Chur Language Reference Manual

Revision History

Version	Date	Name	Description of Changes
1	11/09/10	Nitish Rawat	First Edition
2	08/30/11	Sankha Sil	Revision
3	09/16/11	Sankha Sil	Revision
4	11/04/11	Sankha Sil	Revision
5	02/13/12	Sankha Sil	Revision
6	02/15/12	Debabrata Das	Second Edition
7	03/23/12	Dipu Deshmukh	Review
8	23/04/12	Sankha Sil	Revision (Named Argument description updated, test results reporting updated)
9	16/05/12	Sankha Sil	Revision (Arranging and adding new features of ZUG)
10	19/06/12	Sankha Sil	Review
11	03/07/12	Dipayan Sengupta	Revision (Added examples for NameSpaces)
12	14/08/12	Sankha Sil	Built-in Atoms added
13	25/10/12	Dipayan Sengupta	Added a new Chapter for MVCV
14	13/11/12	Sankha Sil	Revision(Correcting Indexed MVM examples)
15	21/11/12	Sankha Sil	Added new feature: Built In Atom enhanced.
16	21/01/13	Md Sarfaraz Khan	Added a new topic,command line options

Table of Contents

1. Introduction....6
1.1 Document Purpose....6
1.2 Intended Audience....6
2. Anatomy of an Automation Test Suite....7
2.3 TestCases and Molecules Worksheets....9
2.4 Prototypes....13
3. Writing a simple Test Case....14
3.3 Flow of Execution....14
3.4 Command line execution....15
4. Built-in Atoms....16
4.1 Context Variable Atoms....16
4.2 XML Parsing Atoms....18
4.3 Web Automation Atoms....18
4.4 Print Atom....18
4.5 GetValueAtIndex Atom....18
4.6 GetCurrentIndex Atom....18
4.7 GetValueAt Atom....19
5. Exception Handling....20
5.1 Test case or Molecule exception handling....20
5.2 Test suite exception handling....21
6. Context variables....22
6.1 Definition....22
6.2 Declaring a Context variable....22
6.3 Pass arguments by value or by reference....22
6.4 Reading and Altering Context variable value from atoms (ZUG API)....22
6.5 Zuoz pearls....23

6.6 Appending values to a Context variable.....23

6.7 Destroying a Context variable.....23

6.8 ZUG test environment variables.....24

7. Molecules.....26

7.1 Definition.....26

7.2 Designing a Molecule.....26

7.3 Positional and named arguments.....27

8. Multi-valued Macros.....30

8.1 Definition.....30

8.2 Cartesian MVM expansion.....30

8.3 Indexed MVM expansion.....30

8.4 Nested MVMs.....31

8.5 MVM used in Test-cases.....31

8.6 MVM used in Molecules.....32

8.7 Scalar and Vector MVMs.....32

8.8 Indexing Macro values in a molecule without Indexed MVM.....33

9. Multi Valued Context Variable.....34

9.1 Definition.....34

9.2 Cartesian Expansion.....34

9.3 MVCV used in Test-cases.....35

9.4 MVCV used in Molecules.....35

9.5 Scalar and Vector MVCVs.....36

10. Concurrent Execution.....37

10.1 Test case or Molecule concurrency.....37

10.2 Test step concurrency.....37

10.3 Thread safe context variables.....37

10.4 Passing thread safe context variables as arguments.....37

11. Negative Testing.....38

11.1 Definition.....38

11.2 How To Use.....38

12.NameSpaces.....40

12.1 Calling a Molecule from a single external worksheet.....40

12.2 Calling external macros from an external worksheet.....41

12.3 Calling two different macros from different external worksheets.....41

12.4 Multi Level Include Files.....42

13. Command Line Options.....43

13.1 -verbose.....43

13.2 -include.....43

13.3 -macrocolumn.....43

14. ZUG Logs.....44

14.1 Result Log.....44

14.2 Debug Log.....44

14.3 Error Log.....44

14.4 Primitive Log.....44

15. Reporting test results 45

16. Glossary.....46

1. Introduction

CHUR is Automature's formal test specification language. CHUR needs ZUG to execute tests, so please refer to the ZUG manual to find out how to run tests written in CHUR.

CHUR is platform independent. It can execute tests wherever ZUG is supported.

CHUR uses a spreadsheet paradigm to specify test steps. The choice is intentional. Most test engineers aleady use spreadsheets to document testcases. CHUR is simply a more formal way to describe such tests, so that they can be executed automatically.

CHUR is simple in structure, but extremely powerful as a test specification language. With semantic support for concurrent execution, threadsafe context variables, macros, include files, testcase generation, explicit differentiation between test steps and verification steps, and other features, it is not just yet another general purpose programming language. Instead, it is specifically designed to make the design and maintainability of test cases easier.

1.1 Document Purpose

This Programmer’s Guide guides the user through the steps to create a testsuite in CHUR, containing one or more test-cases. This document describes the various features of the language, and provides examples of how they are intended to be used.

1.2 Intended Audience

This user manual is intended for users who want to learn designing test cases using CHUR. It may also be used by quality analysts, test automation engineers, QA managers, QA architects to understand the basic concepts behind Automature's unique solution to Automation.

Though no formal programming background is essential to understand CHUR, some prior basic knowledge of programming concepts, spreadsheets and software testing, would be helpful.

2. Anatomy of an Automation Test Suite

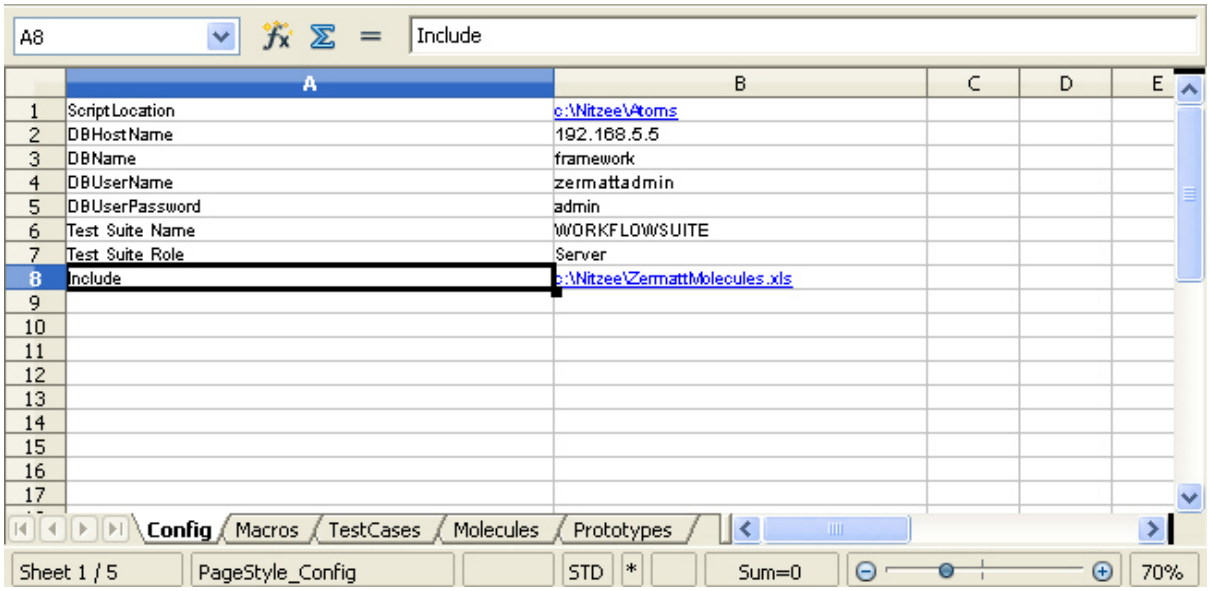
Automation Test Suite is represented in spreadsheet and it consists of following worksheets:

Config, Macros, TestCases, Molecules and Prototypes

2.1 Config Worksheet

This section is where environmental information is described. This information is typically used by ZUG to locate supporting programs, and upload results into the Automature database.

It also contains the configuration for mapping where the Atoms are installed and also if any other spreadsheets need to be included or not.



Following is a brief description of the Config Sheet properties –

Property Name	Property Significance	Values	Optional?
ScriptLocation	**Here the location of the atoms are specified. ZUG will pick the atoms to execute from the location specified. Multiple location may be provided using ';' after each qualified path. The locations can be a fully qualified path name, or a path relative to the current directory.**	e.g. E:\ZUG\Scripts\ is a valid fully qualified path for ScriptLocation whereas "Scripts\Atoms" is a relative path and it will find the folder either from the current location or the location of the testsuite as per the ZUG invocation. For Multipile Atom Locations e.g C:\Zuoz\Atoms\Java;C:\Zuoz\Atoms\Ruby	If no external atoms are userd then this field can be kept blank.
DBHostName	Host name of the Database server	HostMachine :port<localhost:4567>	No
DBName	Database Name	Framework (not mandatory for davos)	No
DBUserName	Name of the User to do Authentication to DB	e.g. davosuser	No
DBUserPassword	Password of the user required for SQL Authentication	e.g. user	No
Test Suite Name	Name of the Test Suite	e.g. ZERMATT Automation	No
Test Suite Role	Role of the node where Test Suite will be executed	**e.g. client , server , etc...).**	No
ProductLogLocations	List of folders (Fully qualified paths) of logs for products under test like ZUG Client, Zermatt server	This should be a comma separated list of paths	Optional
INCLUDE	Allows a reference to another spreadsheet file, which contains Molecules and macros to be included. ZUG allows more than one such declaration and nested include files.	e.g. C:\FileOps.xls e.g for multipile includes C:\MoleculeFile.xls,C:\MacroFile.xls	Optional

2.2 Macros Worksheet

This section allows the test case designer to declare short names called Macros that can then be substituted in the test case specification during execution. This information is integral to the logic of the decomposed test steps, and does not relate to the execution environment.

There are three columns on the Macros worksheet:

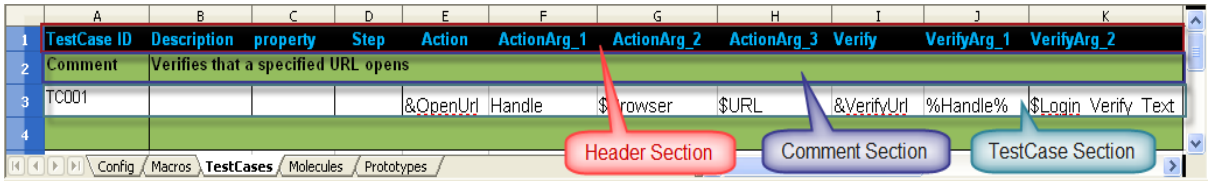
Column Name	Significance	Values	Optional?
Macro Name	**The first column is for the *Macro Name.***	**A Macro name must start with a single dollar ($) . A Macro name can contain letters, numbers or underscores (_).**	No
Value	The second column contains the value assigned to the Macro.	Any string, e.g. HostMachine :port<default 3306>	No
Comment	The third column is for comments.	Any related information to the Macro, something that describes the Macro or its purpose or any comment can be written here. This column is ignored by the engine and therefore, isn't mandatory.	Yes

A sample Macro worksheet:

	A	B	C
1	**Macro Name**	**Value**	**Comment**
2	$URL	http://google.com	**Giving the URL input which the browser will open.**
3	$Browser	firefox	**Specifying which browser to open.**
4	$Login_Verify_Text	Google	**Specifying the text which should be verified in the webpage.**

Config | **Macros** | TestCases | Molecules | Prototypes

2.3 TestCases and Molecules Worksheets



	A	B	C	D	E	F	G	H	I	J	K
1	TestCase ID	Description	property	Step	Action	ActionArg_1	ActionArg_2	ActionArg_3	Verify	VerifyArg_1	VerifyArg_2
2	Comment	Verifies that a specified URL opens									
3	TC001				&OpenUrl	Handle	$browser	$URL	&VerifyUrl	%Handle%	$Login_Verify_Text
4											

Config | Macros | **TestCases** | Molecules | Prototypes

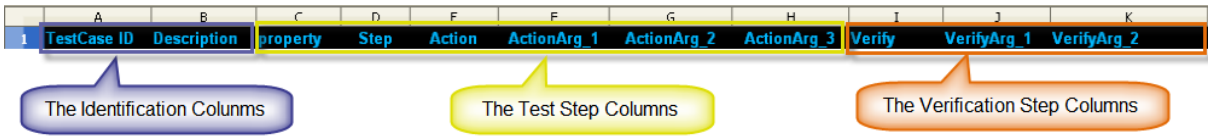
Header Section | Comment Section | TestCase Section

The Test Case and the Molecules worksheets are divided into two logical sections, viz.

