

A Comparison of State-based Modelling Tools for Model Validation

Emine G. Aydal¹, Mark Utting², Jim Woodcock¹

University of York, UK¹

University of Waikato, New Zealand²

Abstract. In model-based testing, one of the biggest decisions taken before modelling is the modelling language and the model analysis tool to be used to model the system under investigation. UML, Alloy and Z are examples of popular state-based modelling languages. In the literature, there has been research about the similarities and the differences between modelling languages. However, we believe that, in addition to recognising the expressive power of modelling languages, it is crucial to detect the capabilities and the weaknesses of analysis tools that parse and analyse models written in these languages. In order to explore this area, we have chosen four model analysis tools: USE, Alloy Analyzer, ZLive and ProZ and observed how modelling and validation stages of MBT are handled by these tools for the same system. Through this experiment, we not only concretise the tasks that form the modelling and validation stages of MBT process, but also reveal how efficiently these tasks are carried out in different tools.

1 Introduction

A model is a schematic description of a system, theory or phenomenon that accounts for its known properties, and may be used for further study of its characteristics. It translates the description of the features of the tested system into a precise presentation of the expected behaviour [13].

Model-Based Testing (MBT) is a new and evolving technique for generating a suite of test cases from requirements [10]. It helps to ensure a repeatable and scientific basis for product testing, gives good coverage of all the behaviour of the product and allows tests to be linked directly to requirements [11]. MBT is also defined as the automation of the design of black-box tests [12, 11].

In the MBT context, a model serves two main purposes: it forms the basis for test-case generation and it acts as an oracle for the System Under Test (SUT). In order to fulfil these purposes, it is crucial that the modelling language in which the model is described is capable of expressing the properties expected from that model. These properties are studied under the title *specification paradigm* in [1]. Depending on the paradigm chosen, the characterisation of the specification differs, i.e., each paradigm describes the system by focusing on different aspects of the system. History-based specifications specify a system by characterising its maximal set of admissible histories whilst state-based specifications use admis-

sible system states at some arbitrary snapshots. Transition-based specifications focus on the transitions from one state to another and functional specifications describe the system as a set of structured collection of mathematical functions [1]. In MBT, the *Model Paradigm* can be described as the combination of a *specification paradigm* and a modelling language. The specification paradigm chosen determines the set of modelling languages that are able to express the properties required for that paradigm. Having said that, it is still open to discussion how successful these languages are in expressing these properties and how well the current tools fulfil our expectations in analysing the models written in these languages.

In this study, we address this issue by modelling the same system in three modelling languages that use the state-based specification paradigm and by analysing these models with four different model analysis tools. Through this study, we explore the impact of the selection of a modelling language and a tool in the modelling and validation processes. In order to achieve this, we focus on the creation of an abstract model of the SUT from informal requirements, and the validation of the model via animation, e.g., snapshot generation. The term *snapshot* means a valid, restricted, arbitrary instance of the model. Validity is checked in accordance with the system invariants. Restrictions may be introduced in order to reduce the search space and to concentrate on the area of interest. The degree of arbitrariness changes from one modelling language/tool to another, but the idea is to be able to generate an instance of the model or to animate the operation with a minimum degree of user interaction.

The next section explains the contribution of this study in further detail. Section 2 provides background information about the modelling languages and the tools used in this study. The basic version of the case study is presented in Section 3. Section 4 gives the extended version of the case study, explains the expectations from the tools, and the results obtained. Finally, the experiment is summarised in Section 5.

1.1 Contribution

The modelling languages covered in this study are UML enriched with OCL, Alloy and Z. All of these languages are classified under the state-based specification paradigm. They all model the system as a collection of variables, declare the invariants that the system must satisfy and define the operations of the system by its pre- and post-conditions [8]. Although they seem to have similar, if not the same, targets, these languages differ a great deal in terms of their syntax and analysis. In the literature, there has been research stating the differences between UML and Alloy [6, 7] and between UML, Z and Alloy [5]. However, these studies mainly focus on the languages and not the tools that parse and analyse these languages.

In this experiment, we model a Course Assignment System in all these three languages by using analysis tools that support these languages. The motivation behind this experiment can be summarised as follows:

- We use this experiment as a magnifying glass on the modelling and validation stages of model-based testing. Through this study, we clarify the tasks carried out during these stages for different modelling languages.
- By concretising the tasks during modelling and validation phases, we also reveal the expectations and the capabilities of the tools that analyse the models.
- In addition to these, in traditional MBT, the general tendency whilst validating an operation is to start from an input state and expect the tool to generate an output state. The drawback of this approach is that the tester is responsible for finding the initial valid state in order to carry out the rest of the process. In this paper, we demonstrate an extension of this approach, where possible, by generating both the input and the output state that represent the operation’s execution.
- Furthermore, all the languages studied in this experiment have different degrees of formalism and they are generally being used by different groups of people in modelling, validating, and testing different systems. However, these communities may not be aware of the capabilities of the languages/tools that are used by other communities and they may not follow the improvements that occur in one another. The lack of such knowledge may have several impacts. For instance, the members of one community may not be able to see the benefits and the power of other modelling languages or analysis tools. Therefore the usage of tools may not be efficient and the perspective necessary to develop better testing tools may be limited.

