TestTalk Language Reference (Version 0.1)

Chang Liu, Debra J. Richardson

liu@ics.uci.edu, djr@ics.uci.edu

Information & Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

UCI-ICS Technical Report No. 99-07

February 28, 1999

1 Introduction

This technical report explains the TestTalk language. TestTalk is a software test description language [1]. It is designed to describe software test cases and test oracles. The goal is to make software tests readable, portable, maintainable, yet executable. We hope that TestTalk can help overcome problems caused by writing software tests in programming languages [2]. This technical report contains the language specification, examples of TestTalk tests, and a few guidelines on TestTalk usage.

2 Lexical Conventions

2.1 Tokens

A token is the smallest element of a TestTalk description that is meaningful to TestTalk parser and translator. The TestTalk parser recognizes these kinds of tokens: keywords, identifiers, strings, variables, and punctuators.

Tokens are separated by "white space". White space can be none or more: blanks, horizontal and vertical tabs, new lines, formfeeds, and comments.

TestTalk is case sensitive.

```
token:
    keyword
  | identifier
  | string
  | number
  | variable
  | punctuator

keyword:
    "Action"
  | "ActionList"
  | "Coverage"
  | "Dialect"
  | "End"
  | "Include"
  | "Oracle"
  | "RuleSet"
  | "Set"
  | "Setting"
  | "Scenario"
  | "TestCase"
  | "TestSuite"
  | "TransformationRule"
  | "TransRuleSet"
  | "TransformationRuleSet"
  | "Vocabulary"
  | "Where"

identifier:
    [a-zA-Z][a-zA-Z0-9_]*

string:
    "\".*\""

number:
    [0-9]+
  | [0-9]*"."[0-9]+
```

```
variable:
    "$"[0-9]+
  | "$"identifier

punctuator:
    "<<"
  | "->"
  | ":"
  | "."
  | ";"
```

Regular expressions are used here. Please note that '|' means "or".

One exception of token production rule is made within transformation rule, where any thing between "<<" and a "." at the beginning of a line is considered part of the rule and is not analyzed by TestTalk parser.

2.2 Comments

A comment is text that TestTalk parser ignores but that is useful to TestTalk writers and readers. To be convenient for people with different "accents", TestTalk recognizes a number of common comment forms.

```
comment:
    "--.*"
  | "#.*"
  | "//.*"
```

3 The Calculator Example

Throughout this report, a simple calculator application will be used to demonstrate the different part of TestTalk. Here is a brief description of the functional part of the calculator.

> *This is a simple calculator with an internal stack. The calculator can do arithmetics (+,-,*,/) on integers. The result of calculation can be stored in the stack. The numbers stored in the stack can be retrieved later as a part of further calculation.*

A brief description of a simple calculator with stack

One implementation of this calculator "calc1" is written in C++ with a line-based interactive interface. A typical scenario of "calc1" is as follows:

```
% calc1
>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
>
```

A complete listing of TestTalk description which uses this scenario to test "calc1" is in Appendix 1.

4. TestTalk: The Language

A TestTalk description consists of Setting definition sections, Dialect definition sections, Transformation rule set definition sections, Oracle definition sections, and Scenario / Action List definition sections, Test suite definition sections. In the future, it will also include Test data adequacy criteria definition sections. These sections can be defined in any order. They can be scattered in different locations, files, as long as TestTalk parser and translator can locate them.

```
TestTalk-description:
    section-list

section-list:
    section
  | sectiionlist section

section:
    setting-definition-section
  | dialect-definition-section
  | transformation-rule-set-definition-section
  | oracle-definition-section
  | scenario-definition-section
  | actionlist-definition-section
  | test-suite-definition-section
  | test-date-adequacy-criteria-definition-section
```

4.1 Setting Definition Sections

A setting definition section defines a setting. A setting represents a testing environment. It usually includes information like the name and path of executable file of application-under-test, dialect name in which tests are described, the file name of generated files. For each setting, a set of automated tests will be generated. Whenever application-under-test advances to a new revision, or testers switch test automation tools, or the application-under-test is transplanted to a new platform, or in any other desired situation, it is recommanded to define a new setting to create a new testing environment.