1. Header Section - The first set of rows in each sheet.

   The following is the header section for a test case sheet:

2. One or more of the following two sections
   1. The comment section - this section will appear as a preamble to a testcase, and is a verbose description of the testcase
   2. The testcase section, which lays out the details about steps, actions and verifications about an individual testcase. This section is divided into three logical column groups, viz.
      1. The ***identification*** group of columns
      2. The ***test step*** group of columns, and
      3. The ***result verification*** group of columns



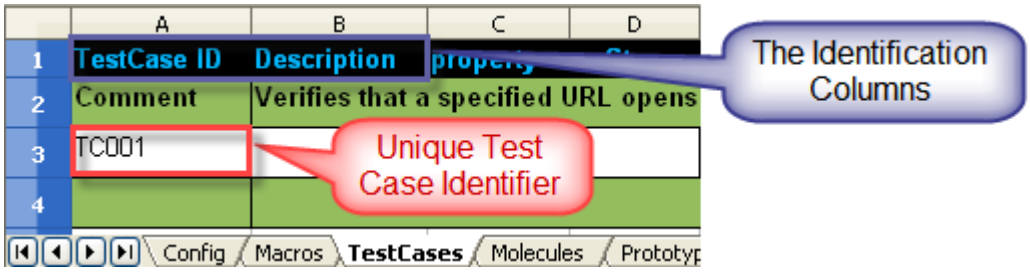
	A	B	C	D	E	F	G	H	I	J	K
1	TestCase ID	Description	property	Step	Action	ActionArg_1	ActionArg_2	ActionArg_3	Verify	VerifyArg_1	VerifyArg_2

The Identification Colunms | The Test Step Columns | The Verification Step Columns

The identification group of columns

These columns identify the test case.

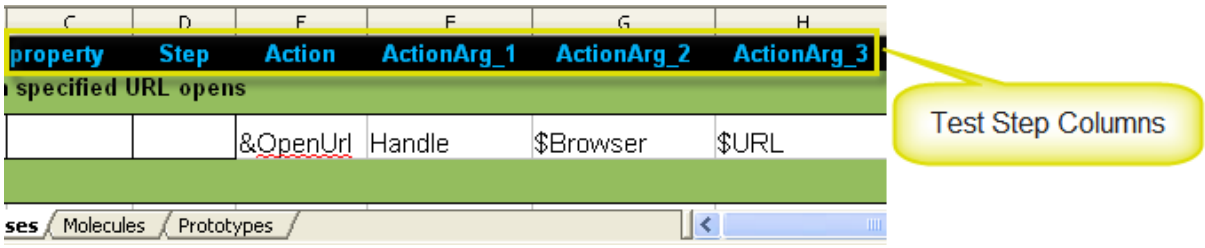
1. The first column contains the testcase or the Molecule identifier. This must uniquely identify each test case. This identification is used to locate the row in the test framework database, where results will be stored.
2. The second column should contain a summary description of the testcase.



The test step group of columns

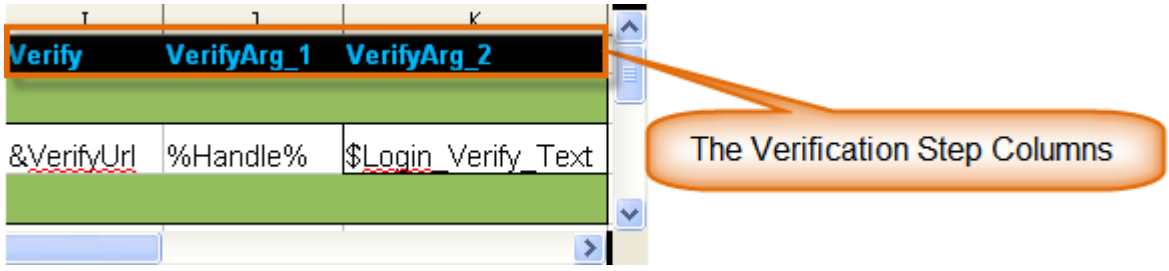
These columns describe the steps involved in executing the test. A test is expected to consist of one or more actions. Actions may be performed in sequence or in parallel. The step identifier allows the designer to designate several steps to be executed in parallel. Each action step allows the specification of input arguments.

The sequence identifier also allows the designer to declare certain steps to be initialization or cleanup steps. By appropriately pairing the sequence identifiers for these steps, the designer can tell ZUG which cleanup steps need to be executed, in case the test fails at any given point.



The result verification group of columns

These columns describe how to verify the outcome of a specific action. Each method can be supplied arguments. The verification method always returns a status. If the status returned is a zero, then the verification is deemed to have succeeded. If all actions for a given test case are successful, then the testcase has passed.



The following table describes the meaning of indiviudal columns in the testcases & Molecules worksheet.

Column Name	Column Description	Acceptable Values	Optional?
TestCase ID* or *Molecule Name	Unique identifier or name for a test case, or Molecule. The cell should be left blank if the action on the corresponding row is a part of the same test case or Molecule declared earlier. Init and Cleanup are reserved names interpreted by the ZUG in a special way	***Init* : This is the initialization Molecule for the entire test plan. If a testcase is named *Init*, ZUG will execute this *testcase* before any test case is executed from the test case worksheet. If all action steps for *Init* succeeds, ZUG will proceed with other test cases. The result of *init* is not stored in the database. If this**	Mandatory, if the row corresponds to the first step of a test case, Molecule, initialization or cleanup. Otherwise, should be left blank

		fails then ZUG looks for a testcase named *cleanup Cleanup* : This is the clean up for the entire test suite. The cleanup testcase is executed at the end after all the test cases have been executed. Like the *Init* testcase, the result for the cleanup Molecule is also not stored in the database.	
Description	Textual Description of the Step.	Currently ignored by the ZUG, it is a good practice for the Test Case Writer to document the intent of the step within the Test Case or Molecule.	Optional, but highly recommended
property	This column is used to tell the ZUG how to execute the action or the testcase. The property can be a testcase or Molecule property, or it can be a test step property. Multiple properties can be specified as a pipe separated list	1. ***Auto or Manual* : (test case or Molecule property) . Means that the test case or Molecule will be executed by the ZUG.** 2. ***GCE* or Seq: (Test case or Molecule property) : GCE stands for "Generate concurrently executing testcases" - This means All the expanded test cases (using MVM) will be executed concurrently (i.e. in parallel) instead of sequentially.** 3. ***REM/Comment or Exec*; (test step property) To prevent a specific step (Action/Verification step) from execution.** 4. **Negative,neg-action,neg-verify for negative testing.**	**Default properties:** ***Auto* | *Seq* | *Exec***
Step:	This should be specified for each Action of a Test Case or a Molecule (but) not for a Verification step. It should be a monotonically increasing number for each test case action. If multiple steps of a test case have the same number, then those steps are executed concurrently. Some steps can be considered to be initialization steps, a few others the actual action steps, and the rest cleanup steps. The	"2i" and "5c" are examples of a valid initialization step and a valid cleanup step	Optional, unless two steps are supposed to be executed in parallel, or are used to pair an initialization step with a cleanup step within a test case or Molecule. The default behavior, when steps are not numbered are that the steps are exceuted sequentially.

	initialization steps should be postfixed by an "i" and the cleanup steps should be postfixed with a "c". This allows ZUG to handle exceptions properly, without the necessity to have a semantic understanding of the actions.		Only those cleanup steps of a testcase or Molecule will be executed, whose initialization steps have been completed or started.
Action	A Test Case is a sequence of one or more Actions. And each Action may take as many arguments as necessary.	**"@" : prefixes the name of a atom. ZUG executes the Action by executing the atom in a separate command shell window. The atom must return a status value indicating success or failure. A zero return indicates success. An non-zero return will cause the testcase or Molecule to abort, and initiate cleanup actions.** **"&" prefixes the name of a Molecule. ZUG supports the notion of Nested Molecules - wherein a Molecule or a normal testcase can refer to a series of Molecules, which can then invoke any number of other Molecules. Note that recursive Molecule calls are not supported.** **Built-in atoms (no prefix) - These are atoms that execute in the same process context as ZUG. These atoms are internal to the ZUG For a complete list of built-in atoms, see Built-in Atoms .** **In-Process atoms (no prefix) – Atoms may also be added by users to execute in-process. The name of the library must precede the name name of the atom.**	
ActionArg_ #	**All Action Arguments should have "ActionArg_#" as a caption for the column header, where # should be an integer, starting from 1 and be incremental. For example to have 3 arguments for the Action one can have 3 columns with the column caption reading "ActionArg_1", "ActionArg_2", "ActionArg_3". The contents may be an actual value, a macro, that would be substituted at "compile" time, or a context variable, that would be substituted at "execution" time. Refer to Macro description to learn about macros. Action arguments also support**	**The contents may be an input argument, referred to by its position, or its formal name, or an actual value, a context variable, or a macro that would be substituted at execution time.** **Context variables may be passed by reference or by value. If a context variable is referenced enclosed within a pair of '%' around it, the variable is passed as by value, otherwise ZUG just passes the context variable by reference and the atom, or Molecule is responsible for reading or setting the value of that context variable** ***Pass By Value* :** **Example: @openbrowser URL=**	Optional

	passing of named formal arguments e.g. Formal_arg1=Actual_arg1. Actual_arg1 may be an actual value, a context variable or a macro. While passing an argument from a test case to a molecule, only the value is passed (and not the formal name). However, if the action happens to be an atom, (action name prefixed by @, the arguments are passed as a *formal name=actual value* pair.). Formal Arguments increase the maintainability of the testsuite.	**%myurl%** ***Pass By Reference* : Example: @getwindowhandle WHandle=WINDOW_HANDLE**	
Verify	Each Action of a Test Case may consist of none, one, or several Verification Methods. And each Verification method may take several arguments as needed.	Any verification method that starts with a "@" will be considered a atom. ZUG will execute the Verification Method by running the atom in its own command shell window. A Verification Method may also invoke a Molecule. Note that verification methods may also be implemented as in-process atoms. In that case, the name of the atom must be prefixed with the package name, in which the atom is implemented. In process atoms should not be invoked using the :@" prefix.	Optional
VerifyArg_#	**All Verification Arguments should have *VerifyArg*_# as the column caption, where # must be a number (integer). This number must be incremental. For example to have 3 arguments for the Verification Method, one must have 3 columns with the column captions VerifyArg_1, VerifyArg_2, VerifyArg_3 respectively.**	The contents may be an input argument, referred to by its position, or its formal name, or an actual value, a context variable, or a macro that would be substituted at execution time.	

2.4 Prototypes

This is the place where one specifies the footprint (i.e. the number and names of formal arguments of invokable entities) for atoms and molecules. This allows ZUG to perform syntax verification.

