Using Korn shell scripts to perform function test of CMVC (a Unix client/server application)

Angel Rivera

TR 29.3407

CMVC Technical Support Team
IBM Software Solutions
Research Triangle Park, North Carolina, USA

ABSTRACT

This technical report provides recommendation on how to use Korn shell scripts to perform a function testing of the CMVC server and line commands in Unix. This type of information might be of interest to testers in general, even if they are not familiar with CMVC.

This document concentrates on a set of scripts that perform a limited function testing of CMVC. Specifically, each CMVC line command is tested at least once, based on the most common usage scenario. These test cases do not test the error messages, that is, they do not try to show that all error messages are eventually displayed; however, the scripts need to capture the situation when a line command fails to execute correctly. Also, the scripts rely in the return code of the executables; thus, this approach is not suitable for GUI applications.

In addition, some general recommendations are described to assist in the development of Korn shell scripts for function testing:

Provide a document that explains the prerequisites and main sequence for running the scripts
Provide a logical grouping of actions
Provide an execution sequence that follows a common usage scenario.
Provide comments and instructions in each shell script
Make an initial backup to create a baseline
Check for input parameters and environment variables
Try to provide “usage” feedback
Try to provide a “silent” running mode
Provide one function to terminate the script when there are errors
When possible, provide functions that do well a single task
Capture the output of each script, while watching the output being produced
Inside each script, capture the return code of each line command
Keep a count of the failed transactions
Highlight the error messages to easily find them in the output file
When possible, generate files “on the fly”
Provide feedback on the progress of the execution of the script
Provide a summary of the execution of the script
Try to provide an output file that is easy to interpret
When possible, provide cleanup scripts and a way to return to the baseline

ITIRC KEYWORDS

Function Testing
Unix
CMVC
Korn shell scripts

ABOUT THE AUTHOR

Angel Rivera

Mr. Rivera is an advisory software engineer with the VisualAge TeamConnection/CMVC development and technical support team. He joined IBM in 1989 and since then has worked in the development and support of library systems.

Mr. Rivera has an M.S. in Electrical Engineering from The University of Texas at Austin, and a B.S. in Electronic Systems Engineering from the Instituto Tecnologico y de Estudios Superiores de Monterrey, Mexico.

Mr. Rivera is currently the team lead for the development of CMVC 2.3 and provides service support for customers of VisualAge TeamConnection.

Introduction

This technical report provides recommendation on how to use Korn shell scripts to perform a function testing of CMVC, a Unix client/server application.

This document concentrates on a set of scripts that perform a limited function testing of CMVC. Specifically, each CMVC line command is tested at least once, based on the most common usage scenario. These test cases do not test the error messages, that is, they do not try to show that all error messages are eventually displayed; however, the scripts need to capture the situation when a line command fails to execute correctly.

In addition, some general recommendations are described to assist in the development of Korn shell scripts for function testing, such as logical division of the testing, how to check for the return code and count for errors, and how to invoke the shell script to capture the output. This type of information might be of interest to testers in general, even if they are not familiar with CMVC.

The audience for this document could be:
Testers of applications that run in Unix. Specifically the chapter about the general recommendations of Korn shell scripts.
Project leaders and testers that use CMVC.

The structure of this document is the following:

Chapter "General recommendations on Korn shell scripts for performing function testing" lists some recommendations to facilitate the creation of this type of Korn shell scripts.
Chapter "Purpose of the CMVC function test cases" explains the overall methodology for running the test cases.
Chapter "Procedure for configuring a CMVC family to be used for function testing' explains how to set the CMVC family server to properly accept the requests from the test cases.
Chapter "Running the function testing Korn shell scripts" explains how to run the scripts and how to interpret the results.
Chapter "Running a basic set of test cases for the GUI" explains the testing for the CMVC GUI.

Disclaimer

This technical report is not an official publication from the CMVC support group. The author is solely responsible for its contents.

Acknowledgments

We would like to acknowledge the contribution of the following co-workers:
Lee Perlov, WebSphere technical support.
Pam Hudadoff, who refined the initial scripts for populating a CMVC family for testing.

Recommendation to use “echo” to ease the porting from Korn to Bourne/Bash

If you have existing Korn shell scripts that you want to port to the Bourne or the Bash shells, you need to take into account that the Korn “print” command is not available in those other shells; you will need to use the “echo” command.

How to get the most up to date version of this technical report

The most up to date version of this technical report can be obtained from the following IBM CMVC ftp site:

ftp://ftp.software.ibm.com/ps/products/cmvc/doc/tr/README.index.txt
ftp://ftp.software.ibm.com/ps/products/cmvc/doc/tr/trfvtksx.pdf

General recommendations on Korn shell scripts for performing function testing

This chapter describes some recommendations that helped to create the Korn shell scripts. These recommendations can be applied to many client/server applications that run in Unix.