Within this context, we take an example system and use all these different tools to model and validate this system. Thus, this study not only concretises the tasks to be done, but also reports the difficulties and the advantages of modelling and validation by using the tools such as Alloy Analyzer, ProZ, ZLive and USE. By doing so, we contribute in bridging the gap between formal, semi-formal and perhaps non-formal environments.

2 Background

In this section, we give a brief overview of the modelling languages and the tools used in this study.

UML Unified Modeling Language, OMG’s most-used specification, offers a rich set of notations to model application structure and architecture as well as business processes and data structures [2]. When used together with the Object Constraint Language (OCL), it also provides a means to describe model behaviour and metamodel constraints. There have been many studies that use UML and OCL in different stages of MBT, ranging from model validation to test case generation [3, 4]. After the popularity of UML/OCL is raised, many tool developers produced tools with different capabilities. Some examples to these tools are the OCL Compiler from the University of Dresden (OCLCUD), **UML Specification Environment (USE)** by Mark Richters in the University of Bremen, the OCL Compiler, produced

by Cybernetic Intelligence GMBH and KeY by the University of Karlsruhe. After careful consideration, we decided to use the USE tool [17, 18] in this study, especially due to its capabilities in generating automatic snapshots of the system and in validating pre-/post-conditions through scenarios in addition to its ability to verify the system invariants.

Z is a formal specification language used for describing and modelling computing systems [14]. It is based on the standard mathematical notation used in axiomatic set theory, first-order predicate logic and lambda calculus. We used two analysis tools in this experiment to parse and analyse the Z model of our case study: ProZ and ZLive. The **ZLive** animator is part of the CZT project, which provides a framework for building formal methods tools, especially for the Z specification language [19]. It provides a simple textual user interface that handles Z in LaTeX and Unicode markup. **ProZ** is an extension of the ProB animator and model checker to support Z specifications. It uses the *Fuzz* type checker by Mike Spivey for extracting the formal specification from a L^AT_EXfile [9].

Alloy is a simple structural modelling language based on first-order logic [15, 16]. Alloy is similar to OCL, the Object Language of UML, but it has a more conventional syntax and a simpler semantics, and is designed for automatic analysis. Alloy is a fully declarative language, whereas OCL mixes declarative and operational elements. Z was a major influence on Alloy, but unlike Z, Alloy is first order. **The Alloy Analyzer** is a tool developed by the Software Design Group at MIT, for analyzing models written in Alloy [15]. It allows the user to generate instances of invariants, animate the execution of operations and check user-specified properties.

3 Case Study : Course Assignment System

The system modeled in this experiment is a simple 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 1. This section provides the details of how the system is modeled in UML, Alloy and Z as well as the expectations from the tools at this stage. The case study is extended further in Section 4 to specify the pre/postconditions of several operations.

3.1 Modelling the static structure of the system

In this phase of the experiment, we create the *basic model* for the Course Assignment Software, i.e., the static structure and the invariants of the system. The expectation from the tool at this stage is to parse the model written by the user and to create a valid, arbitrary instance of the model –an object diagram that satisfies all the system invariants– in a reasonable amount of time. The existence of such an instance increases confidence in the model by guaranteeing that there are no conflicting invariants.

During this phase, we observed analysis tools in terms of their ability to:

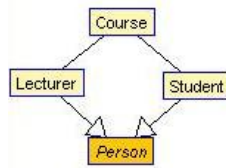
Req. No.	Requirement Description
R0	The system consists of courses, students and lecturers.
R1	Each course must be subscribed by at least one student.
R2	Each course can only be subscribed by students from certain years of their degree and this information is associated to each course.
R3	The total number 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 1. Requirements of Course Assignment Software

- create and visualise a valid instance of the model
- run with less user interaction
- perform in a reasonable amount of time
- return adequate information about the execution of the model

3.2 Modelling in UML and OCL with USE

The USE tool allows users to specify system models, invariants, and pre- and postconditions textually, and allows assertions to be checked [18]. The tool provides a multi-level platform where the model is defined in a .use file, the generation of an instance of the model is managed by an .assl file, the extra optional invariants are imposed in a .invs file and all these files as well as other USE-related commands are executed by calling .cmd files in command prompt of the tool. The class diagram of the system, as shown in Figure 1, consists of 4

**Fig. 1.** Class Diagram derived in USE

classes: Course, Lecturer, Student and Person. Lecturer and Student classes are subclasses of the abstract class Person. For each requirement listed in Table 1, an invariant is written in OCL. The generation of snapshots is mainly driven by the user through the .assl file. Following is the shortened version of the .assl file used in this study.

```

procedure AssignCourses(countCourse:Integer, countStudent:Integer,
    countLecturer:Integer)
var theConstants:SystemConstants, theCourses:Sequence(Course),
theStudents:Sequence(Student), theLecturers:Sequence(Lecturer), aCourse:Course ;
begin
    theConstants := Create(SystemConstants);
    ...    ---values of the constants are assigned
    theCourses := CreateN(Course,[countCourse]);
    for c:Course in [theCourses]
    begin
        [c].cID := Try([Sequence{1..10}
            ->reject(cID1| Course.allInstances.cID->exists(cID2|cID1=cID2))]);
    ... end;
    theStudents := CreateN(Student,[countStudent]);
    for s:Student in [theStudents]
    begin
        [s].year := Try([Sequence{1..4}]);
        ... -- Assignment of other attributes
        aCourse := Try([Course.allInstances->asSequence
            ->select(c1| Course.allInstances
                ->forAll(c2|c1.attendees->size()<=c2.attendees->size()) )]);
        Insert(Assignment,[aCourse],[s]); --link creation
    end;
    theLecturers := CreateN(Lecturer,[countLecturer]);
    for l:Lecturer in [theLecturers]
    begin
        [l].expYear := Try([Sequence{1..40}])
        ... -- Attribute assignment and Link creation with a Course Object
    end;
end;
end;

