SAT Documentation

Release 5.8.0

CEA DES/ISAS/DM2S/STMF/LGLS

May 18, 2021

CONTENTS

SAT is a tool that makes it easy to build on various linux platforms and windows large software, which rely on a lot of prerequisites. It was originally created for the maintenance and the packaging of SALOME platform (its name comes from **Sa**lome**T**ools), its usage is now wider. The following features should be highlighted:

- the **definition** of the application content: which products (prerequisites, codes, modules) are necessary and which versions are required
- the **configuration** of the application : how to get the source of products, how to compile them, which options to use, etc. The configuration can be conditionally overwritten, this feature allows application developers taking into account platform specifics.
- the **preparation** of the complete software: all the required sources with correct versions are retrieved from git/svn/cvs repositories, or from already prepared tarballs.
- management of **patches** if some are required to compile on specific platforms (portage)
- management of the **environment** to set up at compile time and at runtime
- automatic **compilation** of the complete application (the application with all its products).
- production of a **launcher** that sets up the environment and starts the application
- management of **tests**: both unit and integration tests are managed
- **packaging**: creation of binary and/or source packages to distribute the application on various platforms
- **overwriting** the configuration in command line: it allows users setting easily their own preferences or options

SAT uses **python**, and many of its strength come from its power and straightforwardness. It is compatible with both python2 and python3 versions. SAT is a Command Line Interface (CLI[1]) based on python language. It is a suite of commands, which are detailed later in this documentation. These commands are used to perform the operations on the application. SAT enables command completion by sourcing the provided complete_sat.sh script.

Like similar tool, SAT doesn't like modified environments, as this can cause conflicts while compiling products or using applications. It is recommended that SAT users run with a **clean environment**, especially for PATH, LD_LIBRARY_PATH and PYTHONPATH. ~/.bashrc file should be as thin as possible!

[1] https://en.wikipedia.org/wiki/Command-line_interface

CHAPTER

ONE

DOCUMENTATION

1.1 Installation

sat is provided either embedded into a salome package, or as a standalone package. It can also be retrieved from the git repositories.

1.1.1 From git bases

sat git bases are hosted by the salome platform Tuleap forge[1] . Therefore you first have to get an account to this forge. To get started, one has to download sat, and at last one sat project (usually SAT_SALOME project, which contains all the configuration required to build SALOME and its prerequisites). The following script get sat and SAT_SALOME project from git repositories:

```
# get sat
BASE_SAT=https://codev-tuleap.cea.fr/plugins/git/spns/SAT.git
BASE_PROJET=https://codev-tuleap.cea.fr/plugins/git/spns/SAT_SALOME.git
TAG_SAT=master
TAG_PROJET=master
git clone ${BASE_SAT}
cd SAT
git checkout ${TAG_SAT}
cd ..

# get sat project SAT_SALOME
git clone ${BASE_PROJET}
cd  SAT_SALOME
git checkout ${TAG_PROJET}
cd ..

# initialisation de sat

# add SAT_SALOME project to sat, other configurations projects can be added
SAT/sat init --add_project  $(pwd)/SAT_SALOME/salome.pyconf

# record tag and url (not mandatory)
SAT/sat init --VCS $BASE_SAT
SAT/sat init --tag $(git describe --tags)
```

1.1.2 Embedded sat version

sat is provided in salome packages with sources, in order to be able to recompile the sources (**sat** is not provided in salome packages with only binaries).

[1] https://codev-tuleap.cea.fr/projects/salome

Embedded **sat** is always associated to an embedded **sat** project, which contains all the products and application configuration necessary to the package.

```
tar -xf SALOME-9.3.0-CO7-SRC.tgz
cd SALOME-9.3.0-CO7-SRC
ls PROJECT/   # list the embedded sat project
# edit the SALOME-9.3.0 configuration pyconf file
sat/sat config SALOME-9.3.0 -e
```

The user has usually two main use cases with an embedded sat, which are explained in the README file of the archive:

1. recompile the complete application

```
./sat prepare SALOME-9.3.0
./sat compile SALOME-9.3.0
./sat launcher SALOME-9.3.0
```

Please note that the sources are installed in *SOURCES* directory, and the compilation is installed in *INSTALL* directory (therefore they do not overwrite the initial binaries, which are stored in *BINARIES-XXX* directory). The launcher *salome* is overwritten (it will use the new compiled binaries) but the old binaries can still be used in connection with *binsalome* launcher).