- Provide a document that explains the prerequisites and main sequence for running the scripts
- Provide a logical grouping of actions
- Provide an execution sequence that follows a common usage scenario.
- Provide comments and instructions in each shell script
- Make an initial backup to create a baseline
- Check for input parameters and environment variables
- Try to provide “usage” feedback
- Try to provide a “silent” running mode
- Provide one function to terminate the script when there are errors
- When possible, provide functions that do well a single task
- Capture the output of each script, while watching the output being produced
- Inside each script, capture the return code of each line command
- Keep a count of the failed transactions
- Highlight the error messages to easily find them in the output file
- When possible, generate files “on the fly”
- Provide feedback on the progress of the execution of the script
- Provide a summary of the execution of the script
- Try to provide an output file that is easy to interpret
- When possible, provide cleanup scripts and a way to return to the baseline

The rest of this chapter provides more detail on each recommendation.

Provide a document that explains the prerequisites and main sequence for running the scripts

It is important to document, preferably in a single file with a self-describing title, the main ideas behind the function testing, including the pre-requisites, the setup for the server and the client, the overall (or detailed) sequence of the scripts to follow, how to check for success/failures of the scripts, how to perform the cleanup, and to restart the testing.

Provide a logical grouping of actions

If you have a very small list of actions to be performed, then you could put them all in a single shell script.

However, if you have a large list of actions, then it is recommended that you group them into logical sets, such as the server actions in one file and the client actions in another. In that way, you will have finer granularity to perform the testing and the maintenance of the test cases.

Provide an execution sequence that follows a common usage scenario

Once you have decided on the grouping of the actions, then you need to think on performing the actions in a sequence that follows a common usage scenario. The idea is to try to perform what the end users will do more often. In a sense, try to find the approximately 20% of the usage cases that will test 80% of the function that is most commonly invoked.

In the case for the function testing for CMVC, the sequence of the Korn shell scripts is shown below. Notice that to help with the sequencing, the scripts have a suffix with the sequence number:

1. fvt-common-1: To populate the family with the basic CMVC objects.
2. fvt-server-2: To interact with the DBMS to age the defects and other server actions.
3. fvt-client-3: To perform some CMVC actions that can be done from the client.
4. fvt-cleanup: To cleanup the temporary files, in order to prepare for the next round of test cases. It also deletes the files in $HOME/queue and $HOME/queue/messages.

Provide comments and instructions in each shell script

It is good coding practice to provide pertinent comments and instructions in the header of each shell script, in that way, when another tester is assigned to run the scripts, the tester will get a good idea of the scope of the testing done in each script, as well as any pre-requisites and warnings.

A real example is shown below, from the script “fvt-common-1”.

```
#!/usr/bin/ksh
#
# Name: fvt-common-1
#
# Purpose:
#    Populates a CMVC Version 2.3.1 family with objects for FVT
#    purposes: defects, features, tracks, files, levels,
#    components, releases, etc.
#
# Notes:
#
# 1) This script should be run before the other FVT scripts.
#
# 2) This script can be run from either the CMVC family user id
#    or a CMVC client user id, because only client CMVC commands
#    are used.
#
# 3) To invoke this shell script and redirect standard output and
#    standard error to a file (such as fvt-common-1.out) do the
#    following (the -s flag is "silent mode" to avoid prompts to the
#    user):
#
#    ./fvt-common-1  -s  2>&1  | tee fvt-common-1.out
#
# 4) After the execution of this script the following files will be
#    checked out:
#       file: notes.ProjectA   release:  track-level-1.1
#       file: softtar.mk       release:  track-level-1.2
#
# 5) After the execution of this script the following level will be
#    uncommitted:
#       level:  04             release: track-level-1.2
#
# 6) After the execution of this script the following file will be
#    created that has a long file name with blanks and SCCS keywords:
#       file: File with SCCS keywords.txt   relase:  track-level-1.1
#
# Requisite auxiliary files in $HOME/fvt:
# *  sccskeys.binary (It is binary to avoid expansion during
#       the checkout to preserve the original SCCS keywords).
#
# Return codes:
#  0 = All commands were successful
#  1 = At least one command failed, see the output file and search
#      for the keyword "ERROR".
#
###########################################################################
```

Make an initial backup to create a baseline

You may need to perform the function testing several times. Probably the first time that you run it, you may find some errors in your scripts or in the procedures. Therefore, to avoid wasting too much time in recreating from scratch the server environment, specially if a database is involved, you may want to make a backup just before starting with the testing.