```

The order of objects generated by USE and the order of attribute assignment is explicitly stated in the file. In the above case, it is set as SystemConstant object, Course objects, Student objects and then Lecturer Objects. There are both advantages and disadvantages of this approach. The obvious advantage is that it is possible to give realistic values to the attributes. This ensures that the snapshot generated is practical. In addition to this, the user can also control the values assigned through *reject* and *if-then-else* expressions. However, there are several drawbacks of this approach as explained below.

Finding the right order of objects: It is not straightforward to determine which order produces the test cases more efficiently. To explore this more, we changed the order of the object creation and ran the generator to find a valid snapshot of the system. Table 2 shows the distinct orders associated to *variation numbers*, the number of objects in the snapshot, the number of snapshots checked before finding the valid snapshot and the number of seconds spent.

The field *Order* uses the letters S, L and C to represent Student, Lecturer and Course objects respectively. The field *Number of Objects* shows the parameters given to *AssignCourses*, i.e., number of Courses, the number of Student objects

Variation No	Order	Number of Objects	Snapshots checked	Estimated time spent(s)
1	C-S-L	(1,1,1)	5	1s
2	C-L-S	(1,1,1)	129	2s
3	L-S-C	(1,1,1)	30721	312s
4	S-C-L	(1,1,1)	5	1s
1	C-S-L	(2,2,2)	677	4s
4	S-C-L	(2,2,2)	2698	22s
5	S-C-L (*)	(2,2,2)	677	4s
1	C-S-L	(2,3,2)	677	4s
5	S-C-L (*)	(2,3,2)	Stopped after 92105	943s

Table 2. Effects of changes in order of object creation

and the number of Lecturer objects in the snapshot to be generated. Note that the *Estimated time spent* is not an output of the tool, but is calculated by the user. It is clear from the first four rows of Table 2 that the order of object creation specified in the .assl file affects the valid snapshot generation process.

Finding the right order of links: We also implemented another version of variation-4, which has the same object creation, but different link creation order. Instead of generating the instances of the associations between the objects on-the-go, in variation-5, we created all the links at the end and realised that this improved the time spent.

Invariant check: In finding the valid snapshot, USE performs a *Depth-first* search and this is why the order of object creation request becomes an issue to be considered in writing the .assl file. It checks for invariant-conformance after all the objects and links are created. We are aware that some of the invariants can actually be embedded into the .assl file in creating objects by using *reject* and *if-then-else* statements. This may shorten the time to find a valid snapshot by restricting the values assigned to attributes of these objects, but it is, in fact, a repetition of constraints that already exist in the model. We believe that an on-the-fly invariant-check would improve the performance of USE a great deal.

3.3 Modelling in Alloy with Alloy Analyzer

In our first attempt to implement the system in Alloy, we used signatures for each class in UML and defined the associations as attributes of the Course class. For instance, *cAttendees* represented the set of Students in Course class. The problem with this definition of *Course* was that Alloy is unable to analyse the invariants that require a higher-order quantification. An example of these invariants is given below where it states that a student cannot subscribe to more than 4 courses in his/her 4th year.

```
fact LastYearLimit
{all s:Student, cSet : set Course {all c:cSet |
  s in c.cAttendees and s.sYear = 4 => #cSet <= 4}}
```

In our second attempt, we created another signature called *Department* and defined the associations as relations between the classes.

```
some sig Course
{  cID: Int, cAllowedYears: set Int}
one sig Department
{  CourseAssignment : Student some -> some Course ,
   TeachingAssignment : Course -> some Lecturer}
```

After this change, the invariant above could be written in first-order logic as:

```
fact LastYearLimit
{all d:Department, s:Student {s.sYear = 4 => # d.CourseAssignment[s] <= 4}}
```

Moreover, as shown in the definition of *Department*, Alloy also supports the multiplicity concept in relations. The *CourseAssignment* relation imposes that the students should subscribe to at least one course and that the courses should have at least one attendee.

In Alloy, the *run* command generates a valid instance of a given model. One of the first observations we made whilst executing this command is that the bitwidth for integer values is set to 4 by default, i.e., the range of values that an integer attribute can take is limited to $[-8, 7]$. When there is a comparison of an integer and a value that is higher than 7, the value is rounded to this range and the comparison is made without any notification of an error or a warning. For instance, when there are 8 student objects in the system and we request the cardinality of the student objects, the result reads -8. As a workaround, we set the bitwidth for integer to 6 in the *run* command, but we believe that rounding without any notification may cause unforeseen problems in the system.