2. recompile only a part of the application

It is possible to recompile only a part of the products (those we need to modify and recompile). To enter this (partial recompilation mode), one has initially to copy the binaries from *BINARIES-XXX* to *INSTALL*, and do the path substitutions by using the **install_bin.sh** script:

```
# pre-installation of all binaries in INSTALL dir, with substitutions
./install_bin.sh
./sat prepare SALOME-9.3.0 -p GEOM  # get GEOM sources, modify them
./sat compile SALOME-9.3.0 -p GEOM --clean_all  # only recompile GEOM
```

1.1.3 Standalone sat packages

sat is also delivered as a standalone package, usually associated to a sat project. The following example is an archive containing sat 5.3.0 and the salome sat project. It can be used to build from scratch any salome application.

```
# untar a standalone sat package, with a salome project
tar xf sat_5.3.0_satproject_salome.tgz
cd sat_5.3.0_satproject_salome
ls projects  # list embedded sat projects
> salome
./sat config -l  # list all salome applications available for build
```

Finally, the project also provides bash scripts that get a tagged version of sat from the git repository, and a tagged version of salome projects. This mode is dedicated to the developers, and requires an access to the Tuleap git repositories.

1.2 Using SAT

1.2.1 Getting started

SAT is a Command Line Interface (CLI[2]) based on python language. Its purpose is to cover the maintenance and the production of an application which has to run on several platforms and depends upon a lot of prerequisites. It is most of the time used interactively from a terminal, but there is also a batch mode that can be used for example in automatic procedures (like jenkins jobs). SAT is used in command line by invoking after its name a sat option (which is non mandatory), then a command name, followed by the arguments of the command (most of the time the name of an application and command options):

```
./sat [generic_options] [command] [application] [command_options]
```

The main sat options are:

- **-h** : to invoke the **help** and get the list of available options and commands
- **-o** : to **overwrite** at runtime a configuration parameter or option
- **-v** : to change the **verbosity** (default is 3, minimum 0 and maximum 6)
- **-b** : to enter the **batch** mode and avoid any question (this non interactive mode is useful for automatic procedures like jenkins jobs)
- **-t** : to display the compilation logs in the **terminal** (otherwise they are logged in files and displayed by the log command

The main sat commands are:

- **prepare** : to get the sources of the application products (from git repositories or archives) and apply patches if there are any
- **compile** : to build the application (using cmake, automake or shell script)
- **launcher** : to generate a launcher of the application (in the most general case the launcher sets up the run-time environment and starts an exe)
- **package** : to build a package of the application (binary and/or source)
- **config** : to display the configuration
- **log** : to display within a web browser the logs of SAT

1.2.2 Getting help

Help option -h

More details are provided by the help of sat. The help option can be called at two levels : the high level displays information on how to use sat, the command level displays information on how to use a sat command.

```
# display sat help
./sat -h

# display the help of the compile command
./sat compile -h
```

Completion mode

When getting started with sat, the use of the completion mode is convenient. This mode will display by typing twice on the **tab key** the available options, commands, applications or products available. The completion mode has to be activated by sourcing the file **complete_sat.sh** contained in SAT directory:

[2] https://en.wikipedia.org/wiki/Command-line_interface

```
# activate the completion mode
source complete_sat.sh

# list all application available for compilation
./sat compile  <TAB> <TAB>
> SALOME-7.8.2   SALOME-8.5.0   SALOME-9.3.0   SALOME-master

# list all available options of sat compile
./sat compile SALOME-9.3.0 <TAB> <TAB>
> --check             --clean_build_after  --install_flags    --properties
> --stop_first_fail   --with_fathers       --clean_all        --clean_make
> --products          --show               --with_children
```

