2009 International Conference on Software Testing Verification and Validation

# Putting Formal Specifications under the Magnifying Glass: Model-based Testing for Validation

Emine G. Aydal[1], Richard F. Paige[1], Mark Utting[2], Jim Woodcock[1]
University of York, UK[1]
University of Waikato, New Zealand[2]

## Abstract

*A software development process is conceptually an abstract form of model transformation, starting from an end-user model of requirements, through to a system model for which code can be automatically generated. The success (or failure) of such a transformation depends substantially on obtaining a correct, well-formed initial model that captures user concerns.*

*Model-based testing automates black box testing based on the model of the system under analysis. This paper proposes and evaluates a novel model-based testing technique that aims to reveal specification/requirement-related errors by generating test cases from a test model and exercising them on the design model. The case study outlined in the paper shows that a separate test model not only increases the level of objectivity of the requirements, but also supports the validation of the system under test through test case generation. The results obtained from the case study support the hypothesis that there may be discrepancies between the formal specification of the system modeled and the problem to be solved, and that using solely formal verification methods may not be sufficient to reveal these. The approach presented in this paper aims at providing means to obtain greater confidence in the design model that is used as the basis for code generation.*

## 1 Introduction

In recent years, the focus in software development has changed from training users in the capabilities of software, to training software developers in understanding what users expect of their software [7]. The main problem in expecting users to be able to adapt to software is that software then becomes more vulnerable to flaws due to misunderstandings of human psychology. Human Factors Engineering has been one of the disciplines that gained popularity in recent years. This is in part due to the efforts of researchers aiming to bridge the gap between machine and human by considering the factors that affect humans in making decisions whilst using a piece of software [8].

These efforts have introduced changes in almost all stages of software development, from requirements analysis to testing, since the main focus has shifted from software capabilities to user goals and expectations. Therefore, the correct interpretation and complete elicitation of requirements together with domain knowledge have become crucial in order to understand the environment in which the software will serve.

Human factors manifest themselves not only in terms of end-users but also within the development process. From this perspective, viewing the software development process as an abstract form of model transformation, from a model at the user end to another model at developer/analyst end would not be unrealistic. The accuracy of such a transformation depends substantially on obtaining a correct, well-formed initial model, capturing user concerns.

A software engineering initiative that focuses on issues of correctness is the Grand Challenge Program proposed in 2003 [9]. According to this initiative, correctness of a software product is defined in terms of conformity to a formal specification, a mathematical description of the software in a language such as Z [10]. However, a design model based on the formal specification of a system cannot be declared *correct* in isolation, i.e., it can only be said to be correct with respect to the given formal specifications. Thus, the question of whether the formal specification of the system modeled at developer end is an accurate representation of the problem to be solved is still open [14].

In this paper, we contribute to addressing this question by exploring the link between formal modeling and model-based testing processes. By constructing a separate test model from requirements, we not only increase the level of objectivity on the requirements, but also generate test cases that supports the validation of the system under test in formal specification level.

Section 2 gives the high level view of the methodology proposed. Model-based testing in Alloy is explained in Section 3. Section 4 presents an example implementation of this methodology. Finally, the evaluation process together with the preliminary results and the future work are explained in Section 5 and Section 6 respectively.

## 2 Overview of the Methodology

Figure 1 shows a rough presentation of artefacts of a system when only one model is used throughout the whole software development process. The test cases are generated by using the model together with the requirements, and the test suite is run on the code produced by using the same model.

Although this approach may seem reasonable in terms of encouraging reusability in software development, it does not provide the necessary amount of objectivity required for testing. If there are faults in the model, these faults will be carried to both test cases and to code, therefore the test cases will not be able to reveal the faults in the code since they are both based on the same set of (flawed) structures and behaviours in the model.

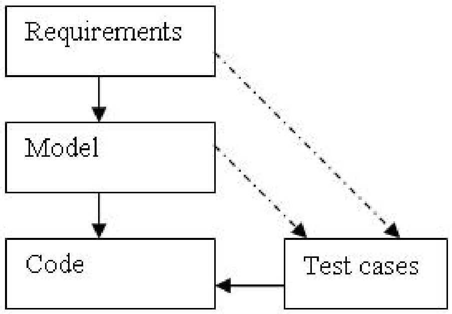

**Figure 1. Artefact generation with one model**

Instead of using one model for both code and test case generation, we propose the use of two models as shown in Figure 2, where the design model is a complete model of the system that is created with the aim of generating code or guiding the implementation, and the test model is the model of system parts that needs to be tested. As the definition suggests, the test model does not have to reflect all the system functions, i.e., it does not have to be a complete model.