After you run the function test cases, then you could restore the server from the backup, and you would be ready for the next round of testing.

Check for input parameters and environment variables

It is a good idea to validate the input parameters and to check if the necessary environment variables are properly set. If there are problems, display the reason for the problem and how to fix it, and terminate the script.

The tester who is going to run this script would appreciate if the script terminates shortly after being invoked in case that a variable is not correct. It is not nice to wait for a long time in the execution of the script to find out that a variable was not properly set.

```
# -------------------------------------------
# Main routine for populating the CMVC family
# -------------------------------------------

CALLER=`basename $0`                    # The Caller name
SILENT="no"                             # User wants prompts
let "errorCounter = 0"

# ----------------------------------
# Handle keyword parameters (flags).
# ----------------------------------

set -- `getopt hs $* 2>/dev/null`       # -h = help & -s = silent

if [ $? != 0 ]
then
 print "Unknown flag(s)"
 usage
fi

while [ "$1" != -- ]                    # While $1 is not the termination char
 do
  case "$1" in
   -h) usage "HELP";            shift;; # Help requested
   -s) SILENT="yes";            shift;; # Prompt is not needed
  esac
 done

shift                                   # Shift pass --

# -----------------------------------------------
# The following CMVC environment variables must be set
# ---------------------------------------------

[ -z "$CMVC_FAMILY" ] && { print "The environment variable CMVC_FAMILY is not
set."; usage; }
[ -z "$CMVC_HOME" ] && { print "The environment variable CMVC_HOME is not
set."; usage; }
[ -z "$DEMO_CLIENT_LOGIN" ] && { print "The environment variable
DEMO_CLIENT_LOGIN is not set."; usage; }
[ -z "$DEMO_CLIENT_HOSTNAME" ] && { print "The environment variable
DEMO_CLIENT_HOSTNAME is not set."; usage; }
```

Notes:

The statement `CALLER=‘basename $0‘` is used to get the name of the script being running. In that way, you do not need to hard code the script name in the script. Thus, when you make a copy of the script, you will have less work to adapt the new derived script.

The statement `set -- ‘getopt hs $* 2>/dev/null‘` is used to get the input arguments when the script is invoked (such as the -h for help and -s for silent mode). Another way is to use the built-in command “getopts” (part of the shell) instead of the external command “getopt”.

The statements `[ -z "$X" ] && { print "The environment variable X is not set."; usage; }` are used to test if the string is null (-z) and if affirmative, then perform the print statement saying that it is not set and invoke the “usage” function discussed below. In case that your script does not use flags, then you could use the variable “$#” which provides the number of arguments that are being passed to the script.

Try to provide "usage" feedback

It is a good idea to provide a "usage" statement that explains how to use the script:

```
# -----------------------------
# Subroutine to print the usage
# -----------------------------

usage()
{
 print "USAGE: fvt-common-1 [-h] [-s]"
 print "WHERE: -h = help       -s = silent (no prompts)"
 print "PREREQUISITES:"
 print " - The CMVC family to be used should be only for FVT purposes."
 print " - "mkdb -d" should have been used to create this CMVC family "
 print " - The workstation login of the CMVC family must have"
 print "   client access to the family as well as superuser privilege."
 print " - The CMVC_FAMILY environment variable must be set. "
 print " - The CMVC_HOME environment variable must be set (/usr/lpp/cmvc )"
 print " - The DEMO_CLIENT_LOGIN environment variable must be set to the "
 print "   name of the client user login. "
 print " - The DEMO_CLIENT_HOSTNAME environment variable must be set to the "
 print "   name of the client user's hostname. "
 print " - The directory, fvt, must exist relative to the CMVC family "
 print "   account's home directory (such as /home/cmfvt/fvt)"
 print " - Run fvt-common-1 from the fvt directory."
 print "fvt-common-1: exiting now with rc=1."
 exit 1
}
```

This "usage" statement can be called when the script is invoked with the "-h" flag, such as:

```
./fvt-common-1  -h
```

Try to provide a “silent” running mode

It might be possible that you want a script to have 2 running modes:

A “verbose” mode (it could be the default) in which the user is prompted to enter a value or to simply press enter to continue.

A “silent” mode, in which the user is not prompted for data.

