Malaysian Journal of Computer Science, Vol. 17 No. 1, June 2004, pp. 42-51

TESTING THE SATISFIABILITY OF Z FORMAL SPECIFICATIONS BY USING PROLOG

Abdullah Mohd Zin
Faculty of Information Science and Technology
Universiti Kebangsaan Malaysia
43600 Bangi, Malaysia

Zarina Shukur
Faculty of Information Science and Technology
Universiti Kebangsaan Malaysia
43600 Bangi, Malaysia

ABSTRACT

Formal specifications are now being used as a basis for communication, design, testing and verification of a software product. For a formal specification to be used effectively, it must be valid, which means that it must be well-formed and reflects the user requirements. The normal technique for validating a formal specification is by using formal reasoning. However, the use of formal reasoning is extremely tedious and time consuming. In this paper we explore alternative techniques for validating a Z formal specification. In particular, we consider the concept of satisfiability as a weaker alternative to validity and discuss how testing can be used to check the satisfiability of a Z formal specification.

Keywords: ***Formal specification, Z formal specification, Validation and verification***

1.0 INTRODUCTION

Formal specifications are now being used as a basis for communication, design, testing and verification of a software product. For a formal specification to be used effectively, it must be valid. A valid formal specification must fulfill two conditions. First it must be well-formed, which means that statements in the formal specification must conform to the syntax and semantics of the formal specification language. The second aspect, which is more important, is to ensure that the properties of the formal specification reflect the user requirements. Having a valid formal specification is very important because the formal specification is taken as the basis for the software development.

A formal specification consists of a state definition and a set of operations. The initial state defines the set of states of the system from which the sequence of operations can be invoked. An operation defines a set of rules that transform the initial state into another state. The resultant state defines the set of states that satisfy the desired result after executing the operation. An approach for validating a formal specification is to consider the state definition and the set of operations as a set of axioms describing a system. A set of axioms is considered to be valid if it fulfills two important criteria: *self-consistency* and *completeness.*

Definition*:* A set of axioms is said to be *self-consistent* if there is no statement in the formal specification that contradicts itself. A set of axioms is said to be *complete* if there are enough statements to describe the behaviour of the system.

The general problem of determining the self-consistency and completeness of an arbitrary set of axioms is known to be undecidable [1]. However, in practice it is normally possible to show that a set of axioms is self-consistent. A standard approach in logic to prove consistency is by interpreting the theory being checked in another theory which consistency is assumed or has been established previously. This approach is cumbersome and unattractive in practice. Another approach is by using the Knuth-Bendix algorithm which determines the consistency of a set of axioms by treating the axioms as rewrite rules rather than equalities [2]. This set of axioms is demonstrated to be consistent if they exhibit the Church-Rosser property. Informally, a set of rewrite rules is Church-Rosser if whenever one applies a rewrite rule to reduce a term, and then a rule to reduce the resulting term, etc, until there is no longer an applicable rule, the final result does not depend upon the order in which the rules were applied. An introduction to the Knuth-Bendix algorithm and its application in theorem proving can be found in a paper by Dick [3].

To show that a formal specification is complete is more difficult. The exact meaning of completeness depends upon the environment in which one is working. Generally, a set of axioms is said to be complete if every well-formed formula or its negation can be proved as a theorem. Since it is not really necessary to show completeness for all formulas, the concept of *sufficient completeness* is normally being used.

***Definition**:* A set of axioms is said to be *sufficiently complete* if and only if all theorems related to the external behaviours of the axioms can be derived from the axioms.

In the development of formal specification, these external behaviours have to be described by the user.