The following sections explain the techniques used whilst forming a test model, generating test cases and transforming test cases to an executable form. The left hand side branch of Figure 2, i.e. the formation of the design model and code generation, is not in the scope of this study.

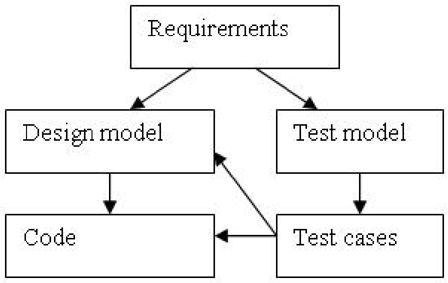

**Figure 2. Separating concerns for different artefacts**

## 3 Model-Based Testing with Alloy

Model-Based Testing (MBT) is a new and evolving technique for generating a suite of test cases from requirements [15]. Figure 3 summarises the test case generation process in Model-based testing [16, 17, 18].

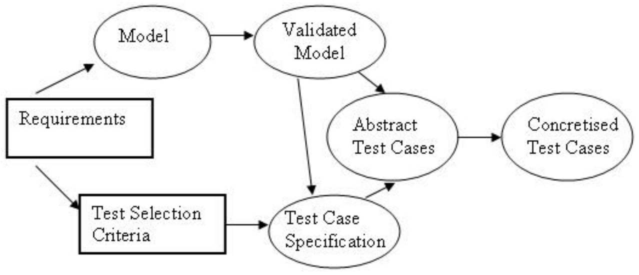

**Figure 3. Model-based Test Case Generation**

As shown in Figure 3, the first step is to build an abstract model of the SUT based on the requirements. Then, the model is validated typically via animation. *Test selection criteria* form the core of a model-based testing technique since they aim to select the test cases that are likely to detect severe and plausible failures at an acceptable cost. In other words, the effectiveness of the test cases heavily depend on the choice of test selection criteria. Once these criteria are defined, they are then transformed into *test case specifications*, i.e., a format that is able to communicate with the model produced. Given the model and the test case specifications, abstract test cases are generated [19, 20, 16, 18].

In the literature, one of the main approaches applied in MBT is to use a commonly accepted modelling language such as UML. Different techniques make use of different diagrams and analyse the information absorbed by these diagrams in their own distinct way. Some of the techniques using UML as the source for test case generation are discussed in [21, 22, 23]. There are other techniques using other modelling languages such as AutoFocus with CLP [25], Simulink Models with Classification Trees [24], etc. The test cases generated by most of the techniques are either in abstract form or concretised into test scripts written in a programming language. For the former approach, there is a need for an adaptor that can translate the format that encapsulates abstract test cases to some programming language. The techniques that create test scripts in a certain programming language limit their usage only to systems implemented in that specific language. This paper introduces a technique that not only allows the user to perform testing both in code and model level, but also gives the user the flexibility to adapt the technique to other modelling/programming languages through different adaptors.

Figure 4 presents a high level view of test case generation process introduced in this paper. The process starts with the modelling of system requirements in Alloy Modelling Language [11]. The reason for choosing Alloy as a modelling language is based on the results of our previous experiments presented in [26]. We validate this model by animating the successful execution of the operations under test. It is important to note that it is not necessary to have a complete model of the system unless we would like to validate all the operations of the system. In other words, it is possible to perform modular testing by only modelling the required parts of the system, so the test model does not have to be a refinement of the design model *per se*.

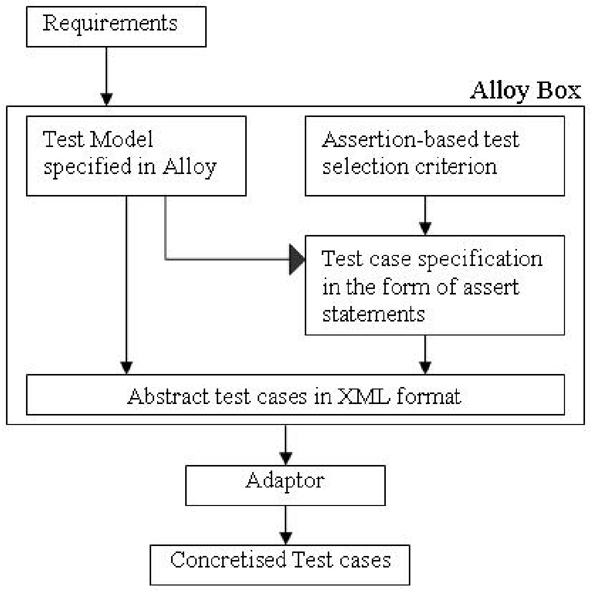
Requirements
Alloy Box
Test Model specified in Alloy
Assertion-based test selection criterion
Test case specification in the form of assert statements
Abstract test cases in XML format
Adaptor
Concretised Test cases

**Figure 4. Assertion-based Test Case Generation in Alloy**

The components of Figure 4 are explained in the rest of this section.

