CI Documentation

Release 0.18.6-python2.7.6

Ivan Keliukh

Sep 04, 2020

Contents

CHAPTER 1

Prerequisites

- OS Linux
- Python version 2.7

1.1 Preinstalled packages

1.1.1 Included to module installation

- `sh` module for Python
- `mechanize` module for Python
- `requests` module for Python

1.2 Optional (used for special VCS types)

1.2.1 Perforce

- p4 CLI (see official installation manual for details)
- P4Python (see official installation manual for details)

1.2.2 Git

- *Git* client (use `sudo apt-get install git`)
- `gitpython` module for Python (use `sudo pip install gitpython`)

1.3 Optional (used for internal tests)

1.3.1 Need to be installed manually

- Docker (Right now only used for internal tests. See official installation manual for details)
- PIP version 9.0 or greater (See official installation manual for details)

1.3.2 Included to module installation

- `pytest` module for Python
- `pylint` module for Python
- `docker` module for Python
- `httpretty` module for Python
- `mock` module for Python
- `sphinx` module for Python
- `sphinx-argparse` extension for Sphinx module

CHAPTER 2

Getting started

Before installing or launching the *Universum*, please make sure your system meets the following *Prerequisites*.

The main script (*universum.py*) is used for performing all CI-related actions. If using raw sources, launch Universum via running this script with *parameters*:

```
$ ./universum.py --help
```

If using a module installed via PyPi, use created `universum` command from any suitable directory:

```
$ universum --help
```

In order to use the CI system with a project, a special `configs.py` file must be created. The contents of such file are described on *Configuring the project* page.

We recommend to place configuration file somewhere inside the project tree. Its location *may also be specified* by the *CONFIG_PATH* environment variable.

CHAPTER 3

Command line

Main script of project *Universum* is `universum.py`. All command-line parameters, general and module-related, are passed to this main script.

Note: Most of command-line parameters can be replaced by setting up corresponding environment variables (see 'env' comment in descriptions)

Universum 0.18.6-python2.7.6

```
usage: universum [-h] [--version] [--build-only-latest] [--no-diff]
                 [--artifact-dir ARTIFACT_DIR] [--no-archive]
                 [--project-root PROJECT_ROOT]
                 [--server-type {tc,jenkins,local}]
                 [--tc-server TEAMCITY_SERVER] [--tc-build-id BUILD_ID]
                 [--tc-configuration-id CONFIGURATION_ID]
                 [--tc-auth-user-id TC_USER] [--tc-auth-passwd TC_PASSWD]
                 [--jenkins-build-url BUILD_URL] [--report-build-start]
                 [--report-build-success] [--report-only-fails]
                 [--report-no-vote] [--out-type {tc,term,jenkins}]
                 [--out {console,file}] [--config CONFIG_PATH]
                 [--filter LAUNCHER.STEP_FILTER] [--report-to-review]
                 [--vcs-type VCS_TYPE] [--git-checkout-id GIT_CHECKOUT_ID]
                 [--git-cherry-pick-id GIT_CHERRYPICK_ID [GIT_CHERRYPICK_ID ...]]
                 [--git-repo GIT_REPO] [--git-refspec GIT_REFSPEC]
                 [--github-token GITHUB_TOKEN]
                 [--github-check-name GITHUB_CHECK_NAME]
                 [--github-check-id GITHUB_CHECK_ID]
                 [--github-api-url GITHUB_API_URL]
                 [--file-source-dir SOURCE_DIR] [--p4-client P4CLIENT]
                 [--p4-sync SYNC_CHANGELIST [SYNC_CHANGELIST ...]]
                 [--p4-shelve SHELVE_CHANGELIST [SHELVE_CHANGELIST ...]]
                 [--p4-force-clean] [--p4-project-depot-path P4_PATH]
                 [--p4-mappings P4_MAPPINGS [P4_MAPPINGS ...]]
                 [--p4-port P4PORT] [--p4-user P4USER]
```

(continues on next page)

(continued from previous page)

```
                [--p4-password P4PASSWD] [--swarm-server-url SWARM_SERVER]
                [--swarm-review-id REVIEW] [--swarm-change SWARM_CHANGELIST]
                [--swarm-pass-link PASS] [--swarm-fail-link FAIL]
                {poll,submit,nonci} ...
```

3.1 Named Arguments

--version Display product name & version instead of launching.

--build-only-latest Skip build if review version isn't latest

Default: False

--no-diff Only applies to build steps where `code_report=True`; disables calculating analysis diff for changed files, in this case full analysis report will be published

Default: False

3.2 Artifact collection

Parameters of archiving and collecting of build artifacts

--artifact-dir, -ad Directory to collect artifacts to. Default is 'artifacts'

Environment variable: $ARTIFACT_DIR

--no-archive By default all directories noted as artifacts are copied as .zip archives. This option turn archiving off to copy bare directories to artifact directory

Default: False

3.3 Source files

Parameters determining the processing of repository files

--project-root, -pr Temporary directory to copy sources to. Default is 'temp'

Environment variable: $PROJECT_ROOT

--report-to-review Perform test build for code review system (e.g. Gerrit or Swarm).

Default: False

--vcs-type, -vt Possible choices: none, p4, git, gerrit, github

Select repository type to download sources from: Perforce ('p4'), Git ('git'), Gerrit ('gerrit'), GitHub ('github') or a local directory ('none'). Gerrit uses Git parameters. Each VCS type has its own settings.

Environment variable: $VCS_TYPE

3.4 Automation server

Automation server options

--server-type, -st Possible choices: tc, jenkins, local

Type of environment to refer to (tc - TeamCity, jenkins - Jenkins, local - user local terminal). TeamCity and Jenkins environment is detected automatically when launched on build agent

3.5 TeamCity variables

TeamCity-specific parameters

--tc-server, -ts TeamCity server URL

Environment variable: $TEAMCITY_SERVER

--tc-build-id, -tbi teamcity.build.id

Environment variable: $BUILD_ID

--tc-configuration-id, -tci system.teamcity.buildType.id

Environment variable: $CONFIGURATION_ID

--tc-auth-user-id, -tcu system.teamcity.auth.userId

Environment variable: $TC_USER

--tc-auth-passwd, -tcp system.teamcity.auth.password

Environment variable: $TC_PASSWD

3.6 Jenkins variables

Jenkins-specific parameters

--jenkins-build-url, -jbu Link to build on Jenkins (automatically set by Jenkins)

Environment variable: $BUILD_URL

3.7 Result reporting

Build results collecting and publishing parameters

--report-build-start, -rst Send additional comment to review system on build started (with link to log)

Default: False

--report-build-success, -rsu Send comment to review system on build success (in addition to vote up)

Default: False

--report-only-fails, -rof Include only the list of failed steps to reporting comments

Default: False

--report-no-vote, -rnv Do not vote up/down review depending on result

Default: False

3.8 Output

Log appearance parameters

--out-type, -ot Possible choices: tc, term, jenkins

Type of output to produce (tc - TeamCity, jenkins - Jenkins, term - terminal). TeamCity environment is detected automatically when launched on build agent.

--out, -o Possible choices: console, file

Define whether to print build logs to console or file. Log file names are generated based on the names of build steps. By default, logs are printed to console when the build is launched on Jenkins or TeamCity agent

3.9 Configuration execution

External command launching and reporting parameters

--config, -cfg Path to project config file (example: -cfg=my/prject/my_conf.py). Mandatory parameter.

Environment variable: $CONFIG_PATH

--filter, -f Allows to filter which steps to execute during launch. String value representing single filter or a set of filters separated by '**:**'. To define exclude pattern use '**!**' symbol at the beginning of the pattern.

A Universum step match specified pattern when 'filter' is a substring of step 'name'. This functionality is similar to 'boosttest' and 'gtest' filtering, except special characters (like '*', '?', etc.) are ignored.

Examples:
* -f='run test' - run only steps that contain 'run test' substring in their names
* -f='!run test' - run all steps except those containing 'run test' substring in their names
* -f='test 1:test 2' - run all steps with 'test 1' OR 'test 2' substring in their names
* -f='test 1:!unit test 1' - run all steps with 'test 1' substring in their names except those containing 'unit test 1'

3.10 Git

--git-checkout-id, -gco A commit ID to checkout. Could be exact commit hash, or branch name, or tag, etc.

