful model. Table 1 summarises the number of CAMLE's consistency constraints. A consistency checker has been implemented as a tool in the CAMLE environment to enable automated checking whether a model satisfies the constraints. If a consistency constraint is violated, the tool reports diagnostic information about the types and causes of inconsistencies in the model under check. Readers are referred to [45, 50] for the details of the consistency constraints and the implementation of the consistency checker in the CAMLE environment.

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

3.2 The Mutation Operators

Based on the syntactic structure of CAMLE language, we defined twenty- four types of mutation operators and developed a software tool to generate mutants of CAMLE models, as shown in Table 2. The tool also classifies mutants and reports statistic data for evaluating the program under test and analysing test adequacy.

In Table 2, each operator type represents a set of mutation operators of similar functions but applicable to various types of diagrams. Mutation operators of the same type are grouped together and implemented as one transformation procedure in the mutation analysis tool in order to improve the efficiency of automatic generation of mutants. Design of the mutation operators ensured that their applications produce syntactically valid mutants.

3.3 The Seed Test Cases and Mutants

In the research on agent-oriented modelling, we have conducted a number of case studies and constructed a number of CAMLE models. We take four of these models as the seeds, which are the evolutionary multi-agent Internet information retrieval system Amalthaea [51, 52, 53], online auction web service [54], agent-oriented modelling of the United Nation's Security Council (see http://www.auml.org/), and self-organised agent communities [55], respectively. Table 3 summarises the scales of the seed models.

Table 2. Mutation Operators

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

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

Table 3. Scales of the seed test cases

Seed	Number of Diagrams				Number of Mutants
	Collaboration Diagram	Behaviour Diagram	Scenario Diagram	Caste Diagram	
Amalthaea	3	8	2	1	3466
Auction	5	6	1	1	3260
Community	4	5	0	1	3244
UNSC	4	2	0	1	1082
Total	16	21	3	4	11052

3.4 Fault Detecting Ability

The case study on data mutation testing of the checker was carried out after the tool was fairly carefully tested and all know bugs were fixed. During the case study, 14 new faults in the implementation of the tools were detected. The faults were identified through the analysis on the tools' reports on the consistency of the mutants. When a report does not comply with the expected consistency status of the mutant, the tool is examined to find the errors in the implementation or design of the software. In addition to implementation errors, weaknesses in the definitions of consistency constraints were also identified through the analysis of the mutation scores. Consequently, 3 constraints were modified and 13 new constraints were introduced.

To investigate the fault detecting ability of the test method, we conducted another experiment using error seeding. A total number of 118 errors in six types [56] were manually inserted into the consistency checker. The consistency 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 on the same test data. A difference between the two outputs on a same test data indicates there is a fault either in the faulty checker (inserted fault) or in the original checker (indigenous fault). Then, the faulty checker is examined to identify the fault. The results are given in Table 4.

The experiment demonstrated 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 given in Table 4 shows, mutant models as test data to detect faults are much more effective than the original seed models. Testing on the mutants also revealed 5 indigenous faults in the program that the original seeds did not detect.

Table 4. Results of Error Seeding Experiment

Fault Type		# Inserted Faults	# Detected Faults	
			By seeds	By mutants
Domain	Missing path	12	5 (42%)	12 (100%)
	Path selection	17	8 (47%)	17 (100%)
Computation	Incorrect variable	24	14 (58%)	21 (88%)
	Omission of statements	31	13 (42%)	31 (100%)
	Incorrect expression	15	9 (60%)	14 (93%)
	Transposition of statements	19	12 (63%)	19 (100%)
Total		118	61 (52%)	114 (97%)

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

3.5 Test Adequacy

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

Because each type of mutants represents a type of errors that a modeller can make, a quantitative measurement of a test suite's adequate is the coverage of the types of mutants generated from the seeds. For example, as shown Figure 1, the mutants generated by applying the mutation operators on scenario diagrams in the Amalthaea model cover all types of mutants. However, not all

seeds are capable of generating all types of mutants. For example, a test set that only contains Auction suite, Community suite and UNSC suite is inadequate. It is in lacking of mutants that are generated by applying operators 6 and 17 on scenario diagrams. Using such a test set will miss some of the program's functions. When a set of seeds are found inadequate due to the lack of certain elements, additional seeds targeting at the absent elements can be made, until the mutants cover all types of mutants.