We also noticed that there is no *string* or *character* datatype. One way of implementing strings is to define a *string* signature and define it as a set of characters, which must also be declared as a signature. In order to simplify our model, we performed a data abstraction for the attributes of type string.

In terms of snapshot generation, Alloy Analyzer is found to be very powerful. Table 3 gives the results for several snapshots generated by using several SAT solvers and different numbers of objects.

No. of objects	No. of vars	No. of primary vars	No. of clauses	SAT Solver	Time spent(ms)
(1,1,1)	6625	643	19609	berkmin	1047+407ms
				sat4j	844+109ms
(2,2,4)	13639	1294	37838	berkmin	2922+625ms
				sat4j	3000+94ms
(2,4,10)	26828	2336	73231	berkmin	11797+1859ms
				sat4j	12125+343ms

Table 3. Snapshot generation with Alloy Analyzer

The numbers of objects are represented in the form of (Course, Lecturer, Student). The time spent, the number of variables, primary variables and clauses checked are given by the tool after finding the snapshot. The field *time spent* is given as an addition of two terms of which the first one represents the time whilst the tool is generating the Canonical Normal Form (CNF) of the model and the second is the duration of finding the snapshot that satisfies the invariants. As shown in Table 3, SAT Solvers have an impact on the time spent, but the fact that the user does not have to have any knowledge about the implementation of these SAT Solvers other than selecting them, makes the process easy.

Another advantage of this tool is that the user's role in the snapshot generation process is minimal. The user is not responsible for creating an extra file to manage the object or link creation. The invariant check is managed by SAT Solvers, so there is no concern as to where and when invariants are checked.

3.4 Modelling in Z with ZLive and ProZ

The Z specification of the course assignment is very similar to that of Alloy in that the Z version has the similar classes in *schema* form. The invariants of the system are embedded into the *Department* schema.

<i>Department</i>
$studentSet : \mathbb{P} Student$ $lecturerSet : \mathbb{P} Lecturer$ $courseSet : \mathbb{P} Course$ $CourseAssignment : Student \leftrightarrow Course$ $TeachingAssignment : Course \rightarrow Lecturer$
$dom\ CourseAssignment \subseteq studentSet \wedge ran\ CourseAssignment \subseteq courseSet$ $dom\ TeachingAssignment \subseteq courseSet \wedge ran\ TeachingAssignment \subseteq lecturerSet$ $\forall s : studentSet \bullet \#(\{s\} \triangleleft CourseAssignment) \leq cMaxCourseSubscription$ $\forall s : studentSet; listOfCourses : \mathbb{P} courseSet \mid$ $\quad listOfCourses = CourseAssignment(\{s\})$ $\quad \wedge s.sYear = cExceptionalYear \bullet \#listOfCourses \leq 4$ $\forall s : studentSet; c : courseSet \mid s \mapsto c \in CourseAssignment \bullet$ $\quad s.sYear \in c.cAllowedYears$ $\forall s : studentSet; c : courseSet; lec : lecturerSet \mid s \mapsto c \in CourseAssignment$ $\quad \wedge c \mapsto lec \in TeachingAssignment \bullet s.age < lec.age$ $\forall c : courseSet \bullet (\#(CourseAssignment \triangleright \{c\}) \leq cMaxAttendees)$ $\forall lec : lecturerSet \bullet lec \in ran\ TeachingAssignment \Rightarrow lec.lExpYear \geq cMinExpYear$ $\forall lec : lecturerSet \bullet (\#(TeachingAssignment \triangleright \{lec\}) \leq cMaxLecturerAssignment)$

As expected, the Z version of *Department* schema is more mathematical in terms of syntax than the other two versions. This mathematical form certainly brings formalism to Z, however it also makes comprehension more difficult for non-mathematicians. Therefore, one of the main expectations from the tools that analyse models written in L^AT_EX or some other markup version of a formal language like Z is to compensate this by providing more guidance to the user. In this respect, the tools we used, ZLive and ProZ behave differently. ZLive allows the

user to communicate with the tool via a set of commands, whereas ProZ provides a menu from which the user can choose what to do next. In terms of information given for syntactical errors, ZLive returns more explanatory information than ProZ.

In the modelling phase, we could parse and initialise the model. However, it is not possible to generate an arbitrary instance of the model other than the initial model. Note that in Z, the developer is responsible for writing a valid instance of the model to initialise the model, whereas the tools that analyse Alloy and UML could generate a valid snapshot without the user having to write the value of each attribute that needs to have a value to satisfy the system invariants.

3.5 Remarks

Table 4 outlines the factors that have an effect on the search for a valid snapshot for Alloy Analyzer and USE. Z tools are not considered here since we were only able to initialise the model with the Init schema written by the user.

Factor	USE	Alloy Analyzer
Order of Object Creation	Relevant	N/A
Order of Attribute Value Assignment	Relevant	N/A
Selection of objects before linking to another object (criteria considered at this stage)	Relevant	N/A
Loop structure of object creation	Relevant	N/A
The point in which the invariants are analysed	Relevant	N/A
The range of variables	Relevant	Relevant
Constraint Solver embedded	N/A	Relevant
The number of objects to be created	Relevant	Relevant
Limitations on formulas in higher order logic	N/A	Relevant
Default Limitations on data types	Not observable	Relevant

Table 4. Factors affecting valid snapshot generation process

In terms of general tool usage, Table 5 summarises our observations.