1.2.3 Build from scratch an application

This is the main use case : build from scratch an application.

```
# get the list of available applications in your context
# the result depends upon the projects that have been loaded in sat.
./sat config -l
>  ...
>  SALOME-8.5.0
>  SALOME-9.3.0
>  SALOME-9.4.0

# get all sources of SALOME-9.4.0 application
./sat prepare SALOME-9.4.0

# compile all products (prerequisites and modules of SALOME-9.4.0)
./sat compile SALOME-9.4.0

# if a compilation error occured, you can access the compilation logs with:
./sat log SALOME-9.4.0

# create a SALOME launcher, displays its path.
./sat launcher SALOME-9.4.0
> Generating launcher for SALOME-9.4.0 :
>   .../SALOME-9.4.0-CO7/salome

# start salome platform
.../SALOME-9.4.0-CO7/salome

# create a binary package to install salome on other computers
./sat package SALOME-9.4.0 -b
```

All the build is done in the *application directory*, which is parameterized by the sat configuration variable *$APPLICATION.workdir*. In the above example this directory corresponds to *.../SALOME-9.4.0-CO7*. SAT can only build applications provided by the projects that have been loaded with *sat init* command. The available applications are listed by *sat config -l* command.

1.2.4 Partial recompilation of a packaged application

Getting all the sources and compile everything is often a long process. The following use case has proven to be convenient for fast usage! It consists to get the application through a sat package containing the binaries, the sources and SAT. This allows using directly the application after the untar (the binary part). And later, if required, it is possible to add a module, or modify some source code and recompile only what was added or modified.

```
# untar a sat package containing binaries (for CentOS7) and sources
tar xfz SALOME-9.4.0-CO7-SRC.tar.gz
```

```
# start salome
SALOME-9.4.0-CO7-SRC/salome

# copy binaries in INSTALL directory, do required substitutions
# to enable recompilation
./install_bin.sh

# get sources of modules we want to recompile
sat/sat prepare SALOME-9.4.0 -p SHAPER,SMESH

# do some modifications and recompile both modules
sat/sat compile SALOME-9.4.0 -p SHAPER,SMESH  --clean_all
```

This use case is documented in the README file of the package

1.2.5 Using SAT bases

Users or developers that have to build several applications, which share common products, may want to mutualise the compilation of the common products. The notion of SAT base follows this objective. It allows sharing the installation of products between several applications, and therefore compile these products only once.

Location

By default the SAT base is located in the parent directory of sat (the directory containing sat directory) and is called BASE. This default can be changed by the user with sat init command :

```
# change the location of SAT base directory
./sat init -b <new base path>
```

Which products go into the base

The application developer has the possibility to declare that a product will go by default in the base. He uses for that the keyword 'base' in the install_dir key within the product configuration file (products pyconf) : *install_dir : 'base'* It is done usually for products that are considered as prerequisites.

At this stage, all products with install_dir set to 'base' will be installed in SAT base directory.

Application configuration

The default behavior of products can be modified in the application configuration, with the **base** flag. Like other application flags (debug, verbose, dev) the **base** flag can be used for a selection of products, or globally for all products.

```
# declare in application configuration that SMESH and YACS are installed in base
products :
{
...
SMESH : {base : "yes"}
YACS : {base : "yes"}
...
}

# declare with a global application flag that all products are installed in base
base : "yes"
```

Mutualisation of products

Products that go in base and have the same configuration will be shared by different applications (it's the objective). SAT does check the configuration to prevent of an application using a product in base with a non compatible configuration. To check the compatibility, SAT stores the configuration in a file called *sat-config-<product name>.pyconf*. In a next build (for example in another application), SAT checks if the new configuration corresponds to what is described in *sat-config-<product name>.pyconf*. If it corresponds, the previous build is used in base, otherwise a new build is done, and stored in a new directory called *config-<build number>*.