The following excerpt illustrates the handling of the invocation flag “-s” to run the script in silent mode:

```
# ---------------------------------------------------
# Everything seems to be OK, prompt for confirmation
# ---------------------------------------------------

if [ "$SILENT" = "yes" ]
then
 RESPONSE="y"
else
 print "This script will perform FVT client actions on the "
 print "CMVC family: $CMVC_FAMILY"
 print "Do you wish to proceed [y or n]? "
 read RESPONSE                          # Wait for response
 [ -z "$RESPONSE" ] && RESPONSE="n"
fi

case "$RESPONSE" in
 [yY]|[yY][eE]|[yY][eE][sS])
 ;;
 *)
  print "$CALLER terminated with rc=1."
  exit 1
 ;;
esac
```

Provide one function to terminate the script when there are errors

It is a good idea to provide a central function to terminate the execution of the script with critical errors are encountered. This function could provide additional instructions on what to do in such situations.

```
# -----------------------------------
# Subroutine to terminate abnormally
# -----------------------------------

terminate()
{
 print "The populate action for the CMVC family was not successful."
 print "Recreating the family may be necessary before running fvt-common-1
again,"
 print "that is, you must use 'rmdb', 'rmfamily', 'mkfamily' and 'mkdb -d'. "
 print "fvt-common-1 terminated, exiting now with rc=1."
 dateTest=`date`
 print "End of testing at: $dateTest"
 print ""
 exit 1
}
```

When possible, provide functions that do well a single task

When possible, provide functions that do well a single task. For example, instead of issuing a big list of long CMVC line commands, such as:

```
# --------------------------------------------------
print ""
print "Creating Access lists..."
# --------------------------------------------------

Access -create -component Development -login ted -authority general -verbose
 if [ $? -ne 0 ]
 then
  print "ERROR found in Access -create -component Development -login ted
-authority general"
  let "errorCounter = errorCounter + 1"
 fi

Access -create -component Development -login pat -authority general -verbose
 if [ $? -ne 0 ]
 then
  print "ERROR found in Access -create -component Development -login pat
-authority general"
  let "errorCounter = errorCounter + 1"
 fi

Access -create -component Development -login jim -authority general -verbose
 if [ $? -ne 0 ]
 then
  print "ERROR found in Access -create -component Development -login jim
-authority general"
  let "errorCounter = errorCounter + 1"
 fi
```

... you could create a function such as the following, which also handles the return code and if needed, increases the error counter.

```
CreateAccess()
{
 Access -create -component $1 -login $2 -authority $3 -verbose
 if [ $? -ne 0 ]
 then
  print "ERROR found in Access -create -component $1 -login $2 -authority $3"
  let "errorCounter = errorCounter + 1"
 fi
}
```

... and then invoke this function in an manner that is easy to read and to expand:

```
# --------------------------------------------------
print ""
print "Creating Access lists..."
# --------------------------------------------------

CreateAccess Development ted    general
CreateAccess Development pat    general
CreateAccess Development jim    general
```

Capture the output of each script, while watching the output being produced

If the script does not automatically send the output to a file, you could exploit some features of the Korn shell to capture the output of the execution of the script, such as:

```
./fvt-common-1  -s  2>&1  | tee fvt-common-1.out
```

Let’s analyze the above command:

*) Using “2>&1” to redirect the standard error to standard output

The string “2>&1” indicates that any errors should be send to the standard output. That is, the Unix file id of 2 is associated with standard error, and the file id of 1 is associated with standard output. If you do not use this string, then you will be capturing only the good messages, and the error messages will not be captured.

*) Using the pipe “|” and the “tee” command

There is a good analogy between the Unix processes and plain plumbing concepts. In this case, we want to make a pipeline in which the input to the pipeline is the output of the desired script. The next thing to decide, is what to do with the output of the pipeline. In this case, we want to capture it in an output file, named “fvt-common-1.out” in our example.

However, besides capturing the output, we also want to watch the output being produced while the script is running. That is the reason that we have to attach a “tee” (T-shape pipe) that will allow 2 things at the same time: placing the output into a file AND displaying the output into the screen. The plumbing analogy would be:

```
 process --> T ---> output file
             |
             V
           screen
```

If you just ONLY want to capture the output and you do not want to see the output being displayed in the screen, then you could just omit the extra plumbing:

```
./fvt-common-1  -s  2>&1  > fvt-common-1.out
```

The plumbing analogy in this case would be:

```
 process --> output file
```

Inside each script, capture the return code of each line command

One way to determine the success or failure of the function testing is by counting the line commands that have failed, that is, that have a return code different than 0. The variable “$?” provides the return code of the command recently invoked; in the example below, it provides the return code of the execution of the CMVC Release command.

```
Release -create $1 -component $2 -owner $3 -description "Release $1" \
        -process $4 -approver $5 -environment $6 -tester $7  -verbose
if [ $? -ne 0 ]
then
 print "ERROR found in Release -create $1 (tracking, approval, level, test)"
 let "errorCounter = errorCounter + 1"
fi
```