Criteria	USE	Alloy An.	ZLive	ProZ
Create and visualise a valid instance of the model	Yes	Yes	No	No
Generation of snapshot without extra effort	No	Yes	N/A	N/A
Perform in a reasonable amount of time	No	Yes	N/A	N/A
Return information about the model execution	Partial	Yes	N/A	N/A
Provide adequate information about the errors that occur during the modelling and execution	Yes	Yes	Yes	No

Table 5. Observations about the tool usage

4 Course Assignment - Extended Version

In the second phase of the experiment, we defined operations such as *Subscribe()*, *Unsubscribe()*, *Assign()*, *Deallocate()* through their pre- and post-conditions. We also created query operations to check the current status of the system. The main aim in the second phase was to ensure that the pre- and post-conditions associated to the operations are *realistic*. The word *realistic* implicitly contains the following statements:

- There is no pre-condition that is false for all possible instances of the system.
- There is at least one post-state that satisfies all postconditions and system invariants when the associated pre-state satisfies all pre-conditions and the system invariants.

One of the indirect objectives of this activity in the validation step of MBT is to find too strong/too weak system invariants, pre- and post-conditions.

In traditional MBT, it is generally the case that a pre-state is given as input to the tool and the tool is expected to find the associated post-state by animating the operation execution. This sort of animation is also called *forward animation* and it is certainly one way of checking the aforementioned objectives related to assertions and system invariants. Opposite of this, *backward animation* needs a post-state as input and generates a pre-state. In our approach, we followed a *non-directed* animation, where possible. Thus, the main expectation from the modelling tool at this stage was to generate a snapshot with two instances of the system where first one (pre-state) is the initial, valid instance of the model and the second one (post-state) presents the system after the execution of the operation under investigation. The advantage of this approach is that the user does not have to bring the system to the initial state by executing other operations, yet it is still possible to impose restrictions on the pre- or post-state.

In addition to this, we also analysed the modelling tools in terms of their capability in animating the operations with less user interaction, generating pre/post states of the operations and returning useful information about the execution.

4.1 Validation in UML and OCL with USE

The implementation of this phase with USE consists of the steps given in the first column of Table 6. The second, third and fourth columns show whether or not the tasks carried out by the user includes any *intellectual* or *procedural* work. *Intellectual* work stands for the tasks where the user has to understand the semantics of the operation and perform an action accordingly. *Procedural* work represents the administrative tasks such as writing the output of the tool into a new file, moving a file to a different location, etc.

Task-1 in Table 6 is explained in Section 3.2. Once the valid snapshot is generated, this is recorded and executed to actually bring the system into the state described by the snapshot. In the third step, the system is brought into another valid state where all the preconditions of the operation under investigation are satisfied. It is possible to combine Task-1 and Task-3 to generate a valid snapshot

Task	Command Execution	Procedural Work	Intellectual Work
1. Generate a valid initial snapshot of the system	Yes	Yes	Yes
2. Record and execute the snapshot	Yes	Yes	No
3. Prepare the snapshot for the operation to be executed (Precondition adjustment)	Yes	Yes	Yes
4. Enter the operation	Yes	No	No
5. Animate the execution of the operation	Yes	Yes	Yes
6. Exit the operation	Yes	No	No

Table 6. Validation of operations in USE

that already satisfies preconditions. For some operations, Task-3 may be void if the operation can run at any valid state of the system. We can analyse these steps further in the following .cmd file for the Course Assignment example:

```
open c:/<root>/CourseAssignment.use
gen start -b c:/<root>/CourseAssignment1.assl AssignCourses(2,2,2)
gen result
read c:/<root>/snapshot.cmd
read c:/<root>/precond_Unsubscribe.cmd
!openter Student1 Unsubscribe(Student1,Course1)
read c:/<root>/Unsubscribe.cmd
!opexit true
```

The first three lines form Task-1 in Table 6. Before executing the next command, the output of the tool is written into the file *snapshot.cmd*. At this point, we have a system where there are 2 course objects, 2 student objects and 2 lecturer objects linked in accordance with the system invariants. The assertions of the *Unsubscribe* function as written in USE is given below:

```
context Student::Unsubscribe(s: Student, c: Course) : Boolean
  pre UnsubscribePre1: s.courses->select(cID = c.cID)->size() = 1
  pre UnsubscribePre2: s.courses->size() >= 1
  post UnsubscribePost1: s.courses->select(cID = c.cID)->isEmpty()
  post UnsubscribePost2: s.courses->size() = s.courses@pre->size() - 1
```

The commands written in *precond_Unsubscribe.cmd* file ensures that the preconditions of *Unsubscribe* function are satisfied when *!openter* command is executed. The *Unsubscribe.cmd* file animates the operation and finally the postconditions of the function are verified after the *opexit* command.

To conclude, the tool provides a platform where the user can animate the execution of an operation, but it is user's responsibility to create the .assl file that generates the initial, valid instance of the model and to make sure that the preconditions are satisfied. Given the tasks are carried out, the tool animates the operation execution. The main drawback of this technique is its dependability on the user. If the assertions of the system are not satisfied on Task-6, that does not mean that there is no state that this operation is run successfully. It may well be that the user could not bring the system in the right state to execute the operation or did not implement the operation correctly.

