**The Wrapper Script (sitar)**

This script will be a single python module that will run outside the virtual environment (although it can also be invoked within the environment as well). It will be placed into the central location for scripts, so it will be in everyone’s path. It will use the python3.6 release that IT has made available. It is basically a command launcher. The syntax would be:

sitar COMMAND ARGS

Were the COMMAND is a string that indicates the command, and ARGS are the specific arguments for that command.

The initial commands that we should work on are related to sda. Some initial commands that would run outside the virtual environment would be:

* -help
  + This command will display a list of possible commands (will depend on if you are running from with a virt env)
* set\_ws NAME
  + This will create a shell for the specified workspace (NAME). The workspace must already exist. The prompt will get changed based on name of the workspace. It will load all of the .cshrc files for the specific project and the new aliases will be available.
* shell
  + This will create a shell for the workspace that is in your cwd. The rest functionality is same as set\_ws.
* command COMMAND [ARGS]
  + This command is similar to shell. After the setup for the workspace is loaded, then the command will be run.
* create\_ws PROJECT [-n NAME] [-i]
  + This will create a new workspace for PROJECT with the name NAME. This will be a replacement for the setup\_workspace.py scripts that we have been using.
* ls\_ws [-a]
  + This will list out the user workspaces
* ls\_prj [FILTER]
  + This will list out the different possible project areas
* join\_ws PROJECT [-n NAME] [-l]
  + This command will allow the user to join a shared workspace
* rm\_ws NAME
  + This command will remove the workspace.
* refresh
  + This command will refresh the sda project/workspace data.
* gui
  + This will call SDA GUI, in future we need to create another GUI.
* repair\_ws NAME
  + This command will repair a workspace
* ws\_info PROJECT
  + This command will email the user with the project information
* jira
  + Create a JIRA ticket for some WTF command [check for logs of sda which is a design sync product , so if someone files a jora ticket it will be ahard to find where thr log for sda update is]

These commands involve the setup workspace and sda commands. There will be 1 configuration file that will be stored in the user home directory that stores settings. We can use the python config module for this (or other modules). The format is mostly just key value, so YAML or XML can be used. This configuration file will store some user settings as well as the sda data for project and workspaces.

The sequence of operation for this script would be:

1. Read in the user configuration file.
2. When user run the script first time, it will generate a config file.
3. Use the results from cached config file and Run the commands.
4. Config file will be forcefully gets updated when Create\_ws, Join\_ws or Remove\_ws commands used.

The background task for updating the sda is to do the following:

The result of this commands will is called as SDA UPDATE File, which is mentioned in Step 3 from FRAMEWORK

1. Run “sda ls -development -report verbose” [project data]
2. Parse the output
3. Run “sda ls -area -report verbose” [user Specific]
4. Parse the output
5. Store the data into a new file in the user home directory

The following sda data needs to be stored for the developments:

1. Development (PROJECT)
2. Data URL
3. Selector
4. Development Path

The following sda data needs to be stored for the areas:

1. Development Area (NAME)
2. Development
3. Assignment
4. Shared
5. Development Area Path

The following data needs to be stored in the config file in the user home directory:

1. Sda project data
2. Sda development data
3. Time of last update
4. Last 10 wtf commands(CWD, logfile)
5. User settings

Here is the flow for each command:

* set\_ws NAME
  1. Look-up the NAME in the sitr details to find out the Development Area Path
  2. Set the env var SYNC\_DEVAREA\_DIR to be the Development Area Path
  3. Run the command:
     + tcsh -c 'source $main\_setup; tcsh'
  4. Source The .cshrc.sitar file which sets .cshrc.project and skill/.cshrc file to set all the environment variables.
* shell
  1. Search the cwd and up to find the .cshrc.project file
  2. Set the env var SYNC\_DEVAREA\_DIR to be the location of the .cshrc.project file
  3. Run the same command as set\_ws
* command COMMAND [ARGS]
  1. Search the cwd and up to find the .cshrc.project file
  2. Set the env var SYNC\_DEVAREA\_DIR to be the location of the .cshrc.project file
  3. Run the command specified like this:
     + tcsh -c 'source $main\_setup; COMMAND ARGS'
* create\_ws PROJECT [-n NAME] [-i]
  1. Look up PROJECT in the sda details to find out the info
  2. Call sda mk to create the workspace
  3. Get the user workspace ready
* ls\_ws [-a]
  1. Display the info about the users areas from the sda data
* ls\_prj [FILTER]
  1. Display the info about the developments from the sda data
  2. Filter the list with the FILTER string if provided
* join\_ws PROJECT [-n NAME] [-l]
  1. Look up PROJECT in the sda details to find out the info about the developments
  2. There are 4 different shared workspaces (shared, tapeout, release\_prep, baseline)
  3. Call sda mk to join the workspace (see the setup\_workspace.py script for details)
  4. Get the user workspace ready (see the setup\_workspace.py script for details)
* ws\_info PROJECT
  1. Look up PROJECT in the sda details to find out the info about the developments
  2. Download and parse the project.xml file for that PROJECT
  3. Generate an email to the user with the project details
* repair\_ws NAME
  1. Look-up the NAME in the sitr details to find out the Development Area Path
  2. Setup the sitar env (see the Sitar.py script)
  3. Launch the stclc shell
  4. Use the Dsync methods to repair the workspace
* rm\_ws NAME
  1. Look up PROJECT in the sda details to find out the info about the developments
  2. Call sda rm to remove the workspace (see the setup\_workspace.py script for details)
* refresh
  1. Run through the update procedure in the foreground
* Jira
  1. Prompt the user for a summary of the issue
  2. Prompt the user for a detailed summary
  3. Gather the wtf history from the config file (should include the cwd and logfile locations)
* gui
  1. This would run sda gui