In the case of a Z formal specification, the self-consistency and completeness of a specification is normally determined by proving the initialisation theorem, investigating preconditions for all operational schemas and proving properties about the specification [4]. Proof of Initialisation Theorem demonstrates the feasibility of the state schema by demonstrating the validity of the *Initialisation Theorem*. This theorem is stated as:

***Initialisation Theorem*:** If **State** is the state schema of a system, and **InitState** is the initialisation of the system, then

$$\vdash \exists State' \,.\, InitState$$

The theorem states that there really is a *State'* system which satisfies the requirement of *InitState*. If it is so, the state shcema is feasible, otherwise it is not.

Operation describes the mechanism for transforming the system from "initial state" to "final state". An operation S can be described by using Hoare notation as

$$\{P\}\ S\ \{Q\}$$

where P is a predicate describing the set of initial states which can guarantee that if the execution of S begun in any of the states, it will reach the final state described by predicate Q. P is called the *precondition* of S. Q in this case is called the *postcondition*. Since precondition determines the condition under which each operation is applicable, it is necessary for the specifier to determine that the precondition for each operation is properly specified. By calculating preconditions, we can identify the range of error conditions which may arise. The consistency of an operation can be determined by checking whether the calculated precondition agrees with our intuition [5].

We have stated earlier that it is sufficient to determine the completeness of a formal specification by checking that all the external behaviour of the specification can be deduced from the specification. In Z specification, all these external behaviours denote the properties of the system being specified. These properties may be demanded in informal requirements for the specification, or they may be identified by the specifier as key points about the specification. In order to show the completeness of a specification, we have to show that these properties can be derived from the specification.

2.0 APPROACHES FOR VALIDATING A FORMAL SPECIFICATION

The normal approach to validate a formal specification is by formal reasoning. Since a formal specification is a set of axioms, the properties of the system can be considered as theorems to be derived from this set of axioms. So the technique for proving these properties is similar to the technique of proving any mathematical theorem. In general, a proof is constructed by reasoning from the axioms, giving a justification for each step that is made, arriving finally at the desired conclusion. Reasoning may be carried out at many different levels depending on how rigorous we are trying to be. At one end there is an informal proof which progresses are justified by appealing to generally accepted facts and to intuition. At the other end of the scale is the completely formal proof, where a fixed set of rules must be strictly applied to justify each step.

However, an experiment of using this approach in software development, conducted by Fields et al [6] indicates that:

> "Formally verifying specifications and development is extremely tedious and time consuming. This is not a problem specific to *mural* but is more to do with the level of detail at which one is forced to work when doing 'fully formal' development"

Proponents of a liberal approach in formal methodology have argued that, in most cases it is not necessary to demonstrate the validity of the specification by using formal reasoning. Other techniques for validating formal specification can also be used.

Formal technical review (FTR) is a class of reviews that include walkthrough, inspection, round-robin reviews and other small group technical assessments [7]. Each FTR is conducted as a meeting which focus on a specific part of the overall specification. One type of FTR is called Fagan's Inspection Method [8] that was developed by Michael Fagan at IBM in the 1970s. FTR is a very effective way for validating specifications of other technical documents because although people are good at catching some of their own errors, large classes of errors escape the originator more easily than they escape anyone else. Experiments of applying Fagan's method by the Seismic Software Support Group (SSSG) at Shell Research in the Netherlands, have managed to uncover substantial errors in 11 software requirement specifications totaling some 500 pages [9]. However, this approach is very informal and requires a lot of human effort.

One approach to formalise FTR is the viewpoint resolution technique. Viewpoint resolution is a process which identifies discrepancies between two different viewpoints, classifies and evaluates those discrepancies, and integrates the alternative solutions into a single representation [10]. One approach of using this technique is comprised of procedures to formalise viewpoints, procedures to analyse the formalised viewpoints, and a special language, VWP1, to represent viewpoints[1]. Thus the viewpoint analysis strategy is basically a process for finding discrepancies between two rule bases, each one representing a different viewpoint. An advantage of the viewpoint resolution technique is that it is formal and thus the result of the analysis will be more reliable. However, this technique requires more than one specification which must be produced by different specifiers.