4.2 Validation in Alloy with Alloy Analyzer

In Alloy, the operations are written as *predicates*. The implementation of *Unsubscribe* function in Alloy is given below. The preconditions ensure that there is a student assigned to a course and that the course has more than 1 attendee. Postconditions state that the student is no longer an attendee of the course, the number of attendees in that course is decremented by one and that no change is made in the *TeachingAssignment* relation that represents the assignment of lecturers to courses.

```
pred Unsubscribe(d, d': DepartmentState) {
  some s: Student, c: Course {
    c in d.CourseAssignment[s]           //pre1
    # c.~(d.CourseAssignment) >= 1       //pre2
    d'.CourseAssignment = d.CourseAssignment - s->c   //post1
    # d'.CourseAssignment[s] = #d.CourseAssignment[s] - 1 //post2
    d'.TeachingAssignment = d.TeachingAssignment}} //extra
```

In the search of pre- and post-states of the system before and after the execution of this operation, we use the following assertion:

```
assert CheckUnsubscribe{all d,d':DepartmentState| not Unsubscribe[d , d']}
```

In this assertion, we claim that the animation of *Unsubscribe* operation is not possible and expect the tool to find a counter example. The existence of counter example would mean that there is at least one state that allows the operation to run and another state that represents the system after the execution of the operation. Note that in both states, all the system invariants must be satisfied.

In Alloy, the call for assertions is managed by *check* command. In executing this command, it is possible to fix the number of objects we would like to see in the snapshot. If we do not specify this information explicitly, then the tool finds the snapshots that have less number of objects.

In the validation process with Alloy Analyzer, we observed that less amount of intellectual work is needed provided that the operation under investigation is defined correctly. There is no concept of entering or exiting an operation. Having said that, the user can still add further criteria to the *assert* statement to narrow down the search space.

4.3 Validation in Z with ZLive

ZLive provides a *command-prompt*-like platform where the user can interact with the tool by using commands that may have Z structure.

In order to facilitate the job of the user, we added *Add_ClassType_* operations into the Z specifications such as *AddStudent*, *AddLecturer*, etc. Thus, the creation of objects are done by calling such operations. For instance, to add a student into the student list, we can call the *AddStudent* schema with:

1.no constraint: *do AddStudent* which creates an arbitrary instance of the Student schema

2.some constraints: *do* [*AddStudent* | *sYear* = 4] which creates a Student object whose *sYear* value is set to 4.

3.specific values: *do*[*AddStudent* | *s?* = $\langle sYear == 4, age == 21, name == 1000 \rangle$] which assigns all the attributes of the student object

This variety gives the user the flexibility to try different channels if the desired instance cannot be produced. Especially when the search space is too large, the tool may not be able to handle a call with no restriction, thus, imposing extra constraints on the schema by using the second and third approach may reduce the search space.

In addition to *do* operator, there is also semicolon *;-* operator that can be used in the same context. The difference between *do* operator and *;* operator is that the latter takes the current state as pre-state, and produces a post-state by animating the execution of the operation. This approach is similar to ProZ as explained in Section 4.4. The *do* operator, on the other hand, does not take the current state into consideration, and, analogous to Alloy, generates a pre- and a post-state that represents the states before and after the execution of the operation under investigation. In fact, it is also possible to request the tool to find the pre-state by specifying an explicit post-state.

When an operation cannot be run for a particular pre-state, the tool returns *no solution*. An advantageous feature of the tool for such situations is the *why* command which gives more insight about why no solution could be found. However, the information given is very low level and assumes a certain level of subject-related knowledge from the user.

Another implicit benefit of the tool is that it introduces only a limited number of keywords and mainly uses Z syntax, thus, a Z-literate user would not have any difficulty interacting with the tool after having a rough look at the keywords.

4.4 Validation in Z with ProZ

ProZ identifies a schema as an operation if all variables of the state and their primed counterpart are declared in the operation, and no other schema in the specification refers to the operation [9].

An advantage of using ProZ is that it provides the user with a Graphical User Interface (GUI) where it is possible to observe the current values of the attributes, the history of the operations that are animated and the list of enabled operations at a given time, i.e., the operations whose preconditions are satisfied. When the tool parses the system model, the first and the only operation available is the *Init* operation. Init operation creates an empty set of student list, course list and lecturer list. In order to facilitate the job of the user in creating objects, we also added operations such as *AddStudent*, *AddLecturer*, etc.

The drawback of the GUI is that the tool actually attempts to list all the possible calls with the parameters in the accepted range as enabled operations. Since the number of such calls is high and both a timeout value and an animation setting limit the number of calls listed, the user is given only some of the enabled operations with a set of parameters. We noticed that the *timeout* button helps

to retrieve the commands that are not visible in the Enabled Operation section by bypassing the timeout value, however this sometimes causes the application to get into a non-responsive mode which cannot be interrupted (other than by killing the application).