Keep a count of the failed transactions

One way to determine the success or failure of the function testing is by counting the line commands that have a return different than 0. However, in my personal experience, I was used to handle only strings in my Korn shell scripts and not integers. The manuals that I consulted were not too clear on how to use integers, and that is the reason I want to expand a little bit here, on how to use integers and additions to count the number of errors (failures of line commands):

First, you need to initialize the counter variable as follows:

```
let "errorCounter = 0"
```

Then, issue the line command and capture the return code using the $? Variable. If the return code is different than 0, then increment the counter by one (see the statement in bold blue)

```
Release -create $1 -component $2 -owner $3 -description "Release $1" \
        -process $4 -approver $5 -environment $6 -tester $7  -verbose
if [ $? -ne 0 ]
then
 print "ERROR found in Release -create $1 (tracking, approval, level, test)"
 let "errorCounter = errorCounter + 1"
fi
```

By the way, the integer variables can be displayed as other variables; that is, by using “print” or “echo”.

Highlight the error messages to easily find them in the output file

When an error (or failed transaction) is encountered, besides increasing the error counter, it is a good idea to print something that indicates that there was an error. Ideally, the string of characters should have a substring such as ERROR or something like that, that will allow the tester to quickly find the error in the output file. Potentially this output file could be large, and it is important to quickly locate the errors.

In the sample, code, the statement in bold blue shows

```
if [ $? -ne 0 ]
 then
  print "ERROR found in Release -create $1 (tracking, approval, level, test)"
  let "errorCounter = errorCounter + 1"
 fi
```

When possible, generate files “on the fly”

In some cases, it is necessary to handle files that will be used by the application. You could use existing files or you could add statements in the script to create them. If the files to be used are long, then it is better to have them as separate entities. If the files are small and the contents if simple or it is not relevant (that is, the important thing is to have a text file, regardless of its contents), then you could decide to create these temporary files “on the fly”.

The following lines of code show an example of how a temporary file is created “on the fly”:

```
cd $HOME/fvt

print ""
print "Creating file softtar.c"

echo "Subject: This is softtar.c" >  softtar.c
echo "This is line 2 of the file" >> softtar.c
```

Notes:

The first echo statement uses the single > to force the creation of a new file.
The second echo statement uses the double >> to append data to the bottom of an existing file. By the way, in case that the file does not exist, then the file will be created.

Provide feedback on the progress of the execution of the script

It is a good idea to include print statements in the script to indicate the logical progress of the execution of the script. You could add something that will quickly identify the purpose of the output.

If the script is going to take more than few seconds to execute, you could try to print the date at the beginning and at the end of the execution of the script. In that way, the output will show these times and you could compute the elapsed time.

In the sample script, some print statements that provide the indication of the progress are shown below.

```
# --------------------------------------------------
print "Subject: CMVC 2.3.1, FVT testing, Common, Part 1"
dateTest=`date`
print "Begin testing at: $dateTest"
print ""
print "Database: $DATABASE"
print "Family:   $CMVC_FAMILY"
print "Testcase: fvt-common-1"
print ""
# --------------------------------------------------

# --------------------------------------------------
print ""
print "Creating Users..."
# --------------------------------------------------
CreateUser pat  $THISUSERID@`hostname` Pat  Anderson  Management

...


# --------------------------------------------------
print ""
print "Creating Access lists..."
# --------------------------------------------------

CreateAccess Development ted    projectlead
```

Provide a summary of the execution of the script

If you are counting the errors or failed transactions, it is recommended to indicate whether or not there were errors. The idea is that the tester could see the bottom of the output file and quickly tell if there errors or not.

In the sample script, the following code statements provide such summary of the execution.

```
# --------------
# Exit
# --------------
if [ $errorCounter -ne 0 ]
then
 print ""
 print "$errorCounter ERRORS found during the execution of this test case."
 terminate
else
 print ""
 print "Yeah! No errors were found during the execution of this test case."
fi

print ""
print "fvt-common-1 complete."
print ""
dateTest=`date`
print "End of testing at: $dateTest"
print ""

exit 0
```

```
# end of file
```

Try to provide an output file that is easy to interpret

It is very helpful to provide some key information in the actual output that is generated by the script. In that way, the tester could easily determine if the file that is being looked at, is relevant and current (versus obsolete or irrelevant). The addition of the date-time stamp is important to give a sense of currency. Also, the summary report helps to determine if there were errors or not; if there were errors, then the tester will have to search for the specified keyword, such as ERROR and identify the individual transactions that failed.