Symbolic execution was first proposed by King [12] as an alternative to the concept of dynamic testing. Symbolic execution of a formal specification is similar to the symbolic execution of a program unit. The idea of using symbolic execution techniques as a method for validating formal specifications has been proposed by Kemmerer [13] and a system for symbolic execution of a formal specification has been described by Kneuper [14]. An advantage of symbolic execution is that it allows one to deal with infinite domains. It also lets the specifier to test a large number of cases at one time. The main problem with symbolic execution is that predicates defining the current state can be unmanageable after executing a sequence of transforms, especially when the transforms contain conditional expressions that cannot be readily resolved. Another disadvantage of using symbolic execution for validating formal specification is the danger that, even though the system might show a mistake, such as referencing a wrong variable, the user might not notice it. This can happen in particular when the results of the symbolic execution look too familiar to the original specification.

The testing technique for validating a formal specification executes the specification against test data. This idea has been proposed by Kemmerer [13] and Jalote [15]. For the testing to be done, the formal specifications must be executable. However, most formal specification notations are non-procedural and thus cannot be executed directly. So before this type of formal specification can be tested, it must be translated into a procedural form that can be executed.

An experiment for testing a formal specification is described by Jalote [15]. In this experiment, Jalote has developed a system for testing the completeness of a formal specification. This system is based on the axiomatic specification of an ADT (Abstract Data Type). The specification language that is employed has two major components: syntactic specifications and semantic specifications. The syntactic specification provides syntactic and type-checking information such as variables and their types, domain and range of operations. Semantic specifications define the meaning of the operation by stating, in the form of axioms, the relationships between the operations. A major cause of incompleteness of a specification is that some of the axioms are not provided.

The advantage of testing is that this technique is well understood and thus, as compared with other techniques it is easier to use. Another advantage of this technique is that the ability to execute formal specifications helps the specifier in understanding the specification. The disadvantage of this technique is similar to the disadvantage of program testing, that is, it cannot be used to establish whether the program is really correct and free of errors.

3.0 A TECHNIQUE FOR TESTING THE SATISFIABILITY OF A Z FORMAL SPECIFICATION

In this section, we describe a technique that can be used to test the validity of a Z formal specification.

As we have mentioned earlier, the purpose of validating a formal specification is to ensure that it reflects the user requirements. Our technique is based on an alternative definition of the validity of a formal specification as given

[1] VWP1 is a rule-based language, which is derived from the PRISM language [11]

by Kemmerer [13].

***Definition**:* A formal specification F is considered to be valid if

$$\forall I \in IMP(F) \;.\; (\forall P \in STATES(I) \;.\; INIT(P) \Rightarrow RESULT(SEQ(P))).$$

where IMP(F) is the set of all possible implementations of F, STATES(I) is the set of all possible states for implementation I, INIT is the initial state, SEQ is a sequence of operations and RESULT is the resultant state.

This definition implies that a formal specification *accurately reflects* the user requirements if for every possible implementation of that specification, it is *relevant* to the user requirements. An implementation is *relevant* to the user requirements if it gives the desired functionality. That is, given the initial state, it will be transformed by a sequence of operations into the resultant state.

However, it is not possible to determine the validity of a formal specification by checking that all possible implementations of that specification produce the desired result since this implies that we have to test for all implementations of the specification. So, in practice, a more reasonable approach is to have the specification define the minimum critical requirements and to determine whether a formal specification is satisfiable with respect to the functional requirements. We define satisfiability as follows:

Definition: A formal specification is said to be *satisfiable* with respect to the functional requirements if there is some implementation of the specification that gives the desired functionality.

In our approach for checking the satisfiability of a Z formal specification, this specification is implemented by translating it into an equivalent Prolog program. Prolog is selected as the target language since both Z and Prolog are based on the first order predicate logic. The program is done animated against some test data. The process of animating Z specifications by using Prolog has been descibed in [16]. We are arguing that if the Prolog program which is equivalent to the specification exhibits the required behaviour as it is defined in the user requirements, then it can increase the level of confidence about the validity of the specification.

We have stated that the validity of a Z specification is normally checked by checking the validity of the initialisation theorem, by investigating preconditions and also by checking the validity of the properties of the specification. The initialisation theorem and properties of the specification are normally defined in a Z specification as predicates (or constraints) to the specification. The satisfiability of these predicates can be checked by animating them.

4.0 A CASE STUDY

To demonstrate the feasibility of the technique, we will use a part of the specification for the Music Library System [17] as a case study.

4.1 The Specification – The Music Library System

A music library information system is used to keep information about members of a music library.

The basic data types required by the system are:

Person: set of all people
Recording: set of all recording items
Copy: set of all identifiers that can be used to identify the occurrence of all recording items.

In Z, these basic types are declared as:

[Person,Recording,Copy]

The state schema for the system is called **MusicLib** which is described as follows:

MusicLib

member: P Person
held: Copy → Recording
loan: Copy → Person
reservation: Recording → Person

---

dom loan ⊆ dom held
ran loan ⊆ member
dom reservation ⊆ ran held
ran reservation ⊆ member

There are a number of operations that need to be done by the system. In this case study, we are going to consider only two operations: **AddNewMember** and **RemoveMember**.

AddNewMember is an operation to add a new member into the system's database. Before this operation can be done, we have to make sure that the new member is not already in the database.

AddNewMember

Δ MusicLib
mem? : Person

---

mem? ∉ member
member' = member ∪ {mem?}
held' = held
loan' = loan
resevation' = reservation

RemoveMember is an operation to remove an existing member from the database.

RemoveMember

Δ MusicLib
mem? : Persoan

---

mem? ∈ member
mem? ∉ ran loan
mem? ∉ ran reservation
member' = member \ {mem?}
held' = held
loan' = loan
resrvation' = reservation

4.2 Translating a Z specification into Prolog

Each schema of the specification is translated into a Prolog predicate. We are going to use the translation technique as described in [16]. For example the state schema is translated into:

```
pMusicLib :-
    update(vmember,V312),
    update(vheld,V313),
    update(vloan,V314),
    update(vreservation,V315),
    dom(V314,V700),
    dom(V313,V701),
    subset(V700,V701),
    ran(V314,V702),
```

```
subset(V702,V312),
dom(V315,V703),
ran(V313,V704),
subset(V703,V704),
ran(V315,V705),
subset(V704,V312),
memberof(vmember,V312),
memberof(vheld,V313),
memberof(vloan,V314),
memberof(vreservation,V315).
```

where "update", "dom", "subset", "ran" and "memberof" are Prolog predicates which we have defined in the Zprolog library. V312 and V313 are examples of variables that are used to store the values of the operation. The values of state variables are kept in Prolog's database by using the structure. However, in order to manipulate the values of these variables, they must be stored in the form of lists. So, at the beginning of every schema, it is important to extract the data into a list form. This is done by using Prolog's predicate "update". At the end of the schema, new values must be written back into the database by using the predicate "memberof".

4.3 Proving The Initial State Theorem

The Initial State Theorem for this specification can be stated as:

MusicLib InitMusicLib

Based on the state schema, the initial state schema for the system can be defined as:

InitMusicLib

MusicLib

---

member = ∅
held = ∅
loan = ∅
reservation = ∅

The satisfiability of the Initial State Theorem can be done by executing pInitMusicLib followed by pMusicLib as follows:

? pInitMusicLib, pMusicLib

The reply given by the Prolog interpreter is

yes

indicating that the initial state given is an acceptable initial state.

Suppose that we give a wrong initial state for the system as follows:

InitMusicLib2

MusicLib

---

member = {m1}
held = {r1}
loan = ∅
reservation = ∅

When we run the command

? pInitMusicLib2, pMusicLib

the reply is

no

indicating that InitMusicLib2 is not a right initial state for the system.

