Introduction

What it does

This tool permits a series of tests to be written addressing specific tags or entry points within a project and act to verify that the return results are as expected for that code. The significance of this is that, when run routinely any time that the project is modified, it will act to indicate whether the intended function has been modified inadvertently or whether the modification has had unexpected effects on other functionality within the project. The set of unit tests for a project should run rapidly (usually within a matter of seconds) and with minimal disruption for developers. Another function of unit tests is that they indicate what the intended software was written to do. The latter may be especially useful when new developers start working with the software or a programmer returns to a project after a prolonged period.

The concept of Unit Testing was already in place before Kent Beck created a tool that he used in the language Smalltalk, and then was turned into the tool Junit for Java by Kent Beck and Erich Gamma. This tool for running specific tests on facets of a software project was subsequently referred to as xUnit, since NUnit was developed for .NET developers, DUnit for Delphi developers, etc. MUnit is the equivalent tool for M developers to use and was originally created in 2003.

The current version of M-Unit adds (thanks to Sam Habiel) the ability to determine the time spent in each unit test API, and removes some text related to a potential installation problem on GT.M systems that was found to be unnecessary. In addition, the current version fixes for Cache systems a problem in the pre-installation processing on systems on which M-Unit had not previously been installed and returned to using the original code for checking and adding the %ut global to the current (not just VISTA) namespace, with error checking on the deprecated code and using a newer API if the old one is not present or fails for any reason. The latter insures that M-Unit will run on older installations and continue to function if the original code is no longer present in the future.

Using M-Unit

The M-Unit functionality is contained in the %ut, %ut1 and %utcover routines. The code was originally written by Joel Ivey when he was working as a developer for the Department of Veteran Affairs. The code had input as suggestions by several other developers both inside and outside of the VA, including Kevin Meldrum and especially Sam Habiel who made significant contributions to the current status including modifications to the preinstall routine for Cache to improve setting the %ut namespace for routines and globals to the current VistA partition. Current development is being continued for OSEHRA.

```
%ut      ;VEN-SMH/JLI - PRIMARY PROGRAM FOR M-UNIT TESTING ;12/16/15  08:40
         ;;1.4;MASH UTILITIES;;APR 11, 2016;
         ; Submitted to OSEHRA Apr 11, 2016 by Joel L. Ivey under the Apache 2 license
(http://www.apache.org/licenses/LICENSE-2.0.html)
         ; Original routine authored by Joel L. Ivey as XTMUNIT while working for U.S. Department of Veterans
Affairs 2003-2012
         ; Includes addition of %utVERB and %utBREAK arguments and code related to them as well as other
substantial additions authored by Sam Habiel 07/2013-04/2014
         ; Additional work by Joel L. Ivey 05/14-12/15
         ;
         ; This routine and its companion, %ut1, provide the basic functionality for
```

; running unit tests on parts of M programs either at the command line level
; or via the M-Unit GUI application for windows operating systems.
;
; Original by Dr. Joel Ivey
; Contributions by Dr. Sam Habiel

From a user's perspective the basic start for unit tests from the command line is the entry point EN^%ut, the first argument is the name of the routine to be tested and is required, but the tag can take up to two additional arguments: a verbose indicator and a BREAK indicator, both of these require a non-zero value to activate them.

        D EN^%ut("ROUTINE_NAME")
or
        D EN^%ut("ROUTINE_NAME,VERBOSE,BREAK)

The command with a single argument will result in the unit tests being run and each successful test is shown by a period ('.') followed by specification of the number of tags entered, the number of tests run, the number of failures, and the number of errors encountered. Instead of the period for successes, failures or errors are indicated by the tag and routine name for the specific test, a description of the test if provided, and a message concerning the failure if provided or the line and routine at which the error occurred. The verbose option will result in a listing of each test that is executed, which may make it more difficult to identify problems if they have occurred. The BREAK option will result in termination of the unit test as soon as a failure or error is encountered, this is not usually recommended, since only a part of the unit tests (and potential problems) will have been examined. The unit tests will normally continue even if errors are encountered.

The code written in a unit test routine has specific entry points that should indicate a specific set of functionality being tested. The tag may have more than one test, but these should all focus on the same aspect being tested. Originally specification of the tags and a description of the functionality being tested by the tag testing were entered following an XTENT tag in the following manner.

XTENT ;
        ;;TEST1;Testing functionality for one feature
        ;;ANEW1;Testing another piece of functionality
        ;;ATHIRD;Testing still something else

More recently, an alternative method was added similar to the annotation used in C#, thanks to the suggestion of Kevin Meldrum. The indicator @TEST is specified as the first string following the semi-colon on the same line as the tag, and a description can then be added following this indicator.

TEST4   ; @TEST another test for different functionality

Since there will frequently be multiple routines with tests created to test a specific project, these can be indicated in a manner similar to the original description of the entry tags, following a XTROU tag. The following could be used to link additional test routines to a ZZUXQA1 test routine.

XTROU ;
        ;;ZZUXQA2
        ;;ZZUXQA3

```
        ;;ZZUXQA4
```

The other routines can also reference these as well, or additional related test routines. Each routine would be included only once, no matter how many of the other routines reference it in this manner.

A test routine can use one of three types of calls for its tests, determining truth, equivalence, or simply indicating failure for the test. In each of these a final argument can be used to specify information about the specific test result.

Truth is tested by the command

        DO CHKTF^%ut(TorF,message)

where 'TorF' is a value to be tested for true (passing the test) or false (failing the test).

Equivalence is tested by the command

        DO CHKEQ^%ut(expected,result,message)

where 'expected' is the value that is expected from the test, and 'result' is the value that was obtained and should be equal to 'expected' if the test is to pass. If a test fails, the expected value and the observed values are shown along with the 'message' indicating the test that failed.

Failure already determined is specified by the command

        DO FAIL^%ut(message)

and is generally used when the processing has reached an area that it shouldn't be expected to reach given the circumstances, and 'message' then describes the situation.