A setting is defined like this:

```
setting-definition-section:
    "Setting" setting-name
       declaration-list
```

```
    "End" "Setting"

setting-name:
    string

declaration-list:
    declaration
  | declaration-list declaration

declaration:
    "Set" variable-name value

variable-name:
    identifier

value:
    string
```

Here, "variable-name" can be any identifier. The tester can refer to the value of these variables in scenario definition or other sections by using these variable names.

A few variables have special meaning to TestTalk translator. They are:

* defaultDialect: defines the default dialect used in scenario, test case or action list definition.

* defaultOracle: defines the default oracle name used in scenario, test case or action list definition when oracle name is omitted in oracle invocation.

* transformationRuleSet: defines the default transformation rule set used to translate scenarios, test cases and action lists.

The following three variables defines the name of generated files:

* extension: the extension of generated file name. The default value is "g_", which is short for "generated".

* prefix: the prefix of generated file name. The default value is ".tt", which is short for "TestTalk".

* executable: if its value is yes, then the file property of the generated file will have an executable attribute.

An example setting definition section is like this:

```
Setting "Calc1 AutoTester With Expect"
  Set defaultDialect "SimpleCalcWithStack"
  Set defaultOracle "check result"
  Set app "../calc/calc1" -- name of application executable
  Set transformationRuleSet "ExpectScriptGenerator"
  Set extension ".1.expect"
  Set prefix "g"
  Set executable yes
End Setting
```

In this case, if a scenario has a name "Typical", then the generated file will be "gTypical.1.expect". And it will be an executable file.

4.2 Dialect Definition Sections

Dialect definition sections define dialects that can be used later in scenario, test case, action list, or oracle definition. A dialect consists of a collection of verbs or predicates. It also defines how each verb should be followed by objects. The meaning of these verbs could be explained informally in comments. Transformation rules are the ultimate authority in defining these verbs' semantics. However, there might be multiple

transforamtion rule sets for a certain dialect. It is up to the TestTalk description writer to make sure they're consistent.

```
dialect-definition-section:
    "Dialect" dialect-name
       declaration-list
       ["Include" "Dialect" dialect-name]
    ["Action" "Vocabulary" ":"
       verb-definition-list     ]
    ["Oracle" "Vocabulary" ":"
       predicate-definition-list     ]
    "End" "Dialect"

dialect-name:
    string

verb-definition-list:
    /* empty */
  | verb-definition
  | verb-definition-list verb-definition

verb-definition:
    verb object-list "[" object-list "]"

verb:
    identifier

object-list:
    /* empty */
  | object
  | object-list object

object:
    variable

predicate-definition-list:
    /* empty */
  | predicate-definition
  | predicate-definition-list predicate-definition

predicate-definition:
    predicate object-list

predicate:
    identifier
```

"[ ]" means optional.

An example dialect definition section is like this:

```
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
```

Here, verb "Feed", and predicates "PromptIs", and "ResultIs" are defined. If a scenario, a test case, or an action list is going to be written in dialect "SimpleCalcWithStack", the writer can use these three words as well as all words in the vocabulary of dialect "Core".