## 3.1 Test Selection Criteria

Test selection criteria define the conditions by which the test cases are generated. The algorithm used to select test cases, the size of the system instances used, the boundaries of the test case search space and the assumptions taken are determined during this stage. The decisions taken here are modelling language independent.

The test selection criteria applied in this study is based on the pre and postconditions of system operations specified in the test model. In order to circumscribe test selection criteria, we first define the following concepts:

- Test case
- Search space
- Size of sample set

The test cases are defined as combinations of pre-state of the System under Test (SuT), Operation under Test (OpuT), parameter values for the OpuT and post-state of the SuT.

The *search space* for test cases is mainly divided into two: those that satisfy both pre and postconditions of the OpuT and those that falsify -ideally- one precondition of the OpuT. By separating the search space into two, we not only aim to validate the system in terms of correct behaviour in the expected areas, but also catch the inconsistencies between design model and test model. These inconsistencies may be due to scope-related differences between the two models, incorrect interpretation of the requirements whilst modelling, lack of capabilities in modelling languages used in building the design model or the test model. Regardless of the cause of the inconsistent behaviour, we are certain that the existence of such behaviour shows the importance of introducing 'redundant' test models in Model-based Testing.

Another reason to include the second category in the search space, i.e. those that falsify ideally one precondition, is based on the the assumption introduced in the *Coupling effect* hypothesis. Coupling effect states that complex faults are coupled to simple faults in such a way that the test data that detects all simple faults in a program will detect the most complex faults [2]. This is also in line with the idea of semantic size of the faults introduced in [4] where Offutt et al. defined the semantic size of a fault as the difference between the possible pre-states of the system for which the post-state is valid and those for which the post-state does not satisfy the postconditions. Thus, by allowing the test case to falsify only one precondition, we keep the semantic difference between a successful execution of an operation and a failed execution of an operation to minimum. This means when the test cases generated by the test model are run on the design model or on the code generated from the design model, we either expect to see the same results, or present the differences between test model and design model in a clear manner.

Finally, the last decision to be made in test selection criteria determination is the *size of sample sets*, i.e. how big the instances of the program will be. The concern at this point is mainly to be able to challenge the instances of the system that differ in size in terms of the number of objects they include.

The method of sampling not only imposes restrictions on the size of pre-states of the system generated, but also reduces the number of test cases that satisfy test case specifications to a manageable amount. This is because, although test specifications determine certain characteristics of the test cases, the number of test cases that satisfy these characteristics may still be immense due to the size of the instances and the possible different combinations of the objects. Thus, by introducing sampling rules, we aim to:

- include groups of test cases validating the operation under test for instances of the system that vary in terms of size
- observe the response of operations when the system is in its minimal and maximum size
- introduce the randomness in test case generation to a certain degree

In light of the aims mentioned above, the samples are categorised as:

- Large: The number of objects is set to its maximum.
- Specific: The number of objects are set to a predefined value
- Random: The number of objects is not fixed.
- Minimum: The number of objects is set to its minimum.

## 3.2 Test case specifications

Test case specifications describe the characteristics of the test cases to be generated by using the test selection criteria. If test selection criteria are considered as metamodels, the test case specifications can be regarded as instances of that model in the sense that they realise the test selection criteria in the modelling language in which the test model is modelled. As the description states, the test case specifications are language-dependent.

As shown in Figure 4, in this study, test case specifications are specified in terms of *assert* statements in Alloy. The Alloy Analyzer is a tool developed by the Software Design Group at MIT, for analyzing models written in Alloy[11, 12]. It allows the user to generate instances of invariants, animate the execution of operations and check user-specified properties. An *assert* statement, when called in Alloy Analyzer, checks the SuT in terms of satisfying the predicate specified in the statement [11].

**Test triggers**

*Test triggers* are calls to the testing environment to find a test case that is compliant with the test case specifications. Depending on the environment, test case specifications may inherently include the behaviour of test triggers, but nonetheless it is important to be able to differentiate them when there is a need.

In the Alloy Analyzer, test case specifications written as assert statements are only visible when called by *check* statements. In other words, in order to invoke the tool to find a test case that satisfies test case specifications, there has to be a *check* statement that calls the corresponding assert statement.
In addition to its main purpose, the test triggers in Alloy also provide the facility to adjust the size of the sample sets predetermined by test selection criteria.