Warning: Please note that only the dependencies between products are considered for the checking. If the compilation options changed, it will not be tracked (for example the use of debug mode with -g option will not produce a second configuration, it will overwrite the previous build done in production mode)

1.2.6 Developing a module with SAT

SAT has some features that make developers' life easier. Let's highlight some of the developers use cases. (if you are not familiar with SAT configuration, you may first read Configuration Chapter before, and come back to this paragraph after)

Activating the development mode

By default *sat prepare* command is not suited for development, because it erases the source directory (if it already exists) before getting the sources. If you did developments in this directory **they will be lost!**.

Therefore before you start some developments inside a product, you should **declare the product in development mode** in the application configuration. For example if you plan to modify KERNEL module, modify SALOME configuration like this:

```
APPLICATION :
{
...
    products :
    {
    # declare KERNEL in development mode (and also compile it
    # with debug and verbose options)
    'KERNEL' : {dev:'yes', debug:'yes', verbose:'yes',
                tag:'my_dev_branch', section:'version_7_8_0_to_8_4_0'}
    ...
    }
}
```

When the dev mode is activated, SAT will load the sources from the git repository only the first time, when the local directory does not exist. For the next calls to *sat prepare*, it will keep the source intact and do nothing!

In the example we have also set the debug and the verbose flags to "yes" - it is often useful when developing.

Finally, we have changed the tag and replaced it with a development branch (to be able to push developments directly in git repo - without producing patches).

Warning: But doing this we have (probably) broken the automatic association done by SAT between the tag of the product and the product section used by SAT to compile it! (see the chapter "Product sections" in the Configuration documentation for more details about this association) Therefore you need to tell SAT which section to use (otherwise it will take the "default" section, and it may not be the one you need). This is done with : **section:'version_7_8_0_to_8_4_0'**. If you don't know which section should be used, print it with SAT config before changing the tag : *./sat config SALOME-9.4.0 -i KERNEL* will tell you which section is being used.

Pushing developments in base, or creating patches

If you have set the tag to a development branch (like in the previous example), you can directly push your developments in the git repository with *git push* command. If not (if you are detached to a tag, you can produce with git a patch of you developments:

```
git diff > my_dev.patch
```

And use this patch either with SAT to apply it automatically with *sat prepare* command, or send the patch for an integration request.

Changing the source directory

By default the source directory of a product is located inside SAT installation, in the SOURCES directory. This default may not be convenient. Developers may prefer to develop inside the HOME directory (for example when this directory is automatically saved).

To change the default source directory, you first have to identify which product section is used by SAT:

```
./sat config SALOME-9.4.0 -i KERNEL
>  ....
>  section = default
```

Then you can change the source directory in the section being used (default in the example above). For that you can modify the **source_dir** field in the file *SAT_SALOME/products/KERNEL.pyconf*. Or change it in command line: **./sat -o "PRODUCTS.KERNEL.default.source_dir='/home/KERNEL'" <your sat command>**. For example the following command recompiles KERNEL using */home/KERNEL* as source directory:

```
# take KERNEL sources in /home/KERNEL
./sat -o "PRODUCTS.KERNEL.default.source_dir='/home/KERNEL'" compile SALOME-master\
      -p KERNEL --clean_all
```

Displaying compilation logs in the terminal

When developing a module you often have to compile it, and correct errors that occurs. In this case, using *sat log* command to consult the compilation logs is not convenient! It is advised to use in this case the **-t** option of sat, it will display the logs directly inside the terminal:

```
# sat -t option put the compilation logs in the terminal
./sat -t -o "PRODUCTS.KERNEL.default.source_dir='/home/KERNEL'" compile\
       SALOME-master -p KERNEL --clean_all
```

1.3 Configuration

1.3.1 Introduction

For the configuration, SAT uses a python module called *config*, which aims to offer more power and flexibility for the configuration of python programs. This module was slightly adapted for SAT, and renamed Pyconf. (see config module[3] for a complete description of the module, the associated syntax, the documentation).

sat uses files with **.pyconf** extension to store the configuration parameters. These *.pyconf* are parsed by SAT, and merged into a global configuration, which is passed to the sat commands and used by them.

