TestTalk, A Test Description Language: Write Once, Test by Anyone, Anytime, Anywhere, with Anything[1]

Chang Liu, Debra J. Richardson

Department of Information and Computer Science
University of California, Irvine, Irvine, CA 92697

Technical Report 99-08

February 28, 1999

Abstract

Software tests are intellectual assets, too, and are as valuable as source code to a software project. Over the long term, maintainable software tests significantly lower a project's cost. It is very difficult, however, to write maintainable software tests, especially executable ones. Existing approaches - including natural language, tables or forms, test scripts, programming languages, and test description languages - all are problematic, as discussed in this paper. Another solution is a test description language that provides a mechanism to specify software tests while separating different concerns of automated software testing.

In this paper, we analyze the current situation of software test description. We propose a description language just for software testers: TestTalk. We present examples to illustrate the benefits of TestTalk and discuss implementation issues of this language. The primary goal of TestTalk is to enable tests to be written once, and then used by anyone (i.e., they are understandable), anytime (i.e., they are maintainable as the software evolves), anywhere (i.e., they can be ported to a new platform), and with anything (i.e., they can be used with any testing tool in any testing environment).

[1] This research was sponsored in part by the Air Force Material Command, Rome Laboratory, and the Defense Advanced Research Projects Agency under agreement number #F30602-97-2-0033. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official position or policy, either expressed or implied, of the U.S. Government, AFMC, Rome Laboratory, DARPA, or the University of California, and no official endorsement should be inferred.

1 Introduction

Software tests are valuable intellectual assets, especially in long-lived, multi-version, and multi-platform commercial software. The highly publicized year 2000 (Y2K) software problem [14] provides a good sense of the problems that arise in this domain as well as how long tests should exist. Updating application software subject to the Y2K problem would be much easier if original tests were still available and usable to verify the modified software. Unfortunately, this is too often not the case.

Tests, especially automated software tests, represent significant investment. Windows NT 5.0 reportedly will have 48 million lines of source code, while associated test code consists of 7.5 million lines of source code. In this case, roughly one seventh of programming resources are spent in test automation work. Window NT 5.0 is going to be maintained over the next several years; maintainable tests would make a big difference in long-term costs.

The topic of this paper is software test description. In Sect. 2 and 3, we briefly survey and evaluate various software test description methods, including natural languages, structured formats, scripting languages, programming languages, and test description languages. In Sect. 4, we propose TestTalk, a new test description language aimed at solving the problems of current methods; we also provide a simple example to illustrate various features of TestTalk and evaluate TestTalk relative to the surveyed methods. We summarize our evaluations in Sect. 5 and then conclude and discuss future plans in Sect. 6.

Let us begin by taking a look at how software tests are currently written and ask the question, "Will these tests be available and usable when the next Y2K-type problem strikes?"

2 Software Test Description Practices

A software test[2] is the collection of all test artifacts related to a particular test execution for the application-under-test (AUT). Tests can be described in a variety of ways. Necessary information for a test includes input data (and/or test steps) and expected output (and/or test oracle, a mechanism for determining success or failure); actual output and execution status information are available (and should be recorded) after test execution. Other information about a test, such as significance, history, identification, etc. may also be desirable. This section surveys major software test description methods. [3]

---

[2] Henceforth, a test refers to a software test.

[3] A few example tests are included in this section; unless stated otherwise, they are hypothetical.

2.1 Natural Languages

Traditionally, tests are described in natural language (NL). NL descriptions included in test documents are typically mixed with test requirements/design, strategy, scenarios, or other information. Sometimes, they are even included in system requirement specifications; specified scenarios may be used directly as test cases or test scenarios.

How rich and detailed NL tests are depends on the effort put into the test planning activity. Tests could be very detailed, where every step is described clearly (side effects and exceptions might even be documented), in which case a new tester could easily test according to the document. Tests could also be described in a very skeletal way, intended to serve merely as quick reminders to "inside" testers.

Two example natural language descriptions of the "same" test are provided: detailed in Fig. 1, skeletal in Fig. 2. The second example could be instantiated very differently from the first example.

```
step 1: start the application:
  select "start" button;
  select "run" menu item;
  type "c:\apps\myapp\sut.exe"
    -- the path of the exe could be different
step 2: open a file:
  select "file" menu;
  select "open" menu item;
  type "c:\test\suit1\test1\temp.cpp"
    -- this file is attached at the end of this test document
  click the "open" button
step 3: modify line 3
    move cursor to the end of line 3, add "class A { int aa; }"
step 4: compile
  select "project" menu;
  select "compile file" menu item;
```

Fig. 1. Example detailed NL test.

```
add a simple class definition to a source file and compile it.
```

Fig. 2. Example skeletal NL test.