A truncated sample output file is shown below:

```
Subject: CMVC 2.3.1, FVT testing, Common, Part 1
Begin testing at: Tue Apr 18 12:50:55 EDT 2000

Database: DB2
Family:   cmpc3db2
Testcase: fvt-common-1


Creating Users...
User pat was created successfully.
...

Yeah! No errors were found during the execution of this test case. Yeah!

fvt-common-1 complete.

End of testing at: Tue Apr 18 12:56:33 EDT 2000
```

An example of the bottom of the output file when errors are encountered is shown below:

```
ERROR found in Report -view DefectView

*** 1 ERRORS found during the execution of this test case. ***
The populate action for the CMVC family was not successful.
Recreating the family may be necessary before running fvt-client-3 again,
that is, you must use 'rmdb', 'rmfamily', 'mkfamily' and 'mkdb -d',
then issue: fvt-common-1 and optionally, fvt-server-2.
fvt-client-3 terminated, exiting now with rc=1.
End of testing at: Wed Jan 24 17:06:06 EST 2001
```

When possible, provide cleanup scripts and a way to return to the baseline

It might be possible that the test scripts generate temporary files; in that case, it is a good practice to have a script that will delete those temporary files. This will avoid mistakes in which the tester may not delete all the temporary files, or worse, delete some needed files that were not temporary.

It also deletes the files in $HOME/queue and $HOME/queue/messages.

In this document, the script fvt-cleanup is used for these purposes.

Purpose of the CMVC function test cases

The objective of function test cases for CMVC is to have a "sanity" test of the basic function of the product (that is, it is not a comprehensive system test). The idea is to run these function test cases whenever there is a need to test the basic functionality of CMVC after a new version (or significant patches) for the operating system (or the DBMS) have been applied.

You will have to manually create a CMVC family and then execute the appropriate Korn shell scripts, in the sequence shown below:

* fvt-common-1:
To populate the family with users, components, releases, defects, files and other CMVC objects. This script can be executed from either the server or the client. This is the main function test cases for the server and the client (line commands).

* fvt-server-2:
To interact with the DBMS to age the defects. This script can be executed ONLY from the server.

* fvt-client-3:
To perform additional CMVC actions that can be done from the client.

* fvt-cleanup:
To cleanup the $HOME/fvt directory of the files that were extracted during the execution of the other scripts. It also deletes the files in $HOME/queue and $HOME/queue/messages.

Limitations

The following limitations apply to the CMVC family to be used for function test:

The defects and features created by the shell scripts will begin with the defect name of “1001”; thus, in case that you use for testing, an existing CMVC family that has already some objects, and if you have more than 1000 defects/features in your family, the shell scripts will fail to create the desired objects.

The idea behind starting at this high defect number is to allow the tester to open defects into a test family (the numbers will start with the defect name of “1”) and then, if so desired, to populate the family with more defects (which will start with the defect name of “1001”).

Suggestions

It is suggested to use the family name of "cmvcfvt". However, this is not necessary. In this document we will use "cmvcfvt" to illustrate the commands.

```
Family name:      cmvcfvt
First superuser:  cmvcfvt
First host login: cmvcfvt
First host:       the same host where the cmvcfvt family is created
```

You can specify an additional user and host for that user, in that way you can run the CMVC function test scripts from either the cmvcfvt account or the account of the additional user.

Objects created in the FVT family

The shell scripts create entries for the following CMVC tables, in the order in which they are accessed:

Users
Host Lists
Components
Access Lists
Notification List
Defects
Features
Verification Records
Releases
Approver Lists
Environment Lists
Files
Levels
Level Members
Tracks
Approval Records
Fix Records
Test Records

Cleaning the files created in the $HOME/fvt directory

The scripts create and extract files in the directory $HOME/fvt. To delete these files, run the script:

```
    fvt-cleanup
```

Procedure for configuring a CMVC family to be used for function testing

This chapter describes the configuration for the CMVC family in order to accept the requests from the shell scripts.

Prerequisites

The procedure in this section assumes that the following prerequisites are installed and configured:

* CMVC Server

For the installation instructions for the Server, refer to the CMVC Server manual:
IBM Configuration Management Version Control, Server Administration and Installation, 2.3

* CMVC Client

For the installation instructions for the Client, refer to the Client manual:
IBM Configuration Management Version Control, UNIX Client Installation and Configuration, 2.3

Configuring the CMVC family for function testing

Configuring the CMVC family for function testing involves creating a new family on the CMVC server. Follow the instructions in Chapter 6 of the Server manual for configuring a new family on the CMVC server, with the following additional requirements:

*) This example uses the family name 'cmvcfvt'.

