An approach to specification-based testing systems
A. Mohd Zin, a A. Al-Amayreh, a E. Foxley b
a Dept. of Computer Science, Universiti Kebangsaan Malaysia, Bangi, Malaysia
b Dept. of Computer Science, University of Nottingham, Nottingham, UK
E-Mail: amz@sun1.ftsm.ukm.my, ef@cs.nott.ac.uk

Abstract

At the end of a software development process, there is a need for software verification. The aim of the software verification is to assess the software product to determine conformance to its specification. There are a few methods which can be used for doing this. However, the most popular method is software testing. For software testing to be done effectively, there is a need to select proper test cases such that all aspects of the software can be tested. In this paper we describe a technique for generating test cases automatically from the formal specification of the software. We also describe a testing system based on this technique.

1. Introduction

Software verification is normally done at the end of a software development process in order to assess the software product to determine conformance to its specification. The prime concern of software verification is the correctness of the implementation. Other than that a software product must also be verified to ensure that it possesses other aspects of software quality, for example, efficiency and maintainability.

Nowadays, formal specifications have been accepted as a means for communicating software specifications. With the use of formal specifications,
there is a need for a specification-based verification. The most suitable technique is the formal verification. Formal verification is a technique where the correctness of a program is proved by using formal reasoning. There are two basic approaches for formal verification, either by using “inference rule” originally developed by Hoare\(^1\) or by using the so-called “predicate transformer” developed by Dijkstra\(^2\). These two approaches although different, are related to each other.

Although formal verification can be considered as the best verification technique, it is not popular because of the complexity involves in formally verifying a piece of software. Other verification techniques have been proposed as alternatives for formal verification. One of the technique is to check the consistency between the program and its specification. One example of an application of this approach was described by Baker and Rose\(^3\). They have used a formal specification notation called SPECS. The consistency checking between programs and their specifications is done by using dataflow perspective of both programs and specifications. An overview of the process is given in figure 1. Each procedure or functions in a program is abstracted into the abstract procedure definition. A representation mapping from the programming language data structure to abstract objects is used to abstract further the abstract procedure definition into an abstract object view of the procedure. In order to see if the abstract object view of the procedure or function is consistent with the specification, the specification is abstracted to provide an analogous functional view of the specification of the operation. This functional view is then compared with the abstract object view of the procedure.

![Figure 1: Checking the consistency between program and specification](image-url)

In most of software development projects, software verification is done by testing the software. One technique for specification-based testing was described by Hayes\(^4\) for testing a module based on its specification written in Z. The basis for his approach is as follows. Since the program is an implementation of a specification, then it must preserve the properties of its
specification. In particular, it must preserve the invariant of the specification and its precondition. Since the invariant of the specification is given by its state schema, then this invariant can be checked by checking that the state schema holds. This can be done by transforming the state schema into an implementation, and the checking can be made by testing this implementation. However, checking invariants and precondition is not a thorough test for an implementation; the implementation could be quite disastrously wrong but still maintain the invariant. To thoroughly check a module, there is also a need to check that it confirms to the input-output relation of the specification. It has been argued that by combining input-output relation checks with invariant and precondition checks, we can get a thorough test mechanism for a module, and the redundancy incorporated in these checks is sufficient to catch any fault manifested during testing.

This paper describes another technique for specification-based testing. In this technique, a formal specification is used as the basis for generating test cases. In section 2, we will discuss this technique and we will also describe our approach for generating test cases from a formal specification. In section 3, we will describe the process of generating test cases in more detailed. The testing system based on this technique is described in section 4. Section 5 is the conclusion.

2. Generating test cases from a formal specification

The theory for using a specification in program testing was given by Goodenough and Gerhart. In their description, a specification and a program are defined as mapping “f” and “P” from some domain set D to some range R:

\[ \text{specification} \ f : D \rightarrow R \]
\[ \text{program} \quad P : D \rightarrow R \]

A testing method is a mapping from a specification and a program to a set of input test cases

\[ \text{method} \quad M : f \times P \rightarrow P \ D \]

A method is said to be reliable if P either passes all test sets \( T_1 \) or fails on all of them. A method is said to be valid if any errors is detected by some \( T_1 \). A good method is both reliable and valid, and it was shown that by demonstrating a method to be both reliable and valid for a particular specification and program, it is equivalent to proving it to be correct.