The MUnit functionality is set up to capture information on errors, and to continue processing the remaining tests within the tag as well as additional tags.

There are four other tags that have meaning to the MUnit functionality - STARTUP, SETUP, TEARDOWN, and SHUTDOWN. Frequently, to provide specific data to use for testing, it may be necessary to add data which is totally temporary, either for all tests in one pass, or before each test is run.

The STARTUP tag specifies code that should be run once when the testing of a routine is starting up. If multiple routines should use the same STARTUP code, they can have a STARTUP tag that then runs the code in one of the routines. Its companion is SHUTDOWN, which if present, will be run only after all of the tests have been completed within a routine. Again, if multiple routines should use the same SHUTDOWN code they can each have a SHUTDOWN tag and then run the code in one of the routines. This is a change from the prior version, where STARTUP was run only at the start of a unit test sequence and SHUTDOWN only at the conclusion of all tests. However, this was found to cause problems if a suite of multiple unit tests from different applications were being run (e.g., by creating a primary unit test routine which referred to multiple test routines creating a suite of tests), and more than one of the applications required its own STARTUP and SHUTDOWN code.

The SETUP tag specifies code that should be run before each test tag in a given routine is run, there could be similar SETUP tags in other routines as well. Its companion is TEARDOWN which, if present, will be run immediately after each test tag is processed.

It should be noted that care should be taken in using these four tags, since they may end up hiding significant functionality from testing or result in problems later if changes are made to the tests (which would then be converted into changes in the project related to the tests).

The extrinsic function ($$ISUTEST^%ut) can be used to determine whether code is currently running within a unit test or not. The value returned will be true if it is currently in a unit test and false if it is not. This can be used within code that would likely be used under testing to determine whether user interaction might be requested or not, or to set a default value for testing purposes.

An additional tag (CHKLEAKS^%ut) is available for checking for variable leaks as a part of a unit test. This functionality can also be called outside of unit tests as well.

```
CHKLEAKS(%zuCODE,%zuLOC,%zuINPT) ; functionality to check for variable leaks on
executing a section of code
      ; %zuCODE - A string that specifies the code that is to be XECUTED and checked for leaks.
      ;        this should be a complete piece of code
      ;           (e.g., "S X=$$NEW^XLFDT()" or "D EN^%ut(""ROUNAME"")")
      ; %zuLOC  - A string that is used to indicate the code tested for variable leaks
      ; %zuINPT - An optional variable which may be passed by reference.  This may
      ;        be used to pass any variable values, etc. into the code to be
      ;        XECUTED.  In this case, set the subscript to the variable name and the
      ;        value of the subscripted variable to the desired value of the subscript.
      ;          e.g., (using NAME as my current namespace)
      ;             SET CODE="SET %zuINPT=$$ENTRY^ROUTINE(ZZVALUE1,ZZVALUE2)"
      ;             SET NAMELOC="ENTRY^ROUTINE leak test"   (or simply "ENTRY^ROUTINE")
      ;             SET NAMEINPT("ZZVALUE1")=ZZVALUE1
      ;             SET NAMEINPT("ZZVALUE2")=ZZVALUE2
      ;             DO CHKLEAKS^%ut(CODE,NAMELOC,.NAMEINPT)
      ;
      ;        If part of a unit test, any leaked variables in ENTRY^ROUTINE which result
      ;        from running the code with the variables indicated will be shown as FAILUREs.
      ;
      ;        If called outside of a unit test, any leaked variables will be printed to the
      ;        current device.
      ;
```

The COV^%ut API can be used to initiate coverage analysis of unit tests. Previously this functionality was limited to the GT.M version of M (MUMPS), but the current release now provides support for coverage analysis in Intersystems Cache as well. In the original release, this functionality was only available by calling COV^%ut1, but the tag has been moved to %ut to make it more convenient to use. A couple of newly added related APIs are described below as well. The COV^%ut API has three arguments

```
      DO COV^%ut (NAMESPACE,CODE,VERBOSITY)
```

where NAMESPACE specifies the routines to be included in the analysis. If the value does not include an asterick at the end, then only the routine matching the specified name would be included (e.g, "KBBPDEB1", would only include the routine KBBPDEB1 in the analysis). If the NAMESPACE value ends in an asterick, then all routines starting with the initial characters will be included in the analysis (e.g., KBBPD* would include all routines with names starting with KBBPD in the analysis).

CODE specifies the code command that should be run for the analysis. Thus, "DO EN^%ut(""KBBPUDE1"")" would run the routine KBBPUDE1 and any that it might call for the coverage analysis.

VERBOSITY determines the amount of detail to be displayed. A value of 0 or 1 will provide only an analysis of the lines covered out of the total number to be counted (non-code lines are not included in the coverage analysis) for each routine in the analysis, as well as covered and totals for all routines. A value of 2 will also include coverage data for each tag in the routines. A value of 3 will provide the data provided by 1 and 2, but also will list each line for a tag that was not covered during running of the routine(s), so that lines lacking coverage can be determined. A value of -1 will return all data in globals for the calling application to evaluate and present.

The COVERAGE^%ut API has been added to make it easier to analyze the coverage data while having it omit the data on routines that shouldn't be included in the analysis (e.g., those routines which are only unit test routines). It also permits different APIs to be called within the same analysis, so that coverage can be better approximated if different pieces of code need to be called (e.g., an entry point to run unit tests without the verbose flag, and another with the verbose flag, since both count as lines of code). Again, this functionality is currently only available for GT.M systems.

DO COVERAGE^%ut(NAMESPACE,.TESTROUS,.XCLUDE,VERBOSITY)

Where NAMESPACE functions in the same manner as described for COV^%ut (e.g., "%ut*")

TESTROUS is an array specifying the desired APIs that should be called and is passed by reference. If the subscript is non-numeric, it will be interpreted as a routine specification to be used. The values of the array may also be a comma separated list of APIs to be used during the analysis. If an API includes a '^' (as either TAG^ROU or ^ROU) then it will be run as DO TAG^ROU or DO ^ROU. If the API does not include a '^' then it will be run as DO EN^%ut("ROU"). An array could look like
    SET TESTROUS(1)="^%ut,^%ut1"
    SET TESTROUS("%utt1")="VERBOSE^%ut1"
which would cause the unit tests   DO ^%ut, DO ^%ut1, DO EN^%ut("%utt1"), and
DO VERBOSE^%ut1 to be run.