When run inside the virtual environment, the behavior is different so the commands to run inside the workspace are different. Once workspace path is set using set\_ws and a user is inside venv, additional script with following command will be available inside and sitewide. Name of the script will be WTF.

**The wtf script**

The script will be a collection of python3.6 scripts and modules that will run inside the virtual environment. It is meant as a replacement for the Sitar.py script. Since it will be available in the path, it will be easier to run. It is also basically a command launcher. The syntax would be:

wtf COMMAND ARGS

Were the COMMAND is a string that indicates the command, and ARGS are the specific arguments for that command. Arguments should start with a – or include the names of modules. Most commands require a list of modules. If no list is provided, then all modules are processed. For all commands a logfile should be generated in the log directory with debug information. This logfile should be displayed at the end of the command. Here is a list of initial commands:

* populate [-force]
  + This command will populate the workspace. The Sitar.py script can be used as a reference. This command does not take any modules.
* pop\_modules [-force]
  + This command will populate the modules that are in update mode. The Sitar.py script can be used as a reference. This command will run the pop command for each module. This command does need to be udpated to collect any errors that were generated and report them at the end.
* submit [-comment COMMENT] [-n TAG] [-pop] [-noemail] [module]
  + This command does a sitr submit. The Sitar.py script can be used as a reference. If the -n is given then a snapshot submit is done with the specified tag. If the -pop option is given, then the module is populated first, then the normal submit is done. The -pop option and -n are mutually exclusive. If neither option is given, then a normal submit is done.

The script will need to check for errors. Once this is complete, then an email should be generated. The address to use can be found in the project.xml file. The command also needs to be updated to check what branch we are on. This can be done with this command (where TOP is the top container):

url tags -btags TOP

If we are on Trunk, then if the TAG is given the name will get prepended with **REL\_** (which is currently done) and **\_v1.1** will be added at the end. For any other branch, the name should be converted to uppercase and prepended to the name with the same suffex.

The command sequence to use for the snapshot submit also needs to be updated. Instead of all of the different steps, only a tag needs to be done. This is a recursive tag on the module (need to check with Bijan about this).

We also may want to update this command to automatically do a checkin of the module before doing the submit (need to check with Bijan).

* show\_checkouts MODULE
  + This command will show the checkouts inside that module. If no modules are provided, then all modules in update mode will be scanned. The Sitar.py script can be used as a reference.
* show\_locks MODULE
  + This command will display the locks for a module. If no modules are provided, then all modules in update mode will be scanned. The Sitar.py script can be used as a reference.
* check\_tag TAG MODULE
  + This command will check the TAG for files inside a module. If no modules are provided, then all modules in update mode will be scanned. The Sitar.py script can be used as a reference.
* Integrate [-local]
  + This command will integrate the next versions of modules. This is an existing command in the Sitar.py file, but it will need to be udpated. The script will need to figure out what branch we are on. This can be done with this command (where TOP is the top container):

url tags -btags TOP

If the selector is Trunk:, then only submits that start with REL\_ or are in the form v1.XX should be allowed. For all other branches, only submits that start with the branch name should be allowed. Another change is the command by default should run through bsub unless the -local optin is given.

* release [-comment COMMENT] [-integrate FILE] [-noemail]
  + This command will show the status of the workspace. The Sitar.py script can be used as a reference. This command does not take any modules. But there will be some changes needed. The skipcheck option should be given for the release command. And by default an email report needs to be generated where the details will come from the project.xml file. If the -noemail option is given, then no email should be generated.

If the -integrate option is given, then a file will specify the versions of modules to select then integrate. Once the integration is complete, then the release can be made. If the -integrate option is given, then this command should run through bsub unless the -local switch is given.