[3] http://www.red-dove.com/config-doc/

1.3.2 Configuration projects

By default SAT is provided with no configuration at all, except is own internal one. The configuration is brought by SAT projects : usually a git base containing all the configuration files of a project (*.pyconf* files). For Salome platform, the SAT project is called SAT_SALOME and can be downloaded from salome Tuleap forge. SAT projects are loaded in sat with the sat init command:

```
# get salome platform SAT configuration project (SAT_SALOME), and load it into SAT
git clone SAT_SALOME
SAT/sat init --add_project $(pwd)/SAT_SALOME/salome.pyconf
```

SAT_SALOME project provides all configuration files for salome applications, and for the products that are used in these applications.

1.3.3 Application configuration

The configuration files of applications contain the required information for SAT to build the application. They are usually located in the application directory of the project:

```
# list applications provided by SAT_SALOME project
ls SAT_SALOME/applications
>   MEDCOUPLING-9.4.0.pyconf            SALOME-7.8.0.pyconf
>   SALOME-8.5.0.pyconf          SALOME-9.4.0.pyconf
```

These files can be edited directly, and also with the SAT:

```
# edit SALOME-9.4.0.pyconf configuration file
SAT/sat config SALOME-9.4.0 -e
```

The application configuration file defines the APPLICATION sections. The content of this section (or a part of it) can be displayed with *sat config* command:

```
# display the complete APPLICATION configuration
sat config SALOME-9.4.0 -v APPLICATION

# display only the application properties
sat config SALOME-9.4.0 -v APPLICATION.properties
```

SAT users that need to create new application files for their own purpose usually copy an existing configuration file and adapt it to their application. Let's discribe the content of an application pyconf file. We take in the following examples the file *SAT_SALOME/applications/SALOME-9.4.0.pyconf*.

Global variables and flags

At the beginning of the APPLICATION sections, global variables and flags are defined:

- **name** : the name of the application (mandatory)
- **workdir** : the directory in which the application is produced (mandatory)
- **tag** : the default tag to use for the git bases
- **dev** : activate the dev mode. In dev mode git bases are checked out only one time, to avoid risks of removing developments.
- **verbose** : activate verbosity in the compilation
- **debug** : activate debug mode in the compilation, i.e -g option
- **python3** : 'yes/no' tell sat that the application uses python3
- **base** : 'yes/no/name' to set up the use of a SAT base

```
APPLICATION :
{
    name : 'SALOME-9.4.0'
    workdir : $LOCAL.workdir + $VARS.sep + $APPLICATION.name + '-' + $VARS.dist
    tag : 'V9_4_BR'
    debug : 'no'
    dev : 'no'
    base : 'no'
    python3 : 'yes'
    ...
```

Please note the workdir variable is defined in the above example with references to other sections defined in other configurations files (i.e. $LOCAL and $VARS). It's a useful Pyconf functionality. Most of the global variables are optionnal, except name and workdir.

Environment subsection

This subsection allows defining environment variables at the application level (most of the time the environment is set by the products configuration).

```
APPLICATION :
{
...
    environ :
    {
        build : {CONFIGURATION_ROOT_DIR : $workdir + $VARS.sep + "SOURCES" +\
                 $VARS.sep + "CONFIGURATION"}
        launch : {PYTHONIOENCODING:"UTF_8"}
        SALOME_trace : "local" # local/file:.../with_logger
        # specify the first modules to display in gui
        SALOME_MODULES : "SHAPER,GEOM,SMESH,PARAVIS,YACS,JOBMANAGER"
    }
}
```

In the example above CONFIGURATION_ROOT_DIR variable will be set only at compile time (usage of *build* key), while PYTHONIOENCODING will be set only at run-time (use of *launch* key). variables SALOME_trace and SALOME_MODULES are set both at compile time and run time.