XCLUDE is an array specifying the names of routines that should be excluded from the coverage analysis, and can also be specified as either arguments or as a comma separated list in the value. Thus,
    SET XCLUDE("%utt1")="%utt2,%utt3,%utt4,%utt5,%utt6,%uttcovr"
would result in only the functioning routines in %ut* being included in the coverage analysis.

The VERBOSITY argument can have the 0 through 3 values as described above.

The MULTAPIS^%ut API has been added to provide capabilities to run multiple sets of unit tests in the same manner as with the COVERAGE^%ut API, but it does not attempt to perform any coverage analyses. It has a single argument is passed by reference and has the same capabilities as TESTROUS above. Usage is as

        DO MULTAPIS^%ut(.TESTROUS)

The new GETUTVAL^%ut and LSTUTVAL^%ut APIs can be used to generate cumulative totals If a routine with code to run multiple unit tests is created by calling the GETUTVAL^%ut API after each test passing a variable (which can be undefined initially) by reference to create an array containing a cumulative total for the tests. At the conclusion, the LSTUTVAL^%ut API can then be called to print the cumulative totals.

        DO GETUTVAL^%ut(.TESTSUM)
Then
        DO LSTUTVAL^%ut(.TESTSUM)
Will present the summary listing of values for the tests.

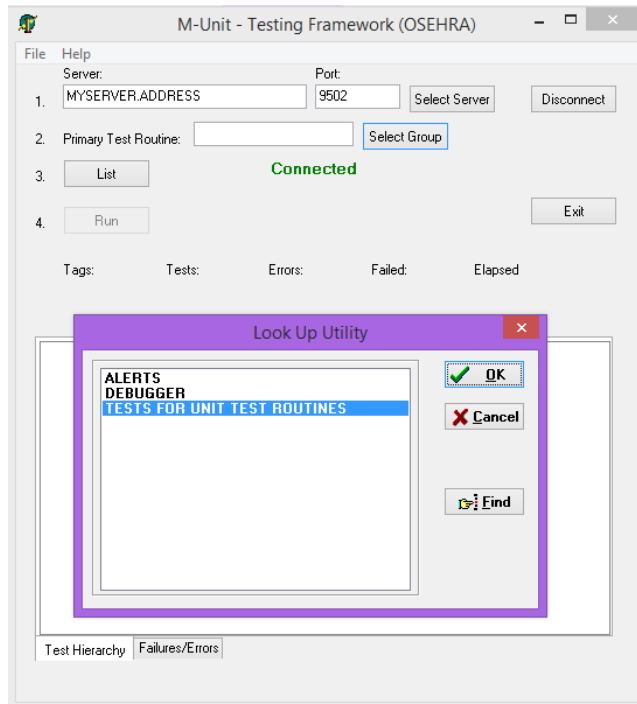The GUI MUnit application provides a visually interactive rapid method for running unit tests on M code.

Figure 1. Selection of an M-Unit test

After specifying the server address and port, the user can sign on and then click the Select Group button to select a unit test from the M-UNIT TEST GROUP file (#17.9001) as shown here (Figure 1), or simply enter the name of a unit test routine in the Primary Test Routine field and click on List. This will bring up a list of the routines and tags in the unit test run (Figure 2).
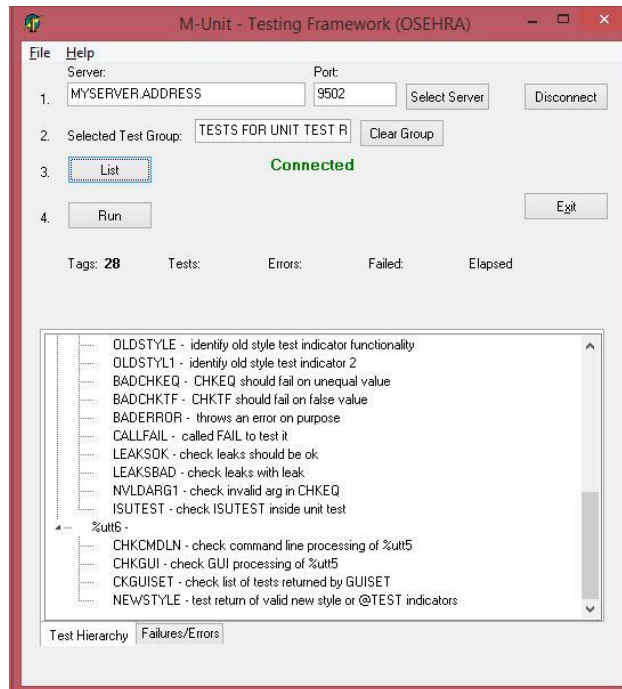
Figure 2. List of Unit tests selected for running

Clicking the Run button will run the unit tests, resulting in a bar which is green if all tests pass or red if any failures or errors are encountered (Figure 3).
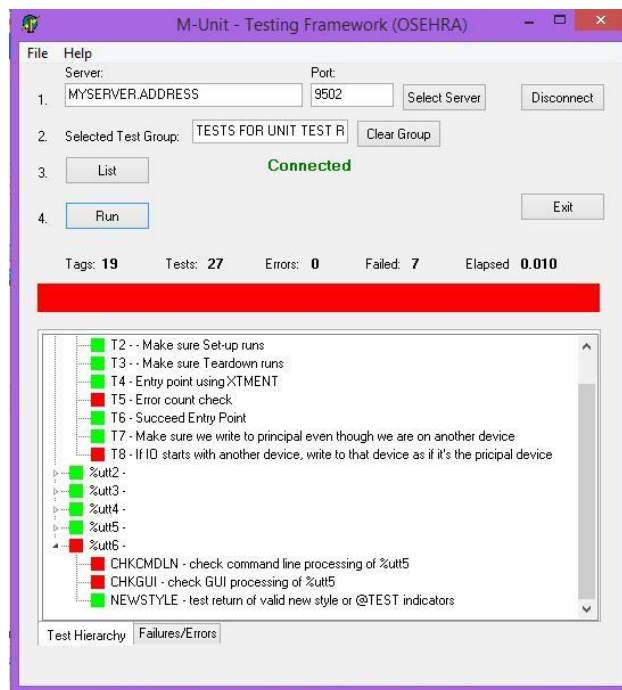

Figure 3. The unit tests run with failures

If failures or errors are encountered, clicking on the Failures/Errors tab at the bottom of the listing opens a display of specific information on the problems.
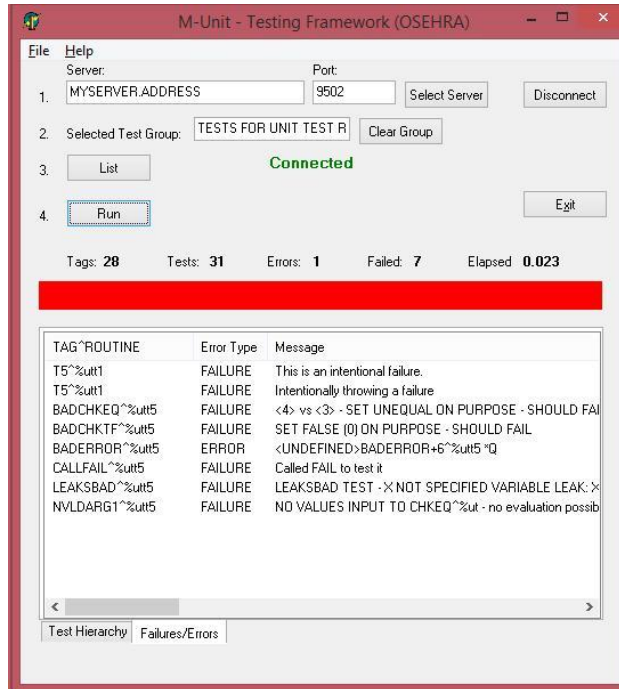

Figure 4. Specifics on failed tests or errors

In the case shown (Figure 4), all of the failures are intentional. Usually, failures and/or errors are not intentional and the user can then edit the routine, and save the changes, then simply click on the Run button again to see the effect of the changes.

To select a new unit test, the user would click on the Clear Group button, then again either select another group or as shown in Figure 5, entering the name of a unit test routine (ZZUXQA1 and related routines are not included with the M-Unit Test code and is shown only as an example) and clicking on the List button.
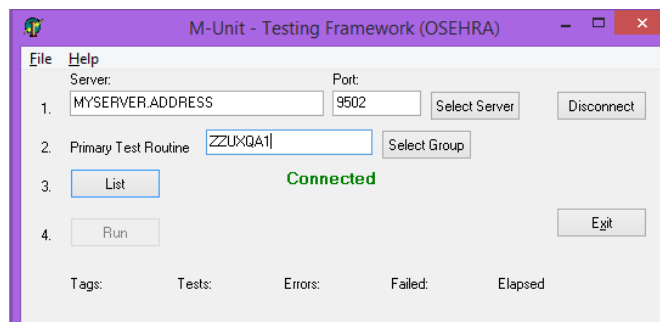

Figure 5. Specification of unit tests by routine name

Again, clicking the Run button will run the unit tests (Figure 6). This figure shows the desired result, a green bar meaning that all tests passed.
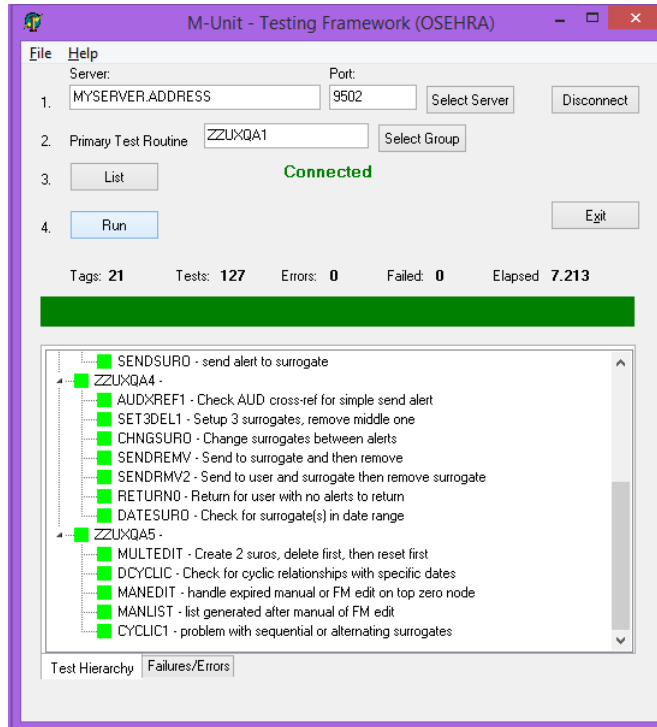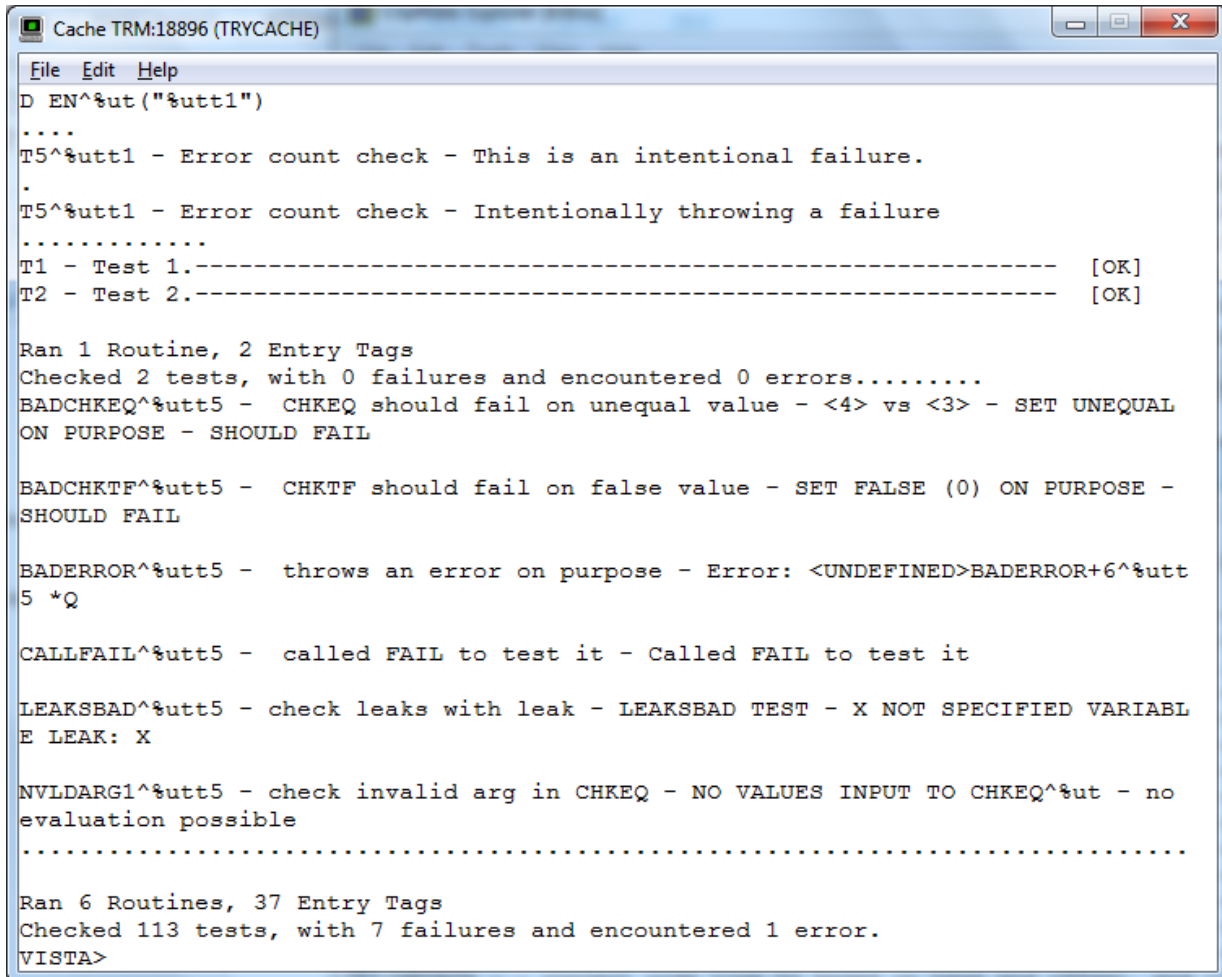

Figure 6. Result from the second group of unit tests

The results of %utt1 and related routines run at the command line without the verbose flag are shown in Figure 7.

```
Cache TRM:18896 (TRYCACHE)                                         ─ ▢ ✕

 File  Edit  Help
D EN^%ut("%utt1")
....
T5^%utt1 - Error count check - This is an intentional failure.
.
T5^%utt1 - Error count check - Intentionally throwing a failure
.............
T1 - Test 1.----------------------------------------------------- [OK]
T2 - Test 2.----------------------------------------------------- [OK]

Ran 1 Routine, 2 Entry Tags
Checked 2 tests, with 0 failures and encountered 0 errors.........
BADCHKEQ^%utt5 -  CHKEQ should fail on unequal value - <4> vs <3> - SET UNEQUAL
ON PURPOSE - SHOULD FAIL

BADCHKTF^%utt5 -  CHKTF should fail on false value - SET FALSE (0) ON PURPOSE -
SHOULD FAIL

BADERROR^%utt5 -  throws an error on purpose - Error: <UNDEFINED>BADERROR+6^%utt
5 *Q

CALLFAIL^%utt5 -  called FAIL to test it - Called FAIL to test it

LEAKSBAD^%utt5 - check leaks with leak - LEAKSBAD TEST - X NOT SPECIFIED VARIABL
E LEAK: X

NVLDARG1^%utt5 - check invalid arg in CHKEQ - NO VALUES INPUT TO CHKEQ^%ut - no
evaluation possible
.................................................................

Ran 6 Routines, 37 Entry Tags
Checked 113 tests, with 7 failures and encountered 1 error.
VISTA>
```