## 3.3 Abstract Test Case Generation

As part of the technique introduced in this paper, we use the counter-example generation capability of Alloy Analyzer in generating test cases. TO be more specific, whilst instantiating the test selection criteria, we assert the negation of what is specified in the test selection criteria, and expect Alloy Analyzer to provide us what we initially aim to find.

If the analyzer cannot find a counter example that falsifies the predicate in the test case specification, it states that the assert statement *may* be valid, otherwise it presents the counter example in different views such as tree view, xml view, graph view, etc. The response of the Alloy Analyzer may be interpreted differently depending on the test selection criteria used. Table 1 presents how the search space is partitioned as test selection criteria and the corresponding test case specifications.

| **Test Case Type** | **Test Selection Criteria** | **Test Case Specification** |
|---|---|---|
| TC-PP | $pre \wedge post$ | assert $\neg(pre \wedge post)$ |
| TC-NP | $\neg pre$ | assert $\neg(\neg(pre))$ |
| TC-NPP | $\neg pre \wedge post$ | assert$(\neg(\neg pre \wedge post))$ |

**Table 1. Summary of the test case generation technique**

The test case types reflect the test case selection criteria. The first P in the name stands for precondition, the second

P for postcondition and N stands for negation. The TC-PP type of test cases present the successful execution of an operation, thus the expected response from the Alloy Analyzer is to produce a counter example to the given test case specification. If no counter example is generated, this may mean that there is an inconsistency within the test model that prohibits the satisfaction of both pre and postconditions at the same time. This inconsistency may be due to one of the following reasons:

- Conflicting pre/postcondition
- Conflicting invariants
- Too strong preconditions
- Incorrectly interpreted requirements

The TC-NP type of test cases provide test cases that do not satisfy preconditions. The idea of generating such test cases may sound strange at first, but these type of test cases are extremely useful in robustness testing of the actual system, and in finding the discrepancies between the design model and the test model.

Opposite to the expectations in TC-PP, in the search of TC-NPP test cases, we do not expect Alloy Analyzer to find any counter examples. If it does, that shows that the test model is not consistent within itself. In fact, the search for a TC-NPP type of test case can be considered as a validation activity rather than a test case search.

To conclude, there are different set of activities to be performed depending on the response received from Alloy Analyzer and the type of test case we aim to generate. The counter examples generated by Alloy have the information about the pre-state of the system, the OpuT, the parameters with which the OpuT is called, and the post-state of the system. The post-state of the system is only valuable to have for TC-PP type of test cases, since post-state is not of concern if the precondition does not hold.

As the final activity within the test-case generation process, we record these counter examples generated by Alloy Analyzer in XML form. For statistical purposes, we add a header to each test case to keep more information about the process steps whilst reaching that point.

A concrete example of model-based testing process with Alloy is given in Section 4.

## 4 Case Study: Course Assignment

The first case study in which we applied the approach is a Course Assignment System, where the students and lecturers are assigned to certain courses. The initial requirements and restrictions of the system are listed in Table 2. It is important to note that non-functional requirements are kept out of the scope of this particular case study.

| **Req. No.** | **Requirement Description** |
|---|---|
| R0 | The system consists of courses, students and lecturers. |
| R1 | Each course must be subscribed by at least one student. |
| R2 | Each course can only be subscribed by students from certain years of their degree and this information is associated to each course. |
| R3 | The total number of students for a course cannot exceed 7. |
| R4 | Only one lecturer must be assigned to each course. |
| R5 | Course ID must be unique. |
| R6 | The lecturer assigned to a course must have at least 3 years of experience. |
| R7 | A student must subscribe to at least 1 course. |
| R8 | A student cannot subscribe to more than 6 courses. |
| R9 | In his/her $4^{th}$ year, the student cannot subscribe to more than 4 courses. |
| R10 | The age of the students taking a course must be less than the age of the lecturer assigned to that course. |
| R11 | A lecturer can be assigned to 3 courses at most. |

**Table 2. Requirements of Course Assignment Software**

Figure 5 presents a general view of the model-based testing process we followed in this case study. The internal structure of *Alloy Box* is provided in Figure 4.