2.2 Structured Formats

Structured formats – such as tables, forms, graphs, diagrams, and structured natural language – are another popular informal means of test description. One approach is to describe partitions of the input domain and expected results with condition-decision tables [8]. Jorgensen ([10] page 5) provides an approach that explicitly describes test cases using a form containing several fields, see Fig. 3. The form is well structured, yet there is no restriction on field contents; any NL description or even a diagram is acceptable.

Test Case ID			
Purpose			
Pre-Conditions			
Inputs			
Expected Outputs			
Post-Conditions			
Execution History			
Date	Result	Version	Run By

Fig. 3. Example structured test form.

2.3 Scripting Languages

A convenient means of describing automated software tests is to use a scripting language. Test scripts can be executed automatically. Expect [11] is one such popular tool used for testing applications with command-line interfaces; it supports describing tests in Tcl scripts [17]. Fig. 4 shows an example Expect/Tcl test, which tests the UNIX program "echo" with test input data "Hello world". Visual Test [2] is another popular tool used to test applications with graphical user interfaces; it supports describing tests in a BASIC-like scripting language. Fig. 5 shows a Visual Test example, which detects changes in display two seconds after a file is opened.

```
send "echo Hello world!\n"
expect {
-re "Hello world.*$prompt $" { pass "Echo test" }
-re "$prompt $"        { fail "Echo test" }
timeout            { fail "timeout" }
}
# "-re" means that the following pattern
#    is a regular expression.
```

Fig. 4. Example Expect test script.

```
Const SCREENS = "Screens.scn"
Const SCREEN1_NAME = "Screenshot 1"
Const SCREEN2_NAME = "Screenshot 2"

Scenario ("Display Chang Detection")
     ScreenShot$ = SCREENS
     Screen1$ = SCREEN1_NAME
     Screen2$ = SCREEN2_NAME

     Viewport On
     Viewport Clear    '*** Remove all text from the Viewport

     RUN app$, NOWAIT  '*** Start the application
     Play "{ESC}" ' to be safe, get rid of possible dialog window

     'Open a source file
     Play "%(FO)"
     Play "test.txt"
     Play "%O"
     Play "{ESC}"
     hApp = WGetActWnd(0)

     '*** Dump the window
     ScnCaptureWindow(ScreenShot$,SCREEN1$,hApp,0)

     Sleep 2 ' Wait 2 seconds

     '*** Dump the window again
     ScnCaptureWindow(ScreenShot$,SCREEN2$,hApp,0)

     ret1 = ScnCompFiles(ScreenShot$,SCREEN1$,ScreenShot$,SCREEN2$,0)
     IF ret1 <> 0 Then Print "Display change detected!"

     Play "%{F4}"  '*** Close the application
End Scenario
```

Fig. 5. Example Visual Test script.

2.4 Programming Languages

Another common approach is developing programs to test software, usually coded in the developer's favorite programming language. Test programs are complete programs that typically set up the environment for a particular test execution, invoke the AUT, feed it test input data, compare the output to expected results, and possibly record the test result. The AUT may be so complex that test code or test hooks must be embedded to capture enough information. In this case, tests are likely to be written in the same programming language as the application, because this facilitates integration and the team is more familiar with the language.

Fig. 6 shows an example test in C++, which tests an editor with a spell-checking function.

```
Bool TestCaseTypo()
{
 // Send this key sequence into editor window
 Key("Thiss is a typo.");
 // invoke "Tool/Spell" menu
 Key("&t&s");
 pText = GetTypo();
 if (strcmp(pText, "Thiss") == 0) {
  ReportSuccess("The typo is catched.\n");
  return TRUE;
 } else {
  ReportFailure("The typo is missed.\n");
  return FALSE;
 }
}
```

Fig. 6. Example C++ test.