4.3 Transformation Rule Set Definition Sections

Transformation rule sets define how scenarios, test cases, action lists, or oracles in a certain dialect should be transformed into some executable formats or other useful formats for test automation. The target content of these transformation rules is an arbitrary value that is interpretable only the chosen test automation tool that's responsible for finally carrying out the test. For example, the target content could be pure test input in a text file, or a test script, or even program source code.

```
transformation-rule-set-definition-section:
    "TransforamtionRuleSet" rule-set-name
        rule-list
    "End" "TransformationRuleSet"

rule-list:
    /* empty */
  | rule
  | rule-list rule

rule:
    left "->" right [ "Where" parameter-trans-rule-list "End"]

left:
    verb-definition
  | predicate-definition

right:
    string
  | "<<" [.\n]* "\n\."

parameter-trans-rule-list:
    /* empty */
  | parameter-trans-rule
  | parameter-trans-rule-list parameter-trans-rule

parameter-trans-rule:
    string "->" string ";"
```

Shorter keywords "RuleSet", "TransRuleSet", or "TransformationRule" can be used instead of the long keyword "TransforamtionRuleSet".

The second production rule of "right" means anything between "<<" and a "." which sits at the beginning of a line.

If a variable appeared in the body of right side of a transformation rule, it will be replaced with the value of the variable or parameter when transformed. For example,

"$app" will be replaced by the value of variable $app, which is "../calc/calc1" in our example, "$1" will be replaced by the first parameter. To keep a "$" from being interpreted as a special character leading variables, the escape character "\" can be used. For example, "\$app" will be translated into "$app" instead of the value of variable $app. "\\" will be translated into "\".

Parameters to a particular rule can be replaced according to parameter transformation rules before it's used. For example, if a parameter transformation rule ("File" -> "&f") is defined for a rule, whenever the string "File" is passed as a parameter, "&f" will be used in the generated files instead of "&f". An example of this is:

```
InvokeMenu $menuItem-> <<
    SendKey(\"$menuItem\");
.
Where
  "File" -> "&f" ;
  "Open" -> "&o" ;
End
```

"HEADER" and "FOOTER" are two special transformation rules. They are automatically invoked before and after the transformation of a scenario.

An example transformation rule set definition section is this:

```
TransformationRuleSet "TestInputGenerator"
  HEADER -> <<
.

  Start $app -> <<
.

  Feed $1 [$2] -> <<
$1
.

  PromptIs $1 -> <<
.
  ResultIs $1 -> <<
.

  Quit $app -> <<
.

  FOOTER -> <<
.

End TransformationRuleSet
```

Transforamtion rule set "TestInputGenerator" basicly says only first object of verb "Feed" is kept in the generated file. Anything else is transforamtion to empty string.

4.4 Oracle Definition Sections

Oracle definition sections define test oracles in a dialect.

```
oracle-definition-section:
    "Oracle" oracle-name fomal-parameter-list ":"
       check-list
    "End" "Oracle"

oracle-name:
    string
```

```
check-list:
    check
  | check-list check

check:
    predicate oracle-parameter-list
  | "CheckWith" oracle-name oracle-parameter-list

oracle-parameter-list:
    /* empty */
  | oracle-parameter
  | oracle-parameter-list oracle-parameter

oracle-parameter:
    variable
  | string
  | number
```

Due to implementation limitation, oracles can only take one parameter in version 0.1. The "CheckWith" clause can't take parameters. These limitations will be changed soon.

An example oracle definition section is as follows:

```
Oracle "check result" $Expected :
  ResultIs $Expected;
  CheckWith "check prompt";
End Oracle
```

This oracle checks result first, then invoke another oracle to check prompt.

4.5 Scenario / Action List Definition Sections

Scenario / Action List definition sections defines steps to carry out software tests. Scenarios and action lists are pretty much the same, except that scenario is a complete action list of application from launching to closing. On the other hand, an action list can start from any point of application execution and end at any point without closing the application. Test case definition are very similar to scenario definition, except that a test case definition is always part of a test suite definition.

```
scenario-definition-section:
    "Scenario" scenario-name
      action-list
    "End" "Scenario"

action-list-definition-section:
    "ActionList" action-list-name
       action-list
    "End" "ActionList"

action-list:
    /* empty */
  | action
  | action-list action

action:
    verb actual-parameter-list [ "[" actual-parameter-list" "]" ]
  | "CheckWith" oracle-name actual-parameter-list
  | "Include" "ActionList" action-list-name

actual-parameter-list:
    /* empty */
  | actual-parameter
  | actual-parameter-list actual-parameter
```

```
actual-parameter
    variable
  | string
  | number
```

An example scenario definition section is as follows:

```
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
```

This scenario is the TestTalk description of the calculator scenario that is listed at the beginning of this document.