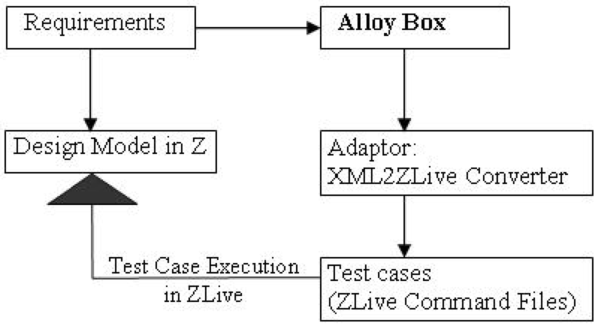
Requirements
Alloy Box
Design Model in Z
Adaptor: XML2ZLive Converter
Test Case Execution in ZLive
Test cases (ZLive Command Files)

**Figure 5. Case Study: Testing a Z Model with test cases generated by Alloy Analyzer**

The system is first modelled in the Z formal specification language [10] and checked in the ZLive Animator that provides a simple textual user interface that handles Z in LaTeX and Unicode markup [13].
In parallel to this process, the test model is produced in Alloy. As mentioned in Section 2, the test specifications are

written as assert statements according to the test selection criteria, and, then, the test triggers are called in Alloy Analyzer. To explain the technique on an example, the Alloy specifications for the operation *Subscribe* is given below.

```
pred Subscribe(d, d': DepartmentState,
               s: Student, c: Course) {
    not c in d.CourseAssignment[s]
    HasSpace [c,d]
    # d.CourseAssignment[s] <
               callMax_CourseSubscription
    IsEligible [s,c ]

    d'.CourseAssignment =
               d.CourseAssignment + s->c
    d'.TeachingAssignment =
               d.TeachingAssignment
    # d'.CourseAssignment[s] =
               #d.CourseAssignment[s] + 1
}
```

Table 3 presents the test case specifications and the test triggers for this operation. When the test triggers are run in Alloy Analyzer, it generates counter examples for TC-PP and TC-NP type of test case specifications. It is important to note that TC-NP and TC-NPP can be repeated for each different precondition, and the result to each, if obtained, would be a different test case.

Another issue that needs to be considered in forming a more complete test case is the requirements that relates to the size of the system. This is achieved by using the test triggers. The size of the instances included in the test triggers for Subscribe function are listed below:

- Maximum: exactly 2 DepartmentState, 20 Person, exactly 5 Course, 5 int
- Specific: exactly 2 DepartmentState, exactly 4 Student, exactly 3 Lecturer, exactly 3 Course, 5 int
- Random/Minimum: 5 int

The size corresponding to the *maximum* instance of the system has 20 people and 5 courses. The number of students and lecturers are decided by the analyzer during the analysis. The value given to the number of objects in this statement is determined after careful consideration of the system invariants.
The size of the instance that corresponds to *Specific* is fixed during the test case generation process. We noticed that when we do not put any limitations on the number of objects to appear in the instance, i.e. we request a random instance, Alloy finds the instance that complies with the test case specifications and has minimum number of objects. Note that in Alloy, the bidwidth for integer values is set to 3 by default, so to use integer values in a larger range, the bidwidth has to be assigned to a higher value by *run* or *check* statement.

**Test Case Specifications**

**TC-PP**

```
assert TC-PP_Subscribe
    {all d,d:DepartmentState|
    not Subscribe[d ,d]}
```

**Test Trigger**

```
check TC-PP_Subscribe for  <size>
```

**TC-NP**

```
assert TC-NP\_Subscribe\_Pre1 {
    all d,d' : DepartmentState,
        s: Student, c: Course |

    (not c in d.CourseAssignment[s] )
    or
    not (
    (HasSpace [c,d])  and
    (# d.CourseAssignment[s] <
        callMax_CourseSubscription)  and
    (IsEligible [s,c ])
    )
}
```

**Test Trigger**

```
check TC-NP_Subscribe_Pre1 for  <size>
```

**TC-NPP**

```
    assert TC-NPP_Subscribe\_Pre1 {
    all d,d' : DepartmentState,
        s: Student, c: Course |

    (not c in d.CourseAssignment[s] )
    or
    not (
    (HasSpace [c,d])        and
    (# d.CourseAssignment[s] <
        callMax_CourseSubscription)  and
    (IsEligible [s,c ])
    )

    or
    not (
    (d'.CourseAssignment =
        d.CourseAssignment + s->c) and
    (d'.TeachingAssignment =
        d.TeachingAssignment) and
    (# d'.CourseAssignment[s] =
        #d.CourseAssignment[s] + 1)
    )
    )\\
}
```

**Test Trigger**

```
check TC-NPP_Subscribe_Pre1 for  <size>
```

**Table 3. Test Case Specifications for Subscribe Operation focusing on precondition$_1$**

After running the test triggers and receiving responses from Alloy Analyzer, we record the results provided by the analyzer in XML format with a header that helps us to build statistical data.