2.5 Test Description Languages

Notations and languages have also been designed to formally describe test input domains. Formal test grammars [3], [7] are very expressive in specifying the input domain for an AUT, but can not be used to specify other test concerns such as expected behaviors. Several testing tools, including TAOS [18], support input grammar specification and generate random input data based upon the grammar. The DGL (Data Generation Language) system [15] supports test generation from enhanced context-free grammars. Grammar-based techniques are most useful when the input has complex forms (such as the input to a compiler),

Other languages have been designed specifically with testing in mind. Both TSL (Test Specification Language) [16] and TDD (Test Data Description) [22] support specifying *categories*, or characteristic *properties*, of the inputs that are partitioned into symbolic values, or *choices*, which are then combined to form test cases (see [4]). As an example, Fig. 7 shows a TDD specification of a property `trans-kind` with four choices corresponding to four types of transactions. These would be combined with choices for other input categories. As with most test description languages, TDD does not specify tests directly but rather an input partition, or "a symbolic description of the desired test input", which must be converted into actual test input data by the tester.

```
Prop trans_kind =
  {read, write, create, delete}
```

Fig. 7. Example TDD description.

The test template framework (TTF) [21] takes a different approach to specifying tests. TTF uses Z [23] to specify the valid input space and then applies test heuristics partition the space. Using TTF, test data must be selected manually. TTF is particularly useful when the requirements have been formally specified in Z, in which case it supports formal documentation of tests derived from formal specifications.

Specification-based testing automatically derives tests from formal specifications. Techniques have been proposed for deriving important specification-based tests [20],

[5], [9], [4] and for deriving oracles for test result checking [19], [22], [6]. We do not evaluate these approaches here, because the languages are not specifically geared toward test description and the capabilities depend on whatever other tool support is developed (which is often by researchers other than the language developers).

3 Evaluation of Current Practices of Software Test Description

Here we evaluate the various methods of test description described in Sect. 2. We have selected a set of criteria we feel are important for describing software tests, including:

- *Writability/readability:* Test description methods should facilitate both the writing and reading of tests.
- *Understandability:* Tests should be understandable. In particular, it should be easy to understand what the test should do; this includes clearly delineating test input and/or steps and expected output and/or oracles.
- *Test data and oracle provision/automation:* To execute a test, actual test data must be provided or possible to generate from what is provided. Likewise, test results can only be checked if test oracle information or expected outputs are provided along with a mechanism for checking against test output. The automation of both aspects is critical, especially when tests must be rerun expeditiously. As recognized in a landmark testing paper, "For all but very small systems, automated tools are required to do an adequate job" of testing [1].
- *Maintainability:* Methods should facilitate maintaining tests over time as the application evolves. It is useful to separate test maintainability into two aspects: test sensitivity to implementation change, and test sensitivity to requirement change.
- *Portability:* It should be easy to take tests to another platform when the application is ported. This includes a change in hardware platform or software platform, including operating system or the automated testing environment.
- *Regression testability:* Regression testing is a critical and very expensive process that affects maintainability and portability. Although a test may be insensitive to requirement, implementation, or platform change, it may require rework to execute it in the face of evolution.
- *Manageability:* Test management involves the organization of tests for the test process, which includes documentation and reporting.
- *Measurability:* Test adequacy measurement determines how well testing has been carried out and when the process is complete according to some adequacy criteria.

Table 1 in Sect. 5 summarizes our evaluation for both current methods and TestTalk, our new approach, which we are designing as an answer to the problems outlined here.

3.1 Natural Languages and Structured Formats

Both natural languages and structured formats are informal approaches for describing tests. As such, they have many similarities with respect to their benefits and

drawbacks. As discussed previously, the quality and richness of an informal test depends entirely on the writer, as does the clarity and granularity of details.

3.1.1 Writability, Readability and Understandability

Tests in natural language (NL) are very expressive; a tester can write whatever she can imagine. This is the power of NL, yet it is inherently ambiguous, which can lead to misunderstandings between writer and reader.

Obviously, NL tests might be considered easiest to understand, since they are closest to natural communication. Whether or not a test can be easily understood, however, depends on the quality of the test description, which varies dramatically.

Although structured formats can make a group of tests and their relationships easier to understand, structured tests remain subject to the same problems as NL tests because the fields within the format are still informally described. In addition, the understandability gained by using a structured approach does not scale up because structures larger than a few pages can not be fully grasped.

3.1.2 Test Data and Oracle Provision/Automation

It is possible for testers to describe actual test data and expected results in natural language and structured formats, although whether or not they do and how consistently depends on their effort. It is somewhat more common in structured formats. Informal tests can not be executed in their natural form, nor is there any capability for automatically using either data or expected results. To take advantage of any test automation tool, testers have to rewrite tests in the form supported by some tool. This is not an easy job but rather is similar to implementing a requirement specification. For the Expect tool [11], for instance, the tester must understand each informal test and write it in the Tcl scripting language.