* status
  + This command will show the status of the workspace. The Sitar.py script can be used as a reference. This command does not take any modules.
* update [-config VERSION] MODULE
  + May want to call this command update. This command will make the specified module editable in the workspace. If no modules are provided, then all modules will be put into update mode. The Sitar.py script can be used as a reference. We do need a fix for this. When the config is not given, the value is hard-coded to Trunk: But we need to find out the correct config to use. This can be done with this command:

url tags -btags MODULE

* restore MODULE
  + This command will put the module back into cached mode. If no modules are provided, then all modules will be put into mcache mode. The Sitar.py script can be used as a reference.
* vhistory MODULE
  + This command will display the version history for the module. This is something that Trupti was working on. I am not sure if it got completed. Basically the vhistory command is run on all specified modules (or all modules if none are specified). The command would be something like:

vhistory -all -report +TYBV-NLHRLAFE MODULE

* showstatus MODULE
  + This is a new command. If no modules are given, then the top module should be used. To show the status, the following command can be used:

showstatus -rec -objects MODULE

The output will need to be formatted to remove the Identical items

* setup\_ws
  + This command will be used to setup a workspace. It should be called when the workspace is initially created.
* pop\_latest [-n TAG] [-noemail]
  + This is a new command that will be used for the flat release. For this command, the script must populate all of the modules in the workspace.

1. [sitr pop]The first step is to run the populate command to bring in all of the latest updates.

2. [populate\_tag on only modules in update mode]. Then for all modules that are in update mode, the tag must be populated in those modules[populate\_tag on only modules in update mode].

[sitr update module -config]For modules that are in cache mode, a lookup must be done to see if there has been a newer submit than the version that is present in the workspace. If there is a version that is newer than the baseline, then that version should be brought into the workspace.

An email report should be generated and sent out unless the -noemail option is given.

* mk\_lib MODULE LIB1 [LIB2] [LIB3]
  + This is an existing command that is in the Cadence.py scipt. It can be used as a reference. The only update to the script is that it should error out if the MODULE is not in update mode.
* release\_mod [-comment COMMENT] [-noemail] [-local] [-config VERSION] MODULE
  + This is a new command that is used to integrate and release a single module. It can follow a similar flow to the release command, but only one module will change
* mk\_release [-comment COMMENT] [-noemail] [-local] TAG
  + This is a new command that is used to create flat release of a workspace. The script must first get a list of the status of all of the modules in the workspace. For all modules in update mode, the snapshot tag must be name (similar to what is done with the submit option). A list of what modules were submitted with this tag must be stored.

Then for all other modules, the versions of these must be stored. Then the script will create a temporary file that contains all of the versions for all modules, then call the release command in the integrate workspace using this file.

* lookup [-o to a file] MODULE
  + This command will run a sitr lookup operation. The Sitar.py script can be used as a reference. Although there are some updates that are neededThe command should also be updated to be able to filter on the specified module(s).
  + n
* compare [-trunk] [-baseline] [-tag TAG] MODULE

**compare -rec -selector 1.10 CONFIG**

* + This is a new command. It would run a compare with the specified module (or all modules). The baseline version (VERSION) will need to be retreived from the sitr\_mods data. The Trunk selector (TRUNK) will also need to be retrieved using the url btags command. If no options are given (or just the -baseline version), then the compare will be using this command:

compare -rec -selector VERSION MODULE

If the -trunk option only is given, then the command would be:

compare -rec -selector TRUNK MODULE

If the tag option only is given, then the command would be:

compare -rec -selector TAG MODULE

If both -trunk and -baseline are given, then the command would be:

compare -rec -selector TRUNK -selector2 VERSION MODULE

If both the -trunk and -tag options are given, then the command would be:

compare -rec -selector TRUNK -selector2 TAG MODULE

If both the -baseline and -tag options are given, then the command would be:

compare -rec -selector VERSION -selector2 TAG

All three options cannot be given.

.

* compare\_ws PATH [REVISIT LATER}
  + This is a new command. The PATH argument must be a pointer to another workspace. This command would run on all modules in update mode. For each module, the rel\_path (REL\_PATH) must be extracted from the sitr\_mods. Then the following command can be run:

compare -rec -path REL\_PATH PATH/REL\_PATH

* showhrefs MODULE
  + This is a new command. This will recursively show the hrefs in a module. To do this, you can use this command:

showhrefs -format list -rec MODULE

* updatehrefs [-xls FILE]
  + This is similar to the udpate hrefs command that we already have, but we need to get more information about what to do.
* check\_mcache
  + This is a new command which takes no arguments. Need to check with Bijan if there is a better way to do this. It first gets the baseline versions of all of the modules from sitr\_mods. Then for each module, the specific module version needs to be found. To do this, you can use this command:

vhistory -all -format list -report +G-NLHALFEVDC MODULE

The output will map the baseversion to the module version. Once all of the module versions are known, then you can run this command to see what modules are in the mcache:

mcache show -format list

* mk\_mod MODULE
  + [Designsync and sitr module}Make a new sitr module and integrate it into the workspace
* mk\_branch BRANCH
  + Make a branch of the current workspace
* repair\_ws
  + This command will repair a workspace
* jira [-log FILE]
  + Create a JIRA ticket for some WTF command

Priorities:

* High
* Med
* Low
* Future