Having the system in Z and the test cases in XML format, we identified the need for an adaptor that would ideally transform the test cases from XML format to some format that can be run on the design model. The following section explains how this is achieved.

## XML2ZLive Converter

Since we can check the consistency of the design model in ZLive, we can also check the validity of the instances of the system using ZLive. Thus, we transform the test cases in XML format to ZLive commands. This process can also be regarded as a model tranformation where test cases in XML present the source model and the ZLive commands present the target model. Kleppe et al. mention in [6] that model transformations must be done according to a transformation definition, where a transformation definition is a set of transformation rules that describe how one or more constructs in the source language can be transformed to one or more constructs in the target language.

The model transformation from XML to ZLive is not the main focus of this paper, therefore we do not present it in detail. However, as mentioned in the previous section, we define a test case as the combination of a pre-state of the SuT, the name of the OpuT, the parameter values to this operation and the post-state where applicable. Thus, during the transformation process, the test cases in XML format are first elevated to this more abstract level, and then specified in ZLive commands. In addition, this process required an additional schema to the Z model which takes the list of objects and the relations and assigns to appropriate objects in the instance of the Z Design model.

In light of this, we have implemented a Java application to carry out the XML to ZLive conversion. This tool expects the user to enter the folder that has all the XML files in it and the folder that will hold the ZLive command files. It, then, converts all the XML test case file to ZLive command files. Table 4 presents a template for a ZLive command file that is executed on the Z model to challenge the Subscribe function.

| |
|---|
| do Init |
| ; [InitReady \| lSet? = <list of lecture objects>$\wedge$ cSet? = <list of course objects>$\wedge$ sSet? = <list of student objects>$\wedge$ CAssignment? = <links from student to course > $\wedge$ TAssignment? = <links from course to lecturer> |
| ; [Subscribe \| s? = <student object>$\wedge$ c? = <course object>] |

**Table 4. Template for ZLive Command File**

## Analysis of Results

Once the ZLive commands are produced, they are run within the ZLive environment, and the results are compared to those from the Alloy Analyzer. Table 5 provides statistics about the case study and the test cases generated through the technique introduced in this paper.

| | |
|---|---|
| # of system operations | 7 |
| # of operations tested | 4 |
| # of system invariants | 9 |
| # of preconditions in Operations under Test | 13 |
| # of postconditions in Operations under Test | 12 |
| # of test cases generated | 38 |
| # of errors found | 9 |

**Table 5. Statistical information about the case study**

Each time there was a discrepancy between the results obtained from Alloy Analyzer and ZLive, we examined the situation. The errors found through such examination can be categorised as shown in Table 6.

| | |
|---|---|
| Z specification error | 4 |
| Alloy specification error | 1 |
| Alloy Analyzer-XML file generator-related error | 2 |
| Model transformation-related error | 2 |

**Table 6. Categorisation of errors**

The kind of errors found related to Z and Alloy specifications include incorrectly specified attribute range, missing or incorrect system invariants, etc.

The errors related to XML file generation capability of Alloy were due to the assumptions we made in parsing the information given in this file. We noticed some of the generalisations we made about the XML file generated by ALLOY were not valid for all. This experiment was carried out on the Alloy Analyzer version 4.0 RC11. We are aware that Alloy Analyzer developers have released a better version of the XML file in version 4.0 RC18, so we expect to see that these sort of problems will not occur in the future.

Model transformation-related errors occur due to the concepts that are expressed differently in Alloy and Z. One obvious example is the *object identifiers* assigned to objects in Alloy. In the instances generated by Alloy Analyzer, the objects are created with distinct object names, therefore even if there is no primary key, the objects are unique by default. However, whilst transforming instances to Z, we use the definition of the object (fields and values), and thus the object names disappear. This data loss is reflected in the test cases as follows: if there are two objects with the same field values, Z actually considers these two objects as one object. Thus, the loss of such information not only changes the structure of the test case completely, but also modifies the size of the instances indirectly. Since the size of an instance and the internal structure of a test case are the properties defined in test selection criteria, changing these two also means creating a completely different test case with an unknown test case specification. This is clearly an issue that needs to be avoided during model-based testing.

The ultimate goal of this study is to find the errors related to the first two categories mentioned: specification errors in design model language and test model language (Z and Alloy in this case), and to eliminate the errors that fall into the other categories as much as possible. As mentioned in Section 1, by finding such errors, we gain confidence in the design model that ideally generates the code behind the SuT, and validate the system in a more objective manner. In order to get closer to this goal, we evaluated our technique by using Mutation Testing [3, 4, 5]. Section 5 explains how Mutation testing is applied to observe the sensibility of our technique against the specification-errors.