4.6 Test Suite Definition Sections

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 application-under-test.

```
test-suite-definition-section:
    "TestSuite" test-suite-name
       test-case-list
    "End" "TestSuite"

test-suite-name:
    string

test-case-list:
    /* empty */
  | test-case
  | test-case-list test-case

test-case:
    "Include" "Scenario" scenario-name
  | "TestCase" test-case-name
       action-list
    "End" "TestCase"

test-case-name:
    string
```

4.7 Test Data Adequacy Criteria Definition Sections

This type of sections is not implemented in version 0.1 yet.

5 Possible Future Enhancement

Besides to finish a few unimplemented features mentioned above, we'd like to see more enhancement to TestTalk in the future. Here's a few of them.

A repetitive mechanism is useful when a certain action pattern is to be repeated for several times. However, we will be careful not to introduce complex repetitive conditions that are common in programming languages.

A TestTalk test is very readable text. Testers could easily embedded it into natural language specifications or test documents, no matter what kinds of documentation systems they're using. Microsoft word, WordPerfect, and HTML are some of the popular document formats people use. It's pretty easy to embed TestTalk test into such documents. What we need is just a simple filter to get the TestTalk spec and the TestTalk translator will do all the rest. We're currently working on enhanced HTML document with a TestTalk tag.

A scenario for software requirement purpose can often be used for testing purpose, and vice versa. If specifiers can use TestTalk to write scenarios, boring duplication and possible loss of accuracy could be avoided. We may add some features to make TestTalk also a good tool for specifiers to write requirement scenarios.

This is still version 0.1 of TestTalk. We are aware that for TestTalk to fit nicely into all kinds of different situations: different platforms, languages, test hooks, documentation systems, etc, a lot of improvement and refinement need to be done. We have a few plans above. We are also prepared to include more after we do more case studies.

6 References

[1] Chang Liu, Debra J. Richardson, "TestTalk, a Test Description Language: Write Once, Test by Anyone, Anytime, Anywhere, with Anything", Technical Report 99-08, Information & Computer Science, University of California, Irvine, February 1999.

[2] 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.

Appendix 1: listing of a complete TestTalk example

This TestTalk example tests "calc1" and an newer version of the same calculator "calc2". "calc2" does exactly the same thing as "calc1" but with slightly input and output formats change.

Pieces of this TestTalk example have been used in this technical report to demonstrate the TestTalk language. We include the complete listing here so readers can have a better understanding of how those pieces relate to each other.