4.4 Precondition Investigation

A precondition for an operation schema **AddNewMember** can be found by removing the after-state variables and outputs from the signature and existentially quantifying them in the predicate. So

preAddNewMember

member: **P** Person
held : Copy → Recording
loan: Copy→ Person
reservation: Recording→ Person
mem? : Person

MusicLib'•
(mem? ∉ member ∧
member' = member ∪ {mem?} ∧
held' = held ∧
loan' = loan ∧
resevation' = reservation)

Suppose that based on our intuition, we predict the precondition to be:

preAddNewMemberExpected

MusicLib
mem? : Person

---

mem? ∉ member

In order to check whether the expected precondition is the right precondition we have defined a Prolog predicate called equivalent which can be used to test whether two predicates are equivalent or not. The definition of equivalent is given as:

Definition: Two prolog predicates are equivalent if the effect of executing these predicates is the same.

By using this predicate, we can test our expected precondition by executing the following Prolog query:

```
? equivalent(pPreAddNewMember(a), pPreAddNewMemberExpected(a),R),
write(R).
```

The reply from Prolog is

```
R = equiv
```

indicating that our expected precondition is possibly the right precondition for that operation schema.

However, suppose that we give a wrong expected precondition, for example

preAddNewMemberExpected2

MusicLib
mem? : Person

---

mem? ∈ member

When we execute

```
? equivalent(pPreAddNewMember(a), pPreAddNewMemberExpected(a),R),
write(R).
```

The reply from Prolog is

```
R = not_equiv
```

4.5 Proof of Properties

There are a few properties or external behaviors of the system. One of them is:
Successful addition of a new member to the system will increase the number of members in the system.

In Z, this statement can be stated as
AddNewMember ∧ Success • #member' = #member + 1

To validate this property, we have to convert each schema into Prolog. The predicate
#member' = #member + 1

is also converted into Prolog as:

```
predicate1 :-
    cardinal(vmember,V301),
    add(V301,1,V302),
    cardinal(vmemberP,V303),
    equal(V303,V302).
```

We can run a Prolog query as:
? pAddNewMember(a), Success, predicate1.
yes
indicating that the property can be deduced from the specification.

Suppose that we give a wrong property, for example
AddNewMember ∧ Success • #member' = #member – 1

The predicate
#member' = #member - 1
is converted into Prolog as:

```
predicate2 :-
    cardinal(vmember,V301),
    minus(V301,1,V302),
    cardinal(vmemberP,V303),
    equal(V303,V302).
```

We can run a Prolog query as:
? pAddNewMember(a), Success, predicate2.
no
indicating that the predicate cannot be deduced from the specification.

5.0 IMPLEMENTATION

The system for testing the satisfiability of Z formal specifications, as shown in Fig. 1, consists of the following components:

- Z compiler: the Z compiler, called **zc** accepts a Z formal specification (written in the Latex format) and does the syntax and type checking. If no errors are found, it will translate the Z specification into an intermediate representation.
- Z to Prolog translator (**zp**): accepts as input the intermediate representation and translates it into Prolog.
- Z Prolog Library: consists of Prolog predicates that define all operators in Z.
- Test predicates: consists of a few Prolog predicates that can be used in the testing process.

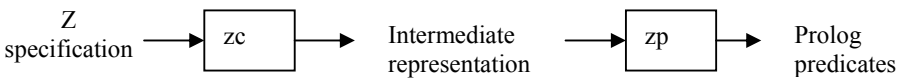

Fig. 1: Z formal specification testing system

The first implementation of **zc** and **zp** was first described in [16]. However, **zc** and **zp** have been reimplemented so that they can be used under Windows operating system.

The specification to be tested needs to be translated into an intermediate representation by using **zc**. If there are no errors in the specification, the output from **zc** can be translated into Prolog by using **zp**. The testing can be done by using a Prolog interpreter for example the Quintus Prolog. The Z Prolog library together with the Prolog program produced by **zp** need to be called.

6.0 CONCLUSION

In this paper we have discussed the use of testing in order to validate Z formal specifications. We have described the technique by using a case study. Although this technique cannot ensure the validity of a formal specification, it can be used to check the satisfiability of a formal specification. In most cases, especially in the case of non-critical software development, the checking of satisfiability is sufficient to ensure that the software can be properly developed.

REFERENCES

[1] J. V. Guttag, "Notes on Types Abstraction (Version 2)". *IEEE Trans Software Engineering*, Vol. 6 No. 1, Jan 1980, pp. 13-23.

[2] D. E Knuth and P. B. Bendix, "Simple Word Problem in Universal Algebra", in *Computational Problem in Abstract Algebra*, Pergamon Press, 1970, pp. 263-297.

[3] A. J. J. Dick, "An Introduction to Knuth-Bendix Completion", *The Computer Journal*, Vol. 34, No. 1, Jan 1991, pp 2-15.

[4] B. Potter, J. Sinclair and D. Till, *An Introduction to Formal Specification and Z*. Prentice-Hall, Inc. 1996.

[5] J. M. Spivey, "Specifying a Real-Time Kernel". *IEEE Software*, Vol. 7, No. 5, Sept 1990, pp. 21-28.

[6] B. Field, and M. Elvang-Gorasson, "A VDM Case Study in Mural". *IEEE Trans Software Engineering*, Vol. 18, No. 4, April 1992, pp. 279-295.

[7] R. S. Pressman, *Software Engineering: A Practioner's Approach*. McGraw-Hill, Inc. 2001.

[8] M. E. Fagan, "Design and Code Inspection to Reduce Errors in Program Development". *IBM System Journal*, Vol. 15, No. 3, 1976, pp. 182-211.

[9] E. P. Doolan, "Experience with Fagan's Inspection Method". *Software – Practice and Experience*, Vol. 22, No. 2, Feb 1992, pp. 173-182.

[10] J. C. S. Leite, and P. A. Freeman, "Requirement Validation Through Viewpoint Resolution", *IEEE Trans Software Engineering*, Vol. 17, No. 12, Dec 1991, pp. 1253-1269.

[11] S Ohlsson, and P. Langley, *PRISM: Tutorial and Manual*. Feb 1986. Dept of Computer Science, Univ of California.

[12] J. C. King, "Symbolic Execution and Program Testing". *Communication of the ACM*, Vol. 19, July 1976, pp. 385-394.

[13] R. A. Kemmerer, "Testing Formal Specifications to Detect Design Errors". *IEEE Trans Software Engineering*, Vol. 11, No. 1, Jan 1985, pp. 32-43.

[14] R. Kneuper, "Symbolic Execution as a Tool for Validation of Specification". *PhD Thesis*, Manchester University, 1989.

[15] P. Jalote, "Testing the Completeness of Specifiations". *IEEE Trans Software Engineering*, Vol. 15, No. 5, May 1989, pp. 526-531.

[16] Abdullah bin Mohd Zin and E Foxley, "Software Tools for Animating a Z Specification". *Sains Malaysiana*, Vol. 24, No. 4, Dis 1995, pp. 67-89.

[17] A. D. Heath, *Introductory Logic and Formal Methods*. Alfred Waller Ltd, Publishers, 1994.

BIOGRAPHY

Abdullah Mohd Zin received his PhD from the University of Nottingham, United Kingdom in 1993. He is currently attached as an Associate Professor at the Faculty of Information Science and Technology, Universiti Kebangsaan Malaysia.

Zarina Shukur received her PhD from the University of Nottingham, United Kingdom. She is currently a lecturer at the Faculty of Information Science and Technology, Universiti Kebangsaan Malaysia.