Environment variable: $GIT_CHECKOUT_ID

--git-cherry-pick-id, -gcp List of commit IDs to be cherry-picked, separated by comma. '–git-cherry-pick-id' can be added to the command line several times

Environment variable: $GIT_CHERRYPICK_ID

--git-repo, -gr See your project home page for exact repository identifier, passed to 'git clone'. If using SSH, '–git-repo' format is 'ssh://user@server:port/detailed/path'

Environment variable: $GIT_REPO

--git-refspec, -grs Any additional refspec to be fetched

Environment variable: $GIT_REFSPEC

3.11 GitHub

GitHub repository settings

--github-token, -ght GitHub API token; for details see https://developer.github.com/v3/oauth_authorizations/

Environment variable: $GITHUB_TOKEN

--github-check-name, -ghc The name of Github check run

Environment variable: $GITHUB_CHECK_NAME

Default: "Universum check"

--github-check-id, -ghi Github check run ID

Environment variable: $GITHUB_CHECK_ID

--github-api-url, -gha API URL for integration

Environment variable: $GITHUB_API_URL

Default: "https://api.github.com/"

3.12 Local files

Parameters for file settings in case of no VCS used

--file-source-dir, -fsd A local folder for project sources to be copied from. This option is only needed when '–driver-type' is set to 'none'

Environment variable: $SOURCE_DIR

3.13 Perforce

--p4-client, -p4c P4 client (workspace) name to be created. Use '–p4-force-clean' option to delete this client while finalizing

Environment variable: $P4CLIENT

--p4-sync, -p4h Sync (head) CL(s). Just a number will be interpreted as united CL for all added VCS roots. To add a sync CL for specific depot/workspace location, write location in the same format as in P4_MAPPINGS with '@<CL number>' in the end,

e.g. '//DEV/Solution/MyProject/... @1234567'. To specify more than one sync CL for several locations, add '–p4-sync' several times or split them with comma

Environment variable: $SYNC_CHANGELIST

--p4-shelve, -p4s List of shelve CLs to be applied, separated by comma. –p4-shelve can be added to the command line several times. Also shelve CLs can be specified via additional environment variables: SHELVE_CHANGELIST_1..5

Environment variable: $SHELVE_CHANGELIST

--p4-force-clean **Revert all vcs within '–p4-client' and delete the workspace.** Mandatory for CI environment, otherwise use with caution

Default: False

--p4-project-depot-path, -p4d Depot path to get sources from (starts with '//', ends with '/...'Only supports one path. Cannot be used with '–p4-mappings' option

Environment variable: $P4_PATH

--p4-mappings, -p4m P4 mappings. Cannot be used with '–p4-project-depot-path' option. Use the following format: '//depot/path/... /local/path/...', where the right half is the same as in real P4 mappings, but without client name. Just start from client root with one slash. For more than one add several times or split with ',' character

Environment variable: $P4_MAPPINGS

--p4-port, -p4p P4 port (e.g. 'myhost.net:1666')

Environment variable: $P4PORT

--p4-user, -p4u P4 user name

Environment variable: $P4USER

--p4-password, -p4P P4 password

Environment variable: $P4PASSWD

3.14 Swarm

Parameters for performing a test run for pre-commit review

--swarm-server-url, -ssu Swarm server URL; is used for additional interaction such as voting for the review

Environment variable: $SWARM_SERVER

--swarm-review-id, -sre Swarm review number; is sent by Swarm triggering link as '{review}'

Environment variable: $REVIEW

--swarm-change, -sch Swarm change list to unshelve; is sent by Swarm triggering link as '{change}'

Environment variable: $SWARM_CHANGELIST

--swarm-pass-link, -spl Swarm 'success' link; is sent by Swarm triggering link as '{pass}'

Environment variable: $PASS

--swarm-fail-link, -sfl Swarm 'fail' link; is sent by Swarm triggering link as '{fail}'

Environment variable: $FAIL

3.15 Additional commands

{poll,submit,nonci} universum poll
universum submit
universum nonci

CHAPTER 4

Code report

The following analysing modules (analysers) are installed by default: `universum_pylint`, `universum_svace`, `universum_uncrustify`. Analysers are separate scripts, fully compatible with Universum. It is possible to use them independently from command line.

All analysers must have an argument for JSON file with analysis results. If you run code report independently, the name must conform to file name standards. If argument is not provided, output will be written to console.

Running analysers from Universum, you need to add `code_report=True` and result file argument mandatory must be set to `"${CODE_REPORT_FILE}"`. `"${CODE_REPORT_FILE}"` is a pseudo-variable that will be replaced with the file name during execution. Also, you are able not to add `code_report=True` option and name file as you wish, in this case result file won't be processed according to the rules defined for analysers and step will be marked as `Failed` if there are analysis issues found.

Note: When using Universum, if file with analysis results is not added to artifacts, it will be deleted along with other build sources and results.

When using via Universum `code_report=True` step, use `--report-to-review` functionality to comment on any found issues to code review system.

4.1 Pylint

Pylint analyzer

```
usage: universum_pylint [-h] [--files FILE_LIST [FILE_LIST ...]]
                        [--rcfile RCFILE] [--python-version {2,3}]
                        [--result-file RESULT_FILE]
```

4.1.1 Named Arguments

--files Python files and Python packages for Pylint.

--rcfile	Specify a configuration file.
--python-version	Possible choices: 2, 3 Version of Python Default: "3"
--result-file	File for storing json results of Universum run. Set it to "${CODE_REPORT_FILE}" for running from Universum, variable will be handled during run. If you run this script separately from Universum, just name the result file or leave it empty.

Config example for `universum_pylint`:

```
from _universum.configuration_support import Variations

configs = Variations([dict(name="pylint", code_report=True, command=["universum_pylint
↪",
                           "--python-version", "3", "--result-file", "${CODE_REPORT_
↪FILE}",
                           "--files", "*.py", "examples/"])
                      ])

if __name__ == '__main__':
    print configs.dump()
```

This file will get us the following list of configurations:

```
$ ./configs.py
[{'command': 'universum_pylint --python-version 3 --result-file ${CODE_REPORT_FILE} --
↪files *.py examples/', 'name': 'pylint', 'code_report': True}]
```

4.2 Svace

Svace analyzer

```
usage: universum_svace [-h] [--build-cmd BUILD_CMD [BUILD_CMD ...]]
                       [--lang {JAVA,CXX}] [--project-name PROJECT_NAME]
                       [--result-file RESULT_FILE]
```

4.2.1 Named Arguments

--build-cmd	Relative path to build script or command itself
--lang	Possible choices: JAVA, CXX Language to analyze
--project-name	Svace project name defined on server
--result-file	File for storing json results of Universum run. Set it to "${CODE_REPORT_FILE}" for running from Universum, variable will be handled during run. If you run this script separately from Universum, just name the result file or leave it empty.

Config example for `universum_svace`:

```
from _universum.configuration_support import Variations

configs = Variations([dict(name="svace", code_report=True, command=["universum_svace",
                           "--build-cmd", "make", "--lang", "CXX",
                           "--result-file", "${CODE_REPORT_FILE}"])
                      ])

if __name__ == '__main__':
    print configs.dump()
```

will produce this list of configurations:

```
$ ./configs.py
[{'command': 'universum_svace --build-cmd make --lang CXX --result-file ${CODE_REPORT_
↪FILE}', 'name': 'svace', 'code_report': True}]
```

4.3 Uncrustify

Uncrustify analyzer

```
usage: universum_uncrustify [-h] [--files [FILE_NAMES [FILE_NAMES ...]]]
                            [--file-list [FILE_LISTS [FILE_LISTS ...]]]
                            [--cfg-file CFG_FILE]
                            [--filter-regex [PATTERN_FORM [PATTERN_FORM ...]]]
                            [--output-directory OUTPUT_DIRECTORY]
                            [--result-file RESULT_FILE]
```

4.3.1 Named Arguments

--files, -f File or directory to check; accepts multiple values; all files specified by both '–files' and '–file-list' are gathered into one combined list of files

--file-list, -fl Text file with list of files or directories to check; can be used with '–files'; accepts multiple values; all files specified by both '–files' and '–file-list' are gathered into one combined list of files

--cfg-file, -cf Name of the configuration file of Uncrustify; can also be set via 'UNCRUSTIFY_CONFIG' env. variable

--filter-regex, -r (optional) Python 2.7 regular expression filter to apply to combined list of files to check

--output-directory, -od Directory to store fixed files, generated by Uncrustify and HTML files with diff; the default value is 'uncrustify'

--result-file File for storing json results of Universum run. Set it to "${CODE_REPORT_FILE}" for running from Universum, variable will be handled during run. If you run this script separately from Universum, just name the result file or leave it empty.

Config example for `universum_uncrustify`:

```
from _universum.configuration_support import Variations

```

(continues on next page)

(continued from previous page)

```
configs = Variations([dict(name="uncrustify", code_report=True, command=["universum_
↪uncrustify",
                           "--files", "project_root_directory", "--cfg-file", "file_
↪name.cfg",
                           "--filter-regex", ".*//.(?:c|cpp)", "--result-file", "$
↪{CODE_REPORT_FILE}",
                           "--output-directory", "uncrustify"])
                      ])

if __name__ == '__main__':
    print configs.dump()
```

will produce this list of configurations:

```
$ ./configs.py
[{'command': 'universum_uncrustify --files project_root_directory --cfg-file file_
↪name.cfg --filter-regex .*//.(?:c|cpp) --result-file ${CODE_REPORT_FILE} --output-
↪directory uncrustify', 'name': 'uncrustify', 'code_report': True}]
```

CHAPTER 5

Configuring the project

In order to use the *Universum*, the project should provide a configuration file. This file is a regular python script with specific interface, which is recognized by the *Universum*.

The path to the configuration file is supplied to the main script via the *CONFIG_PATH* environment variable or `--launcher-config-path` / `-lcp` command-line parameter. Internally the config file is processed by the `_universum.launcher` module. The path is passed to this module in *config_path* member of its input settings.