The following is the listing of "typical1.TestTalk":

```
Setting "Calc1 AutoTester With Expect"
  Set defaultDialect "SimpleCalcWithStack"
  Set defaultOracle "check result"
        -- default oracle could be define in Dialect, but it
        -- could be overriden here.
  Set app "../calc/calc1" -- 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
```

```
Setting "Calc2 AutoTester With Expect"
  Set defaultDialect "SimpleCalcWithStack"
  Set defaultOracle "check result"
  Set app "../calc/calc2" -- name of application executable
  Set transformationRuleSet "ExpectScriptGenerator"

  Set extension ".2.expect"
  Set prefix "g"
  Set executable yes
End Setting

Setting "Test Input"
  Set defaultDialect "SimpleCalcWithStack"
  Set transformationRuleSet "TestInputGenerator"

  Set extension ".in"
  Set prefix "g"
  Set executable no

-- Notice: no defaultOracle and app definition,
-- since it's not needed
End Setting

Setting "Calc1 Parser Base Test"
  Set app "../calc1"
  Set defaultDialect "SimpleCalcWithStack"
  Set defaultOracle "check result 2"
  Set transformationRuleSet "ParserBasedTesterGenerator"
  Set extension ".y"
  Set prefix "g"
End Setting

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

Oracle "check prompt" :
  PromptIs "";
End Oracle

Oracle "check result" $Expected :
  ResultIs $Expected;
  CheckWith "check prompt";
End Oracle

Oracle "check result 2" $Expected :
  ResultIs $Expected;
End Oracle

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

TransformationRuleSet "TestInputGenerator"
  HEADER -> <<
.
  Start $app -> <<
.
  Feed $1 [$2] -> <<
$1
.
  PromptIs $1 -> <<
.
  ResultIs $1 -> <<
.
  Quit $app -> <<
.
  FOOTER -> <<
.
End TransformationRuleSet

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

TransformationRuleSet "ParserBasedTesterGenerator"
  HEADER -> <<
%{
#include &lt;stdio.h
#include &lt;stdlib.h
bool testResult = TRUE;

char * testData[] = {
.

  Start $app -> <<
.

  Feed $1 [$2] -> <<
"$1",
$2
.

  ResultIs $1 -> <<
"$1",
.

  Quit $app -> <<
.

  FOOTER -> <<
} ;

int result_index = 0;
int yylex();
int lineno=1;
int yyerror(char * msg)
{
        printf("line %d: %s at [%s]\\n", lineno, msg, yytext);
}
void output(char * s);

%}

%union {
long     val;
}

%token <val>  NUM

%%
output:    /* empty */
        | output result
;

result:    '>' { /* ignore */ }
         | '\\n' {}
         | NUM {}
         | NUM '\\n' {}
         | NUM '\\n' '>' {
          int expected = atoi(testData[result_index*2+1]);

          if (expected == \$1)
            {
              printf("%d: %d found\\n",result_index, \$1);
            }
```

```
          else
            {
              printf("Warning: Expecting %d while getting %d\\n",
                expected, \$1);
              testResult = FALSE;
            }
          result_index ++;
        }
;

%%

main()
{
  int i;
  FILE * myfp;
  char * inputFileName = "t.in";
  char * outputFileName = "t.out";
  char cmdBuf[500];

  printf("Generating test input %s...\\n",inputFileName);
  myfp = fopen(inputFileName,"w");
  if (myfp==NULL)
    {
      printf("unable to open file %s to write.\\n",inputFileName);
      exit(1);
    }
  for (i=0; i<sizeof (testData) / sizeof (char *); i=i+2)
    fprintf(myfp,"%s\\n", testData[i]);
  fclose(myfp);

  printf("Executing test and saving result in %s...\\n",
    outputFileName);
  sprintf(cmdBuf,"$app < %s > %s",
    inputFileName, outputFileName);
  system(cmdBuf);

  printf("Analyzing test result...\\n");
  yyin = fopen(outputFileName,"r");
  yyparse();
  if (testResult==TRUE)
        printf("Pass!!!\\n");
  else
        printf("Failure found!!!\\n");

}

void output(char * s)
{
    printf("%s",s);
}
.

End TransformationRuleSet
```

Listing of "typical1.TestTalk"

Appendix 2: How to parse and translate a TestTalk description

For version 0.1, here's the location and executable file name of TestTalk parser and translator.

```
cliu1@redondo-beach.ics.uci.edu% pwd
/home/cliu1/public_html/research/TestTalk/v0.1/parser
cliu1@redondo-beach.ics.uci.edu% testtalk < typical1.TestTalk
```

If the TestTalk file passed parsing, output files will be generated under the name that's specified in the TestTalk file. If no name is specified, the default output name is "g_SCENARIONAME.tt", where SCENARIONAME stands for the name of corresponding scenario. "g" is short for "generated". "tt" is short for "TestTalk".