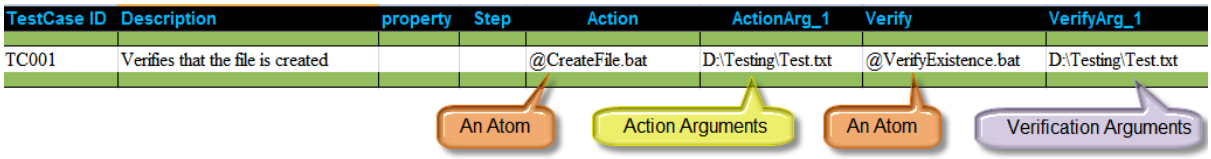
Specifying prototypes for molecules or atoms is optional. It is generally good practice to do so, because ZUG can automatically do compatibility checks between different versions of molecules, perhaps packaged inside include files.

3. Writing a simple Test Case

3.1 Introduction

To write test cases, you should copy the template spreadsheet file (***Testsuite_template.xls***) from the *templates* folder of your ZUG installation directory. Test cases are written in the *TestCases* worksheet of the spreadsheet and the spreadsheet with all its test cases is called a test suite.

The *TestCases* worksheet of the spreadsheet has a number of columns. For ZUG to execute a test case, the first row in the *TestCase ID* and the *Action* columns cannot be empty. Test cases are separated by the green rows, which contain a description of the intent of the test cases, and each test case must contain at least one row.



The above test case shows how to create a file and also verifies whether the file is created or not.

3.2 Action/Verification steps and their arguments

A test case in CHUR consists of one or more Action steps and their corresponding verification steps.
To write an automated test case you need to write action steps in the Action column and verification steps in the Verify column.

These action steps can be references to Atoms or Molecules. We shall use only Atoms for the Action and Verify steps in this section.

- An Atom is a user-written program or a CHUR built-in or user-extended function, that implements a very specific task e.g. open an application, click a button, save a file, navigate to a webpage, etc.

Atoms accept arguments just like any other program or function. Arguments can be plain text, Context Variables or Macros. Macros that contain some value and are defined in the Macros worksheet.

- A Macro is a name given to a constant string value (text). Macros are defined in the *Macros* worksheet and the values are replaced at runtime in the *TestCases* and *Molecules* worksheet of the spreadsheet wherever the corresponding Macro names are used.

3.3 Flow of Execution

ZUG starts to execute test cases on the *Testcases* worksheet of the spreadsheet in the order they are defined.

ZUG executes the Atom in the Action column of the test case by passing the argument list to the Atom and launching the Atom using the interpreter defined in the **ZugINI.xml** file and then it executes the Atom in the Verify column by passing the argument list, if any, to it.

The default flow of execution of a test case is sequential, i.e. row by row. That is, first the Action step is executed and then its verification step, if any, is executed, which is in the same row[1] of the *Testcases* worksheet. Once both the Action and Verify step are executed of the first row of the test case, the control jumps to the second row

1 Note that one action may have more than one verification step. Though it may increase readability if the steps were packaged as a molecule, and invoked as a single step, CHUR allows multiple verification steps to be specified inline, on consecutive rows, under the Verify column.

of the test case, if any, and starts to execute the Action and Verify steps and so on.

3.4 Command line execution

In order to execute your testsuite from command line you should type the following on the console:

```
C:\>runzug C:\Testsuites\Template.xls -verbose
```

runzug is the command to launch ZUG from command line, the second argument is the path to your testsuite file and the third argument *-verbose* tells ZUG to show the detailed output of the executiion of the test suite.

4. Built-in Atoms

There are some atoms which are internally included in ZUG and can be used just like other Atoms in CHUR test suite. These atoms are often referred to as built-in atoms. To invoke built-in atoms refer to them without the usual "@" prefix.

Built-in atoms execute in the same process context as the engine that drives the test cases. Therefore they are significantly less expensive to invoke, than out-of-process atoms.

CHUR also supports user-written atoms that are executed in the same process as the engine. Currently, only atoms written in Java, can be invoked as in-process atoms. To invoke these atoms, the name must be prefixed with the name of the package in which they are contained.

4.1 Context Variable Atoms

These built-in atoms implement the Context Variable features available in CHUR. See the section below for the argument list for these atoms

Name of Atom	Description
SetContextVar	Declare and set the context variable value for given name.
UnSetContext Var	Delete the context variable value for given name.
AppendToCont extVar	It will append value of key=value pair to given context variable value.

4.1.1 SetContextVar

Parameter Name	Purpose	Type	Valid Values	Direction	Optional?	Default Value
name=value	name represents Context variable name and value represent the Context variable value to be set. If context variable already exist then its value will be update to new value.	Scalar	any string	Input	Required	none

4.1.2 UnSetContextVar

Parameter Name	Purpose	Type	Valid Values	Direction	Optional?	Default Value
name	string represents Context variable name to be delete if exist.	Scalar	any string	Input	Required	none

4.1.3 AppendToContextVar

Parameter Name	Purpose	Type	Valid Values	Direction	Optional?	Default Value
contextvar	Context variable name to which value to be append. It should be valid context variable name.	Scalar	context variable name.	Input	Required	None
key	Value to be append to context variable value. It should be in key=value pair. Minimum one "key=value" pair should be their. Their can be multiple "key=value" pair if user wants different value to be append. Each pair will be different ActionArg or VerifyArg. For ex. Action will be "AppendToContextVar" and its Action arguments will be contextvar=path, Text=\, FileName=abc, Dot=., FileExtension=txt respectively. Then contextvar "path" value will be modified to C:\abc.txt where "C:" is original value for contextvar "path".	Scalar	key=value pair or key1=value1,key2=value2 etc.	Input	Required	None

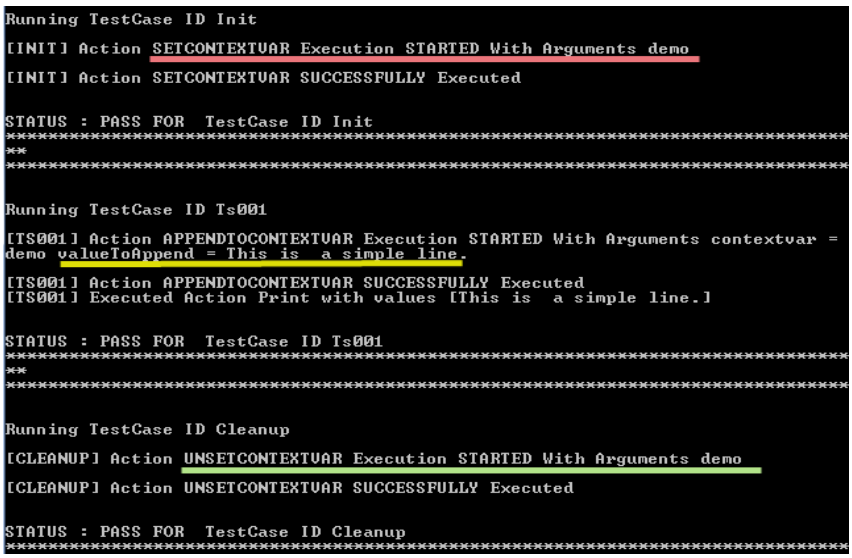
The following example show how to set a context variable, append values to the context variable and also unset the same.

TestCase ID	Description	property	Step	Action	ActionArg_1	ActionArg_2	ActionArg_3
Init				SetContextVar	demo		
Ts001				AppendToContextVar	contextvar=demo	value1=This is	value2= a simple line.
				Print	%demo%		
Cleanup				UnsetContextVar	demo		

or

FR 667				SetContextVar	Ctx##=243	Rty##=567	GG##=sw2
				Print	%Ctx##%	%GG##%	
				Print	%Rty##%		
				UnsetContextVar	Ctx##	Rty##	GG##
FR 667 EXT				SetContextVar	Cct={1,2,4}	Ffg={2,45,78}	
FR 667 WOR				Print	$$%Cct%	$$%Ffg%	

The following example show how to set a context variable, append values to the context variable and also unset the same. Here is the output of the above testcase.



4.2 XML Parsing Atoms

These in-process atoms are part of the ZUOZ atom library, and can be used to automate testing of Web Applications. The atoms must be prefixed with ***Zxml.*** since they are dynamically loaded as a package at runtime . Please refer to the ZUOZ Reference Guide for documentation on these and other atoms.

- **For further reference visit ZUG Forum.**

4.3 Web Automation Atoms

These in-process atoms are part of the ZUOZ atom library, and can be used to automate testing of Web Applications. The atoms must be prefixed with ***Zbrowser.*** since they are dynamically loaded as a package at runtime . Please refer to the ZUOZ Reference Guide for documentation on these and other atoms.

- **For further reference visit ZUG Forum.**

4.4 Print Atom

This atom just shows the values passed to it on console. It basically can be used for debugging.

4.5 GetValueAtIndex Atom

This atom returns the value of the MVM or MVCV passed as scalar at the specific index.

GetValueAtIndex($MVM/%MVCV%,INDEX,ContextVariable)

4.6 GetCurrentIndex Atom

This atom returns the current index, starting at , of the iterator being used for the specifed MVM or MVCV passed as scalar value.

GetCurrentIndex($MVM/%MVCV%,INDEX,ContextVariable)

4.7 GetValueAt Atom

This atom returns the value of target MVM or MVCV at the position corresponding to the SourceValue in the source MVM or MVCV as scalar value.

GetValueAt($TargetMVM/%TargetMVCV,$SourceMVM/%SourceMVCV,SourceValue,ContextVariable)

5. Exception Handling

5.1 Test case or Molecule exception handling

An *exception* is an event, which occurs during the execution of a program, that disrupts the default sequential flow of the program's instructions. When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an ***exception object***, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called ***throwing an exception***.

In CHUR, the idea is similar, but it works slightly differently. The idea is that certain steps are designated to be cleanup actions for other actions. An example of a cleanup action for opening a file, would be closing the open file. Cleanup is a good idea to restore the state of the system, so that subsequent tests can execute from a clean state, and not be affected by previous errors.

In CHUR, cleanup actions are matched with their corresponding “initialization” actions using a step index. Thus, initialization and cleanup steps are designated in pairs, both having the same step index. Initialization step indexes increase monotonically, while cleanup step indexes decrease monotonically, thus forming nested pairs of initialization and cleanup steps.

The step column is used to designate specific actions to be either an initialization action or cleanup actions. To identify the pair, numbers are used. For a given pair, the initialization action is a numeral, postfixed by an "i" and the cleanup action is a numeral, postfixed with a "c". So, whenever an initialization step fails the corresponding cleanup action (bearing the same index) is executed.

For example, If any Atom fails in a test step which is designated with “3i” in the step column on the TestCases or Molecules worksheet of the spreadsheet, then execution will jump to the corresponding clean up step which is designated “3c ”. The scope of the initialization and cleanup pairs is within a single test case, or a single molecule.

Note that, after an exception, execution resumes at the cleanup step corresponding to the last executed initialization step. The subsequent steps may, or may not be cleanup steps (corresponding to initialization steps executed earlier).

1	TestCase ID	Description	property	Step	Action	ActionArg_1	ActionArg_2	Verify	VerifyArg_1	VerifyArg_2
2										
5	TC-002	Create a file		1i	@CreateFile.bat	$file_location	$file_name_1	@VerifyExistence.bat	$file_location	$file_name_1
6					@Wait.bat	$timeout				
7				2i	@CreateFile.bat	$file_location	$file_name_2	@VerifyExistence.bat	$file_location	$file_name_2
8					@Wait.bat	$timeout				
9				3i	@CreateFile.bat	$file_location	$file_name_3	@VerifyExistence.bat	$file_location	$file_name_3
10					@Wait.bat	$timeout				
11				3c	@DeleteFile.bat	$file_location	$file_name_3			
12					@Wait.bat	$timeout				
13				2c	@DeleteFile.bat	$file_location	$file_name_2			
14					@Wait.bat	$timeout				
15				1c	@DeleteFile.bat	$file_location	$file_name_1			
16										

In steps 1i, 2i and 3i, we are creating three different files and also verifying their creations respectively.
Now if any steps between 1i and 2i fails, it directly jumps to 1c which deletes the first file. It simply skips the steps in between.
Similarly if any steps between 2i and 3i fails, it jumps to 2c and executes the remaining steps.