Note: Generally there should be no need to implement complex logic in the configuration file, however the *Universum* doesn't limit what project uses its configuration file for. Also, there are no restriction on using of external python modules, libraries or on the structure of the configuration file itself.

The project is free to use whatever it needs in the configuration file for its needs.

5.1 Project configuration

Project configuration (also mentioned as *build configuration*) is a simple complete way to test the project: e.g., build it for some specific platform, or run some specific test script. Basically, *project configuration* is a single launch of some external application or script.

For example:

```
$ ./build.sh -d --platform MSM8996
$ make tests
$ ./run_regression_tests.sh
```

— are three different possible project configurations.

5.2 Minimal project configuration file

Below is an example of the configuration file in its most basic form:

```
from _universum.configuration_support import Variations

configs = Variations([dict(name="Build", command=["build.sh"])])
```

This configuration file uses a *Variations* class from the *_universum.configuration_support* module and defines one build configuration.

Note: Creating a *Variations* instance takes a list of dictionaries as an argument, where every new list member describes a new *project configuration*.

- The *_universum.configuration_support* module provides several functions to be used by project configuration files
- The *Universum* expects project configuration file to define global variable with name *configs*. This variable defines all project configurations to be used in a build.

The minimal project configuration file example defines just one project configuration with the following parameters:

1. **name** is a string *"Build"*
2. **command** is a list with a string item *"build.sh"*

Note: Command line is a list with (so far) one string item, not just a string. Command name and all the following arguments must be passed as a list of separate strings. See *List of configurations* for more details.

5.3 Execution directory

Some scripts (using relative paths or filesystem communication commands like `pwd`) work differently when launching them via `./scripts/run.sh` and via `cd scripts/ && ./run.sh`.

Also, some console applications, such as `make` and `ant`, support setting working directory using special argument. Some other applications lack this support.

That is why it is sometimes necessary, and sometimes just convenient to launch the configuration *command* in a directory other then project root. This can be easily done using *directory* keyword:

```
from _universum.configuration_support import Variations

configs = Variations([dict(name="Make Special Module", directory="specialModule",␣
↪command=["make"])])
```

To use a *Makefile* located in *"specialModule"* directory without passing "-C SpecialModule/" arguments to `make` command, the launch directory is specified.

5.3.1 get_project_root()

By default for any launched external command *current directory* is the actual directory containing project files. So any internal relative paths for the project should not cause any troubles. But when, for any reason, there's a need to refer

to project location absolute path, it is recommended to use *get_project_root()* function from *_universum.configuration_support* module.

Note: The *Universum* launches in its own working directory that may be changed for every run and therefore cannot be hardcoded in *configs.py* file. Also, if not stated otherwise, project sources are copied to a temporary directory that will be deleted after a run. This directory may be created in different places depending on various *Universum* settings (not only the *working directory*, mentioned above), so the path to this directory can not be hardcoded too.

The *_universum.configuration_support* module processes current *Universum* run settings and returns actual project root to the config processing module.

See the following example configuration file:

```
from _universum.configuration_support import Variations, get_project_root

configs = Variations([dict(name="Run tests", directory="/home/scripts", command=["./
↪run_tests.sh", "--directory", get_project_root()])])
```

In this configuration a hypothetical external script *"run_tests.sh"* requires absolute path to project sources as an argument. The *get_project_root()* will pass the actual project root, no matter where the sources are located on this run.

5.4 List of configurations

The *Universum* gets the list of project configurations from the *configs* global variable. In the basic form this variable contains a flat list of items, and each item represents one *project configuration*.

Below is an example of the configuration file with three different configurations:

```
from _universum.configuration_support import Variations, get_project_root
import os.path

test_path = os.path.join(get_project_root(), "out/tests")
configs = Variations([dict(name="Make Special Module", command=["make", "-C",
↪"SpecialModule/"], artifacts="out"),
                      dict(name="Run internal tests", command=["scripts/run_tests.sh
↪"]),
                      dict(name="Run external tests", directory="/home/scripts",␣
↪command=["run_tests.sh", "-d", test_path])
                      ])
```

The example configuration file declares the following *Universum* run steps:

1. Make a module, located in *"specialModule"* directory
2. Run a *"run_tests.sh"* script, located in *"scripts"* directory
3. Run a *"run_tests.sh"* script, located in external directory *"home/scripts"* and pass an absolute path to a directory *"out/tests"* inside project location
4. Copy artifact directory *"out"* to the working directory

Note: Concatenating *get_project_root()* results with any other paths is recommended using `os.path.join()` function to avoid any possible errors on path joining.

5.5 Common *Variations* keys

Each configuration description is a python dictionary with the following possible keys:

name Specifies the name of the configuration. The name is used as the title of the build log block corresponding to the build of this configuration. It is also used to generate name of the log file if the option for storing to log files is selected. A project configuration can have no name, but it is not recommended for aesthetic reasons. If several project configurations have the same names, and logs are stored to files (see `--launcher-output` / `-lo` command-line parameter for details), all logs for such configurations will be stored to one file in order of their appearances.

command The list with command line for the configuration launch. For every project configuration first list item should be a console command (e.g. script name, or any other command like `ls`), and other list items should be the arguments, added to the command (e.g. `--debug` or `-la`). Every command line element, separated with space character, should be passed as a separate string argument. Lists like `["ls -a"]` will not be processed correctly and thus should be splat into `["ls", "-a"]`. Lists like `["build.sh", "--platform A"]` will not be processed correctly and thus should be plat into `["build.sh", "--platform", "A"]`. A project configuration can have an empty list as a command. Such configuration won't do anything except informing user about missing it.

environment Python dictionary of required environment variables, e.g. `environment={"VAR1": "String", "VAR2": "123"}` Can be set at any step level, but re-declaring variables is not supported, so please make sure to mention every variable only one time at most.

artifacts Path to the file or directory to be copied to the working directory as an execution result. Can contain shell-style pattern matching (e.g. *"out/*.html"*), including recursive wildcards (e.g. *"out/**/index.html"*). If not stated otherwise (see `--no-archive` command-line parameter for details), artifact directories are copied as archives. If *'artifact_prebuild_clean' key* is either absent or set to *False* and stated artifacts are present in downloaded sources, it is considered a failure and configuration execution will not proceed. If no required artifacts were found in the end of the *Universum* run, it is also considered a failure. In case of shell-style patterns build is failed if no files or directories matching pattern are found. Any project configuration may or may not have any artifacts.

report_artifacts Special artifacts for reporting (e.g. to Swarm). A separate link to each of such artifacts will be added to the report. Unlike *artifacts*, *report_artifacts* are not obligatory and their absence is not considered a build failure. A directory cannot be stored as a separate artifact, so when using `--no-archive` option, do not claim directories as *report_artifacts*. Please note that any file can be claimed as *artifact*, *report_artifact*, or both. A file, claimed both in *artifacts* and *report_artifacts*, will be mentioned in a report and will cause build failure when missing.

artifact_prebuild_clean Basic usage is adding `artifact_prebuild_clean=True` to configuration description. By default artifacts are not stored in VCS, and artifact presence before build most likely means that working directory is not cleaned after previous build and therefore might influence build results. But sometimes deliverables have to be stored in VCS, and in this case instead of stopping the build they should be simply cleaned before it. This is where `artifact_prebuild_clean=True` key is supposed to be used. If set without any *artifacts* or *report_artifacts*, this key will be ignored.

directory Path to a current working directory for launched process. Please see the *Execution directory* section for details. No *directory* is equal to empty string passed as *directory* and means the *command* will be launched from project root directory.

critical Basic usage is adding `critical=True` to configuration description. This parameter is used in case of a linear step execution, when the result of some step is critical for the subsequent step execution. If some configuration has *critical* key set to *True* and executing this step fails, no more configurations will be executed during this run. However, all already started *background* steps will be finished regardless critical step results.

background Basic usage is adding `background=True` to configuration description. This parameter means that configuration should be executed independently in parallel with all other steps. All logs from such steps are

written to file, and the results of execution are collected in the end of *Universum* run. Next step execution begins immediately after starting a background step, not waiting for it to be completed. Several background steps can be executed simultaneously.

finish_background Basic usage is adding `finish_background=True` to configuration description. This parameter means that before executing this particular step all ongoing background steps (if any) should be finished first.

code_report Basic usage is adding `code_report=True` to configuration description and `--result-file="${CODE_REPORT_FILE}"` to 'command' arguments. Specifies step that performs static or syntax analysis of code. Analyzers currently provided by Universum: `universum_pylint`, `universum_svace` and `universum_uncrustify` (see code_report parameter for details).

pass_tag, fail_tag Basic usage is adding `pass_tag="PASS", fail_tag="FAIL"` to the configuration description. These keys is implemented only for TeamCity build. You can specify one, both or neither of them per step. Defining `pass_tag="PASS"` means that current build on TeamCity will be tagged with label `PASS` if this particular step succeeds. Defining `fail_tag="FAIL"` means that current build on TeamCity will be tagged with label `FAIL` if this particular step fails. Key values can be set to any strings acceptable by TeamCity as tags. It is not recommended to separate words in the tag with spaces, since you cannot create a tag with spaces in TeamCity's web-interface. Every tag is added (if matching condition) after executing build step it is set in, not in the end of all run. `pass_tag` and `fail_tag` can also be used in configurations multiplications, like this:

```
make = Variations([dict(name="Make ", command=["make"], pass_tag="pass_")])

target = Variations([dict(name="Linux", command=["--platform", "Linux"], pass_tag=
↪"Linux"),
                     dict(name="Windows", command=["--platform", "Windows"], pass_
↪tag="Windows")
                     ])

configs = make * target
```

This code will produce this list of configurations:

```
$ ./configs.py
[{'command': 'make --platform Linux', 'name': 'Make Linux', 'pass_tag': 'pass_
↪Linux'},
{'command': 'make --platform Windows', 'name': 'Make Windows', 'pass_tag': 'pass_
↪Windows'}]
```

This means that tags "pass_Linux" and "pass_Windows" will be sent to TeamCity's build.

Note: All the paths, specified in *command*, *artifacts* and *directory* parameters, can be absolute or relative. All relative paths start from the project root (see *get_project_root()*).

5.6 Dump configurations list

Class *Variations* have a build-in function *dump()*, that processes the passed dictionaries and returns the list of all included configurations.

Below is an example of the configuration file that uses *dump()* function for debugging:

```
#!/usr/bin/env python

from _universum.configuration_support import Variations, get_project_root
import os.path

test_path = os.path.join(get_project_root(), "out/tests")
configs = Variations([dict(name="Make Special Module", command=["make", "-C",
↪"SpecialModule/"], artifacts="out"),
                      dict(name="Run internal tests", command=["scripts/run_tests.sh
↪"]),
                      dict(name="Run external tests", directory="/home/scripts",␣
↪command=["run_tests.sh", "-d", test_path])
                      ])

if __name__ == '__main__':
    print configs.dump()
```

The combination of `#!/usr/bin/env python` and `if __name__ == '__main__':` allows launching the *configs.py* script from shell.

For `from _universum.configuration_support import` to work correctly, *configs.py* should be copied to *Universum* root directory and launched there.

When launched from shell instead of being used by *Universum* system, *get_project_root()* function returns current directory instead of actual project root.

The only thing this script will do is create *configs* variable and print all project configurations it includes. For example, running the script, given above, will result in the following:

```
$ ./configs.py
[{'artifacts': 'out', 'command': 'make -C SpecialModule/', 'name': 'Make Special␣
↪Module'},
{'command': 'scripts/run_tests.sh', 'name': 'Run internal tests'},
{'directory': '/home/scripts', 'command': 'run_tests.sh -d /home/Project/out/tests',
↪'name': 'Run external tests'}]
```

As second and third build configurations have the same names, if log files are created, only two logs will be created: one for the first build step, another for both second and third, where the third will follow the second.

5.7 Combining configurations

The Variations class provides a way to generate a full testing scenario by simulating the combination of configurations.

For this class *Variations* has built-in `+` and `*` operators that allow creating configuration sets out of several *Variations* instances.

5.7.1 Adding build configurations

See the following example:

```
#!/usr/bin/env python

from _universum.configuration_support import Variations

one = Variations([dict(name="Make project", command=["make"])])
```

(continues on next page)

(continued from previous page)

```
two = Variations([dict(name="Run tests", command=["run_tests.sh"])])

configs = one + two

if __name__ == '__main__':
    print configs.dump()
```

The addition operator will just concatenate two lists into one, so the *result* of such configuration file will be the following list of configurations:

```
$ ./configs.py
[{'command': 'make', 'name': 'Make project'},
{'command': 'run_tests.sh', 'name': 'Run tests'}]
```

5.7.2 Multiplying build configurations

Multiplication operator can be used in configuration file two ways:

1. multiplying configuration by a constant
2. multiplying configuration by another configuration

Multiplying configuration by a constant is just an equivalent of multiple additions:

```
>>> run = Variations([dict(name="Run tests", command=["run_tests.sh"])])
>>> print (run * 2 == run + run)
True
```

Though the application area of multiplication by a constant is unclear at the moment.

Multiplying configuration by a configuration combines their properties. For example, this configuration file:

```
#!/usr/bin/env python

from _universum.configuration_support import Variations

make = Variations([dict(name="Make ", command=["make"], artifacts="out")])

target = Variations([dict(name="Platform A", command=["--platform", "A"]),
                     dict(name="Platform B", command=["--platform", "B"])
                     ])

configs = make * target

if __name__ == '__main__':
    print configs.dump()
```

will *produce* this list of configurations:

```
$ ./configs.py
[{'artifacts': 'out', 'command': 'make --platform A', 'name': 'Make Platform A'},
{'artifacts': 'out', 'command': 'make --platform B', 'name': 'Make Platform B'}]
```

- *command* and *name* values are produced of *command* and *name* values of each of two configurations
- *artifacts* value, united with no corresponding value in second configuration, remains unchanged

Note: Note the extra space character at the end of the configuration name *"Make "*. As multiplying process uses simple adding of all corresponding configuration settings, string variables are just concatenated, so without extra spaces resulting name would look like "MakePlatform A". If we add space character, the resulting name becomes "Make Platform A".

5.7.3 Combination of addition and multiplication

When creating a project configuration file, the two available operators, + and $\star$, can be combined in any required way. For example:

```
#!/usr/bin/env python

from _universum.configuration_support import Variations

make = Variations([dict(name="Make ", command=["make"], artifacts="out")])
test = Variations([dict(name="Run tests for ", directory="/home/scripts", command=[
↪"run_tests.sh", "--all"])])

debug = Variations([dict(name=" - Release"),
                    dict(name=" - Debug", command=["-d"])
                    ])

target = Variations([dict(name="Platform A", command=["--platform", "A"]),
                     dict(name="Platform B", command=["--platform", "B"])
                     ])

configs = make * target + test * target * debug

if __name__ == '__main__':
    print configs.dump()
```

This file *will get us* the following list of configurations:

```
$ ./configs.py
[{'artifacts': 'out', 'command': 'make --platform A', 'name': 'Make Platform A'},
{'artifacts': 'out', 'command': 'make --platform B', 'name': 'Make Platform B'},
{'directory': '/home/scripts', 'command': 'run_tests.sh --all --platform A', 'name':
↪'Run tests for Platform A - Release'},
{'directory': '/home/scripts', 'command': 'run_tests.sh --all --platform A -d', 'name
↪': 'Run tests for Platform A - Debug'},
{'directory': '/home/scripts', 'command': 'run_tests.sh --all --platform B', 'name':
↪'Run tests for Platform B - Release'},
{'directory': '/home/scripts', 'command': 'run_tests.sh --all --platform B -d', 'name
↪': 'Run tests for Platform B - Debug'}]
```

As in common arithmetic, multiplication is done before addition. To change the operations order, use parentheses:

```
>>> configs = (make + test * debug) * target
```

5.8 Excluding configurations

At the moment there is no support for − operator. There is no easy way to exclude one of configurations, generated by adding/multiplying. But there is a conditional including implemented. To include/exclude configuration depending on environment variable, use *if_env_set* key. When script comes to executing a configuration with such key, if there's no environment variable with stated name set to either "true", "yes" or "y", configuration is not executed. If any other value should be set, use `if_env_set="VARIABLE_NAME == variable_value"` comparison. Please pay special attention on the absence of any quotation marks around *variable_value*: if added, *$VARIABLE_NAME* will be compared with *"variable_value"* string and thus fail. Also, please note, that all spaces before and after *variable_value* will be automatically removed, so `if_env_set="VARIABLE_NAME == variable_value "` will be equal to `os.environ["VARIABLE_NAME"] = "variable_value"` but not `os.environ["VARIABLE_NAME"] = "variable_value "`.

$VARIABLE_NAME consist solely of letters, digits, and the '_' (underscore) and not begin with a digit.

If such environment variable should not be set to specific value, please use `if_env_set="VARIABLE_NAME != variable_value"` (especially `!= True` for variables to not be set at all).

If executing the configuration depends on more than one environment variable, use `&&` inside *if_env_set* value. For example, `if_env_set="SPECIAL_TOOL_PATH && ADDITIONAL_SOURCES_ROOT"` configuration will be executed only in case of both *$SPECIAL_TOOL_PATH* and *$ADDITIONAL_SOURCES_ROOT* environment variables set to some values. If any of them is missing or not set in current environment, the configuration will be excluded from current run.

CHAPTER 6

'configuration_support' module

See also:

Please see *Configuring the project* for more examples and detailed explanation

6.1 _universum.configuration_support

combine(*dictionary_a*, *dictionary_b*)

Combine two dictionaries using plus operator for matching keys

Parameters

- **dictionary_a** – may have any keys and values
- **dictionary_b** – may have any keys, but the values of keys, matching *dictionary_a*, should be the same type

Returns new dictionary containing all keys from both *dictionary_a* and *dictionary_b*; for each matching key the value in resulting dictionary is a sum of two corresponding values

For example:

```
>>> combine(dict(attr_a = "a1", attr_b = ["b11", "b12"]), dict(attr_a = "a2",␣
↪attr_b = ["b2"]))
{'attr_b': ['b11', 'b12', 'b2'], 'attr_a': 'a1a2'}
```

```
>>> combine(dict(attr_a = {"a1": "v1"}, attr_b = {"b1": "v1"}), dict(attr_a = {"a2
↪": "v2"}))
{'attr_b': {'b1': 'v1'}, 'attr_a': {'a1': 'v1', 'a2': 'v2'}}
```

class Variations

Variations is a class for establishing project configurations. This class object is a list of dictionaries:

```
>>> v1 = Variations([{"field1": "string"}])
>>> v1
[{'field1': 'string'}]
```

Build-in method all() generates iterable for all configuration dictionaries for further usage:

```
>>> for i in v1.all(): i
{'field1': 'string'}
```

Built-in method dump() will generate a printable string representation of the object. This string will be printed into console output

```
>>> v1.dump()
"[{'field1': 'string'}]"
```

Adding two objects will extend list of dictionaries:

```
>>> v2 = Variations([{"field1": "line"}])
>>> for i in (v1 + v2).all(): i
{'field1': 'string'}
{'field1': 'line'}
```

While multiplying them will combine same fields of dictionaries:

```
>>> for i in (v1 * v2).all(): i
{'field1': 'stringline'}
```

When a field value is a list itself -

```
>>> v3 = Variations([dict(field2=["string"])])
>>> v4 = Variations([dict(field2=["line"])])
```

multiplying them will extend the inner list:

```
>>> for i in (v3 * v4).all(): i
{'field2': ['string', 'line']}
```

__add__(*other*)

This functions defines operator + for *Variations* class objects by concatenating lists of dictionaries into one list. The order of list members in resulting list is preserved: first all dictionaries from *self*, then all dictionaries from *other*.

Parameters **other** – *Variations* object to be added to *self*

Returns new *Variations* object, including all configurations from both *self* and *other* objects

__mul__(*other*)

This functions defines operator * for *Variations* class objects. The resulting object is created by combining every *self* list member with every *other* list member using *combine()* function.

Parameters **other** – *Variations* object to be multiplied to *self*

Returns new *Variations* object, consisting of the list of combined configurations

all()

Function for configuration iterating.

Returns iterable for all dictionary objects in *Variations* list

dump(*produce_string_command=True*)
Function for *Variations* objects pretty printing.

Parameters **produce_string_command** – if set to False, prints "command" as list instead of string

Returns a user-friendly string representation of all configurations list

filter(*checker*, *parent=None*)
This function is supposed to be called from main script, not configuration file. It uses provided *checker* to find all the configurations that pass the check, removing those not matching conditions.

Parameters

- **checker** – a function that returns *True* if configuration passes the filter and *False* otherwise
- **parent** – an inner parameter for recursive usage; should be None when function is called from outside

Returns new *Variations* object without configurations not matching *checker* conditions

set_project_root(*project_root*)
Function to be called from main script; not supposed to be used in configuration file. Stores generated project location for further usage. This function is needed because project sources most likely will be copied to a temporary directory of some automatically generated location and the CI run will be performed there.

Parameters **project_root** – path to actual project root

get_project_root()
Function to be used in configuration file. Inserts actual project location after that location is generated. If project root is not set, function returns current directory.

Returns actual project root

CHAPTER 7

Other modules documentation

7.1 _universum.modules.launcher

check_if_env_set(*configuration*)

Predicate function for *_universum.configuration_support.Variations.filter()*, used to decide whether this particular configuration should be executed in this particular environment. For more information see *Excluding configurations*

```
>>> from _universum.configuration_support import Variations
>>> c = Variations([dict(if_env_set="MY_VAR != some value")])
>>> check_if_env_set(c[0])
True
```

```
>>> c = Variations([dict(if_env_set="MY_VAR != some value && OTHER_VAR")])
>>> check_if_env_set(c[0])
False
```

```
>>> c = Variations([dict(if_env_set="MY_VAR == some value")])
>>> os.environ["MY_VAR"] = "some value"
>>> check_if_env_set(c[0])
True
```

Parameters **configuration** – *Variations* object containing one leaf configuration

Returns True if environment satisfies described requirements; False otherwise

CHAPTER 8

Integration with TeamCity

The main design goal of Universum is to *integrate with various CI systems*. E.g. continuous integration/automation, version control, static analysis, testing, etc.

One of popular continuous integration systems is TeamCity, and this particular tutorial will explain how to integrate it with Universum.

The proposed scenario includes the following steps:

0. Install and configure both TeamCity and Universum systems, prepare TeamCity build servers for project building/running/testing/etc.
1. Create a project on TeamCity. Configure common parameters for the project:
   a) project parameters, if any
   b) parameters for Universum

   TeamCity configurations automatically inherit all project settings, so configuring them once on project-level allows to avoid multiple reconfiguring of same parameters in each configuration, and all changes applied to project settings are automatically applied to all inherited settings
2. Create a common TeamCity meta-runner. TeamCity offers two ways of using meta-runners:
   a) creating a new configuration out of existing meta-runner
   b) using an existing meta-runner as a build step

   Both of this scenarios have their own benefits: when creating a configuration, all parameters are inherited too; but when using a build step, any change in meta-runner automatically affects all configurations using it
3. Create all needed configurations, such as:
   - precommit check
   - postcommit check
   - testing
   - etc.

This scenario pays a lot of attention to reusing settings instead of just copying them. It is most important in cases when some of these settings have to be changed: if generalized on project level, there's only one place to fix. And if all similar settings are duplicated, tracking changes becomes more difficult the more times they are copied.

8.1 Install Universum on build agents

0. Download Universum sources to the build server
1. Install Universum on build server by running `sudo pip install .` in Universum sources root directory
2. Install and configure TeamCity server
3. Install build agents on build server, add them to the project pool

Please refer to TeamCity official manuals for details.

8.2 Create a top-level project

0. Create new project for all Universum configurations or add a sub-project to an existing one
1. Go to *Parameters* section in Project Settings and add the following parameters:

   env.BUILD_ID `%teamcity.build.id%`

   env.CONFIGURATION_ID `%system.teamcity.buildType.id%`

   env.TEAMCITY_SERVER server URL to refer to in reports

   env.CONFIGURATION_PARAMETERS will be used in meta-runner; leave empty so far

Making all projects using Universum sub-projects to this one will automatically add all these parameters to their settings.

8.3 Create a common meta-runner

0. Create an .xml file with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<meta-runner name="Run build using CI system">
  <description>Basic project configuration</description>
  <settings>
    <build-runners>
      <runner name="Download and run" type="simpleRunner">
        <parameters>
          <param name="script.content"><![CDATA[
#!/bin/bash

EXITCODE=0

HOST=`hostame | sed -e "s/_/-/"`
USER=`whoami | sed -e "s/_/-/"`
P4CLIENT="Disposable_workspace_"$HOST"-"$USER

cmd="universum --p4-client ${P4CLIENT} --p4-force-clean %env.CONFIGURATION_
↪PARAMETERS%"
```

(continues on next page)

(continued from previous page)

```
echo "==> Run: ${cmd}"
${cmd} || EXITCODE=1

echo "##teamcity[setParameter name='STOPPED_BY_USER' value='false']"

exit $EXITCODE]]></param>
          <param name="teamcity.step.mode" value="default" />
          <param name="use.custom.script" value="true" />
        </parameters>
      </runner>
      <runner name="Clean" type="simpleRunner">
        <parameters>
          <param name="script.content"><![CDATA[
#!/bin/bash

if [ %STOPPED_BY_USER% == true ]
then
echo "==> User interrupted, force cleaning"

EXITCODE=0

HOST=`hostame | sed -e "s/_/-/"`
USER=`whoami | sed -e "s/_/-/"`
P4CLIENT="Disposable_workspace_"$HOST"-"$USER

cmd="python -u ./universum.py --p4-client ${P4CLIENT} --p4-force-clean %env.
↪CONFIGURATION_PARAMETERS% --finalize-only --artifact-dir finalization_artifacts"
echo "==> Run: ${cmd}"
${cmd}

else
echo "==> Additional cleaning not needed, skipping"
fi
          ]]></param>
          <param name="teamcity.step.mode" value="execute_always" />
          <param name="use.custom.script" value="true" />
        </parameters>
      </runner>
    </build-runners>
  </settings>
</meta-runner>
```

Note: Universum default VCS type is Perforce, so this meta-runner is oriented to be used with P4. But the same meta-runner can be used for configurations using any other VCS type. Unused P4 parameters will be just ignored.

1. In *Project Settings* find *Meta-Runners* page and press `Upload Meta-Runner`
2. Select your newly created .xml file as a *Meta-Runner file*

8.4 Configure project using Perforce

0. Create a sub-project to a created earlier top-level project
1. Go to *Parameters* in *Project Settings*

2. Add `env.CONFIG_PATH`: a relative path to project *configuration file*, starting from project root
3. Also add all required project-wide Perforce parameters:

   env.P4USER Perforce user ID

   env.P4PASSWD user <env.P4USER> password

   env.P4PORT Perforce server URL (including port if needed)

   env.P4_MAPPINGS Perforce mappings in *special format*. Also can be replaced with legacy `env.P4_PATH` (but not both at a time)

8.5 Create basic postcommit configuration

0. After creating new build configuration, go to *Build Configuration Settings*
1. To get artifacts from default artifact directory, go to *General Settings*, find *Artifact paths* field and add `artifacts/*` line there
2. To trigger builds via TeamCity but download via Universum, go to *Version Control Settings*, attach required VCS Root and set *VCS checkout mode* to `Do not checkout files automatically`
3. Go to *Triggers* and add *VCS Trigger* with required settings
4. Go to *Build steps*, press `Add build step`, in *Runner type* scroll down to your project runners and select a meta-runner created earlier

After setting up all the environment variables right, you must get the fully working configuration.

8.6 Create configuration for custom builds

0. As in postcommit, specify `artifacts/*` in *Artifact paths* and add your meta-runner as a *Build step*
1. Attaching *VCS root* is not necessary because custom build configurations usually do not use *VCS Trigger*; instead of this, add the following parameters to configuration:

   env.SYNC_CHANGELIST can be a CL number or a list of sync CLs for several different *P4_MAPPINGS*, see *'–p4-sync' option description*

   env.SHELVE_CHANGELIST one or several coma-separated CLs to unshelve for the build

8.7 Integrate with Swarm

0. Go to *Build Configuration Settings* (or to *Project Settings*, if you plan on having more than one Swarm-related configuration)
1. Create `env.REVIEW`, `env.PASS` and `env.FAIL` parameters and leave them empty
2. In *Build Configuration Settings* –> *Parameters* and add `--report-to-review` option in `env.CONFIGURATION_PARAMETERS`
3. If needed, add other *Swarm options*, such as `--report-build-start` and `--report-build-success`
4. Go to Swarm project settings, check in *Automated tests* check-box and follow this instruction

The resulting URL you should insert in text field. The URL should look like:

http://<user>:<password>@<TeamCity URL>/httpAuth/action.html?add2Queue=<configuration>
&name=env.SHELVE_CHANGELIST&value={change}&name=env.PASS&value={pass}&name=env.FAIL&value={fail}
&name=env.REVIEW&value={review}

where

user is a name of a TeamCity user triggering Swarm builds (preferably some bot)

password is that user's password

TeamCity URL is actual server URL, including port if needed

configuration is an ID of your Swarm configuration (see `Build configuration ID` in settings)

or, if your TeamCity supports anonymous build triggering, *user & password* can be omitted along with `httpAuth/` parameter.

1. Probably, in the *POST Body* field you should additionally insert below line:

   name=STATUS&value={status}

or, any other parameter. This is a workaround for TeamCity requirement for using POST method to trigger builds.

8.8 Configure project and configurations using Git

0. Create a sub-project to a top-level project for Universum configurations
1. In *Parameters* set `env.CONFIG_PATH` relative to project root
2. Add oject-wide Git parameters:

   env.GIT_REPO a parameter to pass to `git clone`, e.g. `ssh://user@server/project-name/`

   env.GIT_REFSPEC if some non-default git refspec is needed for project, specify it here
3. Create post-commit configurations *as described above*
4. When creating custom build configurations, use the following parameters instead of P4-specific:

   env.GIT_CHECKOUT_ID parameter to be passed to `git checkout`; can be commit hash, branch name, tag, etc. (see official manual for details)

   env.GIT_CHERRYPICK_ID one or several coma-separated commit IDs to cherry-pick (see official manual for details)

CHAPTER 9

Other usage examples

This page contains some examples of Python scripts using Universum.

9.1 Export Perforce repository to Git

Here's an example script that ports certain directory with commit history from Helix Perforce (P4) repository to Git. It sequentially reproduces directory state for each submitted CL, and submits this repository state to Git using the same commit description.

Note: To port the whole repository you just have to set Perforce source directory and Git destination directory to repository root

9.1.1 Command line description

--p4-port	Source Perforce repository; format equals to Perforce *P4PORT* environment variable and usually looks like `example.com:1666`
--p4-user	Perforce account used to download (sync) files from Perforce server
--p4-password	Corresponding password for *–p4-user*
--p4-client	Name of a temporary Perforce client (workspace) to be created automatically; this client will be deleted after the script finishes its work
--p4-depot-path	Particular folder in source Perforce repository to be ported; should be provided in Perforce-specific format, e.g. `//depot/path/...`
--p4-client-root	Local folder, where the source commits will be downloaded to; should be absolute path; should be anywhere inside destination Git repository
--git-repo	Destination Git repository; should be absolute path; repository should already exist

--git-user	Git account used to commit ported changes
--git-email	Mandatory Git parameter for committer; should correspond to *–git-user*

Note: `--p4-client-root` should not necessarily equal `--git-repo`; it just has to be somewhere inside repository

9.1.2 Preconditions

- Git repository already exists. It can be cloned from remote or created via `git init`
- Git account used for porting is authorized to commit
- Perforce account used for porting is authorized to download (sync) source folder
- Perforce client does not exist

9.1.3 Script

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

import argparse
from P4 import P4
import universum


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--p4-port")
    parser.add_argument("--p4-user")
    parser.add_argument("--p4-password")
    parser.add_argument("--p4-client")
    parser.add_argument("--p4-depot-path")
    parser.add_argument("--p4-client-root")
    parser.add_argument("--git-repo")
    parser.add_argument("--git-user")
    parser.add_argument("--git-email")
    args = parser.parse_args()

    p4 = P4()

    p4.port = args.p4_port
    p4.user = args.p4_user
    p4.password = args.p4_password
    client_name = args.p4_client

    p4.connect()

    depot_path = args.p4_depot_path

    client = p4.fetch_client(client_name)
    client["Root"] = args.p4_client_root
    client["View"] = [depot_path + " //" + client_name + "/..."]
    p4.save_client(client)
```

(continues on next page)

(continued from previous page)

```
37      p4.client = client_name
38
39      changes = p4.run_changes("-s", "submitted", depot_path)
40      cl_list = []
41      for change in changes:
42          cl_list.append(change["change"])
43
44      for cl in reversed(cl_list):
45          line = depot_path + '@' + cl
46          p4.run_sync("-f", line)
47
48          universum.main(["submit",
49                          "-ot", "term",
50                          "-vt", "git",
51                          "-cm", p4.run_describe(cl)[0]['desc'],
52                          "-gu", args.git_user,
53                          "-ge", args.git_email,
54                          "-pr", args.git_repo,
55                          "-gr", "file://" + args.git_repo,
56                          "-grs", "master"])
57
58      p4.delete_client(client_name)
59
60
61  if __name__ == "__main__":
62      main()
```

9.1.4 Possible script modifications

In this example commit messages are preserved, but all changes are committed to Git from one user. To port commit users as well use `p4.run_describe(cl)[0]['user']` to find P4 user and replace incoming parameters `--git-user`, `--git-email` with mapping of P4 users into Git user parameters (`-gu`, `-ge`) that are passed to submitter. See lines 52-53 for the reference.

Also this script only can process the contents of one P4 folder, creating a single mapping for it in `client["View"]`. To use multiple mappings, edit `client["View"]` accordingly instead of parsing `--depot-path` parameter. See line 35 for the reference.

CHAPTER 10

Change log

10.1 0.18.6 (2020-04-27)

10.1.1 Bug fixes

- **p4: fix fix the bug with failing workspace cleanup on attempt to revert entire workspace,** because it requires admin access to the perforce server. The buggy code was introduced by the fix of the issue with reverting files, when there is no file system access to them.

10.2 0.18.5 (2020-04-24)

10.2.1 New features

- **submit:** create and delete real CL to not interfere with any changes in default CL

10.2.2 Bug fixes

- **p4: do not try to revert local files as they can be no longer accessible for write** to avoid creation of undeletable CLs and workspaces
- **launcher:** fix unsuccessful step launch in Ubuntu 14.04 (Python 2.7.6)

10.3 0.18.4 (2020-04-06)

10.3.1 Bug fixes

- **p4:** force clean not deleting CLs leading to buid failures when client exists and contains CLs

- **jenkins_plugin:** steps coloring not working when not using Jenkins Pipeline
- **docs:** update command line arguments in docs to correspond to real ones

10.4 0.18.3 (2020-01-10)

10.4.1 New features

- **launcher command line arguments renamed**
  - see *'Output' and 'Configuration execution' arguments* for new options list
  - old options *–launcher-config-path* and *–launcher-output* are still supported but not recommended to use any more
- **launcher:** add '–filter' option filtering steps to be executed
- **nonci:** add nonci mode of running Universum
- **jenkins_plugin:** expand failed steps by default
- **test:** add Java & JS tests for Jenkins plugin

10.4.2 Bug fixes

- **jenkins_plugin:** fix collapsing with timestamps plugin usage

10.5 0.18.2 (2019-10-09)

10.5.1 New features

- **vcs:** add 'github' VCS type
- **vcs:** implement GitHub as code review system
- **github:** add inline comments for code_report
- **jenkins_plugin:** add jenkins plugin for Universum logs pretty printing
- **test:** clean environment for tests

10.5.2 Bug fixes

- **handle SIGTERM properly**
- **p4:** ignore only expected exceptions on file revert
- **test:** single poll fails because httpretty conflicts with docker-py
- **test:** whitespaces in local paths

10.6 0.18.1 (2019-07-23)

10.6.1 New features

- **out:** add Universum version as log identifier for Jenkins plugin

10.6.2 Bug fixes

- **artifacts:** rewrite 'make_archive' to use ZIP64 extensions
- **tests:** fails if VCS is set globally via env
- **out:** remove old Jenkins block labels because of upcomming plugin update

10.7 0.18.0 (2019-05-28)

10.7.1 BREAKING CHANGES

- **remove setting default VCS type to p4.** *–vcs-type* is now a required option

10.7.2 New features

- **docs:** restructure documentation, switch README to Markdown
- **docs:** add logo, favicon and community docs
- **docs:** add example P4-to-Git porting script
- **args:** VCS type can now be defined via environment variable

10.7.3 Bug fixes

- **incorrect checks of parameters**
- **argument error message for subcommands**
- **docs:** reference to *artifact_prebuild_clean*
- **submit:** git module returns error if there are no files
- **p4:** no error on sync if depot is empty
- **git:** bug with unicode on newer GitPython

10.8 0.17.0 (2019-02-01)

10.8.1 New features

- **api:** add 'file-diff' for Git & Gerrit

10.8.2 Bug fixes

- **code_report:** fixed missing project_home parameter in arguments
- **setup:** specify python version in setup.py, merge 'source_doctest' make target into 'test'

10.9 0.16.2 (2018-12-13)

10.9.1 New features

- **configs:** add support of *setting the environment variables* for build steps
- **code_report:** add parameter '–output-directory' for Uncrustify fixed files
- **code_report:** read HtmlDiff argument value from Uncrustify config
- **api:** add initial API support and 'file-diff' as example usage

10.9.2 Bug fixes

- **p4:** remove 'master CL check' feature as it doesn't work correctly
- **p4:** fix ascii decoding on p4 diff

10.10 0.16.1 (2018-11-22)

10.10.1 New features

- **code_report:** replace wildcards with directory names processing for Uncrustify
- **code_report:** add regexp support in pattern filter for Uncrustify

10.10.2 Bug fixes

- **p4:** fix 'Related Change IDs' bug with wrong current review determining

10.11 0.16.0 (2018-11-07)

10.11.1 New features

- **launch:** add critical background steps
- **vcs:** make VCS-related packages (e.g. `gitpython`) not reqired if not used
- **code_report:** add separate entry points for all *static analysers*
- **code_report:** add *Uncrustify* static analyser
- **out:** add pretty step numbering padding

10.11.2 Bug fixes

- **args:** fix required argument check to not accept empty values as valid
- **launcher:** finish background steps after foreground steps failing
- **out:** add reporting failed background steps to TC

10.12 0.15.4 (2018-09-26)

10.12.1 Bug fixes

- **swarm:** fix not adding current Swarm CL number to list of CLs to unshelve

10.13 0.15.3 (2018-09-26)

10.13.1 Bug fixes

- **swarm:** fix '[Related change IDs]' parsing

10.14 0.15.2 (2018-09-26)

10.14.1 New features

- **swarm:** add '[Related change IDs]' parsing for Swarm reviews

10.15 0.15.1 (2018-09-17)

10.15.1 BREAKING CHANGES

- **create unified entry point for all universum subcommands.** New usage is `universum poll` and `universum submit`

10.15.2 New features

- **launcher:** add *'finish_background'* key to Variations

10.15.3 Bug fixes

- **submit:** fix p4 submit fails for files opened in another workspace

10.16 0.15.0 (2018-09-04)

10.16.1 BREAKING CHANGES

- **swarm:** stop legacy support of 'SHELVE_CHANGELIST' environment variable for Swarm CL number

10.16.2 Bug fixes

- **jenkins:** fix Jenkins relative artifact paths/links

10.17 0.14.7 (2018-08-17)

10.17.1 New features

- **out:** add Jenkins plug-in specific labels for log collapsing

10.18 0.14.6 (2018-08-15)

10.18.1 New features

- **review:** add '–build-only-latest' option for skipping review builds of not latest review revisions
- add hidden '–clean-build' option for repeated debugging

10.19 0.14.5 (2018-08-09)

10.19.1 New features

- **swarm:** rename environment variable for Swarm CL ('SWARM_CHANGELIST') old name is still supported though

10.20 0.14.4 (2018-08-03)

10.20.1 Bug fixes

- **swarm:** fix Swarm review revision processing

10.21 0.14.3 (2018-08-01)

10.21.1 Bug fixes

- **swarm:** fix latest Swarm review revision detection

10.22 0.14.2 (2018-07-30)

10.22.1 Bug fixes

- **gerrit:** add exceptions on wrong Gerrit review parameters
- **swarm:** return voting for specified review version
- **swarm:** add review revision to comment text

10.23 0.14.1 (2018-07-23)

10.23.1 New features

- **report:** add '–report-no-vote' option for vote skipping

10.23.2 Bug fixes

- **configs:** remove outdated code style functions, fix get_project_root
- **code_report:** fix duplication of found issues message
- **launcher:** remove stderr from console output for launcher output type 'file'

10.24 0.14.0 (2018-06-25)

10.24.1 New features

- **code_report:** add svace analysis tool
- **main:** add finalizing execution even if interrupted by user
- **main:** add '–finalize-only' option for cleaning without execution
- **artifacts:** add recursive wildcards (**) to artifacts
- **utils:** add PyCharm case to environment detection
- **submit:** fix submitted P4 CL number in logs

10.24.2 Bug fixes

- **submit:** skip P4 submit if default CL has any files before reconciling
- **setup:** specify httpretty version to avoid SSL import errors

10.25 0.13.6 (2018-05-18)

10.25.1 New features

- **p4:** create environment variables for each mapping's sync CL

10.25.2 Bug fixes

- **docs:** fix change log

10.26 0.13.5 (2018-05-10)

10.26.1 BREAKING CHANGES

- **p4:** remove `allwrite` option in p4 client; please set '+w' modifier for files in VCS to be edited
- **configs:** *if_env_set* variables should now be splat with `&&` only

10.26.2 New features

- **report:** add support of *tagging* TeamCity builds
- **swarm:** `PASS` and `FAIL` parameters are no longer mandatory
- **submit:** new files are now added to VCS by submitter with '+w' modifier
- **report:** add link to build log to successful reports
- **report:** move link to review to 'Reporting build started' block

10.26.3 Bug fixes

- **p4:** fix unhandled 'no file(s) to reconcile' P4Exception
- **out:** fix bug with decoding non-ascii strings
- **docs:** documentation fixed and updated; please pay special attention to *prebuild artifact cleaning* *Variations* key

10.27 0.13.4 (2018-04-13)

10.27.1 New features

- **code_report:** add number of issues to build status
- **artifacts:** add link to artifact files to build log

10.27.2 Bug fixes

- **p4:** p4 client now is created with allwrite option
- **gerrit:** report all issues to review with a single request
- **code_report:** return error if pylint is not installed

10.28 0.13.3 (2018-03-22)

10.28.1 New features

- **configs:** add *negative 'if_env_set' values*

10.28.2 Bug fixes

- **add return of exit codes to all main scripts**
- **report:** fix bug with multiple success reporting

10.29 0.13.2 (2018-03-07)

10.29.1 New features

- **artifacts:** add CONFIGS_DUMP.txt to build artifacts
- **code_report:** add support for pylint3 for ubuntu14, restore LogWriterCodeReport
- **report:** update build result reporting, add skipped steps
- **report:** add option to only report failed steps

10.29.2 Bug fixes

- **report:** remove duplicating comment
- **out:** fix skipped steps reporting
- **configs:** fix critical step handling while merging one-element Variations

10.30 0.13.1 (2018-02-16)

10.30.1 Bug fixes

- **poll:** fix wrong order of polled changes

10.31 0.13.0 (2018-02-14)

10.31.1 New features

- **report:** add driver for processing Jenkins builds
- **launcher:** add critical steps for groups
- **setup:** add entry points for all high level scripts

10.31.2 Bug fixes

- **files:** fix cleaning sources function in finalize for Git
- **tests:** add stderr and exception/traceback detection
- **tests:** remove pylint error ignoring
- **code_report:** add exit codes for *code_report*

10.32 0.12.5 (2018-02-06)

10.32.1 Bug fixes

- **gerrit:** update 'Verified' to work with non-default labels
- **artifacts:** fix exception message when encountering existing artifacts
- **docs:** doc files removed from *master* branch

10.33 0.12.4 (2018-01-31)

10.33.1 New features

- **code_report:** implement static analysis support

10.34 0.12.3 (2018-01-19)

10.34.1 New features

- **code_report:** add *code_report* stub for further static analysis support
- **tests:** make errors in finalize affect exit code

10.34.2 Bug fixes

- **docs:** update TeamCity-related documentation
- **tests:** fix docker images makefiles

10.35 0.12.2 (2017-12-27)

10.35.1 New features

- **artifacts:** change to shell-style wildcards instead of old limited ones
- **submit:** reconcile files and directories from list
- **submit:** reconcile using wildcards
- **report:** update list of all performed steps, add successful
- **docs:** new *Variations keys* described

10.35.2 Bug fixes

- **report:** fix reporter message for build started
- **p4:** exit committed CL precommit check wihout failing
- **tests:** remove docker container caching where not necessary
- **tests:** fix import thirdparty detection

10.36 0.12.1 (2017-12-11)

10.36.1 New features

- **artifacts:** clean artifacts before build
- **git:** add user and email to Git module parameters

10.36.2 Bug fixes

- **vcs:** roll back of import fixes from release 0.10.2 causing Swarm builds of submitted CLs to fail
- **tests:** set user and email in testing Git repo

10.37 0.12.0 (2017-11-29)

10.37.1 BREAKING CHANGES

- **swarm:** the `--swarm` flag is replaced with `--report-to-review`. All pre-commit check configuration must be updated to reflect this change

10.37.2 Bug fixes

- **submit:** fix incorrectly back-ported fix from the new architecture, which prevented submit to git from working
- **gerrit:** fix bug with accessing url path by incorrect index and with including username into url in build log on pre-commit check

- **gerrit:** fix bug with adding apostrophe character (‘) to the ssh command line and failing to submit build start report to gerrit review

10.38 0.11.2 (2017-11-24)

10.38.1 New features

- **launcher:** add support for critical steps - now steps can be marked with *“critical” attribute* to fail entire build in case of step failure. By default the build continues even if some steps have failed

10.38.2 Bug fixes

- **submit:** fix setup script to actually install submitter module and to create console script called “universum_submit”
- **submit:** add support for executing commit message hooks by using external git utility instead of gitpython module (required to submit to gerrit)

10.38.3 Known issues

- **submit:** commit message hook is not downloaded from gerrit during cloning of the repository. As a workaround add installation of commit message hook to configs.py:

```
configs += Variations([dict(name="Install commit message hook",
                            command=["scp", "-p", "-P", "29418",
                                     "<user>@<server>:hooks/commit-msg", ".git/
↪hooks/"])])
```

- **submit:** by default, submit uses “temp” subfolder of the current folder as working directory. As a workaroung add the explicit setting of project root to configs.py:

```
configs += Variations([dict(name="Submit",
                            command=["universum_submit",
                                     "-pr", get_project_root(),
                                     "--vcs-type", "gerrit",
                                     "--commit-message", "Publish artifacts",
                                     "--file-list", "out/module.bin"])])
```

10.39 0.11.1 (2017-11-22)

10.39.1 New features

- **review:** add link to review page on server to logs
- **docs:** add instructions for TeamCity integration
- **tests:** add gravity tests for cases found by coverage
- **tests:** extend *test_git_poll* test suite with special merging cases

10.39.2 Bug fixes

- **report:** remove special characters from report message
- **launcher:** fix paths processing

10.40 0.11.0 (2017-11-09)

10.40.1 New features

- **submit:** add submit functionality for Git & Gerrit
- **tests:** add coverage report
- **tests:** add test for checking referencing dependencies

10.41 0.10.7 (2017-11-07)

10.41.1 Bug fixes

- **gerrit:** resolving issues fixed

10.42 0.10.6 (2017-11-06)

10.42.1 New features

- **tests:** add submitter initial tests

10.42.2 Bug fixes

- **files:** fix module construction order in main module and git *refspec* processing errors

10.43 0.10.5 (2017-11-03)

10.43.1 New features

- **files:** add repository state file
- **poll:** add poller for Git and initial tests

10.44 0.10.4 (2017-10-17)

10.44.1 New features

- **submit:** add an external script for submitting to repository

10.44.2 Bug fixes

- **p4:** remove reusing of existing p4 clients

10.45 0.10.3 (2017-10-17)

10.45.1 Bug fixes

- **git:** typo fix

10.46 0.10.2 (2017-10-10)

10.46.1 New features

- **git:** add *git checkout*, *git cherry-pick* and *refspec* functionality
- **gerrit:** add Gerrit support
- **configs:** add quotes and warning if space is detected within parameter in *command* item

10.46.2 Bug fixes

- **tests:** make unused vcs module import non-obligatory

10.47 0.10.1 (2017-09-22)

10.47.1 New features

- **git:** add initial Git support; change `--no-sync` into switch of `--vcs-type`

10.47.2 Bug fixes

- **p4:** fix 'Librarian checkout' exceptions

10.48 0.10.0 (2017-09-13)

10.48.1 New features

- **p4:** add `--p4-force-clean` instead of `--p4-no-clean` option: p4client is now not deleted by default

10.48.2 Bug fixes

- **Project 'Universe' renamed into 'Universum' to avoid name duplication**
- **reporter:** TeamCity-related parameters are no longer mandatory

10.49 0.9.1 (2017-08-25)

10.49.1 New features

- **launcher:** add support for *custom environment variables values*

10.50 0.9.0 (2017-08-22)

10.50.1 New features

- **Project 'Universe' transformed into a Python module, installable with pip**

10.50.2 Bug fixes

- **docs:** update documentation on module arguments

10.51 0.8.1 (2017-08-03)

10.51.1 New features

- **configs:** remove unnecessary nesting of configurations

10.51.2 Bug fixes

- **launcher:** append sys.path with config_path to import any subsidiary modules
- **report:** fix non-existing report_artifacts processing - ignore non-existing directories
- **launcher:** fix empty variable names - ' & name' is now processed correctly

10.52 0.8.0 (2017-07-26)

10.52.1 New features

- **CI Framework renamed into project 'Universe'**
- **docs:** add *description* of main script command-line parameters

10.52.2 Bug fixes

- **docs:** fix table content width, remove unnecessary scroll bars

10.53 0.7.0 (2017-07-21)

10.53.1 New features

- **docs:** add *system prerequisites page* to user manual
- **docs:** add documentation for `_universum.configuration_support` module
- **launcher:** add support for more than one environment variable to *filter configurations*

10.53.2 Bug fixes

- **launcher:** fix *configuration filtering*: filter artifacts as well as configurations
- **output:** use TeamCity built-in methods of stderr reporting for correct in-block error highlighting

10.54 0.6.3 (2017-07-13)

10.54.1 Bug fixes

- **docs:** fix product name and version display in documentation

10.55 0.6.2 (2017-07-11)

10.55.1 New features

- **report:** add *direct links to build artifacts* into reports

10.56 0.6.1 (2017-07-05)

10.56.1 New features

- **files:** add *working directory* reference to logs

10.56.2 Bug fixes

- **p4:** bring back reverting in 'prepare repository' step and add more logs

10.57 0.6.0 (2017-07-05)

10.57.1 New features

- **launcher:** add *configuration filtering*
- **artifacts:** wildcard initial support

10.58 0.5.0 (2017-06-06)

10.58.1 New features

- **tests:** add docker-based testing for p4poll
- **launcher:** change stderr printing to real-time instead of united report

10.59 0.4.1 (2017-05-30)

10.59.1 Bug fixes

- **artifacts:** fix artifacts reference before creation

10.60 0.4.0 (2017-05-30)

10.60.1 New features

- **artifacts:** artifacts are now collected to a separate directory
- **main:** add version numbering

10.61 0.3.0 (2017-05-25)

10.61.1 New features

- **swarm:** less default comments to Swarm, more optional
- **tests:** add pylint check
- **tests:** add doctest collecting

10.61.2 Bug fixes

- **test:** fix bug with stopping all test types once one type detects failure
- **swarm:** fix reporting to Swarm builds that did not execute actual build steps
- **launcher:** fix artifact collecting interruption
- **launcher:** fix extra dot directory in artifact archives

10.62 0.2.1 (2017-05-17)

10.62.1 Bug fixes

- **swarm:** Swarm double prefixes fixed

10.63 0.2.0 (2017-05-16)

10.63.1 New features

- **p4:** switch to disposable workspaces
- **p4:** add multiple VCS roots support
- **poll:** add perforce server polling to trigger builds by opening specified URL
- **tests:** add test stub
- **tests:** switch to py.test

10.63.2 Bug fixes

- **p4:** fix argument processing & list sorting
- **p4:** add p4client name changing
- **tests:** fix configs.py
- **tests:** add missing thirdparty dependency - module 'py'

10.64 0.1.1 (2017-04-26)

10.64.1 Bug fixes

- **output:** add warnings display

10.65 0.1.0 (2017-04-26)

10.65.1 New features

- **docs:** add change log
- **launcher:** add asynchronous step execution
- **docs:** update system configuring manual

10.65.2 Bug fixes

- **launcher:** change default 'command' launch directory back to project root

Project *Universum* is a continuous integration framework, containing a collection of functions that simplify implementation of the automatic build, testing, static analysis and other steps. The goal of this project is to provide unified approach for adding continuous integration to any project. It currently supports Perforce, Git, Gerrit, Swarm, Jenkins and TeamCity.

Sometimes *Universum* system can be referred to as the framework or just CI.

Python Module Index

Index

Symbols

A

C

D

F

G

S

V