Hall discusses an approach for generating test case from Z specifications. This was done by first identifying the test domains. This was carried out by looking for a partition of input and output sets and states which are given in
the declaration part of the specification, and the condition contained in the predicate part of the specification. Each set or state was observed independently and its partition was observed at combinations of these sets and states, and at possible non-orthogonal combinations. Having established the test-domains, “typical” elements were selected from the sets. The next step was to consider the boundaries of the test domains, and any further test cases that were necessary to test the operation at and adjacent to this domain. This is necessary because it is recognised that problems frequently arise at the boundaries of the test domains. The technique described above can be shown to be able to generate test cases which are reliable and valid. However, the main problem with this technique is that it is not capable of being used for generating test cases automatically.

Our approach for generating test cases from a formal specification is different from the one described by Hall. Instead of generating test cases directly from the specification, we translate the specification into a Prolog implementation, and then we use the Prolog implementation to generate the test cases, as shown in figure 2.

![Diagram](image)

**Figure 2: Our approach for generating test cases**

It has been shown that the process of generating a Prolog implementation from a formal specification can be done almost automatically. A program for generating a Prolog implementation from a subset of Z formal specification has been implemented. Another work in the same area was described by West. With the availability of an automatic Z to Prolog translator, the problem of generating test cases from a Z formal specification can now be reduced into the problem of generating test cases from a Prolog program.

3. **Generating test cases from Prolog implementation**

Several methods for generating test cases from Prolog has been discussed by
many people, for example by Bourge and Gorlick. Another technique for generating test cases from Prolog-based specifications is described by Denney.

If we have a Prolog predicate “pred(A,B)”, the easiest way to generate test cases would be to issue the goal

?- pred(A,B)

to the Prolog interpreter. Since “A” and “B” are variables, Prolog will repeatedly use its backtracking feature to generate values for “A” and “B”.

However, uncontrolled backtracking would generate one particular pattern infinitely, never getting around to generating other patterns which should be generated by the predicate. To appreciate this problem, let us consider the following Prolog program:

reverse([],[],true).
reverse([X],[X],true).
reverse([X|Tail],Z,true):-
   reverse(Tail,Z1,true), append(Z1,[X],Z).
reverse([X],[Y],false):-
   X \= Y.
reverse([X|Tail],Z,false):-
   append(Z1,[X],Z), reverse(Tail,Z1,false).
append([],L,L).
append([X|L1],L2,[X|L3]):-
   append(L1,L2,L3).

If we simply issue a general goal to the Prolog interpreter as

?- reverse(List1,List2,Result)

we will get the output as shown in figure 3. The problem occurs in the third predicate as there is an infinite backtracking loop, such that the control will never get around to generate other cases such as those specified in the forth and fifth predicate of “reverse”.
To solve this problem, a Prolog meta-interpreter need to be written. Our approach for writing the meta-interpreter is similar to the approach proposed by Denney, that is by using a deterministic automaton. The formalism used for this automaton relies on mapping Prolog’s goal-reduction states to states of an automaton. This automaton can be generated automatically from the specification by translating goal reduction states to automaton states. Goal reduction is the strategy followed by Prolog to solve a goal by replacement of goal by its body, whose head is identical to the chosen goal. The meta-interpreter can then monitor the execution of the predicate by recording the paths through a corresponding predicate automaton. By using our meta-interpreter, the test cases will consider all possible cases, as shown in figure 4.

Figure 3: Test cases generated by issuing a general goal

<table>
<thead>
<tr>
<th>C#</th>
<th>List1</th>
<th>List2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[]</td>
<td>[]</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>[67]</td>
<td>[67]</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>[67, 13]</td>
<td>[13, 67]</td>
<td>true</td>
</tr>
<tr>
<td>4</td>
<td>[67, 13, 59]</td>
<td>[59, 13, 67]</td>
<td>true</td>
</tr>
<tr>
<td>5</td>
<td>[67, 13, 59, 05]</td>
<td>[05, 59, 13, 67]</td>
<td>true</td>
</tr>
</tbody>
</table>

Figure 4: Test cases generated from meta-interpreter

The second problem which need to be solved is related to the input-output format. The test cases produced need to be modified so that it is suitable and compatible with the program to be tested. In order to do that, we developed an Input-Output Analyser to perform the necessary modification to the test cases by taking into consideration the input-output format of the program. The analyser will generate an oracle table $T$, which is a set of all test cases

$$ T = \{ C_i = (I_i, O_i) \} $$
where \( i \) is no of test cases, \( C_i \) is the test case \( i \), \( L_i \) is the input and \( O_i \) is the expected output. The generated output from the input-output analyser is shown in figure 5.