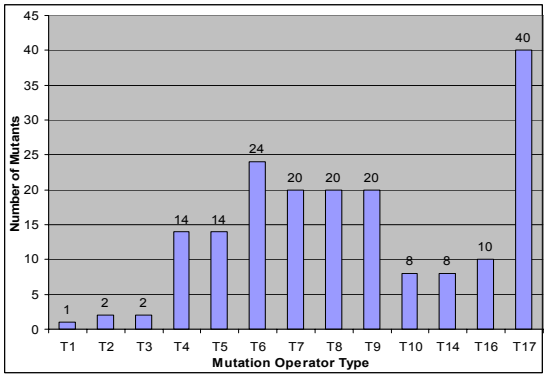

Figure 1. Distribution of Mutants of Scenario Diagrams

The coverage of mutant types helps to evaluate, but does not guarantee, the adequate coverage of the program under test. The CAMLE modelling language defines each consistency constraint as a predicate. The checker examines the consistency of a model by checking if the model satisfies the constraints one by one and reports the error according to how the constraint is violated. Therefore, for each violation of constraint, the checker may report one or more error message. Each error message has a unique identifier and represents a type of inconsistency that the checker can detect. There are totally 19 different error messages that the early version of the checker can produce, and 21 errors messages and 10 warning messages the revised version of the checker can report.

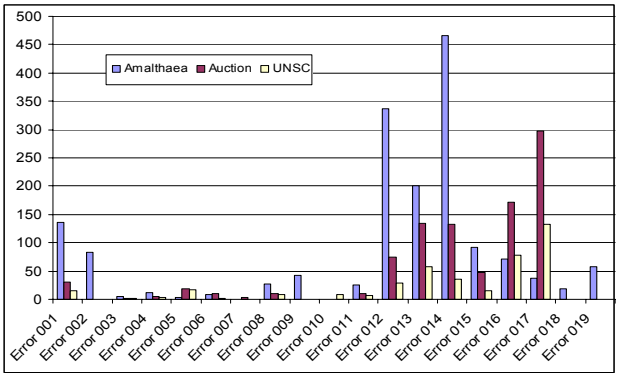

Figure 2. Distribution of Error Messages in Mutant Sets

If a consistency constraint is never violated by any mutant in a test, there are two possible reasons. (1) The set of mutants is not sufficient so that the type of inconsistency is absent in the test suite. Hence, the part of code in the checker that detects such inconsistency cannot be exercised in testing on these test cases. (2) The consistency constraint is not well defined or implemented so that it can never be violated. Therefore, the test adequacy of a test can be measured by the coverage of the types of inconsistency that the checker detects in the test. Figure 2 gives the distribution of mutants in terms of the error messages produced by the checker.

The results shown in Figure 2 demonstrated that mutant test cases caused the checker to produce every type of error messages during testing. Therefore, the test data have achieved 100% coverage of the types of inconsistency. The case study clearly demonstrated that the test method can achieve a very high test adequacy.

3.6 Test Cost

The main cost of applying the test method includes (a) the design and implementation of mutation operators and the integration of them into the modelling tools, (b) the development of seed test cases, and (c) the analysis of the correctness of the software under test on each test case. In our case study, the seed test cases were already there in previous case studies of the modelling tool. The development of 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 found out the cost of this task. Based on the experiment, it was estimated that checking the correctness of the software on all mutants manually would take about 2 man-month efforts to complete.

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 error detecting; hence improve the efficiency of testing. The test efficiency and effectiveness as measured by the achieved test adequacy (see section 3.5) and fault detecting ability (see section 3.4) over the cost can be relatively high.

4. CONCLUSION

In this paper, we proposed the data mutation testing method as an automatic test case generation method for testing software modelling tools. The case study on the testing method clearly demonstrated that the method has a high ability to detect faults and to achieve high test adequacy. Generally speaking, the cost efficiency of the test method is satisfactory though there is space to further improve it.

The testing method is presented for testing software modelling tools. However, we believe it is applicable to all software systems whose input data have complex structures. A direction of further work is to apply the method to other software systems and other application domains that have structurally complex inputs.

REFERENCES

[1] Howden, W. E., Methodology for the generation of program test data, *IEEE Trans. on Computers*, 24(5), 1975, 554-560.

[2] Clarke, L., A system to generate test data and symbolically execute programs, *IEEE TSE*, 2(3), 1976, 215-222.

[3] Korel, B., Automated software test data generation, *IEEE TSE*, 16(8), 1990, 870-879.

[4] Beydeda, S. & Gruhn, V., BINTEST - Binary Search-based Test Case Generation, *COMPSAC'03*, 28-33.

[5] Gupta, N., Mathur, A.P. & Soffa, M.L., Generating Test Data for Branch Coverage, *ASE'00*, 219-227.

[6] Pargas, R.P., Harrold, M.J. & Peck, R. R., Test data generation using genetic algorithms, *STVR* 9(4), 1999, 263-282.

[7] DeMillo, R. A. & Offutt, A. J., Constraint-based automatic test

data generation, *IEEE TSE*, 17(9), 1991, 900-909.
[8] DeMillo,R.&Offutt, J., Experimental results from an automatic test case generator, *ACM TOSEM*, 2(2), 1993, 109-127.
[9] Tai, K-C., Predicate-based test generation for computer programs, *ICSE'93*, 267-276.
[10] Denney, R., Test-case generation from Prolog-based specifications, *IEEE Software*, March 1991, 49-57.
[11] Bouge, L., Choquet, N., Fribourg, L. & Gaudel, M.-C., Test set generation from algebraic specifications using logic programming. *JSS,* 6, 1986, 343-360,
[12] Doong R.K.&Frankl, P.G., The ASTOOT approach to testing object-oriented programs, *ACM TOSEM*, 3(2), 1994, 101-130.
[13] Chen, H. Y. Tse, T. H. & Chen, T. Y., TACCLE: a methodology for object-oriented software testing at the class and cluster levels, *ACM TOSEM***,** 10(1)**,** 2001.
[14] Fujiwara, S., Bochmann, G., Khendek, F., Amalou, M. & Ghedamsi, A., Test selection based on finite state models, *IEEE TSE*, 17(6), 1991, 591-603.
[15] Lee, D. & Yannakakis, M., Principles and methods of testing finite state machines -- a survey, *Proc. of the IEEE*, 84, Aug 1996, 1090-1123.
[16] Hierons, R. M., Checking states and transitions of a set of communicating finite state machines, *Microprocessors and Microsystems*, 24(9), 2001, 443-452.
[17] Morasca, S. & Pezze, M., Using high-level Petri nets for testing concurrent and real-time systems, *Real-Time Systems, Theory and Applications*, H. Zedan ed., North Holland, 1990.
[18] Zhu, H. & He, X., A methodology of testing high-level Petri nets, *IST*, 44(8), 2002, 473-489.
[19] Zhu, H., Jin, L., Diaper, D., Software requirements validation via task analysis, *JSS*, 61(2), 2002, 145-169.
[20]Hartman, A. & Nagin, K., The AGEDIS tools for model based testing, *ISSTA '04*, 129-132.
[21]Offutt, J.& Abdurazik,A., Using UML collaboration diagrams for static checking and test generation., *UML'00*, 383-395.
[22] Tahat, L. H., Bader, A. J., Vaysburg, B., Korel, B., Requirement-Based Automated Black-Box Test Generation**,** *COMPSAC'01*, 489-495.
[23] Vaysburg, B., Tahat, L., Korel, B., Dependence analysis in reduction of requirement based test suites, *ISSTA'02*, 107-111.
[24] Li, S., Wang, J., Qi, Z-C., Property-oriented test generation from UML statecharts, *ASE'04*, 122-131.
[25] Pretschner, A., Prenninger, W., Wagner, S., Kühnel, C., Baumgartner, M., Sostawa, B., Zölch, R. & Stauner, T., One evaluation of model-based testing and its automation, *ICSE'05*, 392-401.
[26] McMinn, P., Search-based Software Test Data Generation: A Survey, *STVR*, 14(2), 2004, 105-156.
[27] McMinn, P. & Holcombe, M., The state problem for evolutionary testing, *GECCO'03*, Springer-Verlag, 2488-2497.
[28] Whittaker, J. A. & Poore, J. H., Markov analysis of software specifications, *ACM TOSEM***,** 2(1), 1993, 93-106.
[29]Prowell, S. J., Using Markov Chain Usage Models to Test Complex Systems, *HICSS'05*, p318.
[30]Bertolini, C., Farina, A. G.., Fernandes, P. & Oliveira F. M., Test Case Generation Using Stochastic Automata Networks: Quantitative Analysis, *SEFM'04*, 251-260.
[31] Farina,A.G., Fernandes, P.&Oliveira,F.M., Measurement and empirical software engineering: Representing software usage models with stochastic automata networks, *SEKE'02*,401-406.
[32] Shai Fine, S. & Ziv, A., Coverage directed test generation for functional verification using Bayesian networks, *DA'03*, 286-291.
[33] Zhang, J., Xu, C., & Cheung, S.C., Automatic Generation of Database Instances for White-box Testing, *COMPSAC'01*, 161-165.
[34] Fisher, M., Cao, M., Rothermel, G., Cook, C., & Burnett, M., Automated test case generation for spreadsheets, *ICSE'02*, 141-151.
[35] Li, J. B., & Miller, J., Testing the Semantics of W3C XML Schema, *COMPSAC'05*, 443-448
[36] Lee, S.C. & Offutt, J., Generating Test Cases for XML-Based Web Component Interactions Using Mutation Analysis, *ISSRE'01*, 200-209.
[37] DeMillo, R.A., Lipton, R.J. & Sayward, F.G., Hints on test data selection: Help for the practising programmer, *Computer*, 11, April 1978, 34-41.
[38] King, K.N. & Offutt, A.J., A FORTRAN language system for mutation-based software testing, *SPE*, 21(7), 1991, 685-718.
[39] Budd, T. A., Mutation analysis: Ideas, examples, problems and prospects, *Computer Program Testing*, Chandrasekaran, B., & Radicchi, S., (eds), North- Holland, 1981, 129-148.
[40] Howden, W.E., Weak mutation testing and completeness of test sets, *IEEE TSE*, 8(4), 1982, 371-379.
[41] Woodward, M. R. & Halewood, K., From weak to strong -- Dead or alive? An analysis of some mutation testing issues, *TAV-2*, July 1988, 152-158.
[42] Gopal, A. & Budd, T., *Program testing by specification mutation*, Technical report TR 83-17, University of Arizona, November 1983.
[43] Woodward, M.R., Errors in algebraic specifications and an experimental mutation testing tool, *SEJ*, July 1993, 211-224.
[44] Meek B., & Siu, K.K., The effective of error seeding. *SIGPLAN Notices*, 24(6), 1989, 81-89.
[45] Zhu, H. & Shan, L., Caste-Centric Modelling of Multi-Agent Systems: The CAMLE Modelling Language and Automated Tools, *Model-driven Software Development*, Beydeda, S. & Gruhn, V. (eds.), Springer, 2005, 57-89.
[46] Fowler, M., *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd Edt., Addison Wesley, 2004.
[47] Yourdon E., *Modern structured analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1989
[48] Zhu, H. & Shan, L., Well-formedness, Consistency and Completeness of Graphic Models. *UKSIM'06*, in press.
[49] Zhu, H., Hall, P. & May, J., Software unit test coverage and adequacy, *ACM Computing Survey*, 29(4), 1997, 366~427.
[50] Shan, L. & Zhu, H., Consistency Check in Modeling Multi-Agent Systems, *COMPSAC'04*, 114-121.
[51]Moukas, A., Amalthaea: Information discovery and filtering using a multi-agent evolving ecosystem, *J. Applied AI*, 11(5), 1997, 437-457.
[52] Zhu, H., Formal Specification of Evolutionary Software Agents, *ICFEM'02*, Springer LNCS 2495, 249-261.
[53] Shan, L., & Zhu, H., Modelling and specification of scenarios and agent behaviour, *IAT'03*, 32-38.
[54] Zhu, H. & Shan, L., Agent-Oriented Modelling and Specification of Web Services, *WORDS'05*, 152-159.
[55] Zhu, H. & Wang, F., Formal Analysis of Emergent Behaviours of Autonomous Agent Communities in Scenario Calculus, Sept. 2005. (*Submitted to the Journal of Autonomous Agents and Multi-Agent Systems*)
[56] Harrold, M. J, Offutt, J, A. & Tewary, K., An approach to fault modeling and fault seeding using the program dependence graph, *JSS*, 1997(3), 273-296.