## 5 Evaluation

Mutation Testing is a testing technique that evaluates the test quality by analysing whether a test set is able to reveal the program under test from a set of alternative programs[1]. There are two hypothesis behind Mutation Testing that supports the idea of injecting simple faults rather than a collection of faults: *Competent Programmer hypothesis* which states that competent programmers produce programs that are close to being correct and *Coupling Effect hypothesis* as previously mentioned in Section 2 [2, 3].

In addition to the research studies that are based on these two hypothesis, there are those that take into account the size of a fault, i.e. scope of the difference between a correct and incorrect version of the program under analysis [4]. This concept is categorised as *syntactic* size of a fault (number of statements or tokens that need to be changed) and semantic size of a fault (the relative size of the subdomain D for which the output mapping R is incorrect, where D⇒R represents the program). Table 7 provides information about the relation between size of a fault and error detection [4].

With these concepts in mind, we have taken the following decisions in setting up the mutation testing environment to evaluate our technique:

- Mutated specifications are produced by injecting one mutant only.

| **Syntactic Size** | **Semantic Size** | **Error Detection** | **Test case usefulness** |
|---|---|---|---|
| large | large | easy to detect | low |
| large | small | reasonably hard to detect | medium |
| small | large | generate noise | medium |
| small | small | hard to detect | high |

**Table 7. The relation between size of a fault and error detection**

- Mutant operators generate a syntactically correct specification of the system under test.
- Mutant operators are grouped in such a way that the operators replacing the original ones are in the same group of operators.
- Mutants are injected into the operations under test.
- For one mutant operator, two different mutated specifications are generated: one where the mutant operator is replaced by an operator that is semantically close, and one it is replaced by a random operator that is listed in the same category of operators.

After having defined the rules of the mutation testing applied in this study, we determined the mutant operators. There are many studies in the literature focusing on subsume relationship between the mutant operators with the aim of reducing the type of mutants to a manageable size[1, 2, 5]. In the light of these studies and the focus of this study, we fix the type of mutants to the following list of operators:

- Set Operator Replacement
  $\{\{\in, \notin\}, \{\subset, \subseteq\}, \{\cup, \cap, \setminus\}, \{\mathbb{P}, \mathbb{P}_1\}, \{\mathbb{F}, \mathbb{F}_1\}, \{\mathbb{Z}, \mathbb{N}, \mathbb{N}_1\}\}$
- Relation Operator Replacement
  $\{\{\mathrm{dom}, \mathrm{ran}\}, \{\lhd, ⩤, \rhd, ⩥\}, \{+, *\}, \{\rightarrow, ⇸, ⤔, ↣$
  $, ⤀, ↠, ⤖, ⇻, ⤕\}\}$
- Relational Operator Replacement
  $\{\{=, \neq\}, \{<, \leq, \geq, >\}\}$
- Logical Operator Replacement
  $\{\{\wedge, \vee, \Rightarrow, \Leftrightarrow\}, \{\forall, \exists, \exists_1\}\}$
- Arithmetic Operator Replacement
  $\{+, -, *, /, \%\}$

To automate the mutation injection process, we implemented a tool, that searches for the operators listed above in a given Z specification and creates a mutated version of that specification for each occurrence of the operator. The tool generated 51 mutated specifications out of which 43 are type-checked. Out of these 43 mutants, 37 of them have been killed by at least one test case. 2 of them stayed alive due to the 'large search space error' in ZLive whilst replacing $\mathbb{P}$ with $\mathbb{P}_1$. The remaining 4 mutated specifications that could not be detected were those that replaced $\leq$ with $<$ ,$\setminus$ with $\cap$ and vice versa, thus proving the difficulty of the errors that are semantically small.

## 6 Future Work and Conclusion

In this paper, we presented a novel technique that introduces a test model to generate test cases by using the assertions of the operations to invoke the counter example generation capability of Alloy Analyzer. The technique allows multi-platform usability since the test cases produced are in XML format, and there is a fixed metamodel for test cases due to the automatic generation.
In order to show the effectiveness of the testing technique in model level instead of testing on the code level, we carried out a case study where the design model is written in Z, and has been exercised by the test cases transformed from XML to ZLive commands. The results obtained from this case study has supported the hypothesis that there may be discrepancies between the formal specification/modelling of the system and the problem to be solved at user end. Thus, we believe this technique can certainly help to gain more confidence in the design model.