5.2 Test suite exception handling

Similar to the above section, where the issue of exception handling inside a test case or a molecule was discussed, CHUR provides the ability to initialize the testsuite, prior to executing any testcase, as well as cleanup the environment after all testcases have been executed. The purpose of the cleanup is to ensure that the machine is left in a stable and consistent state, where other testsuites can be executed properly.

For this purpose, CHUR uses two special keywords as reserved names for test cases.

A test case can be defined as an initialization test case by naming the test case as ***Init.*** This ensures that the test case is executed before any other the test cases defined in that test suite. This can be helpful if you need to create some resources, or load a database table with test data at the beginning of the execution of other test cases so that they can use the resources or data without having to create them.

Similarly, a test case can be defined as a cleanup test case by naming the test case as ***Cleanup.*** This ensures that the test case is executed at the end, even if all other test cases have failed. This can be helpful to cleanup all the resources created during the execution of the test suite.

Even though these are called test cases, and are syntactically identical to normal test cases, their outcomes are not reported. Also, unlike a failure in a normal test case, a failure in the ***Init*** test case will prevent any of the other test cases from being executed, and only the ***cleanup*** test case will be executed.

6. Context variables

6.1 Definition

Context variables are named containers that can be read and written to at run time. They are used to hold and pass information between atoms and molecules. The scope of a context variable is global (visible to all test cases) - must be declared inside the test suite.

6.2 Declaring a Context variable

To declare a context variable you need to use the *SetContextVar* atom in the Action column. *SetContextVar* is a built-in Atom that creates a context variable by the name that is passed to it as the first argument.

A value can be assigned to a context variable at the time of declaration. To do that the first argument can be replaced key-value pair

Context variables are accessible throughout the test suite and hence, if another context variable is declared with the same name, its value will be overwritten.

Context variables are different from Macros as the values of the Macros cannot be changed at runtime, by atoms of any sort. However, a context variable's value can be read or altered by atoms, or molecules.

6.3 Pass arguments by value or by reference

Once a context variable has been declared it can be passed to any atom or molecule as an argument. There are two reasons why you would like to pass context variables to an atom or a molecule as arguments. Either you want to send some information to the atom/molecule, or you want to receive some information from the atom/molecule.

When you need to send some information that is stored in a context variable, you should pass the context variable by value. The syntax to do that is to enclose the name of the context variable inside two % symbols, e.g. %BROWSER%, where the value of BROWSER could be SAFARI, and you want to send the content, i.e. SAFARI.

When you need to receive some information from the Atom, you should pass the context variable by reference. The syntax to do that, is to use the name of the context variable without any '%' signs enclosing it. e.g. PAGE_CONTENT, so the atom can return the contents of the page inside the PAGE_CONTENT context variable.

6.4 Reading and Altering Context variable value from atoms (ZUG API)

The context variables that are declared on the CHUR speadsheet can be accessed by Atoms using methods inside the ZugAPI library installed at the time of ZUG installation.

For example, to send the value of a context variable, as an argument, it should be referenced as ***%contextvariable%***, ZUG retrieves the value of the context variable and then sends it to the Atom. So, when you pass the value of the context variable, the Atom or Molecule does not know about the actual name of the context variable used.

In contrast, if the context variable is passed by reference, then the Atom must use the ZugAPI methods to read

the value of the context variable or write some value into the context variable.

ZugAPI has two methods to read from and write to a context variable:

GetContextVar is used to read the value of a context variable and ***AlterContextVar*** is used to write some value into the context variable. Here are some example of how you can access the ZugAPI methods in different languages:

Windows Platform

Ruby/Watir

```
ZugCOMObj = WIN32OLE.new("Automature.ZugAPI")
ZugCOMObj.AlterContextVar(ARGV[1], "MyValue")
```

AutoIT

```
$ZugCOMObj = ObjCreate("Automature.ZugAPI")
$MyValue = $ZugCOMObj.GetContextVar($NameOfContextVar)
```

JScript

```
var ZugCOMObj = new ActiveXObject("Automature.ZugAPI");
ZugCOMObj.AlterContextVar(WScript.Arguments.Item(0),"MyValue");
```

VB Script

```
Set ZugCOMObj = CreateObject("Automature.ZugAPI")
ZugCOMObj.AlterContextVar WScript.Arguments.Item(0), "MyValue"
```

Batch script

You can invoke functions to read or write to a context variable from from the command line using the ZUGUtility[2].

6.5 Zuoz pearls

Automature provides a collection of pre-built Atoms that can automate testing of a web application, fetch performance metrics, invoke web-services, etc. For more information on how to get this library, please contact support@automature.com

6.6 Appending values to a Context variable

The value stored inside a context variable is plain text and you can append more text/value to the context variable. The built-in function *AppendToContextVar* can be used to append values to an existing context variable.

6.7 Destroying a Context variable

To destroy (Unset) a context variable you should use the built-in function *UnsetContextVar.*

2 The **ZugUtility** provides a command line interface to invoke methods inside the ZugAPI

6.8 ZUG test environment variables

ZUG defines and maintains some global context variables. These variables may be used by the test case designer for developing test cases.

Nr	Name	Purpose	Comments
1	ZUG_TPSTARTTIME	Timestamp when Test Suite execution started.	Value set by ZUG when the test suite execution is started
2	ZUG_TCSTARTTIME	Timestamp when Test Case execution started	Its value updated by ZUG when each new test case execution started
3	ZUG_TSSTARTTIME	Timestamp when Test Step execution started	Variable value updated by ZUG when new test step execution started
4	ZUG_TESTSUITEID	Test Suite Name	
5	ZUG_BASETCID	Base Test Case ID	
6	ZUG_TCID	Generated Test Case ID	
7	ZUG_TCYCID	Test Cycle ID	
8	ZUG_TCYCLENAME	Test Cycle Name	The name created by ZUG if no previous Test Cycle ID provided.
9	ZUG_TSEXDID	Test Execution Detail Id	Value is automatically generated by the Reporting Database for every test case execution
10	ZUG_TESTPLANID	Test Plan ID	
11	ZUG_TOPOSET	Topology SET ID	Topology Set ID
12	ZUG_TOPO	Topology ID	Comma Separated list of Topology ID's participating in the Automation for testing.
13	ZUG_TESTSUITE_TIMEOUT	Test Suite Timeout	This will contain the maximum time in seconds for which a test suite can execute. By default its value will be 2*60*60 seconds. One can use this context variable to change the default behavior of timeout dynamically during the test suite execution.
14	ZUG_TESTSTEP_TIMEOUT	Test Step Timeout	This will contain the maximum time in seconds for which a test step can execute. By default its value will be 30*60 seconds. One can use this context variable to change the default behavior of timeout dynamically during test suite execution.
15	ZUG_EXCEPTION	Test Step Exceptions	If negative property is set to any Test Step then it will set this Context Variable with The error message of that Test Step and the Test will pass.
16	ZUG_ACTION_EXCEPTION	Test Step(Action) Exceptions	If neg-action property is set to any Test Step then it will set this Context Variable with the error message of that action test step
17	ZUG_VERIFY_EXCEPTION	Test Step(Verify) Exceptions	If neg-verify property is set to any Test Step then it will set this Context Variable with the error message of that verify test step

Example:

In the following example we show how to automate search between two destinations on a given date for a travel sit.

The macros sheet:

	A	B	C
1	**Macro Name**	**Value**	**Comment**
2	$URL	http://www.cleartrip.com/	a travel site
3	$Homepage_verify_text	Cleartrippers	Verifies this text in the webpage
4	$Browser	ie	The browser that will be launched
5	$From	Washington	Place of Departure
6	$To	New York	Place of Arrival
7	$Origin_Field	origin	Origin Field name in the webpage
8	$Destination_Field	destination	Destination Field name in the webpage
9	$Date_Field	depart_date	Date Field in the webpage
10	$Date	01/06/2012	Date of Departure
11	$Button_id	button_flight_search	Search Button
12	$Result_Page_Title	Cleartrip | International Air Search Results	Text in the Resulting page which should be displayed

The Test case sheet:

1	TestCase ID	Description	propert	Step	Action	ActionArg_1	ActionArg_2	ActionArg_3
2								
3	Init	Creating a variable named Handle			SetContextVar	Handle		
4		Open a browser			Zbrowser.Initialize	Handle	$Browser	
5								
6	TC001	Selecting a trip			Zbrowser.GoToURL	%Handle%	$URL	
7		Selecting the Origin place			Zbrowser.SetTextByName	%Handle%	$Origin_Field	$From
8		Selecting the Destination place			Zbrowser.SetTextByName	%Handle%	$Destination_Field	$To
9		Selecting a date			Zbrowser.SetTextByName	%Handle%	$Date_Field	$Date
10		Clicking the Search Button			Zbrowser.ClickButtonById	%Handle%	$Button_id	
11		Waiting for search result			Zbrowser.WaitForWindow	%Handle%	$Result_Page_Title	30
12		Destroying the variable			UnsetContextVar	Handle		
13								

The two highlighted rows shows the declaration and destruction of a Context variable.

7. Molecules

7.1 Definition

A Molecule is a collection of several test steps describing either a complete test case, or some part thereof. It is referenced by its name and can be invoked by a test case, or another Molecule (known as a complex Molecule).

It accepts input arguments, and can also pass data back to the caller. It can invoke another Molecule, an external, or a "built-in" atom.

A molecule behaves similar to a procedure, in generic programming languages, in that, it is not expected to return anything to the caller. When a molecule fails executing an atom, its execution sequence is interrupted, and the returned to the caller.

7.2 Designing a Molecule

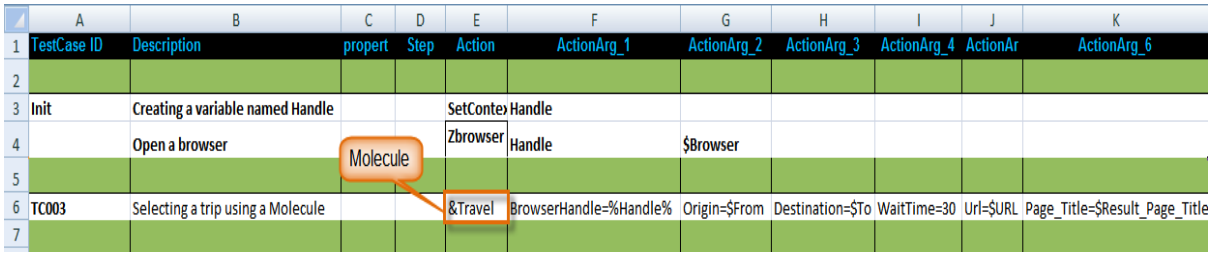
Molecules are designed to promote reuse of the logic of a test case. For example, the test case shown below consists of Atoms and is written on the TestCases worksheet of the spreadsheet.

A test case is implemented in a way that the logic cannot be used by other test cases. However, if we design a Molecule with a similar logic then it can be used by different test cases with an entirely different set of data, or purpose.

A Molecule is syntactically very similar to a test case. The only difference between the definition of a Molecule and a test case is that it has a Molecule ID whereas a test case has a TestCase ID. That is why TestCases worksheet and Molecules Worksheet are exactly similar except the first column that is TestCase ID or Molecule ID. Molecules also increase the readability of the Testcases worksheet.

The Molecules that are defined in the Molecule worksheet can be called by the test cases defined in the TestCases worksheet. Molecules that are defined in the Molecules worksheet can also call other Molecules defined there.

The same above test case could be written as follows:



	A	B	C	D	E	F	G	H	I	J	K
1	TestCase ID	Description	propert	Step	Action	ActionArg_1	ActionArg_2	ActionArg_3	ActionArg_4	ActionAr	ActionArg_6
2											
3	Init	Creating a variable named Handle			SetConte	Handle					
4		Open a browser			Zbrowser	Handle	$Browser				
5											
6	TC003	Selecting a trip using a Molecule			&Travel	BrowserHandle=%Handle%	Origin=$From	Destination=$To	WaitTime=30	Url=$URL	Page_Title=$Result_Page_Title
7											

Here a Molecule **Travel** is defined in the Actions column along with its respective arguments.
The Molecule **Travel** simply does the same operations as the test case TC001 shown before.

	Molecule ID	Description	Property	Step	Action	ActionArg_1	ActionArg_2	ActionArg_3	ActionArg_4	ActionArg_5	ActionArg_6
2											
3	Travel				#define_arg	BrowserHandle	Origin	Destination	WaitTime	Url	Page_Title
4		Go to the specified URL			Zbrowser.GoToURL	#BrowserHandle	#Url				
5		Selecting the Origin place			Zbrowser.SetTextByName	#BrowserHandle	$Origin_Field	#Origin			
6		Selecting the Destination place			Zbrowser.SetTextByName	#BrowserHandle	$Destination_Field	#Destination			
7		Selecting a date			Zbrowser.SetTextByName	#BrowserHandle	$Date_Field	$Date			
8		Clicking the Search Button			Zbrowser.ClickButtonById	#BrowserHandle	$Button_id				
9					UnsetContextVar	BrowserHandle					

7.3 Positional and named arguments

When a Molecule is called by a test case or another Molecule, there are two different ways of passing the arguments: namely, positional argument passing and named argument passing.

Positional argument passing is the simpler way of passing arguments to a Molecule where you just specify a text value, the name of a Macro or a context variable. The Molecule can access these arguments by their positional index in the argument list.

The first row of the Molecule body must contain **#define_arg** as its Action step and the name of the *formal arguments* of the Molecule in the Action argument columns, prefixed with a #.If there are two arguments passed to a molecule, then in the molecule sheet equal number of arguments must also be defined and in

Example-

1.When the argument is a context variable that has been passed in by reference and value

TestCase ID	Description	property	Step	Action	ActionArg_1	ActionArg_2
comment	Using a positional argument where the argument is a context variable that has been passed in by reference					
P001ARG				SetContextVar	Name=John Smith	
				&mol_pos	Name	$NameVerify

Molecule ID	Description	Property	Step	Action	ActionArg_1	ActionArg_2
mol_pos				#define_arg	name	verify_name
				Print	#name is	%#name%
				Zstring.Compare	%#name%	#verify_name

```
Working on Test Case Variable Combination mol pos
[P001ARG_MOL_POS] Executed Action Print with values [Name is, John Smith]
[P001ARG_MOL_POS] Execution Started Action Zstring.Compare with values [John Smith, John Smith]

[P001ARG_MOL_POS] Action ZSTRING.COMPARE SUCCESSFULLY Executed
```

In the above example *#name* is passed in as reference and *%#name%* is passed in as value.
Hence #name=Name and %#name%=John Smith

2.Using positional argument passing,where the argument has been passed in by reference and value in a SetContextVar,

TestCase ID	Description	property	Step	Action	ActionArg_1	ActionArg_2
comment	Using a named argument in a SetContextVar, where the named arg has been passed in by as reference and value					
P002ARG				SetContextVar	Name=John Smith	
				&mol_pos2	Name	$NameVerify

Molecule ID	Description	Property	Step	Action	ActionArg_1	ActionArg_2
mol_pos2				#define_arg	name	verify_name
				SetContextVar	NameRef=**#name**	
				SetContextVar	Nameval=**%#name%**	
				Print	%NameRef% is	%Nameval%
				Zstring.Compare	%Nameval%	#verify_name

```
[P002ARG_MOL_POS2] Action SETCONTEXTVAR SUCCESSFULLY Executed
[P002ARG_MOL_POS2] Executed Action Print with values [Name is, John Smith]
[P002ARG_MOL_POS2] Execution Started Action Zstring.Compare with values [John Smith, John Smith]

[P002ARG_MOL_POS2] Action ZSTRING.COMPARE SUCCESSFULLY Executed
```

In the above example we set a context variable NameRef=#name (passed in as reference).
Hence NameRef contains the value *Name*.
We also set the context variable Nameval=%#name% (passed in as value)
Hence Nameval contains the value John Smith.

3.Using an argument in a AppendToContextVar, where the named arg has been passed in as reference and value and also returns a value.

TestCase ID	Description	property	Step	Action	ActionArg_1	ActionArg_2	ActionArg_3
comment	Using a named argument in a AppendToContextVar, where the named arg has been passed in as reference and value						
P003ARG				SetContextVar	Name=John Smith		
				SetContextVar	String		
				&mol_pos3	Name	%Name%	String
				Zstring.Compare	%String%	$Name	

Passing by Value

Passing by Reference

Molecule ID	Description	Property	Step	Action	ActionArg_1	ActionArg_2	ActionArg_3	ActionArg_4
mol_pos3				#define_arg	name	value	stringVar	
				Print	#name is	#value		
				AppendToContextVar	#stringVar	%#name%	can only be	#value

```
Working on Test Case Variable Combination mol pos3
[P003ARG_MOL_POS3] Executed Action Print with values [Name is, John Smith]

[P003ARG_MOL_POS3] Action APPENDTOCONTEXTVAR Execution STARTED With Arguments contextvar = String valueToAppen
d = John Smith can only be John Smith

[P003ARG_MOL_POS3] Action APPENDTOCONTEXTVAR SUCCESSFULLY Executed

********** Molecule P003ARG_mol_pos3 Execution Finished **********


[P003ARG] Execution Started Action Zstring.Compare with values [John Smith can only be John Smith, John Smith
can only be John Smith]

[P003ARG] Action ZSTRING.COMPARE SUCCESSFULLY Executed
```

Named argument passing is more descriptive than positional argument passing as the arguments are passed as key-value pairs and hence it increases the readability and maintainability of the test case and the Molecule.

When using named argument passing, an additional row must be added in the Molecule body. The first row of the Molecule body must contain **#define_arg** as its Action step and the name of the *formal arguments* of the Molecule in the Action argument columns, prefixed with a #. The *formal argument names* should be used as the key in the **key-value** pair, when invoking the Molecule using *named argument passing* mechanism, as mentioned above.

When names arguments are used to invoke a Molecule, the order of arguments is not significant.

Example:

1.When the named argument is a context variable that has been passed in by reference and value

TestCase ID	Description property	Step	Action	ActionArg_1	ActionArg_2	ActionArg_3
comment	Using a named argument where the argument is a context variable that has been passed in by reference and value					
N001ARG			SetContextVar	Name=John Smith		
			&mol_pos	name=Name	verify_name=$NameVerify	

The molecule mol_pos is same as mol_positional stated above in testcase P001ARG

2.Using a named argument in a SetContextVar, where the named arg has been passed in by as reference and value

TestCase ID	Description property	Step	Action	ActionArg_1	ActionArg_2	ActionArg_3
comment	Using a named argument in a SetContextVar, where the named arg has been passed in by as reference and value					
N002ARG			SetContextVar	Name=John Smith		
			&mol_pos2	name=Name	verify_name=$NameVerify	

The molecule mol_pos2 is same as mol_pos2 stated above in testcase P002ARG

3.Using a named argument in a AppendToContextVar, where the named arg has been passed in as reference and value and also returns a value.

TestCase ID	Description property	Step	Action	ActionArg_1	ActionArg_2	ActionArg_3
comment	Using a named argument in a AppendToContextVar, where the named arg has been passed in as reference and value					
N003ARG			SetContextVar	Name=John Smith		
			SetContextVar	String		
			&mol_pos3	name=Name	value=%Name%	stringVar=String
			Zstring.Compare	%String%	$Name	

The molecule mol_pos3 is same as mol_pos2 stated above in testcase P003ARG

Please Note - **Zug cannot pass a macro having key=value to a molecule using positional arguments.**

Example: In the following testcase we have a macro *$key_value_macro* having values *abc=123*

TestCase ID	Description property	Step	Action	ActionArg_1	ActionArg_2
comment	Zug cannot pass a macro defined as key=value to a molecule using positional arguments.				
NARG008			&mol008	$key_value_macro	

Here is the molecule sheet:

Molecule ID	Description	Property	Step	Action	ActionArg_1
mol008				#define_arg	key_val
				Print	**#key_val**

The above output does not print *abc=123.* Rather it prints *#key_val.*
To pass this macro to a molecule, Named Argument passing is needed.

NARG008				&mol008	key_val=$key_value_macro

8. Multi-valued Macros

8.1 Definition

Multi-valued macros (MVM) contain lists of values, e.g. $FileTypes={DOC,DOCX,PPT,XLS}, or $ITERATOR={1..10}

Multi-valued Macros are often represented by a pair of $ symbols, i.e. $$. Use of the $$ is significant only while the MVM is being referenced inside a Testcase, or a Molecule. It is not significant during declaration in the MACROS worksheet.

Lists may consist of discrete values, or sequences of values.

$ITERATOR={1..4} is equivalent to writing $ITERATOR={1, 2, 3, 4}

Multi-valued macros in CHUR offer a powerful iteration mechanism, and is a core concept. Whenever MVMs are encountered inside a testcase or a molecule, that testcase or molecule is executed multiple times, once for each value of the MVM.

8.2 Cartesian MVM expansion

Cartesian MVM expansion can be useful when all the combinations of all the MVMs present in a test case or Molecule, need to be exercised. If more than one MVM is used in a test case then a cartesian product over the set of values of all the MVMs is calculated. e.g. If there are two MVMs:

$FILE_NAME={File1.txt, File2.rtf}
$LOCATION={**C:\TempFolder**, \\Network\SharedFolder}

and a testcase references both these MVMs as $$ FILE_NAME, and $$LOCATION, then four test cases will be generated, one for each of the following values of these MVMs.

1. [File1.txt, C:\TempFolder],
2. [File2.rtf, C:\TempFolder],
3. [File1.txt, \\Network\SharedFolder],
4. [File2.txt, \\Network\SharedFolder]

TestCase ID	Description	propert	Step	Action	ActionArg_1	ActionArg_2	ActionArg_3	ActionArg_4	ActionAr	ActionArg_6
Init	Creating a variable named Handle			SetConte	Handle					
	Open a browser			Zbrowser	Handle	$Browser				
TC004	Selecting a Way using Cartesian MVM			&Travel	BrowserHandle=%Handle%	Origin=$$Source_	Dest=$$Destination_	WaitTime=30	Url=$URL	Page_Title=$Result_Title

Macros declared for Cartesian MVM

In the above example we can search all the combinations of Source to Destinations.

8.3 Indexed MVM expansion

Indexed MVM expansion can be useful when you need only the combinations of corresponding element index of another MVM. The only condition is that all the MVMs specified as an indexed MVM *must* have an equal number of elements. e.g. If there are two MVMs

$FILE_NAME={File1.txt, File2.rtf} and $LOCATION={**C:\TempFolder**, \\Network\SharedFolder}

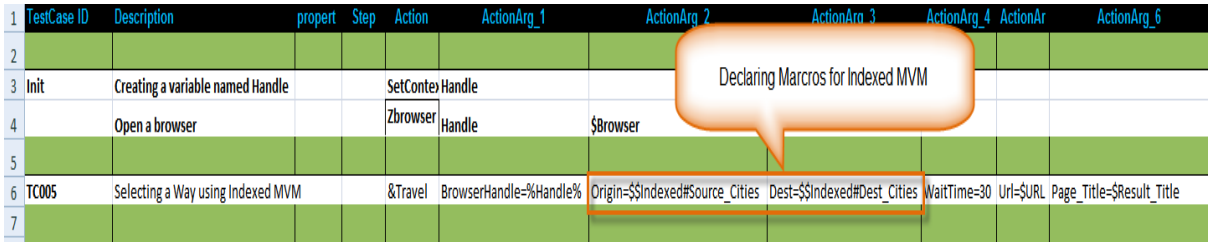
then an indexed MVM is defined as another MVM containing the names of the above MVM, viz.

$FILE_NAME_LOCATION={$$FILE_NAME, $$LOCATION}

Note that the third MVM, i.e. $FILE_NAME_LOCATION, contains the names of the other two MVMs. This is how indexed MVMs are declared. The syntax to expand an indexed MVM is to reference the MVM which contains the names of the other MVMs to be expanded as an indexed MVM, and append the name of the MVM prefixed by a hash (#).

e.g. $$FILE_NAME_LOCATION#FILE_NAME and $$FILE_NAME_LOCATION#LOCATION will expand into two value combinations, viz.

1. [File1.txt, C:\TempFolder],
2. [File2.txt, \\Network\SharedFolder]



In the above example we show indexed MVM works.
We define two macros
$Source_Cities={Washington,Detroit}
$Dest_Cities={London,Liverpool}
$Indexed={$$Source_Cities,$$Dest_Cities}

The macros are called in the test case as Origin=$$Indexed#Source_Cities and Dest=$$Indexed#Dest_Cities

The resulting search will be
i)from Washington to London and
ii)from Detroit to Liverpool.

If any Macro definition is defined as $$, e.g $$Source_Cities={London,Detroit} then Zug will show a error message
in console as

```
Error occured while getting the values from Macro Sheet Macros
Exception Message is Excel/GetKeyValuePair : Error occured while getting
values from the macros Sheet.

 Message Excel/ExpandMacrosValue: Exception raised is ..: Indexed Macros
could not be found in Macro Table.: .
```

By changing the macro deifinition to $Source_Cities in macro worksheet of the test suite file, This error can be removed.

8.4 Nested MVMs

Not yet available.

This refers to the syntax, where the value of an element in an MVM list, is itself another MVM.

8.5 MVM used in Test-cases

When used inside the *TestCases* worksheet of the spreadsheet, is interpreted to auto-generate unique testcases using a new combination of values from each of the MVMs used in the testcase. Test case identifiers are automatically generated by ZUG, by concatenating the base id with the combination values. The auto-generated

testcases are executed sequentially, unless otherwise specified.[3]

8.6 MVM used in Molecules

When used inside the *Molecules* worksheet of the spreadsheet, invokes the Molecule multiple times sequentially[4], each time with a new combination of values (all executing within the context of the parent test case).

MVM used in Molecules using Named Arguments

If you are using named arguments to replace the macros in the molecules, then to use Cartesian MVM the named arguments are referenced as ***##named_argument***. The following example shows how cartesian MVM is used in a molecule using named arguments.
There are two Macros $MVM={value1,value2} and $MVM2={valueA,valueB}. The testcase calls a molecule which simply does cartesian product of these two MVMs.

TestCase ID	Description	property	Step	Action	ActionArg_1	ActionArg_2
MVM006				SetContextVar	StringCartesian	
	Verifies MVM using named arguments			&MoleculeMVM	list1=$MVM	list2=$MVM2
				Zstring.Compare	%StringCartesian%	$CartesianMVMList

Molecule ID	Description	Property	Step	Action	ActionArg_1	ActionArg_2	ActionArg_3
MoleculeMVM				#define_arg	list1	list2	
				AppendToContextVar	contextvar=StringCartesian	v1=##list1	v2=##list2

8.7 Scalar and Vector MVMs

MVMs are defined on the Macros worksheet of the spreadsheet. The syntax to declare an MVM, as we have already learned, is to prefix it with a single or a double dollar sign($$). When the MVM is used in the TestCases or Molecules worksheet, it should be prefixed with a double dollar($$) so that ZUG can expand the test case or Molecule for each value present in that MVM. This is referred to as a vector MVM.

However, if immediate expansion of the MVM is not desired, the MVM can be referred to with a single dollar sign. This allows the engine to interpret a list as a single text string value (including the braces, and the separators).

This distinction is useful when calling molecules, where the iteration needs to be delegated to the called molecule, instead of iterating the caller..

3 Auto-generated testcases can also be executed concurrently, when the GCE property is set for the testcase.

4 Auto-generated molecules can also be executed concurrently, when the GCE property is set for the molecule.

8.8 Indexing Macro values in a molecule without Indexed MVM

We can use the Index MVM feature in molecule without using Indexed MVM. The following example shows how we can perform Indexing of two MVMs passed as scalar Macro to a Molecule.

There are three Macros $TYPE={Animal,Fish,Tree,Bird} , $EXAMPLE={Lion,Shark,Oak,Pigeon} and $Cardinality={1,2,3,4} (where cardinality Macro is for number of elements in the MVMs).

TestCase ID	Description	property	Step	Action	ActionArg_1	ActionArg_2	ActionArg_3
Comment:	Indexing two mvm in molecule without using Index MVM feature.						
TC001				SetContextVar	listA	listB	
				&FormatList	$TYPE	listA	
				&FormatList	$EXAMPLE	listB	
				&PrintIndexdValues	listA	listB	$CARDINALITY
				unsetcontextvar	listA	listB	

Molecule ID	Description	Property	Step	Action	ActionArg_1	ActionArg_2	ActionArg_3
Comment	Molecule to remove the '{' '}' and replace ' , 'with ' :'						
FormatList				#define_args	str	list	
				SetContextVar	string=#str		
				Zstring.eval	trim('%string%', '{}')	string	
				Zstring.eval	Replace('%string%' , ',',':')	string	
				AppendToContextVar	#list	%string%	
				unsetcontextvar	string		
Comment	Takes two colon separated list and performs Index MVM						
PrintIndexdValues				#define_args	list1	list2	card
				setContextVar	v1	v2	v3=##card
				Zstring.GetElementInListByIndex	%#list1%	%v3%	v1
				Zstring.GetElementInListByIndex	%#list2%	%v3%	v2
				print	%v1%	%v2%	
				unsetcontextvar	v1	v2	v3

First the molecule FormatList is called twice with a Macros and a Context Variable .This Molecule formats the MVM and stores it in the Context Variable as a colon separated list (such as Animal:Fish:Tree:Bird).Then the two formatted list are passed to another Molecule PrintIndexValue which also takes the cardinality Macro and performs the Indexing of the two list and then print the indexed values.

9. Multi Valued Context Variable

9.1 Definition

Multi-valued Context Variable (MVCV) contains a list of values, e.g. mvcv1={value1,value2}, or mvcv1={1..10} where mvcv1 is a context variable.

Action	ActionArg_1
SetContextVar	mvcv1={value1,value2}
SetContextVar	mvcv2={valueA,valueB}

Multi-valued context variables are often represented by a pair of $ symbols, i.e. $$. Use of the $$ is significant only while the MVCV is being referenced inside a Testcase, or a Molecule. It is not required when declaring the context variable.

TestCase ID	Description	property	Step	Action	ActionArg_1
comment	Printing the values of a context variable as mvcv				
MVCV004				print	$$%mvcv1%

Multi-valued context variable in CHUR offer a powerful iteration mechanism, and is a core concept. Whenever MVCVs are encountered inside a testcase or a molecule, that testcase or molecule is executed multiple times, once for each value of the MVCV.
In the above example, a context variable *mvcv1* was declared in the Initialization Block. The testcase MVCV004 is executed twice taking each of the values one at a time.

```
TestCase ID      Status          Time Taken(In milli-seconds)    Comments

MVCV004_value1   pass            25
MVCV004_value2   pass            23
********************************************************************************
```

9.2 Cartesian Expansion

Cartesian MVCV expansion can be useful when all the combinations of all the MVCV present in a test case or Molecule, need to be exercised. If more than one MVCV is used in a test case then a cartesian product over the set of values of all the MVCV is calculated. e.g.
If there are two MVCVs:

mvcv1={valueA,valueB}
mvcv2={value1,value2}

and a testcase references both these MVCMs as $$%mvcv1% and $$%mvcv2%, then four test cases will be generated, one for each of the following values of these MVCVs.

1. [valueA,value1],
2. [valueA,value2],
3. [valueB,value1],
4. [valueB,value2]

comment	Cartesian Product of two MVCVs					
MVCV004				print	$$%mvcv2%	$$%mvcv1%

The above testcase will be executed 4 times -

```
TestCase ID      Status          Time Taken(In milli-seconds)    Comments

MVCV004_valuea_value1    pass             27
MVCV004_valueb_value1    pass             23
MVCV004_valuea_value2    pass             24
MVCV004_valueb_value2    pass             31
********************************************************************************
```

9.3 MVCV used in Test-cases

When used inside the *TestCases* worksheet of the spreadsheet, is interpreted to auto-generate unique testcases using a new combination of values from each of the MVCVs used in the testcase. Test case identifiers are automatically generated by ZUG, by concatenating the base id with the combination values. The auto-generated testcases are executed sequentially, unless otherwise specified.[5]

9.4 MVCV used in Molecules

When used inside the *Molecules* worksheet of the spreadsheet, invokes the Molecule multiple times sequentially[6], each time with a new combination of values (all executing within the context of the parent test case).

MVCV used in Molecules using Named Arguments

If you are using named arguments to replace the MVCVs in the molecules, then to use Cartesian MVCV the named arguments are referenced as ***##named_argument***. The following example shows how cartesian MVCV is used in a molecule using named arguments.
There are two MVCVs defined in the Init block : mvcv1={value1,value2} and mvcv2={valueA,valueB}. The testcase calls a molecule which simply does cartesian product of these two MVCVs.

TestCase ID	Description	property	Step	Action	ActionArg_1	ActionArg_2
comment	cartesian product in a molecule using named argument					
MVCV006				SetContextVar	StringCartesian	
				&MoleculeMVCV	list1=%mvcv1%	list2=%mvcv2%

Molecule ID	Description	Property	Step	Action	ActionArg_1	ActionArg_2	ActionArg_3
MoleculeMVCV				#define_arg	list1	list2	
				AppendToContextVar	contextvar=StringCartesian	v1=##list1	v2=##list2

5 Auto-generated testcases can also be executed concurrently, when the GCE property is set for the testcase.

6 Auto-generated molecules can also be executed concurrently, when the GCE property is set for the same.

In the above example we have passed the values of the context variables as *%mcvc1%* and *%mvcv2%* to the molecule *MoleculeMVCV.* These values are then expanded in the molecule using *##named_argument.* The following example shows how to use MVCVs when passed as reference to a molecule.

TestCase ID	Description	property	Step	Action	ActionArg_1	ActionArg_2
comment	cartesian product in a molecule using named argument when passed as reference					
MVCV007				SetContextVar	StringCartesian	
				&MoleculeMVCV2	list1=**mvcv1**	list2=**mvcv2**

Molecule ID	Description	Property	Step	Action	ActionArg_1	ActionArg_2	ActionArg_3
MoleculeMVCV2				#define_arg	list1	list2	
				AppendToContextVar	StringCartesian	**$$%#list1%**	**$$%#list2%**

It gives the same output for both the testcases.

9.5 Scalar and Vector MVCVs

MVCVs can be defined on the Testcase and Molecules worksheet of the spreadsheet. The syntax to declare an MVCV, as we have already learned, is to prefix it with a single or a double dollar sign($$). When the MVCV is used in the TestCases or Molecules worksheet, it should be prefixed with a double dollar($$) so that ZUG can expand the test case or Molecule for each value present in that MVCV. This is referred to as a vector MVCV.

However, if immediate expansion of the MVCV is not desired, the MVCV can be referred to without any dollar sign. This allows the engine to interpret a list as a single text string value (including the braces, and the separators).

This distinction is useful when calling molecules, where the iteration needs to be delegated to the called molecule, instead of iterating the caller.

Look at the following example-

TestCase ID	Description	property	Step	Action	ActionArg_1	ActionArg_2
comment						
MVCV001				SetContextVar	mvcv1={abc,def,ghi}	
				&mol_001	mvcv1	String

Molecule ID	Description	Property	Step	Action	ActionArg_1	ActionArg_2
mol_001				#define_args	mvcv	string
	to print mvcv name			print	**#mvcv**	
	to print mvcv value as scalar			print	**%#mvcv%**	
	print each value of mvcv (vector mvcv)			AppendToContextVar	#string	**$$%#mvcv%**

10. Concurrent Execution

CHUR allows test cases, as well as Molecules to execute concurrently. Concurrent execution can be specified in multiple ways, viz.

10.1 Test case or Molecule concurrency

Property is a column in the *Testcases* and *Molecules* worksheets, which allows the testcase designer to specify the properties of a test case or a Molecule. Multiple properties may be specified as a pipe separated list. **GCE,** which is an abbreviation of ***Generate Concurrent Execution,*** is a test case or Molecule property. It means that if a test case or a Molecule contains an MVM (Multi Valued Macro), and the property column specifies GCE, then the test case (or the Molecule) will be expanded for each combination of the MVMs and will run them concurrently (i.e. in parallel) instead of sequentially.

10.2 Test step concurrency

Step is a column of the Testcases and Molecules worksheets. This relates to each Action of a Test Case or a Molecule (but) does not apply to a Verification step. It should normally be a monotonically increasing number for each test case action. If two or more actions of a test case/Molecule have the same step number, then these steps may be executed concurrently.[7]

10.3 Thread safe context variables

CHUR does not support the notion of scope, so, different threads creating or updating a context variable with the same name would conflict with other existing variables. To get around this, CHUR supports the notion of a thread specific instance of a context variable. It is the responsibility of the Test Suite designer to create and use thread specific context variables, when appropriate.

A ## (double hash) appended to the name of a context variable, automatically refers to the context variable with the thread id appended to the name, thus creating a unique instance of the variable. When a thread specific context variable is referenced in the main thread, ## equates to a null string.

10.4 Passing thread safe context variables as arguments

We refer to a thread safe context variable, for the purposes of argument passing in the following ways, viz

context_variable_name## - this is a thread specific instance of the context variable. CHUR will automatically replace ## with the thread id. This is equivalent to the scheme of passing by reference, except that CHUR will pass the thread specific name of the context variable.

%context_variable_name##% - this is the value of the thread specific instance of the context variable. CHUR will automatically replace ## with the thread id and retrieve the thread specific value of the context variable.

7 The number of threads available for concurrent execution, is a global setting. The concurrent execution of test steps or test cases is limited by the number of threads available. When the threads are exceeded, execution is sequentialized.

11. Negative Testing

11.1 Definition

Testing the system using negative data is called negative testing, e.g. testing the password where it should be minimum of 8 characters so testing it using 6 characters is negative testing.

11.2 How To Use

To use any test step as a negative testing, put the word "**negative**" in property column. Irrespective of testcase sheet or molecules sheet , this feature can be used. This negative testing feature will only work for atoms not for any molecule. When the atom fails and shows the Exception in console if negative property is set to it, then ZUG_EXCEPTION context variable will be set with the Exception message and the test step will pass.

We can put **"neg-action"** for testing an action step and **"neg-verify"** for testing a verify step. They will set exception messages separately in ZUG_ACTION_EXCEPTION and ZUG_VERIFY_EXCEPTION.

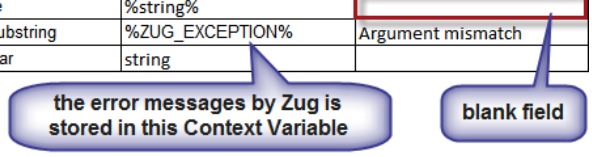
Example-

In the following test case, we are doing negative testing for Compare atom by giving incorrect number of arguments. The error given by Zug is stored in the context variable ZUG_EXCEPTION (which is an in-built context variable). In the next test step, we are checking whether the error thrown was Argument mismatch or not.

To be noted –

If we call a molecule in a test case stating negative property beside the molecule, the test case will fail. The negative property works ***only*** *for test steps.*

comment	Invalid number of arguments					
ZS001NG3	**Compare** atom- negative testing			SetContextVar	string	
				AppendToContextVar	contextvar=string	Value=10
		negative		Zstring.compare	%string%	
				Zstring.matchSubstring	%ZUG_EXCEPTION%	Argument mismatch
				UnSetContextVar	string	



The following figure shows the various outcomes of a test step for different properties of Zug.

Property	Action	Verification	Test Step Outcome
Negative	Pass	not-executed	FAIL
Negative	Fail	Pass	FAIL
Negative	Fail	Fail	Pass
Negative-Action	Pass	not-executed	FAIL
Negative-Action	Fail	Pass	PASS
Negative-Action	Fail	Fail	FAIL
Negative-Verify	Pass	Pass	FAIL
Negative-Verify	Pass	Fail	Pass
Negative-Verify	Fail	not-executed	FAIL

Example:

In the following test case, we have used the **Negative** property and have deliberately made the action and verification to fail. Even though the action and the verifications fail, the status of the test case is pass.

TestCase ID	Description	property	Step	Action	ActionArg_1	ActionArg_2	ActionArg_3	Verify	VerifyArg_1	VerifyArg_2
comment	making the action step fail and verify step fail and the test step outcome should pass									
ZS001NGC3	**Compare** atom- negative testing			SetContextVar	incorrecttext					
				AppendToContextVar	contextvar=incorrecttext	text1=Comparing	text2= string			
		negative		Zstring.compare	$INCORRECT_TEXT	%incorrecttext%		Zstring.matchSubstring	%ZUG_EXCEPTION%	Intentionally failing the verify step
				UnSetContextVar	incorrecttext					

(Enlarge to view)

In the next example, we have used the **Negative-Action** property and have deliberately made the action fail and then verify that the correct error message is displayed. The status of the test case is pass.

TestCase ID	Description	property	Step	Action	ActionArg_1	ActionArg_2	ActionArg_3	Verify	VerifyArg_1	VerifyArg_2
comment	making the action step fail and verify step pass and the test step outcome should pass									
ZS001NGC5	**Compare** atom- negative testing			SetContextVar	incorrecttext					
				AppendToContextVar	contextvar=incorrecttext	text1=Comparing	text2= string			
		Negative-Action		Zstring.compare	$INCORRECT_TEXT	%incorrecttext%		Zstring.matchSubstring	%ZUG_ACTION_EXCEPTION%	String do not match
				UnSetContextVar	incorrecttext					

(Enlarge to view)

In the next example, we have used the **Negative-Verify** property and have deliberately made the action pass and verification fail. The status of the test case is pass.

TestCase ID	Description	property	Step	Action	ActionArg_1	ActionArg_2	ActionArg_3	Verify	VerifyArg_1	VerifyArg_2
comment	making the action step pass and verify step fail and the test step outcome should pass									
ZS001NGC8	**Compare** atom- negative testing			SetContextVar	text					
				AppendToContextVar	contextvar=text	text1=Comparing	text2= string			
		Negative-verify		Zstring.compare	$text	%text%		Zstring.matchSubstring	$text	Intentionally failing the verify step
				UnSetContextVar	text					

12.NameSpaces

ZUG supports the inclusion of external worksheets that contain Macros, Prototypes or Molecules. Since, such inclusions can potentially lead to conflicts in names, imported names are placed inside specially named containers, called namespaces. All names declared within a namespace must be unique, and are referenced by prefixing the name of the entity with the name of the container, where it is defined.

Namespaces promote reuse of Molecules, and atoms, by providing libraries of pre-built containers.

To call the Molecules use '&' sign and to call the macros use '$' sign before the namespaces. For instance, to refer to an external Macro you should write $Spreadseet_Name.Macro_Name and to refer to an external Molecule you should write &Spreadsheet_Name.Molecule_Name, where Spreadseet_Name is the name of the external spreadsheet file, where the macro, or the molecule has been defined.

Zug can include multiple external files from the Config Sheet.

To call any molecule or any macro from an external file, the external file must be included in the Config sheet first.

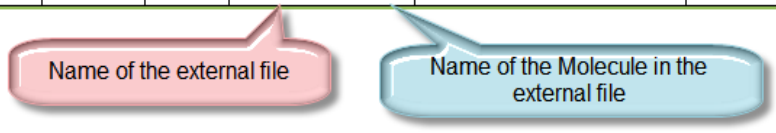
Example -

Include	C:\Documents and Settings\Administrator\Desktop\final\IncludeFile.xls,C:\Documents and Settings\Administrator\Desktop\final\ThirdIncludeFile.xls

12.1 Calling a Molecule from a single external worksheet.

The following example shows how to call a Molecule from an external file (which is IncludeFile.xls.)

TestCase ID	Description	property	Step	Action	ActionArg_1	ActionArg_2
comment	Calling a molecule from another File					
TestInclude001				&IncludeFile.Mol	FirstVal=$value1	SecondVal=$value2

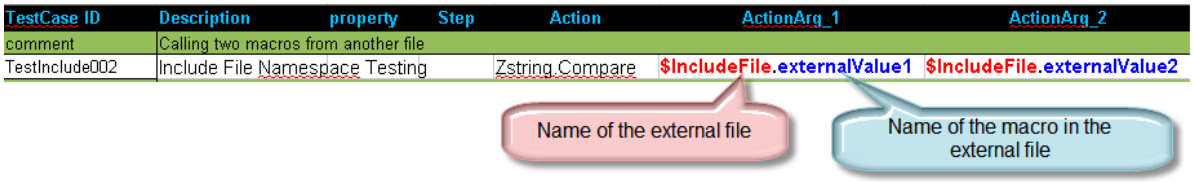


The macros **$value1** and **$value2** are passed to the Molecule **Mol** which is written in **IncludeFile.xls** spreadsheet. Here is the snapshot of the molecule sheet in the external file. It simply compares the two values passed from the testcase.

Molecule ID	Description	Property	Step	Action	ActionArg_1	ActionArg_2
Mol				#define_args	FirstVal	SecondVal
				Zstring.Compare	#FirstVal	#SecondVal

12.2 Calling external macros from an external worksheet.

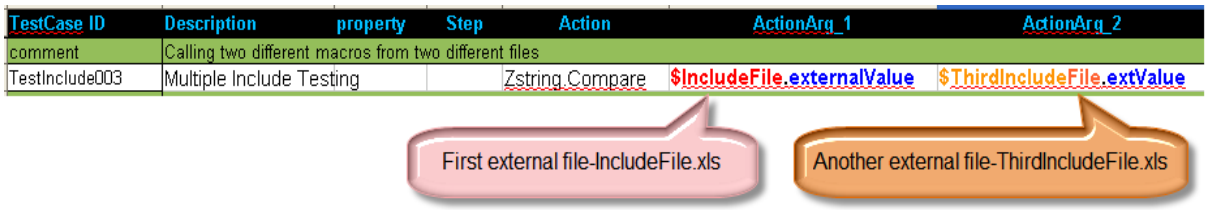
The following example shows how to call two external macros from an external worksheet (which is IncludeFile.xls.)



The macros **$externalValue1** and **$externalValue2** are written in the macros sheet of the external sheet IncludeFile.xls.

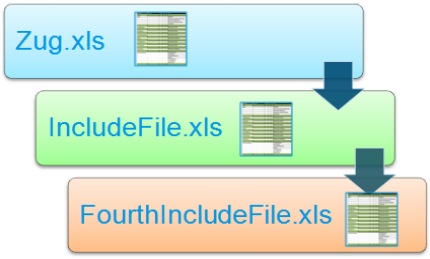
12.3 Calling two different macros from different external worksheets.

The following example shows how to call two external macros from from two different external worksheet (which are IncludeFile.xls and ThirdIncludeFIle.xls) The **IncludeFile.xls** contains **$externalValue** macro in the macros sheet and the **ThirdIncludeFile.xls** contains **$extValue** in its macros sheet. Here we compare these two macros written in different files in a single testcase.



12.4 Multi Level Include Files

Zug can include files in Multi level. i.e. Zug can include an external file which can again include another external file. In the following example we have used three test suites.*Zug.xls* contains the testcase which calls upon a molecule **Mol2** in the *IncludeFile.xls.* The molecule **Mol2** calls upon another molecule **Mol** in the *FourthIncludeFile.xls.* The config sheet of *Zug.xls* includes the *IncludeFile.xls* but **not** *FourthIncludeFile.xls.* The *IncludeFile.xls* includes the *FourthIncludeFile.xls*



Here is the snapshot of the testcase written in Zug.xls file. It calls the molecule **Mol2.**

TestCase ID	Description	property	Step	Action	ActionArg_1	ActionArg_2
comment	Calling a molecule from another File which calls another molecule from another file					
TestInclude004	Nested Include testing			&IncludeFile.Mol2	FirstVal=$value1	SecondVal=$value2

Here is the snapshot of the molecule **Mol2** written in IncludeFile.xls file. It calls the molecule **Mol2.**

Molecule ID	Description	Property	Step	Action	ActionArg_1	ActionArg_2
Mol2				#define_args	FirstVal	SecondVal
				&FourthIncludeFile.Mol	val1=#FirstVal	val2=#SecondVal

Here is the snapshot of the molecule **Mol** written in FourthIncludeFile.xls file. Here val1 and val2 contains the values of the macros **$value1** and **$value2** written in the Zug.xls file.

Molecule ID	Description	Property	Step	Action	ActionArg_1	ActionArg_2
Mol				#define_args	val1	val2
				Zstring.Compare	#val1	#val2

13. Command Line Options

ZUG supports various command line options. Some of them are

13.1 -verbose

-verbose option is used to display the execution result of the test suite in the console.

13.2 -include

As we have seen that we can include test suite by providing test suite's file name or path name along with the file name in the config sheet. Another way of including test suite is by including test suites at run time using the -include command line option .We can include multiple test suites from command line using the -include option. The test suites mentioned in the -include option must be separated by a comma. For example

>runzug E:\testsuites\testCMDInclude.xls -verbose -include=IncludeS2.xls,IncludeS1.xls

Here IncludeS1.xls and IncludeS2.xls are being included at runtime.Thus it provides a dynamic way of including test suites.Also the files included from command line has a higher prioirty over the files included from the config sheet.If a file is is mentioned in the include section of config sheet and a file with same name is being included from command line then the file mentioned in the config sheet is discarded and the one included from command line is used in the execution.

One more powerful feature of this option is to provide name spaces to the test suite included from command line(run time).The default name space of a test suite is the name of the test suite file .Consider the above example ,in this case the name space will be testCMDInclude. However ZUG provides another way of giving name space to the test suites included at run time .We can override the default name space of a test suite included at run time using the -include option. The syntax for giving name space in the -include option is

-include=namespace1#filename1,namespace2#filename2

where the file name can be either a file name or file path along with the file name.

13.3 -macrocolumn

We can have multiple value column in a test suite for a macro .At run time we can select a particular column

	Macro Name	Value	Value2	Value3	Comment
2	$field	{username,password}		{id,password}	
3	$values	{PatAdmin,123456}	{admin,admin}	{khan,123}	
4					
5					

using the -macrocolumn option. The syntax for -macrocolumn is

-macrocolumn=file identifier:column number, file identifier:column number

where the file identifier is the file name,default name space or the optional name space provided in the -include option. If we have provided any optional name space in the -include option then it should be the file identifier in the -macrocolumn.

If any cell of the selected column is empty then it takes the value of the default column cell(1st column i.e, the “value” column).For example if we run the above test suite by giving option in the -macrocolumn to choose the 2nd column of the macro value then the substituted value of $field will be {username,password}.

14. ZUG Logs

ZUG records output from the test suite execution in different text files. The files are stored in the default location (**%appdata%** under Windows) in a directory called ***ZugLogs*** in case of Linux it is saved under home/<users> directory with the same name. The log files can be archived in a central repository, accessible to Zermatt. Please refer to ***Zermatt User Manual*** for details on how to configure the location of the archive.

14.1 Result Log

This log records the console output, and contains the sequence in which test cases have been executed, the specific molecules and atoms that were invoked, along with the actual arguments that they were invoked with.

14.2 Debug Log

This log is for Automature's internal use only.

14.3 Error Log

This log contains a summary of any errors encountered by the atoms and molecules during test case execution.

14.4 Primitive Log

This log contains information specifically written by the executing atoms. Atoms can choose to provide custom trace information, or other useful output regarding their execution, using the ZugAPI's logging method.

15. Reporting test results

DAVOS is Automature's Webservices interface into ZERMATT, the Planning and Reporting Tool. Normally, test case designers need not be aware of the existence of DAVOS, since ZUG - the engine handles the reporting of test cases written in CHUR implicitly.

Under special circumstances, if the testcase needs to store special data, then DAVOS webservices atoms provide a mechanism for doing so. For more information on this, refer to the **DAVOS Reference Manual**.

To report to the database you need to configure the “configsheet” of your test suites.

If you the mandatory fields are not set and still you want to report to database then it will prompt for the database user name and password credentials.

To report you need to provide the command line switches of ZUG viz. -testplanid -topologysetid. These switches are mandatory for reporting.

16. Glossary

Action Arguments

The behavior of an action can be modified by sending it certain extra information. This extra information is sent as a list of “arguments”.

Action Step

A test case is a sequence of one or more Actions. An Action may be a Test Action, or a Verification Action. Each Action may take as many arguments as necessary. Actions carry out the logic of a test case, and also can be used to verify the outcomes of Test Actions.

Atom

Atoms are the basic units of execution in CHUR. They are entities, such as programs or scripts that may be executed at a command line level in a shell (e.g. the Command Prompt in Windows, or bash in Linux). Atoms may be invoked from the Test Cases or Molecules worksheets in the Test Suite spreadsheet. An example of an atom can be a program that enters text into a form field inside a web page, or can simulate a button click. In CHUR atoms are prefixed with the sign @.

Example: **@ClickLinkByHref.rb**

In this example ***ClickLinkByHref.rb*** is an atom written in Ruby, that allows the testcase to click on a link contained in a web page. Atoms can take any number of arguments, to qualify their action.

An atom is expected to return an exit status code that implicitly tells ZUG if the action was successful. By convention, a non-zero status is interpreted as a failure. When ZUG encounters a failure status, it automatically invokes the appropriate clean up steps specified in the CHUR spreadsheet for the corresponding test case.

Exceptions

In CHUR, when atoms return an error, an exception happens, causing sequential execution to be suspended, and cleanup actions to be performed. When a molecule encounters an exception, control is transferred back to the calling molecule, or test case, after executing the cleanup steps inside that molecule. When control is transferred back, execution does not resume sequentially. Instead, the caller's cleanup steps are executed. When a testcase encounters an exception, the testcase's cleanup steps are executed, and then the next test case is started.

Molecule

Molecules are a collection of atoms, or other Molecules, in a sequence, with the added ability to express more complex logic. Molecules may call atoms directly, or through other nested Molecules. Test Cases, themselves could be considered Molecules themselves, except that no other test case or Molecule can call them. Example of a Molecule can be to simulate a user login, by using the atom in the example above. In CHUR Molecules are prefixed with the sign &

Example: **&TwikiLogin**

Note that, Molecules, unlike atoms, can only be executed within ZUG. Also, Molecules do not return any exit status.

Namespace

CHUR allows test case designers to reuse test logic and data. Reuse involves the ability to package and distribute reusable molecules and macros in separate spreadsheet files, and referenced from the spreadsheet where the test cases are specified. Since names used in such distributions may not be unique, and therefore cause unexpected behavior, CHUR uses the concept of Namespaces, to make names unique. Therefore, when referring to an entity, such as a macro, or a molecule, that is defined in an external file, the name must be prefixed with the name of the file, where the entity is defined. This allows names, that are unique within the file, to be also globally unique.

Property

Testcase and molecules can have associated properties, that can be used to control certain behavior. Propeeties can be used to tell the engine to execute these entities in specific ways, e.g. Concurrently, or sequentially.

Test Case

A test case is a collection of steps whose purpose is to verify a certain feature of a product. Test-cases are the smallest units of reporting about the execution of tests.

Test Cycle

During the course of a project, multiple test sessions may be executed, related to a single test plan. These sessions would have several attributes in common, but some may differ. Usually, the difference would be the build of the product under test. These sessions are known as Test Cycles.

A Test Case would record only one result within the scope of a single test cycle. Multiple executions of the same test case within a single cycle, would overwrite previous results. ZUG, unless specifically requested to do otherwise, always creates a new test cycle, each time a test suite is executed.

Test Plan

A test plan is a plan of how a set of features shall be tested within the scope of a project schedule. A Test Plan is defined in Zermatt, and provides a context within which the test outcomes may be stored. Note that Test Plans are only relevant if the results are expected to be stored.

Test Step

A test case is a sequence of one or more Steps. A Test Step corresponds to a single row in a spreadsheet, that is not empty, or a comment. A test step results in an invocation of an Atom or a Molecule. A test step may be explicitly enumerated. If enumerated, it should be a monotonically increasing number for each step. If two steps of a test case are given the same number, then the two steps are intended for ***concurrent execution***.

Some of these steps may be designated ***initialization steps***, others may be designated to be ***cleanup steps***. The numbers are post-fixed with the letters “i” or “c” respectively. Unless they are explicitly designated to be such, test steps are understood to be normal actions carrying out the logic of the testcase.

Test Steps may implement two kinds of logic. The logic may be integral to the execution of the test itself, or it may verify the intermediate or final state of the system under test.

Test Suite

A test suite is a collection of test cases, grouped according to some criteria, such as a related set of requirements. In CHUR, a test suite is represented by a single spreadsheet file, containing one or more test cases. The test suite file may make use of other files, where additional dependencies may have been specified. However, the test cases themselves must reside in a single file.

Verification Step

Each Action of a Test Case may consist of none, one, or several Verification steps. Verification Steps may reference verification Molecules or verification atoms. And each Verification step may take several arguments as needed. As with other atoms, verification atoms are also expected to return an exit status code that implicitly tells ZUG if the verification was successful. By convention, a non-zero status code is interpreted as a failure. When ZUG encounters a failure status in verification, it automatically invokes the corresponding clean up steps.

In CHUR, verification actions, and their corresponding arguments, are identified by a set of columns explicitly set aside for this purpose.

Verification Arguments

The behavior of a verification can be modified by sending it certain extra information. This extra information is sent as a list of “arguments”.