<table>
<thead>
<tr>
<th>C#</th>
<th>List1</th>
<th>List2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[]</td>
<td>[]</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>[z]</td>
<td>[z]</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>[f,l]</td>
<td>[l,f]</td>
<td>true</td>
</tr>
<tr>
<td>4</td>
<td>[d,k,w]</td>
<td>[w,k,d]</td>
<td>true</td>
</tr>
<tr>
<td>5</td>
<td>[z]</td>
<td>[h]</td>
<td>false</td>
</tr>
<tr>
<td>6</td>
<td>[s,o]</td>
<td>[o,v]</td>
<td>false</td>
</tr>
<tr>
<td>7</td>
<td>[a,t,k]</td>
<td>[k,r,m]</td>
<td>false</td>
</tr>
</tbody>
</table>

Figure 5: The oracle table

The third problem which is faced when using Prolog for generating test cases is the variables’ instantiation. Uninstantiated variable is the one which has not yet been bounded to any particular value. For example, suppose we have a Prolog predicate

\[
pred(A,B) :-
A \text{ is } B*20.
\]

In this predicate, the variable \( B \) is unbounded before it is used at the right hand side of the assignment statement. In order to solve this problem, our meta-interpreter will bind any unbounded variable with a random value. For example, consider a specification for a triangle, which is translated into Prolog as follows:

\[
\begin{align*}
\text{validtriangle}(X,Y,Z) & : - \\
& X < Y+Z, Y < X + Z, Z < X + Y. \\
\text{class}(X,X,X,\text{equilateral}) & : - \\
& \text{validtriangle}(X,X,X). \\
\text{class}(X,X,Z,\text{isosceles}) & : - \\
& \text{validtriangle}(X,X,Z). \\
\text{class}(X,Y,X,\text{isosceles}) & : - \\
& \text{validtriangle}(X,Y,X). \\
\text{class}(X,Y,Y,\text{isosceles}) & : - \\
& \text{validtriangle}(X,Y,Y). \\
\text{class}(X,Y,Z,\text{scalene}) & : - \\
& \text{validtriangle}(X,Y,Z). \\
\text{class}(X,Y,Z,\text{invalid}). &
\end{align*}
\]
The test cases generated by the meta-interpreter will be instantiated to random values as shown in figure 6.

<table>
<thead>
<tr>
<th>C#</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>231</td>
<td>231</td>
<td>231</td>
<td>equilateral</td>
</tr>
<tr>
<td>2.</td>
<td>493</td>
<td>334</td>
<td>493</td>
<td>isosceles</td>
</tr>
<tr>
<td>3.</td>
<td>108</td>
<td>214</td>
<td>214</td>
<td>isosceles</td>
</tr>
<tr>
<td>4.</td>
<td>102</td>
<td>129</td>
<td>60</td>
<td>scalene</td>
</tr>
<tr>
<td>5.</td>
<td>263</td>
<td>631</td>
<td>185</td>
<td>invalid</td>
</tr>
</tbody>
</table>

Figure 6: An example of test cases with variable instantiation

4. Specification-based testing system

Our testing system consists of four components: the Z to Prolog translator, test case generator, input-output analyser and a test driver. It accepts the formal specification for the program, I/O format and the program executable code and produce the test result as shown in figure 7. Some of the components were originally designed for a UNIX-based system. It has now been ported to the PC Windows 95 environments.

---

The test driver is responsible for doing the testing. It accepts the oracle table as its input and then run the program. The result of the program execution is then compared with the expected result in the oracle table. This comparison is done by using the oracle program\(^1\).

One of the implementation problem which is faced is concerning the Z-to-Prolog translator. The version which is available to us does not produce a suitable Prolog program to be used by the test case generator. In this situation, the Prolog program need to be modified by hand. We are now working towards improving the Z-to-Prolog translator. Another possible approach is to write a post-processor for the Z-to-Prolog translator so that its output can be modified automatically to conform to the requirement of the test case generator.

5. Conclusion

In this paper we have described an approach for a specification-based testing system. The core for testing system is the test case generator which generates test cases from a Prolog program. The advantage of using this approach is that the test case generation can be done almost automatically. Once the test cases are available, the testing can be done almost automatically by the test driver. The automatic testing which we are using is adopted from the CEILIDH system\(^2\), which is an automatic marking system for students’ programs.

Our experience with the system has been very limited. At present most of the work is concentrated in generating test cases from students’ problems as specified in the CEILIDH system. Some problems which are now specified informally have been rewritten into formal specifications by using Z formal specification notation. Test cases are then generated from these specifications. The result of our experiments convinced us that the technique is feasible and usable. Further work will involve trying to generate test cases and hence automatically testing more complicated problems.

Key Words

Formal specification, Test case generation, Testing system

References:


