So, You Want to Test Your Compiler?

Theodore S. Norvell
Electrical and Computer Engineering
Memorial University

October 19, 2005

Abstract

We illustrate a simple method of system testing by applying it to the problem of testing compilers. The result is a user friendly system that greatly facilitates incremental development. A similar methodology can be applied to testing all sorts of systems, not just compilers.

1 Introduction

1.1 System Testing and Test Driven Development

System testing tests a complete software system from inputs to outputs. Unit testing, on the other hand, tests smaller components such as subroutines and classes in isolation.[1]

System testing has often been seen as a means to test a completed product and is often left to a separate group or even department. Recently process methods such as Extreme Programming [2] have advocated test driven development in which tests are constructed before the software. Tests are used as a method to understand and testably document the problems to be solved in future, often in the next increment (analysis); as a means to verify that these problems have been solved (verification); and as a means to check that future development work does not break earlier development work (regression testing). Combined with small increments, this leads to an approach of very frequent testing. Each test will be rerun after each future increment and so automation of testing is essential.

Most discussion of the role of testing in Extreme Programming and related process methodologies has focussed on unit testing; and some excellent tools exist for unit testing [3]. Once the system has reached a certain level of maturity, one where data can be run through the system from input to output, system tests can be used in place of unit tests. When the goal of an increment is to provide new features, then system level testing may be more appropriate, as it checks not only that the units are behaving correctly, but also that their combined behaviour implements the desired feature.

1.2 The Teaching Machine

The Teaching Machine [4] is a tool for displaying the operation of programming examples in classrooms and via the World Wide Web. It can be thought of as consisting of a set of compilers

[1]Binder [1] defines a "unit" to be anywhere from a single class to a complete executable, however says that usually a unit is a small cluster of one or more classes. I am using the term in the latter sense as a test done in terms of the interfaces to classes, rather than the interface to the system.

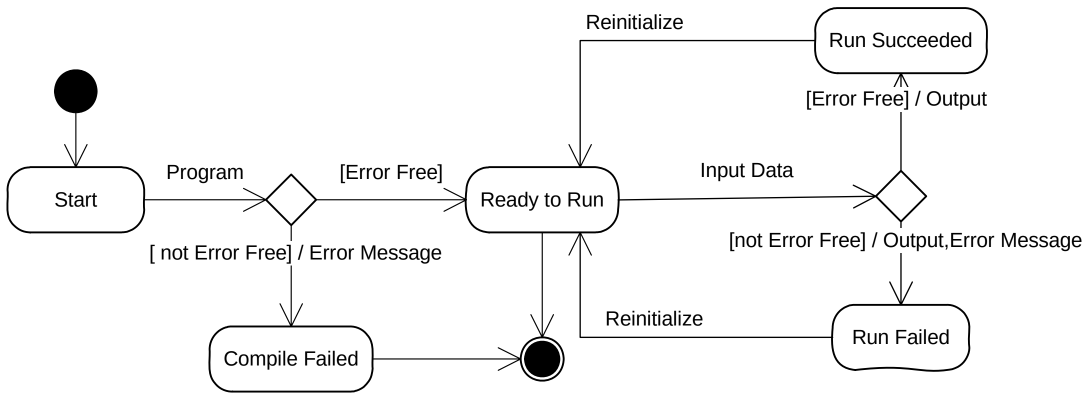


Figure 1: A state machine model of a compiler / machine pair.

(currently for a subset of C++ and for a subset of Java), a virtual machine, and a user interface, which includes a representation of the standard input and the standard output channels.

Once support for C++ had reached an acceptable level, we started to work on Java support. The strategy was to as quickly as possible reach a point where simple programs would run. This meant we needed some support for classes, static functions, and expressions. We then began building a set Java of programs that worked or that we hoped would soon work. However, there was no way to automate the execution of these programs nor the checking of their output. I therefore created the system testing tool described in the remainder of this paper.

At this point the Teaching Machine could compile and execute simple programs but could not send output to the standard output channel. Thus, we could only test that programs that should compile and execute without error, did compile and execute without error (execution tests), and that conversely programs that should lead to reports of errors did lead to the right error message being produced. There was no way to check that the execution tests did the correct calculations. However even this limited form of automated testing proved extremely useful. Compile-time test cases worked fine. For execution test cases, that the correct computation was being done could be checked manually via the Teaching Machine's graphical user interface. Once an execution test case works once, any subsequent regression will usually produce an error message rather than a change in the output.

We then directed efforts toward producing some limited output to the standard output. This required implementing support for multiple packages, importing, static variables, and calling non-static functions. This allowed us to output the results of execution tests and to check they were as expected. Throughout the implementation of these features, automated testing proved very valuable, as it has continued to be as we continue to move toward implementing almost the whole Java language.

2 Analysis

The system under test can be viewed as a state-machine. We can lump states and stimuli together to produce a finite abstraction of the state machine. In the case of a compiler, paired with an execution platform such as a CPU or a virtual machine,[2] we can view the specification of the system under test as the state machine shown in Figure 1. The input to each test is a program, written in Java or C++, and a sequence of input strings, which are sent to the standard input channel. We are not interested in testing the interactive behaviour of test programs and so we present all the input data at once. Based on this state-machine, we can enumerate the things that can go right and that can go wrong. Just focussing on tests of the compilation stage we have

- Error free input, no error message. (pass)
- Error free input, error message (fail)
- Erroneous program, no error message (fail)
- Erroneous program, correct error message (pass)
- Erroneous program, incorrect error message (mismatch)

Extending the first case to a single input we have further

- Input data that should not lead to a run-time error, no error message, correct output (pass)
- Input data that should not lead to a run-time error, no error message, incorrect output (fail)
- Input data that should not lead to a run-time error, error message (fail)
- Input data that should lead to a run-time error, no error message (fail)
- Input data that should lead to a run-time error, correct error message (pass)
- Input data that should lead to a run-time error, incorrect error message (mismatch)

The pass cases lead back to the “Ready to Run” state and thus can be extended with further inputs.

The above analysis demonstrates a general methodology on the particular example of compilers and execution platforms.

3 Implementation

We needed an easy to use method of describing test cases. I created a little language for describing the kinds of tests we wanted to run. These descriptions are embedded within the actual Java or C++ code of the test program as comments, marked with an initial ‘!’ character. Such an arrangement means that all the information pertaining to the test is in a single file, while still allowing that file to be a regular source file that can be submitted to the Teaching Machine or a regular compiler without use of any infrastructure. Figure 2 shows a couple of test programs with embedded test descriptions.

[2]Testing a compiler in isolation is possible, but very difficult and not conducive to test-driven development, as any single source can correspond to a huge number of correct machine language implementations. By testing the compiler together with a platform, we can relegate the machine code to the status of an intermediate data structure.

```
//! Compile. Execute with input ""; expect output equals "0" endl.
public class Hello {
        public static void main( String[] args ) {
              System.out.println( 0 ) ; } }
```

```
//! Compile; expect error matches /.*line 6.*No such constructor is accessible or applicable.*/
public class Constructor12 {
        Constructor12( int i ) { }

        public static void main( String[] args) {
              Constructor12 p = new Constructor12() ; } }
```

Figure 2: Some annotated test programs.

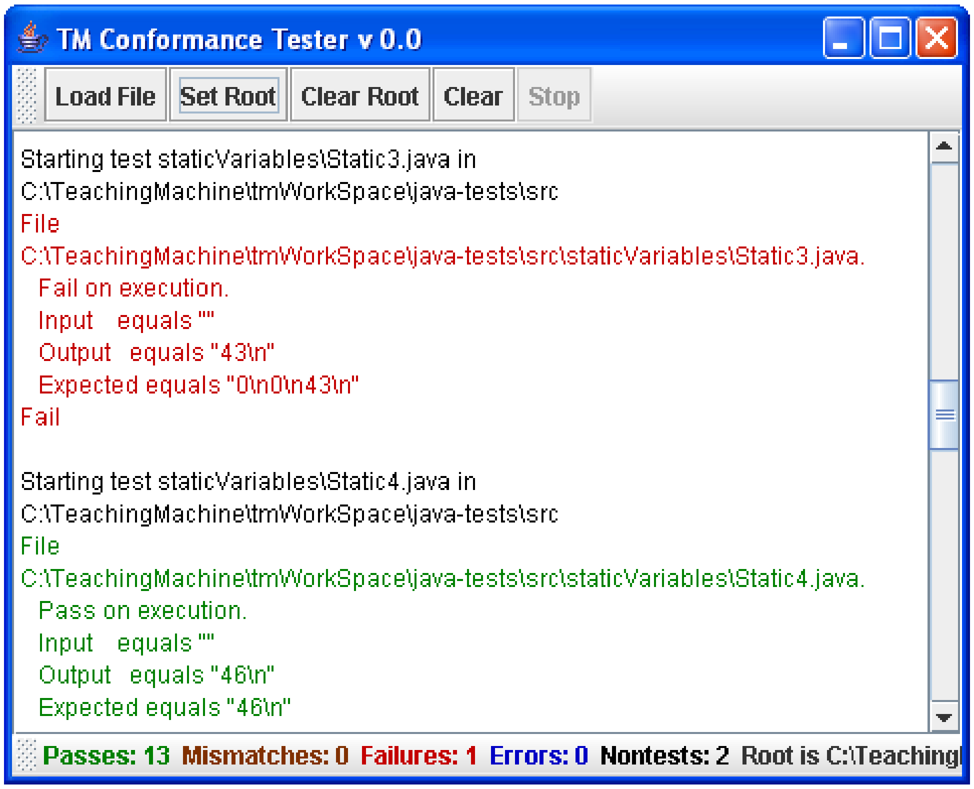
In essence, the grammar of the language is

$$
\begin{array}{rcl}
\textit{Compile} & \rightarrow & \texttt{compile}\ \textit{Error} \\
\textit{Error} & \rightarrow & (\textit{Executes})^{*} \\
 & | & \texttt{expect error}\ \textit{Match} \\
\textit{Execute} & \rightarrow & \texttt{execute}\ \textit{Input Output} \\
\textit{Input} & \rightarrow & \texttt{with input}\ \textit{String} \\
\textit{Ouput} & \rightarrow & \texttt{expect error}\ \textit{Match} \\
 & | & \texttt{expect output}\ \textit{Match} \\
\textit{Match} & \rightarrow & \texttt{equals}\ \textit{String} \\
 & | & \texttt{matches}\ \textit{String}
\end{array}
$$

In the last production, the difference between `equals` and `matches`, is that `equals` demands an exact match, whereas `matches` uses regular expression matching, as implementing in Java's, java.util.rgex package. Note how the structure of this grammar mirrors the structure of the state machine in Figure 1.

The actual test system, which we call the "Conformance Tester" allows the user to interactively select a single test file or a directory. For each file selected, an instance of the Teaching Machine is created and the test script is followed until a failure or mismatch occurs or the script ends. The result is classified as one of the following: *pass*; *fail*; *mismatch*, meaning that an error was produced as expected, but not the right one; *error*, meaning that the test script did not follow the correct grammar; and *nontest*, meaning the file did not appear to be a test file. A short description of each test run is printed to a console output. The output for failures is coloured red for easy location. See Figure 3.

The interface to the Teaching Machine is via a thin interface type called CompilerAdapter, which could easily be reimplemented to allow other compilers to be used. It would not be hard to turn the Conformance Tester into a framework into which plug-ins for various compilers could be plugged, without requiring recompilation of the Conformance Tester.



```
Starting test staticVariables\Static3.java in
C:\TeachingMachine\tmWorkSpace\java-tests\src
File
C:\TeachingMachine\tmWorkSpace\java-tests\src\staticVariables\Static3.java.
  Fail on execution.
  Input    equals ""
  Output   equals "43\n"
  Expected equals "0\n0\n43\n"
Fail

Starting test staticVariables\Static4.java in
C:\TeachingMachine\tmWorkSpace\java-tests\src
File
C:\TeachingMachine\tmWorkSpace\java-tests\src\staticVariables\Static4.java.
  Pass on execution.
  Input    equals ""
  Output   equals "46\n"
  Expected equals "46\n"
```

```
Passes: 13  Mismatches: 0  Failures: 1  Errors: 0  Nontests: 2  Root is C:\Teaching
```

Figure 3: The Conformance Tester

4 Conclusion

The method used to design the Conformance Tester is similar to some of the techniques in [1]. It could be applied to many kinds of systems whether they are transformative, interactive, or even real-time. As always with testing using a solved problem oracle, the determinism of the system is crucial to making the testing efficient of developer time.

In the development of the Teaching Machine, the use of the Conformance Tester has allowed us to adopt a test-driven style of incremental development that has proven effective. Tests serve as a means of documenting work to be done, of documenting work done, communicating between team members, and ensuring that later increments do not break the progress achieved in earlier increments. As documented in Section 1.2 the desire to do system testing strongly influenced the order of language features tackled.

Because the Teaching Machine has a graphical user interface, certain aspects can not be tested using the Conformance Tester. Some of these aspects, especially the response to various interactive commands, can be tested using JSnoopy [5]. Others, such as the display system, can only be tested manually. However the vast majority of features that need to be tested concern the correct implementation of language constructs and these are all testable using the Conformance Tester. The same level and thoroughness of testing could not have been achieved by unit testing alone.

References

[1] Robert V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, 2000.

[2] Kent Beck, *Extreme Programming Explained*, Addison-Wesley, 2000.

[3] Kent Beck and Erich Gamma, "Test infected: Programmers love writing tests," *Java Report*, vol. 3, pp. 37–50, 1998, also at http://junit.sourceforge.net/doc/testinfected/testing.htm.

[4] Theodore S. Norvell and Michael P. Bruce-Lockhart, "Taking the hood off the computer: Program animation with the teaching machine," in *Canadian Electrical and Computer Engineering Conference*, May 2000.

[5] Theodore S. Norvell, "Automating regression testing of java programs the JSnoopy way," in *Newfoundland Electrical and Computer Engineering Conference*, November 2003.