Although the idea of guiding the user by showing the enabled operations is sensible, this actually takes the freedom of choosing the parameters of the function from the user. It also makes it impossible to start from an arbitrary, valid state other than the *Init* state. For instance, in order to test the behaviour of *Subscribe* function when there are n Students, m Courses in the system, we need to call the *AddStudent* operation n times and *AddCourses* operation m times. In other words, it is not possible to create a valid, initial snapshot with n Students and m Courses with one command. This means that the user would not know whether it is possible to have a valid instance of the system with that many objects without actually going through the process of creating these objects. This is an obvious restriction especially if the tool is to be used in test-case generation.

5 Conclusion

In this paper, we focused on the modelling and validation steps of MBT. The tasks included in these steps are analysed in the scope of different model analysis tools. We modeled a Course Assignment system by using three state-based modelling languages, namely Z, Alloy and UML, and analysed the models by using four different tools: USE, Alloy Analyzer, ProZ and ZLive.

Table 7 provides a quick summary of our observations. Further explanations

Criteria	USE	Alloy An.	ZLive	ProZ
Animation with less user interaction	Partial	Yes	Yes	Yes
Generation of pre- and post-states	No	Yes	Yes	No
Information about the execution	Yes	Yes	No	Partial
Requires expertise in one modelling language only	No	Yes	Yes	Yes

Table 7. Observations about the tool usage

about our observations are given under the following titles.

Animation in both directions: Alloy and ZLive recognise the pre- and post-states of the system and they are both capable of performing a non-directed, forward and backward animation of an operation. ProZ is able to simulate a forward animation and undo it, but it cannot discover a pre-state from a post-state. USE is able to carry out forward animation only.

Reduction in the search space (Introducing further constraints): ZLive, USE and Alloy have different ways of reducing the search space in generating snapshots. USE takes advantage of .ins files by introducing extra invariants to the

system, Alloy allows the user to write such invariants within *assert* statements and ZLive uses the schema form as explained in Section 4.3. As far as we know, ProZ does not provide such flexibility to the user.

Search mechanisms: Due to the SAT Solvers embedded in ProZ and Alloy Analyzer, these tools treat the model as a set of constraints and thus they perform the search requests faster compared to ZLive and USE. The Alloy Analyzer is essentially a compiler that translates the problem to be analyzed into a huge boolean formula. After the formula is handed to a SAT solver, the solution is translated back by the Alloy Analyzer into the language of the model. All problems are solved within a user-specified scope that bounds the size of the domains, and thus makes the problem finite and reducable to a boolean formula [15]. Technically, the Alloy Analyzer is a model finder, not a model checker since given a logical formula, it finds a model of the formula. ZLive and USE, on the other hand, make use of depth-first search algorithms to find the requested snapshot, i.e., some valid state(s) of the system. In USE, as briefly described in Section 3.2, the order of objects to be generated is explicitly written in .assl file and this order certainly affects the search time. ZLive determines the order of object creation on the fly based on an optimisation algorithm.

Speaking the language of the tool: In USE, the user has to learn how to write .cmd, .assl and .use files in addition to OCL in which the assertions of the operations and invariants are specified. Alloy Analyzer requires the user to learn Alloy and to know the subtle differences between its constructs such as fact, predicate, assert, etc. ProZ and ZLive both require the user to be able to write the model in Z. With the command prompt ZLive provides, the user can also interact with the tool by using several other keywords and Z schemas. In ProZ, the user can only use the features provided in Graphical User Interface. This may be an advantage in that the user does not have to learn extra keywords/syntax, however, this also restricts the advanced user from being able to carry out complicated yet insightful queries about the model.

In addition to above observations, we also examined the validation techniques in close detail. Each tool has a different way of handling the tasks required to accomplish a certain type of validation and in some cases, tools were not able to realise certain tasks due to their inherent limitations. Table 8 provides a detailed overview of the tasks performed during validation and the tool’s capabilities.

The tasks in Table 8 are divided into three categories: Valid State Generation, Operation Animation and Sequencing. Valid State Generation includes the tasks that involve in generating one state only. The first task in this group can be done by all the tools, whereas the the task 1.2 can be achieved only by USE and Alloy Analyzer. USE, however, needs intellectual input from the user in order to accomplish this task. The definition of *intellectual input* and further details about the task are given in Section 4.1. In ZLive and ProZ, the user has to start the validation process with the *init* schema and it is mostly the case that this schema initialises the system with no objects and therefore it would be fair to put a *No* to these fields in the table. However, it is technically possible, though not practical, to write an *init* schema that creates objects. Having said that, this

Task No.	Task	USE	Alloy An.	ZLive	ProZ
1	Valid State Generation				
1.1	Initialise the system with no objects	Yes	Yes	Yes	Yes
1.2	Automatically generate a valid non-empty state	Yes (intellect.)	Yes	Yes (intellect.)	Yes (intellect.)
1.3	Generate a valid non-empty state by using system operations	Yes (intellect.)	No	Yes	Yes
1.4	Generate a valid state with constraints	Yes (intellect.)	Yes	Yes	No
2	Operation Animation				
2.1	Forward Animation with input values supplied by the user	Yes	Yes	Yes	No
2.2	Forward Animation with no input values given	Yes (lim.)	Yes	Yes (lim.)	Yes (lim.)
2.3	Backward Animation with output values supplied by the user	No	Yes	Yes	No
2.4	Backward Animation with no output values given	No	Yes	Yes (lim.)	No
2.5	Non-directed animation	No	Yes	Yes (lim.)	No
2.6	Non-directed animation with constraints	No	Yes	Yes (lim.)	No
3	Sequencing				
3.1	Update the current state after the animation of an operation	Yes	No	Yes	Yes
3.2	Animate a user supplied sequence of operations	Yes (lim.)	No	Yes	Yes
3.3	Automatically explore all sequences of operations*	No	No	No	Yes

Table 8. Validation techniques and the state-based modelling tools

would require the user to put intellectual work in writing the schema in order to make sure that what is created in *init* schema does not conflict with system invariants. In addition, it is not possible to represent the creation of m number of objects in the *init* schema without explicitly defining each of them.