*) Specify the following CMVC environment variables in the .profile

```
export CMVC_HOME=/usr/lpp/cmvc
export CMVC_FAMILY=$LOGNAME
export CMVC_SUPERUSER=$LOGNAME
export CMVC_BECOME=$LOGNAME
export CLIENT_LOGIN=$LOGNAME
# For AIX use:
export CLIENT_HOSTNAME=`hostname -s`
# For HP-UX use:
export CLIENT_HOSTNAME=`hostname`
# For Solaris use:
export CLIENT_HOSTNAME=`/usr/ucb/hostname`
```

*) Specify the following additional environment variables in the .profile and which will be used by the shell scripts to create an additional user and host from where that user can access CMVC:

```
export DATABASE=DB2    # (or INFORMIX or SYBASE or ORACLE)
export DEMO_CLIENT_LOGIN=$LOGNAME         # (set as appropriate)
export DEMO_CLIENT_HOSTNAME=‘hostname -s‘ # (set as appropriate)
```

*) Logoff and login again in order to activate the changes done to the .profile.

*) Run the following to create the directories and files for the CMVC family:

```
mkfamily
```

*) Run mkdb with the -d option to load the required default configurable fields for CMVC defects and features:

```
mkdb -d
```

*) Create a directory where to store a backup of the recently created database:

```
mkdir $HOME/backup
```

*) Make a backup of the database.

*) Start the cmvcd server daemons for the CMVC family (if they are not already started) by following the procedure in Chapter 13 of the Server manual. For example, issue the following command that will start 2 daemons:

```
cmvcd $LOGNAME 2
```

*) Verify that the CMVC daemons are running:

```
ps -ef | grep cmvcd
```

The result should be something like this:

```
cmvcfvt 12952 20119   0 12:02:41      -  0:00 cmvcd cmvcfvt 2
cmvcfvt 20119     1   0 12:02:41      -  0:00 cmvcd cmvcfvt 2
cmvcfvt 24721     1   0 12:02:39      -  0:00 cmvcd cmvcfvt 2
cmvcfvt 26258 24721   0 12:02:39      -  0:00 cmvcd cmvcfvt 2
cmvcfvt 30625 16001  13 12:04:57  pts/4  0:00 ps -ef
cmvcfvt 16001  9600   0 11:59:11  pts/4  0:00 -ksh
cmvcfvt 24994 16001   4 12:04:57  pts/4  0:00 grep cmvcfvt
```

Note: If you invoke 2 cmvc daemons, then 2 pairs of daemons are created: 2 parents with their 2 children. Therefore, the number of entries given by the ps command is the double of the number used when the cmvcd command was invoked. In this example, cmvcd was invoked with 2, thus, ps shows 4 entries for cmvcd.

*) Start also the notification daemon:

```
notifyd
```

Configuring the CMVC client from another host (optional)

You can use the CMVC client installed in the same host where the CMVC server is installed.

However, in case that you want to test the client from another host, you will need to configure the client with the proper data.

*) This example uses the client name ’cmvcclt’.

*) Specify the following CMVC environment variables in the .profile

```
export CMVC_HOME=/usr/lpp/cmvc
export CMVC_FAMILY=familyName@hostName@portNumber
export CMVC_BECOME=$LOGNAME
```

Note: You need to specify the proper values for CMVC_FAMILY.

*) Specify the following additional environment variables in the .profile and which will be used by the shell scripts to create an additional user and host from where that user can access CMVC:

```
export DEMO_CLIENT_LOGIN=$LOGNAME          # (set as appropriate)
export DEMO_CLIENT_HOSTNAME=‘hostname -s‘ # (set as appropriate)
```

*) Logoff and login again in order to activate the changes done to the .profile.

*) Ensure that you can issue the following CMVC commands:

```
Report -testServer
Component -view root
```

If there are problems with the above commands, such as error messages saying that there is no host list for the client, then fix the problem by adding the appropriate host list entry, when logged in as the CMVC family administrator user id.

Copying the Korn shell scripts into the proper directory

The Korn shell scripts and the associated files need to be copied into the directory structure of the CMVC user id who is going to conduct the function testing. Although the user id used in this section is the CMVC family administrator user id, the same principle applies to other userids, such as cmvcclt (see previous discussion on using a CMVC client from another host).

*) Login to the cmvcfvt account. You should be in the home directory of the family's account on the CMVC Server. Let's assume that is /home/cmvcfvt

*) Create a directory for the test cases:

```
mkdir fvt
```

*) Copy the Korn shell scripts and the associated files. Obtain the zip file mentioned in the chapter "How to get the files mentioned in this document" and place it under $HOME. Then unzip it as follows:

```
unzip trfvtksx.zip
```

*) Change the proper file permissions, in order to execute the files:

```
chmod u+x fvt*
```

Running the function testing Korn shell scripts

This chapters describes how to run the Korn shell scripts for the function testing of CMVC. It is assumed that the steps mentioned in the previous chapter were executed and that the family is ready to be used:

*) Login into the CMVC family user id.