Figure 7. Command line unit tests for %utt1

The results of the single %utt1 unit test routine (and its related routines) run with the VERBOSE option, that some people prefer, specified permits the individual tests and their results to be seen, but makes the results more difficult to interpret (Figure 8).
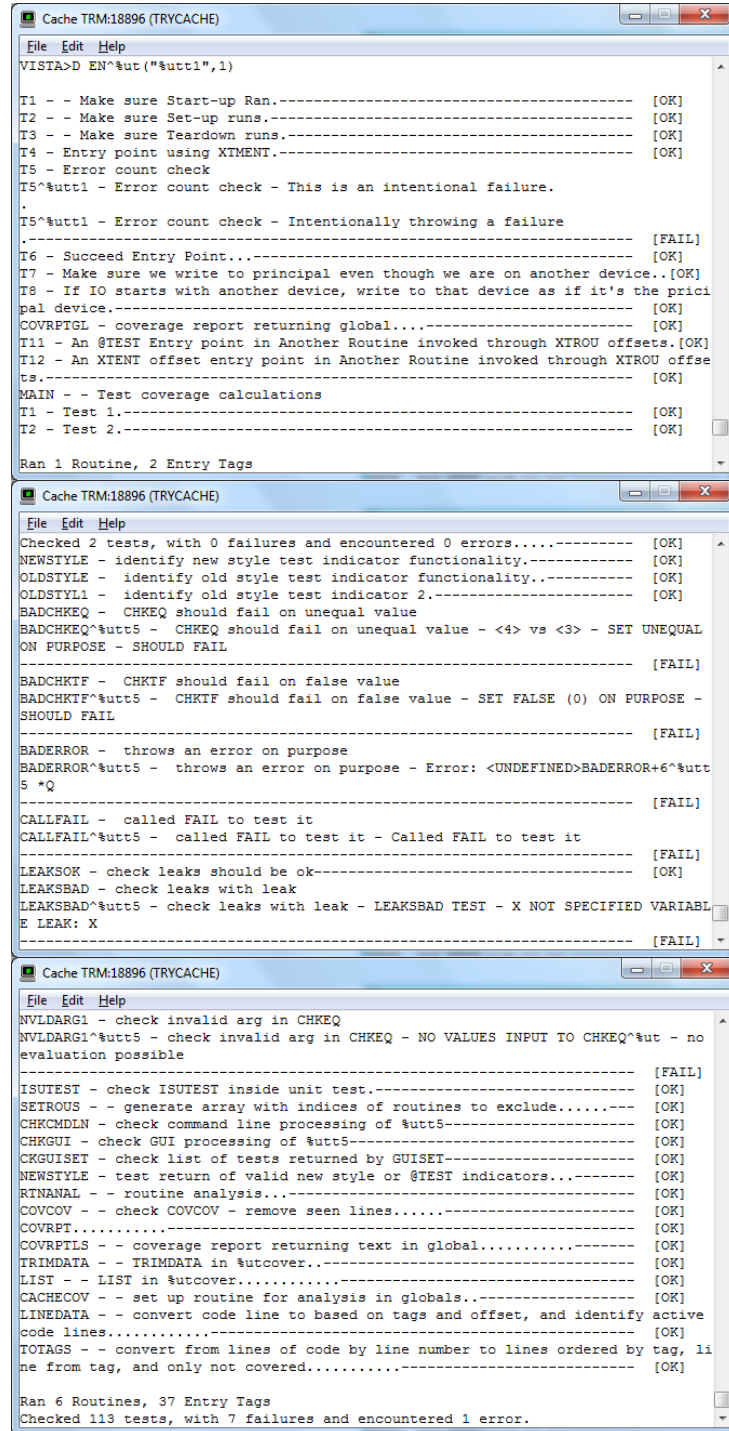

Figure 8. Command line unit tests for %utt1 with VERBOSE option

Running Coverage Analysis of Unit Tests for the M-Unit code

Running the routine ^%uttcovr from the top (I.e., D ^%uttcovr) runs a series of unit tests including from the top of each of the routines ^%ut, ^%ut1, and ^%utcover, which each run unit tests when run from the top, as well as the regular unit tests. As noted earlier, the fourth argument of the COVERAGE^%ut API determines the verbosity of the results, and the ^%uttcovr analysis runs the verbosity 3, which the most detailed level. The results of the COVERAGE^%ut API are presented in the order of most detailed first so that the final output is the summary, and the user can scroll back for details.   As the analysis runs, the output lists each set of unit tests preceeded by a header indicating (e.g.,

       "------------------- RUNNING %utt1 ------------------- " or
       "------------------- RUNNING ^%utt1 -------------------"

where the routine name without the up arrow indicates it was started with the command D EN^%ut(, while those with an up arrow indicate it was started by running the routine from the top or from a specified tag.)

When the coverage is run with a verbosity value of zero or one, it presents only the least detailed item, the coverage by routine and total coverage for the analyzed routines (the coverage on a Cache system is shown)

       Routine %ut   (100.00%) 266 out of 266 lines covered
       Routine %ut1   (88.56%) 240 out of 271 lines covered
       Routine %utcover   (100.00%) 79 out of 79 lines covered

       Overall Analysis 585 out of 616 lines covered (94% coverage)

When the coverage is run with a verbosity value of two, it presents the coverage by tag within the routines as well. Before presenting the summary by routine and overall. The output for %ut1 is shown below for a Cache system (the output for %ut and %utcover both showed 100.00% for all tags).

       Routine %ut1   (88.56%) 240 out of 271 lines covered
       - Summary
       Tag %ut1^%ut1   (100.00%) 2 out of 2 lines covered
       Tag ACTLINES^%ut1   (100.00%) 8 out of 8 lines covered
       Tag CACHECOV^%ut1   (100.00%) 12 out of 12 lines covered
       Tag CHECKTAG^%ut1   (100.00%) 10 out of 10 lines covered
       Tag CHEKTEST^%ut1   (100.00%) 9 out of 9 lines covered
       Tag COV^%ut1   (52.31%) 34 out of 65 lines covered
       Tag COVCOV^%ut1   (100.00%) 9 out of 9 lines covered
       Tag COVRPT^%ut1   (100.00%) 5 out of 5 lines covered
       Tag COVRPTGL^%ut1   (100.00%) 14 out of 14 lines covered
       Tag COVRPTLS^%ut1   (100.00%) 31 out of 31 lines covered
       Tag FAIL^%ut1   (100.00%) 12 out of 12 lines covered
       Tag GETTAG^%ut1   (100.00%) 4 out of 4 lines covered
       Tag GETTREE^%ut1   (100.00%) 7 out of 7 lines covered
       Tag GETVALS^%ut1   (100.00%) 11 out of 11 lines covered
       Tag ISUTEST^%ut1   (100.00%) 1 out of 1 lines covered
       Tag LINEDATA^%ut1   (100.00%) 9 out of 9 lines covered
       Tag NEWSTYLE^%ut1   (100.00%) 4 out of 4 lines covered

Tag NVLDARG^%ut1   (100.00%)   11 out of 11 lines covered
Tag RESETIO^%ut1   (100.00%)   2 out of 2 lines covered
Tag RTNANAL^%ut1   (100.00%)   29 out of 29 lines covered
Tag SETIO^%ut1   (100.00%)   2 out of 2 lines covered
Tag TOTAGS^%ut1   (100.00%)   13 out of 13 lines covered
Tag UP^%ut1   (100.00%)   1 out of 1 lines covered

When the same analysis is run with a verbosity value of 3 (the value used when the ^%uttcovr routine is run from the top) the output for %ut1 (the tags following COVRPT^%ut1 all showed 100.00% coverage as before) also shows the lines that were NOT covered in the COV^%ut1 tag.

Routine %ut1   (88.56%)   240 out of 271 lines covered
 - Detailed Breakdown
Tag %ut1^%ut1   (100.00%)   2 out of 2 lines covered
Tag ACTLINES^%ut1   (100.00%)   8 out of 8 lines covered
Tag CACHECOV^%ut1   (100.00%)   12 out of 12 lines covered
Tag CHECKTAG^%ut1   (100.00%)   10 out of 10 lines covered
Tag CHEKTEST^%ut1   (100.00%)   9 out of 9 lines covered
Tag COV^%ut1   (52.31%)   34 out of 65 lines covered
  the following is a list of the lines **NOT** covered
   COV+17 . N %ZR ; GT.M specific
   COV+18 . D SILENT^%RSEL(NMSP,"SRC") ; GT.M specific. On Cache use $O(^$R(
RTN)).
   COV+19 . N RN S RN=""
   COV+20 . W !,"Loading routines to test coverage...",!
   COV+21 . F S RN=$O(%ZR(RN)) Q:RN="" W RN," " D
   COV+22 . . N L2 S L2=$T(+2^@RN)
   COV+24 . . S L2=$TR(L2,$C(9)," ") ; change tabs to spaces ; JLI 160316 in
serted to replace above
   COV+27 . . I $E($P(L2," ",2),1,2)'=";;" K %ZR(RN) W !,"Routine "_RN_" rem
oved from analysis, since it doesn't have the standard second line" ; JLI 160316
 inserted to replace above
   COV+29 . M RTNS=%ZR
   COV+30 . K %ZR
   COV+31 . Q
   COV+48 . K ^TMP("%utCOVCOHORTSAV",$J)
   COV+49 . M ^TMP("%utCOVCOHORTSAV",$J)=^TMP("%utCOVCOHORT",$J)
   COV+50 . K ^TMP("%utCOVRESULT",$J)
   COV+51 . S ^TMP("%utcovrunning",$J)=1,%utcovxx=1
   COV+53 . I (+$SY=47) VIEW "TRACE":1:$NA(^TMP("%utCOVRESULT",$J)) ; GT.M
START PROFILING
   COV+55 . I (+$SY=0) D ; CACHE CODE TO START PROFILING
   COV+56 . . S STATUS=##class(%Monitor.System.LineByLine).Start($lb(NMSP),$
lb("RtnLine"),$lb($j))
   COV+64 . . SET $ETRAP="Q:($ES&$Q) -9 Q:$ES W ""CTRL-C ENTERED"""
   COV+65 . . USE $PRINCIPAL:(CTRAP=$C(3))
   COV+66 . . Q
   COV+82 . . D TOTAGS(COVERSAV,0),TOTAGS(COVER,1)

```
    COV+83  . . D ##class(%Monitor.System.LineByLine).Stop()
    COV+84  . . Q
    COV+85  . D COVCOV($NA(^TMP("%utCOVCOHORT",$J)),$NA(^TMP("%utCOVRESULT",$J
))) ; Venn diagram matching between globals
    COV+87  . I VERBOSITY=-1 D
    COV+88  . . K ^TMP("%utCOVREPORT",$J)
    COV+89  . . D COVRPTGL($NA(^TMP("%utCOVCOHORTSAV",$J)),$NA(^TMP("%utCOVCOH
ORT",$J)),$NA(^TMP("%utCOVRESULT",$J)),$NA(^TMP("%utCOVREPORT",$J)))
    COV+90  . . Q
    COV+91  . E  D COVRPT($NA(^TMP("%utCOVCOHORTSAV",$J)),$NA(^TMP("%utCOVCOHO
RT",$J)),$NA(^TMP("%utCOVRESULT",$J)),VERBOSITY)
    COV+92  . Q
 Tag COVCOV^%ut1      (100.00%)  9 out of 9 lines covered
 Tag COVRPT^%ut1      (100.00%)  5 out of 5 lines covered
  ...
```

On a GT.M system the summary output shows

```
    Routine %ut    (100.00%)   256 out of 256 lines covered
    Routine %ut1     (88.43%)   237 out of 268 lines covered
    Routine %utcover    (100.00%)   79 out of 79 lines covered

    Overall Analysis 572 out of 603 lines covered (94% coverage)
```

The output from a GT.M system and shows the following output for %ut1 and lines that were also listed in the Cache system output as not being covered have been marked by adding a '+' at the beginning of the line.

```
    Routine %ut1       (84.87%)   230 out of 271 lines covered
     - Detailed Breakdown
    Tag %ut1^%ut1       (100.00%)  2 out of 2 lines covered
    Tag ACTLINES^%ut1      (100.00%)  8 out of 8 lines covered
    Tag CACHECOV^%ut1      (100.00%)  12 out of 12 lines covered
    Tag CHECKTAG^%ut1      (100.00%)  10 out of 10 lines covered
    Tag CHEKTEST^%ut1      (100.00%)  9 out of 9 lines covered
    Tag COV^%ut1        (53.85%)  35 out of 65 lines covered
     the following is a list of the lines **NOT** covered
     COV+34  . S NMSP1=NMSP I NMSP["*" S NMSP1=$P(NMSP,"*")
     COV+35  . I $D(^$R(NMSP1)) S RTNS(NMSP1)=""
     COV+36  . I NMSP["*" S RTN=NMSP1 F  S RTN=$O(^$R(RTN)) Q:RTN'[NMSP1  S RTN
S(RTN)=""
     COV+37  . Q
    +COV+48  . K ^TMP("%utCOVCOHORTSAV",$J)
    +COV+49  . M ^TMP("%utCOVCOHORTSAV",$J)=^TMP("%utCOVCOHORT",$J)
    +COV+50  . K ^TMP("%utCOVRESULT",$J)
    +COV+51  . S ^TMP("%utcovrunning",$J)=1,%utcovxx=1
    +COV+53  . I (+$SY=47) VIEW "TRACE":1:$NA(^TMP("%utCOVRESULT",$J))  ; GT.M
START PROFILING
```

```
   +COV+56   . . S STATUS=##class(%Monitor.System.LineByLine).Start($lb(NMSP),$
lb("RtnLine"),$lb($j))
   COV+57   . . I +STATUS'=1 D DecomposeStatus^%apiOBJ(STATUS,.ERR,"-d") F I=1
:1:ERR W ERR(I),!
   COV+58   . . I +STATUS'=1 K ERR S EXIT=1
   COV+59   . . Q
   COV+73   . I (+$SY=0) ; CACHE SPECIFIC
   COV+74   . K %utcovxx,^TMP("%utcovrunning",$J)
   COV+75   . Q
   COV+78   . I (+$SY=0) D  ; CACHE SPECIFIC CODE
   COV+79   . . S COVERSAV=$NA(^TMP("%utCOVCOHORTSAV",$J)) K @COVERSAV
   COV+80   . . S COVER=$NA(^TMP("%utCOVCOHORT",$J)) K @COVER
   COV+81   . . D CACHECOV(COVERSAV,COVER)
   +COV+82   . . D TOTAGS(COVERSAV,0),TOTAGS(COVER,1)
   +COV+83   . . D ##class(%Monitor.System.LineByLine).Stop()
   +COV+84   . . Q
   +COV+85   . D COVCOV($NA(^TMP("%utCOVCOHORT",$J)),$NA(^TMP("%utCOVRESULT",$J
))) ; Venn diagram matching between globals
   +COV+87   . I VERBOSITY=-1 D
   +COV+88   . . K ^TMP("%utCOVREPORT",$J)
   +COV+89   . . D COVRPTGL($NA(^TMP("%utCOVCOHORTSAV",$J)),$NA(^TMP("%utCOVCOH
ORT",$J)),$NA(^TMP("%utCOVRESULT",$J)),$NA(^TMP("%utCOVREPORT",$J)))
   +COV+90   . . Q
   +COV+91   . E  D COVRPT($NA(^TMP("%utCOVCOHORTSAV",$J)),$NA(^TMP("%utCOVCOHO
RT",$J)),$NA(^TMP("%utCOVRESULT",$J)),VERBOSITY)
   +COV+92   . Q
 Tag COVCOV^%ut1      (100.00%)   9 out of 9 lines covered
 Tag COVRPT^%ut1       (100.00%)   5 out of 5 lines covered
 Tag COVRPTGL^%ut1      (100.00%)   14 out of 14 lines covered
 Tag COVRPTLS^%ut1      (100.00%)   31 out of 31 lines covered
 Tag FAIL^%ut1        (100.00%)   12 out of 12 lines covered
 Tag GETTAG^%ut1       (100.00%)   4 out of 4 lines covered
 Tag GETTREE^%ut1       (100.00%)   7 out of 7 lines covered
 Tag GETVALS^%ut1        (0.00%)   0 out of 11 lines covered
  the following is a list of the lines **NOT** covered
   GETVALS+2   N LINE,MORE,ROUNAME,RSET,VAL,X
   GETVALS+4   S RSET=##class(%ResultSet).%New("%Monitor.System.LineByLine:Res
ult")
   GETVALS+5   S ROUNAME=##class(%Monitor.System.LineByLine).GetRoutineName(RO
UNUM)
   GETVALS+6   S LINE=RSET.Execute(ROUNAME)
   GETVALS+7   F LINE=1:1 S MORE=RSET.Next() Q:'MORE  D
   GETVALS+8   . S X=RSET.GetData(1)
   GETVALS+9   . S VAL=$LI(X,MTRICNUM)
   GETVALS+10   . S @GLOB@(ROUNAME,LINE,"C")=+VAL ; values are 0 if not seen,
otherwise positive number
   GETVALS+11   . Q
   GETVALS+12   D RSET.Close()
```

```
   GETVALS+13  Q
Tag ISUTEST^%ut1        (100.00%)  1 out of 1 lines covered
Tag LINEDATA^%ut1       (100.00%)  9 out of 9 lines covered
Tag NEWSTYLE^%ut1        (100.00%)  4 out of 4 lines covered
Tag NVLDARG^%ut1         (100.00%)  11 out of 11 lines covered
Tag RESETIO^%ut1        (100.00%)  2 out of 2 lines covered
Tag RTNANAL^%ut1         (100.00%)  29 out of 29 lines covered
Tag SETIO^%ut1         (100.00%)  2 out of 2 lines covered
Tag TOTAGS^%ut1         (100.00%)  13 out of 13 lines covered
Tag UP^%ut1           (100.00%)  1 out of 1 lines covered
```

Coverage on the two systems varies due to sections that are system specific. When the output for the two routines are compared there were 16 lines that are not executed in both Cache and GT.M systems so a combined coverage across both systems for the %ut1 routine is 94% and coverage across both systems for the routines %ut, %ut1, and %utcover is actually 97%.

On-going/Future plans for M-Unit functionality:

As a unique program in the realm of M[UMPS] code testing but following in the footsteps of other well established unit test frameworks, the M-Unit software will continue to move forward and improve (as the @TEST indicator was added based on changes in NUnit and Junit and coverage analysis for both GT.M and Intersystems). M-Unit will likely branch out and expand the types of checks that are available, matching the functions of other established test beds.

Summary

M-Unit provides a tool which can assist in writing and modifying routines in M projects with an aim to minimizing flaws in development and in the ongoing life of the software.