The difference between the tasks 1.2 and 1.3 is that 1.2 outputs a non-empty state, e.g. m number of objects of type x and n number of objects of type y , in one step with no user interaction. The task 1.3, on the other hand, may need $m + n$ steps in order to reach the same state.

The task 1.4 is useful especially when the user would like to start from a state that has certain characteristics, e.g., a field is assigned to a particular value, the range of some input is limited, etc. The only tool that is unable to perform this task is ProZ since the GUI of the tool does not allow the user to enter such inputs.

In terms of valid state generation, Alloy Analyzer is found to be the most powerful tool except when a valid non-empty state is to be generated by using

system operations. The reason for this exception is related to the tasks in the third category explained later in this section.

The second category in Table 8 is concerned with Operation Animation, i.e., finding the pre- and/or post-states of a given operation. The reason for marking USE as *limited* in Task 2.2 is that without the limitations in the range of variables and maximum number of objects to be created, USE is not always capable of finding an input state. The limitation of ProZ in performing the same task (forward animation with no input values given) -strangely- comes from one of its strengths. The tool is able to list the enabled operations with possible parameter values. However, the list contains only a certain number of operations, thus a user cannot execute the operation with certain parameter values unless it is listed in the list. The section 4.4 gives further details about this issue.

In performing the tasks included in the second category, both Alloy and ZLive are proved to be competent. In terms of efficiency, ALLOY performs better since ZLive struggles to find a solution when the search space is too big.

In the final category -Sequencing-, the tasks focus mainly on animating a sequence of operations. The only tool that is not capable of realising the tasks 3.1 and 3.2 is Alloy Analyzer. The reason for this is that Alloy Analyzer does not keep track of state changes after an animation, i.e., it does not change the current state to the post-state. Thus, according to our observations, it is superb in performing one request at a time, but does not have the concept of carrying out a sequence of related actions one after the other.

The task 3.3 (essentially model checking) is not directly related to the main targets of this study, yet the tools' capabilities in performing the task could be observed. Other tasks that have not been explored in this study, but can be classified under this category include, but are not limited to, deadlock detection, feasibility check, etc.

It is our belief that the experiences reported in this paper shed light for potential users as well as for the developers of such tools in understanding the modelling and validation steps of MBT and the expectations in using model analysis tools.

References

1. Van Lamsweerde A., Formal Specification; a Roadmap: The Future of Software Engineering, Anthony Finkelstein (Ed.), ACM Press, ISBN 1-58113-253-0, 2000.
2. UML Resource Page, <http://www.uml.org/>, viewed 2007.
3. Bertolino A., Marchetti E., Muccini H., Introducing a reasonably complete and coherent approach for MBT, Electr. Notes Theor. Comput. Sci., 116, 85-97, 2005.
4. Cavarra A., Crichton C., Davies J., Hartman A., Jeron T., Maunier L., Using UML for automatic test case generation, TACAS, 2002.
5. Jackson D., A Comparison of Object Modelling Notations: Alloy, UML and Z, MIT Lab for Computer Science, 1999.
6. He Y., Comparison of the Modelling Languages Alloy and UML, <http://ww1.ucmss.com/books/LFS/CSREA2006/SER4949.pdf>, 2006.

7. Georg G., Bieman J., France R., Using Alloy and UML/OCL to Specify Run-Time Configuration Management: A Case Study, LNI, Vol7, Workshop of the pUML-Group held together with the UML, 2001.
8. Utting M., Pretschner A., Legeard B., A taxonomy of model-based testing, Working paper series, University of Waikato, Department of Computer Science, 04/2006.
9. Using ProZ for Animation and Model Checking of Z Specifications, <http://asap0.cs.uni-duesseldorf.de/trac/prob/wiki/Using%20Z%20with%20ProB>.
10. 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.
11. Utting M., Position paper: Model-based testing, Verified Software: Theories, Tools, Experiments(VSTTE), 2006.
12. Utting M., Legeard B., Practical Model-Based Testing, Morgan Kauffman, 2007.
13. Bernard E., Bouquet F., Charbonnier A., Legeard B., Peureux F., Utting M., Torrebore E., Model-based Testing from UML Models, Lecture Notes in Informatics, pp. 223230, 2006.
14. Z Notation, http://en.wikipedia.org/wiki/Z_notation.
15. The Alloy Analyzer, <http://alloy.mit.edu/>.
16. Jackson D., Software Abstractions: Logic, Language, and Analysis, MIT Press, 2006.
17. UML Specifications Environment, <http://www.db.informatik.uni-bremen.de/projects/USE/>.
18. 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.
19. CZT ZLive, <http://czt.sourceforge.net/zlive/index.html>.