3.1.3 Maintainability, Portability, and Regression Testability

NL or structured tests are insensitive to application implementation change, since usually implementation details such as size and position of a window are not described in exact detail in test documents. However, when the application requirement changes, whether or not it is easy to change tests accordingly depends on the quality of the original tests. Informal tests are easy to transport to another hardware platform and/or operating system because platform-specific details are usually described conceptually so as to be applicable on any platform.

A significant amount of work is often required before informal tests can be executed; tests must be reworked before retesting the application. Thus, informal tests have low regression testability.

3.1.4 Manageability, Measurability and Automation

The quality of tests in natural languages and structured formats determines the process issues. If tests are described separately and relationships among them are explicitly specified, test management is facilitated. Tests that are described in a confusing manner and intermingled, on the other hand, lead to impossible management either manually or automatically.

In general, it is extremely difficult to measure test adequacy with informal tests, because there is no formal notion of adequacy or relation to typical test adequacy criteria.

3.2 Scripting Languages and Programming Languages

Test scripts and test programs are very similar; the major advantage is that they are automated. We have done a detailed investigation of the harmful effects of using programming languages for describing software tests [13].

3.2.1 Writability, Readability and Understandability

Writing tests in programming languages is as difficult as programming, because it is just that. A test that only needs five minutes to run manually may take much longer to fully develop, especially for testers who are not very familiar with special test hooks and test libraries. Tests are easier to write in scripting languages than in programming languages, although scripting languages are usually less expressive. Part of the additional cost of writing a test script or program involves testing and debugging it. Defects found in automated tests are often in the test itself and not in the AUT. These defects are categorized as "test bugs", as opposed to "product bugs", and significant resources are needed to fix test bugs plus deal with the false failure alarms.

3.2.2 Test Data and Oracle Provision/Automation

Although test scripts and programs provide test data, they are nontrivial to discover in the code. The test data or steps are often mingled with other worries in scripting and programming languages (such as memory allocation, data structure and exception handling), thus it is often difficult to understand *what* is being tested. Furthermore, one test may be divided into several procedures or functions or distributed in several files because of the nature of the language, making it even harder to understand.

Test oracles tend to be even more disguised in test scripts and programs, embedded with other actions. It is likely arduous to discover which outputs or behaviors are checked and against what expected output or property. When an automated test "passes", it is hard to tell if this means every outcome is correct or only part of the outcome has been checked. The test might check only an external result or only some internal data structure, which are very different behaviors.

Test scripts and code provide automated support for test data and oracles (test result checking), which makes their repeated use expedient (as long as no changes are required).

3.2.3 Maintainability, Portability, and Regression Testability

Software test scripts or programs typically compare exact actual output to expected output, and thus are very sensitive to the application evolution of any sort – that is, implementation, requirement, or platform changes. Suppose there is a slight change of order, size, or position of a GUI object in an application's interface, often a test script or program that should still report success will go awry. Likewise, when an AUT switches platform and/or operating system, for example from Unix to Windows,

existing tests probably will not work (test programming in pure Java may improve this situation). Moreover, if testers switch testing tool, such as switching from in-house test automation to a commercial test environment, tests would have to be recoded.

After a large number of tests have been coded, it is both costly and time-consuming to modify all tests in the face of such evolution. Thus, although rapid testing is supported by test scripts and programs (perhaps the primary reason testers use this approach for tests that need to be repeated frequently), the modifications required to cope with application change inhibits their regression testability.

3.2.4 Manageability, Measurability and Automation

Automated test scripts or programs inhibit test management, primarily because scripts and programs are difficult to analyze and understand by tools other than the language translator. Sometimes several test cases are included in one script or program. It is hard to separate them into independent tests, let alone manage them. Likewise, it is difficult to analyze test scripts or programs for test data adequacy, unless the tests are executed in an environment that captures execution statistics that can be analyzed for coverage of test criteria.

3.3 Test Description Languages

3.3.1 Writability, Readability and Understandability

Existing test description languages are mostly partition description languages. They are relatively simple, making tests easy to write, read, and understand.

3.3.2 Test Data and Oracle Provision/Automation

Test description languages are not occupied with test data, rather they support partitioning the input domain. The step to provide final test input data is left to the tester, which may not be easy. Nor do test description languages deal with test oracles or expected results

Most of the test description languages are accompanied by tool support. Some include execution support for test input (such as TDD); some do not (such as TTF).

3.3.3 Maintainability, Portability, and Regression Testability

Test descriptions are free of implementation details, thus they are not sensitive to implementation evolution. For application requirement changes, maintainability depends on the language used and its closeness to the requirement specification. Test descriptions are platform-independent (hardware and operating system, at least) and thus have high portability. Porting to another test tool can be costly, but the organization and formality of the description language will facilitate this.

3.3.4 Manageability and Measurability

As discussed above, test descriptions specify input partitions rather than tests directly. Other tools are needed to derive or generate tests from the partition, yet this makes

4.2 Overview of TestTalk Language

A TestTalk description consists of the following definition sections: setting, dialect, transformation rule set, scenario / action list, oracle, and test suite. These sections can be provided in any order and can be scattered in different locations and files, as long as the TestTalk parser and translator can locate them. Each section has a name. There can be more than one definition section of a type as long as their names differ. The complete language reference can be found at [12]. Here, we briefly introduce the purpose of different sections of a TestTalk description. The example in the next section will show the value of tests written in TestTalk.

A *setting definition* section defines the "one-for-all" testing environment. It includes information like the executable file name and path for the AUT, the dialect name for describing tests, and generated file names. Whenever the AUT advances to a new revision or platform or testers switch test tool, a new setting should reflect the new environment.

A *dialect definition* section defines a dialect used in scenarios, test cases, action lists, or oracle definitions. A dialect consists of a collection of verbs or predicates, which are the vocabulary of a customized TestTalk.

A *transformation rule set definition* section defines how scenarios, test cases, action lists, or oracles in a certain dialect should be transformed into executables or other useful formats for test automation. The target content of these transformation rules is an arbitrary value that is interpretable in the chosen test environment, where the tests are finally carried out.

Scenario / Action List definition sections define steps to carry out tests.

Oracle definition sections define test oracles using a dialect.

A *test suite* definition section defines a set of test cases that are logically related. These test cases are grouped together to achieve a certain coverage or to check certain aspects of the AUT.

With information from all these TestTalk definition sections, the TestTalk parser and translator generates executable tests from test descriptions. We will show a simple example in the next section.

4.3 The Calculator Example

A simple calculator problem demonstrates both the simplicity and the power of TestTalk. Fig. 8 shows a brief description of the functional part of the calculator. Input/output formats are not specified in this description.

> *This is a simple calculator with an internal stack. The calculator can do arithmetic (+,-,*,/) on integers. The stack can store as many numbers as the computer memory can possibly hold.*

Fig. 8. A brief description of a simple calculator with stack

The first implementation, `calc.v1`, has a command-line interactive interface. A typical scenario of `calc.v1` appears in Fig. 9 (showing exact input and output): (1+3) is calculated, the result is pushed onto the stack; (5+7) is calculated, the result is

pushed onto the stack; the depth of stack is queried; (`pop + top`) is calculated; the depth of stack is queried again.

```
% calc.v1
>1
Answer: 1
>+
Answer: 1
>3
Answer: 4
>push
Answer: 4
>5
Answer: 5
>+
Answer: 5
>7
Answer: 12
>push
Answer: 12
>depth
the depth of stack: 2
Answer: 2
>pop
the number popped from stack: 12
Answer: 12
>+
Answer: 12
>top
the number at the top of stack: 4
Answer: 16
>depth
the depth of stack: 1
Answer: 1
>
```

Fig. 9. A typical scenario.

Fig. 10 provides the TestTalk scenario definition for the scenario in Fig. 9. `Start`, `Feed`, and `Quit` are vocabulary defined in the dialect section that appears in Fig. 11. The tester of the calculator defines the dialect `SimpleCalcWithStack` so that he/she can easily describe tests, including the term `Feed`. `Start` and `Quit` are defined in the dialect `Core`, a more general dialect reusable for other applications, which defines vocabulary for launching and halting an AUT. The fact that a dialect can include another dialect gives testers great convenience to further customize TestTalk based on other testers' work.

```
Scenario "Typical"
 Start $app ;
 Feed "1" [1];
 Feed "+" [1];
 Feed "3" [4];
 Feed "push" [4];
 Feed "5" [5];
 Feed "+" [5];
 Feed "7" [12];
 Feed "push" [12];
 Feed "depth" [2];
 Feed "pop" [12];
 Feed "+" [12];
 Feed "top" [16];
 Feed "depth" [1];
 Quit $app;
End Scenario
```

Fig. 10. A TestTalk Scenario Definition.

```
Dialect "Core"
Action Vocabulary:
 Start $app;  -- Launch the application
 Quit $app;   -- Halt the application
 CheckWith $1;
End Dialect
Dialect "SimpleCalcWithStack"
 Set defaultOracle "check result"
 Include Dialect "Core"
Action Vocabulary:
 Feed $1 [$2];
  -- Feed string $1 as input into the application,
  -- and check result with default oracle with
  -- a parameter $2
Oracle Vocabulary:
 PromptIs $1;   -- Check if the prompt is $1
 ResultIs $1;   -- Check if result matches $1
End Dialect
```

Fig. 11. A TestTalk Dialect Definition [4]

Oracles can be invoked explicitly or implicitly. Square brackets within a TestTalk scenario stand for implicit invocation of the default oracle, where actual parameters appear between "[" and "]". Fig. 12 defines the default oracle `check_result`. An oracle definition can be nested (e.g., `check_prompt` within `check_result`), which allows testers to build sophisticated oracles based on existing oracles and makes it easier to maintain consistency and clarity of TestTalk descriptions.

```
Oracle "check_prompt" :
 PromptIs ">";
End Oracle
Oracle "check_result" $Expected :
 ResultIs $Expected;
 CheckWith "check_prompt";
End Oracle
```

Fig. 12. A TestTalk Oracle Definition.

The transformation rule set defines how actions are transformed into executable forms in the target test environment. A rule is defined as "`SOURCE -> TARGET`". "<<" means that all lines between that "<<" and a "." at the beginning of a line are the target.

---

[4] Anything between "--" and end of line is comment.

The target format of transformation rule set `ExpectScriptGenerator`, in Fig. 13, is Expect/Tcl script. The right hand side of rule `ResultIs` only checks the number after string `Answer:`, any other outputs in Fig. 9 are ignored.

```
TransformationRuleSet "ExpectScriptGenerator"
 HEADER -> <<
#!/opt/public/bin/expect -f
.
 Start $app -> <<
spawn $app
set timeout 3
expect ?
.
 Feed $1 [$2] -> <<
send "$1\\n"
$2
.
 PromptIs $1 -> <<
expect "$1"
.
 ResultIs $1 -> <<
expect {
  "Answer: $1" {
  }
  timeout {
     puts "failed!\\n"
     exit
  }
}
.
 Quit $app -> <<
.
 FOOTER -> <<
puts "\\nSuccessfully reached the end of this test case!\\n"
.
End TransformationRuleSet
```

Fig. 13. A TestTalk Transformation Rule Set Definition.

The name of default oracle and other environment settings are defined in a setting section, in Fig. 14.

```
Setting "Calc.v1 AutoTester With Expect"
 Set defaultDialect "SimpleCalcWithStack"
 Set defaultOracle "check result"
 Set app "../calc/calc.v1"
  -- name of application executable
 Set transformationRuleSet "ExpectScriptGenerator"

-- File name of the generated test program
 Set extension ".1.expect"
 Set prefix "g"
 Set executable yes
End Setting
```

Fig. 14. A TestTalk Setting Definition.

Using transformation rule set definition `ExpectScriptGenerator` and the setting, dialect, and oracle definitions, the TestTalk parser and translator translates the TestTalk test into an Expect script that can be automatically executed to test `calc.v1`.

4.4 Evaluation of the Calculator Example

The real power of TestTalk starts when numbers of tests are large and when changes occur. Suppose the calculator is a complex program and we have hundreds of thousands of tests just like this example test. Suppose that the output format of the calculator is changed. Instead of printing "`Answer: <NUMBER>`" (Fig. 9), the new version of calculator simply prints "`<NUMBER>`". If the large number of tests we have are all written in Expect, we will have to comb through every single one of them and find all "`Answer:`" to modify one by one. With TestTalk, we simply change the transformation rule for `ResultIs` to:

```
ResultIs $1 -> <<
expect {
  "$1" {
  }
  timeout {
    puts "failed!\\n"
    exit
  }
}
```

Voila! All those hundreds of thousands of tests can be up and running in no time.

The same thing can happen a revision of the calculator employs a graphical user interface instead of a command-line interface. Expect does not work for GUIs, thus a GUI test automation tool (such as Visual Test) must be used to automate tests. Again, all TestTalk tests survive. We need only write a new set of transformation rules to transform TestTalk tests into forms recognized by the new GUI test automation tool. Likewise, the calculator could be transplanted from Unix to another platform such as Windows NT. Again, all TestTalk tests can be easily transplanted by writing a new set of transformation rules for the new platform.

4.5 Evaluation of TestTalk

Here we evaluate TestTalk along the same criteria as we evaluated the surveyed methods of test description in Sect. 3.

4.5.1 Writability, Readability, and Understandability

TestTalk descriptions are fairly easy to write and read. Transformation rule sets involve test code to automate the tests, which might be difficult for some applications. This is the work, however, testers must do to benefit from test automation. With the help of TestTalk, they only have to do this difficult job once rather than writing code for every single automated test.

TestTalk tests are described by dialects, which are defined by testers themselves. This ensures that TestTalk tests are very easy to understand for those testers as well as other developers and testers who have the same domain knowledge.

4.5.2 Test Data and Oracle Provision/Automation

TestTalk tests have separate and clearly delineated test data and test oracle sections, which are translated into an automated test including test input and test result checking via transformation rules.

4.5.3 Maintainability, Portability, and Regression Testability

For application implementation changes, only affected transformation rules must be modified; all existing TestTalk tests remain the same. Since tests are based upon the application requirement, TestTalk tests may have to be modified in light of requirement changes that affect behavior. But because of the high understandability of the TestTalk language and dialects, these changes require minimal effort.

The transformation rule set and setting definition sections greatly enhance portability of TestTalk tests. When the AUT is transported to a new platform or a new test tool is used, only a new set of transformation rules and a new setting definition are required. All TestTalk tests remain unchanged.

4.5.5 Manageability and Measurability

The fact that all tests are described separately in different TestTalk scenario or test case definitions makes it easy to use with test management tools. TestTalk does not yet have built-in test adequacy measurement, yet likewise, the separation of TestTalk tests makes it easy to integrate with test adequacy measurement tools. Moreover, the flexibility of the TestTalk transformation rule mechanism makes it possible to implement mechanisms for test adequacy measurement.

Another benefit of TestTalk is that it is possible to write tests before either the test harness or AUT is available. As long as dialects are defined, the TestTalk parser can check the syntax of test descriptions. When the AUT and test harness are ready, automated test execution can start immediately. This enables testers to work in parallel with programmers, which is very critical to commercial software projects where time reserved for testing are counted by days rather than months.

5 Summary of Evaluation

Table 1 summarizes the benefits of using TestTalk and compares it with the other surveyed methods of software test description.

6 Conclusion and Future Work

TestTalk is a software test description language designed for describing automated software tests in a manner natural to the software testing process rather than the programming or development process. Software tests in TestTalk are understandable, complete, maintainable and portable, yet executable.

Test automation alone can not assure software quality. The number of possible tests for any complex software system is so large that even with automated support, all of them can not be tested. Test adequacy measurement is an important complement to test automation. We plan to add support for test adequacy measurement to TestTalk, using the TestTalk transformation rule mechanism.

A TestTalk test is very readable. Testers can easily embed them into natural language specifications or test documents, no matter what type of documentation is being employed. The TestTalk test can be extracted by simple filter and the TestTalk translator automates the rest. We are currently working on an enhanced HTML document with a TestTalk tag. We also plan to add features to TestTalk to facilitate its

use by specifiers to write requirement scenarios. A scenario specified within software requirements can often be used for testing purposes and vice versa. If specifiers can use TestTalk to write scenarios, useless duplication and possible loss of accuracy can be avoided.

Finally we plan to conduct case studies where commercial software development teams use TestTalk. Only real projects can prove the utility of a software language or tool.

Table 1: Evaluation of software test description practices.

Evaluation Criteria	**Informal Languages**	**Scripting or Programming Languages**	**Test Description Languages**	**TestTalk**
Writability/ Readability	High, but difficult to control quality	Low	High	***Very high***
Understandability	Potentially high, but ambiguous	Low	High	***Very high***
Test Input Data Provision/ Automation	Possible (depends on effort), more common in structured formats; Not automated	***Yes***, but mixed with other information; Automated	No, test data generation left to tester; Automation depends on tool support	***Yes***, clearly separated; Automated
Test Oracle Provision/ Automation	Possible (depends on effort), expected outputs common in structured formats; Not automated	***Yes***, but mixed with other information; Automated	No; Not automated	***Yes***, clearly separated; Automated
Maintainability	Potentially high, but depends on quality	Very Low	High	***Very high***
Portability	***Very high***	Low	***Very high***	***Very high***
Regression Testability	Low, and high rework required	Low, and high rework required	Medium, depends on tool support	***High*** (only transformation rules change
Manageability	Depends on quality	Very low	Possible, depends on tool support	***Very High***
Measurability	Depends on quality	Low	***High***, for black box testing	Possible (future work)

References

1. W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky, "Verification, Validation, and testing of Computer Software", ACM Computing Surveys, 14(2):159-192, June 1982.
2. Thomas R. Arnold, "Visual Test 6 Bible", IDG Books Worldwide, November 1998, ISBN: 0764532553.
3. J. A. Bauer and A. B. Finger, "Test Plan Generation Using Formal Grammars", Proceedings of the Fourth International Conference on Software Engineering, Munich, September 1979.
4. Juei Chang, "Automating Specification-based Test Coverage Measurement and Test Selection", Ph.D. Dissertation, University of California, Irvine, March 1998.
5. Juei Chang, Debra J. Richardson, and Sriram Sankar, "Structural Specification-based Testing with ADL," in Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'96), January, 1996, San Diego, California.
6. Laura K. Dillon and Q. Yu, "Oracles for checking temporal properties of concurrent systems", In Symposium on Foundations of Software Engineering. 140-153. December 1994.
7. A.G. Duncan and J.S. Hutchison, "Using attributed grammars to test designs and implementations", In Proceedings of the 5th International Conference on Software Engineering, April 1981.
8. John B. Goodenough and Susan L. Gerhart, "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, SE-1(2):156-173, June 1975
9. P. Jalote, "Specification and testing of abstract data types", Computing Languages, 17: 75-82. 1992.
10. Paul C. Jorgensen, "Software Testing: A Craftsman's Approach", CRC Press, 1995, ISBN 0-8493-7345-X.
11. Don Libes, "Exploring Expect", O'Reilly, 1995, ISBN 1565920902.
12. Chang Liu, Debra J. Richardson, "TestTalk Language Reference", Technical Report 99-07, Information & Computer Science, University of California, Irvine, February 1999.
13. Chang Liu, Debra J. Richardson, "Programming Languages Considered Harmful in Writing Automated Software Tests", Technical Report 99-09, Information & Computer Science, University of California, Irvine, February 1999
14. Robert A. Martin, "Dealing with Dates: Solutions for the Year 2000", IEEE Computer, March 1997.
15. Peter M. Maurer, "Generating test data with enhanced context free grammars" IEEE Software, 7(4):50-56, July 1990.
16. Thomas J. Ostrand and Marc J. Balcer, "The Category-Partition Method For Specifying and Generating Functional Tests", Communications of the ACM, Volume 31, Number 6, June 1998.
17. John Ousterhout, "Tcl: An Embedded Command Language", Proceedings of The Winter 1990 USENIX Conference, Washington, D.C., Jan 22-26, 1990.
18. Debra J. Richardson, "TAOS: Testing with analysis and oracle support", In Proceedings of the 1994 International Symposium on Software Testing and Analysis, Seattle, August 1994, ACM Press.
19. Debra J. Richardson, S.L. Aha, and T.O. O'Malley, "Specification-based test oracles for reactive systems", In Proceedings of the 14th International Conference on Software Engineering, May 1992.

20. Debra J. Richardson, Owen O'Malley, and Cindy Tittle, "Approaches to Specification-Based Testing", In Proceedings of the ACM SIGSOFT'89 Third Symposium on Software Testing, Analysis, and Verification (TAV3), pages 86-96, Key West, Florida, December 1989, ACM SIGSOFT.
21. P. Stocks and D. Carrington, "Test Templates: a Specification-Based Testing Framework", In Proceedings of the 15th International Conference on Software Engineering, May 1993
22. Sriram Sankar, Roger Hayes, "Specifying and Testing Software Components using ADL", SMLI TR-94-23, Sun Microsystems Laboratories, Inc., April 1994.
23. J.M. Spivey, "The Z Notation: A Reference Manual", Prentice Hall, New York, 1989.