At the moment, we are in the process of applying the technique on other platforms with different modelling languages. We also aim to implement adaptors from XML to these modelling languages in order to increase the level of automation.

In addition to extending the applicability of the technique by using different languages, we also aim to evaluate the technique with other coverage criteria such as Condition Coverage, Modified/Condition Decision Coverage (MC/DC), etc. in order to have a better understanding of its value.

## 7 Acknowledgments

We would like to thank Hon. Antonia Johnson for providing the "William Gibbs Trust Award" which made the research visit to University of Waikato for this study possible.

## References

[1] Kim S., Clark J.A., McDermid J.A., The Rigorous Generation of Java Mutation Operators Using HAZOP, In: Proceedings of the 12th International Conference on Software and Systems Engineering and their Applications (ICSSEA'99). Paris, France.

[2] Offutt A.J., Lee A., Rothermel G., Untch R.H., Zapf C., An Experimental Determination of Sufficient Mutant Operators, ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 5 , Issue 2, Pages 99-18, 1996.

[3] Offutt A.J., Lee S.D., An Empirical Evaluation of Weak Mutation, IEEE Transactions on Software Engineering, v.20 n.5, p.337-344, 1994.

[4] Offutt A.J.,Hayes J.H., A Semantic Model of Program Faults, Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis, Pages 195 - 200, 1996.

[5] Black P.E., Okun V., Yesha Y., Mutation Operators for Specifications, Proceedings of the 15th IEEE international conference on Automated software engineering, 2000.

[6] Kleppe A., Warmer J., MDA Explained. The Model Driven Architecture: Practice and Promise. Addison-Wesley, 2003.

[7] Jackson D., Jackson M., Separating Concerns in Requirements Analysis: An Example, Rigorous Development of Complex Fault-Tolerant Systems , 210–225, 2006.

[8] Viller S., Bowers J., Rodden T., Human Factors in requirements engineering: A survey of human sciences literature relevant to the improvement of dependable systems development processes. Interacting with Computers 11(6): 665-698 (1999)

[9] Jones C., OHearn P., Woodcock J., Verified Software: A Grand Challenge, IEEE Computer Society (2006).

[10] Z Notation, http://en.wikipedia.org/wiki/Z_notation.

[11] The Alloy Analyzer, http://alloy.mit.edu/.

[12] Jackson D., Software Abstractions: Logic, Language, and Analysis, MIT Press, 2006.

[13] CZT ZLive, http://czt.sourceforge.net/zlive/index.html.

[14] Coryoth, A Case for Formal Specification, http://www.kuro5hin.org/story/2005/7/29/04553/9714, 2005.

[15] Dalal S.R., Jain A., Karunanithi N., Leaton J.M., Lott C.M., Patton G.C., Horowitz B.M., Model-based testing in practice, Proceedings of International Conference of Software Engineering ICSE, 1999.

[16] Utting M., Pretschner A., Legeard B., A taxonomy of model-based testing, Tech. report, 2006.

[17] Pretschner A., Phillips J., Methodological issues in model-based testing, Model- Based Testing of Reactive Systems, LCNS 3472 (2005), 281291.

[18] Pretschner A., Model-based testing in practice, Proceedings of Formal Methods 2005 (2005), 537541.

[19] Utting M., Position paper: Model-based testing, Verified Software: Theories, Tools, Experiments(VSTTE) (2006).

[20] Prenninger W., Pretschner A., Abstractions for model-based testing, Proc.2nd International Workshop on Test and Analysis of Component Based Systems (TACoS04) (2005), 5971.

[21] Gogolla M., Buettner F., Richters M.,USE: A UML-based specification environment for validating UML and OCL. Sci. Comput. Program. 69(1-3): 27-34 ,2007.

[22] Cavarra A., Crichton C., Davies J., Hartman A., Jeron T., Maunier L., Using UML for automatic test case generation, TACAS, 2002.

[23] Bernard E., Bouquet F., Charbonnier A., Legeard B., Peureux F., Utting M., Torreborre E., Model-based Testing from UML Models, Lecture Notes in Informatics, pp. 223230, 2006.

[24] Conrad M., Doerr H., Fey I., Model-based generation and structural presentation of test scenarios, WOrkshop on Software-Embedded Systems Testing (WSEST) (1999).

[25] Pretschner A., Slotosch O., Aiglstorfer E., Kriebel S., Model-based testing for real: The inhouse card case study, J.SoftwareTools for Technology Transfer (2004), 140157.

[26] Aydal E.G., Utting M., Woodcock J., A Comparison of State-based Modeling Tools for Model Validation, TOOLS-Europe08, Switzerland, July 2008.