*) Change the directory where the scripts were copied:

```
cd $HOME/fvt
```

*) From $HOME/fvt run the script that will populate the test CMVC family. This script only issues CMVC commands, and thus, it could be run from the CMVC family user id or another CMVC client user id:

```
./fvt-common-1 -s  2>&1  | tee fvt-common-1.out
```

*) Look at the bottom of the output file “fvt-common-1.out” and see the conclusion of the summary report.

*) From $HOME/fvt of the CMVC family user id, run the script to perform server related tasks, such as aging the defects. This script requires direct access to the DBMS and thus it can only be issued from the CMVC family server user id:

```
./fvt-server-2 -s  2>&1  | tee fvt-server-2.out
```

*) Look at the bottom of the output file “fvt-server-2.out” and see the conclusion of the summary report.

*) Run the script that will perform several CMVC actions that are client related, such as checkout a file (and keep a lock), release extract, etc.

```
./fvt-client-3 -s  2>&1  | tee fvt-client-3.out
```

*) Look at the bottom of the output file “fvt-client-3.out” and see the conclusion of the summary report.

*) If you are ready to start the test cycle again, then you need to perform the following actions:

+ Cleanup the $HOME/fvt and $HOME/queue directories by executing:

```
./fvt-cleanup
```

+ Stop the family and recreate it:

```
cd  $HOME
stopCMVC
rmdb -s
rmfamily -s
mkfamily -s
mkdb -d -s
cmvcd $LOGNAME 2
```

Running a basic set of test cases for the GUI

Once the CMVC family has been populated with objects, you can manually perform a basic function test cases for the GUI. The GUI cannot be tested by using Korn shell scripts, and you either have to conduct the test manually or use a tool that can interact with the GUI (which was not used for CMVC).

*) Login to the account on a CMVC user id.

*) Ensure that you are in the home directory:

```
cd $HOME
```

*) Set the DISPLAY environment variable, if required (replace "hostname"):

```
export DISPLAY=hostname:0
```

*) Then invoke the graphical user interface by issuing the following command:

```
cmvc
```

*) You can explore the CMVC objects that were populated by the test scripts. You can start by clicking the desired task from the "IBM CMVC - Tasks" window (the "main" window).

*) Do the following actions, which will test the minimum functionality of the GUI.

- Report -testServer (if it is in the task list)

- Look at all the defects (Windows -> Defects -> Defects...)
  - File -> Open List, then expand the Command Box and specify the query:
    name like '%' order by addDate

- Create a new defect. From the defects window:
  - Actions -> Open, then use component "root" and fill out the rest.

- Look at component tree (Windows -> Components -> Components Tree...)
  - expand the complete component tree: select "root" node and then
    from the right mouse button: Expand Downward Fully

- Look at the help (Help -> On Process...)

- Look at the version number (Help -> On Version...)

How to get the files mentioned in this document

All the tools used here are available via the public Internet. The tools might be updated in the future. The tools are zipped into a single file called **trfvtksx.zip** and it can be downloaded as follows:

FTP site for CMVC

You can download the code from our external FTP site for CMVC, by doing:
1. ftp ftp.software.ibm.com
2. login as **anonymous** and for password give your email address.
3. cd   ps/products/cmvc/doc/tr
4. binary
5. get trfvtksx.zip
6. quit

Obtaining Info-ZIP

The VisualAge TeamConnection and CMVC team uses the Info-Zip **zip** and **unzip** tools to package compressed files (in which the files to be packaged are compressed first).

The main advantages of Info-ZIP are:
- Compatibility: these tools are compatible with other ZIP programs.
- Portability: they are available in ALL the platforms that are supported by VisualAge TeamConnection and CMVC.
- Cross-platform: A zip file prepared in Unix can be unzipped in the correct format in Windows NT and vice versa.

Info-ZIP's software is free and can be obtained for the desired platforms from various anonymous ftp sites, including the URL:

ftp://ftp.uu.net:/pub/archiving/zip/

Because of the general value of these tools, it is recommended that you add the unzip and zip tools in a directory in the PATH that is accessible to all the users for the machine.

How to unzip files

- To only view the contents of the zip file (without actually unpackaging and uncompressing the files) do: **unzip -l trfvtksx.zip**
- To unpackage and uncompress the zip file do: **unzip trfvtksx.zip**

Copyrights, Trademarks and Service marks

***** END OF DOCUMENT *****