products subsection

This subsection will specify which products are included in the application. For each product, it is possible to specify in a dictionnary:

- **tag** : the tag to use for the product
- **dev** : activate the dev mode.
- **verbose** : activate verbosity in the compilation
- **debug** : activate debug mode

If these flags are not specified, SAT takes the default application flag. In the following example, SAT uses the default tag V9_4_BR for products SHAPER, KERNEL and MEDCOUPLING. For LIBBATCH it uses the tag V2_4_2. KERNEL is compiled in debug and verbose mode.

```
APPLICATION :
{
...
    tag : 'V9_4_BR'
...
    products :
    {
```

```
    'SHAPER'
    'LIBBATCH' : {tag :'V2_4_2'}
    'KERNEL' : {debug:'yes', verbose:'yes'}
    'MEDCOUPLING'
    ...
```

properties

Properties are used by SAT to define some general rules or policies. They can be defined in the application configuration with the properties subsection:

```
APPLICATION :
{
...
    properties :
    {
        mesa_launcher_in_package : "yes"
        repo_dev : "yes"
        pip : 'yes'
        pip_install_dir : 'python'
    }
```

In this example the following properties are used:

- **mesa_launcher_in_package** : ask to put a mesa launcher in the packages produced by sat package command
- **repo_dev** : use the development git base (for salome, the tuleap forge)
- **pip** : ask to use pip to get python products
- **pip_install_dir** : install pip products in python installation directory (not in separate directories)

1.3.4 Products configuration

The configuration files of products contain the required information for SAT to build each product. They are usually located in the product directory of the project. SAT_SALOME supports a lot of products:

```
ls SAT_SALOME/products/
ADAO_INTERFACE.pyconf     homard_bin.pyconf          PyQtChart.pyconf
ADAO.pyconf               homard_pre_windows.pyconf  PyQt.pyconf
alabaster.pyconf          HOMARD.pyconf              pyreadline.pyconf
ALAMOS_PROFILE.pyconf     HXX2SALOME.pyconf          Python.pyconf
ALAMOS.pyconf             HYBRIDPLUGIN.pyconf        pytz.pyconf
Babel.pyconf              idna.pyconf                qt.pyconf
BLSURFPLUGIN.pyconf       imagesize.pyconf           qwt.pyconf
boost.pyconf              ispc.pyconf                requests.pyconf
bsd_xdr.pyconf            Jinja2.pyconf              RESTRICTED.pyconf
CALCULATOR.pyconf         JOBMANAGER.pyconf          root.pyconf
CAS.pyconf                KERNEL.pyconf              ruby.pyconf
CDMATH.pyconf             kiwisolver.pyconf          SALOME_FORMATION_PROFILE.pyconf
CEATESTBASE.pyconf        lapack.pyconf              SALOME_PROFILE.pyconf
certifi.pyconf            lata.pyconf                SALOME.pyconf
cgns.pyconf               LIBBATCH.pyconf            SAMPLES.pyconf
chardet.pyconf            libjpeg.pyconf             scipy.pyconf
click.pyconf              libpng.pyconf              scons.pyconf
cmake.pyconf              libxml2.pyconf             scotch.pyconf
colorama.pyconf           llvm.pyconf                setuptools.pyconf
compil_scripts            markupsafe.pyconf          SHAPER.pyconf
COMPONENT.pyconf          matplotlib.pyconf          SHAPERSTUDY.pyconf
CONFIGURATION.pyconf      MEDCOUPLING.pyconf         sip.pyconf
COREFLOWS_PROFILE.pyconf  medfile.pyconf             six.pyconf
```

```
COREFLOWS.pyconf          med_pre_windows.pyconf     SMESH.pyconf
cppunit.pyconf            MED.pyconf                 snowballstemmer.pyconf
cycler.pyconf             mesa.pyconf                SOLVERLAB.pyconf
Cython.pyconf             MeshGems.pyconf            solvespace.pyconf
dateutil.pyconf           metis.pyconf               sphinxcontrib_applehelp.pyconf
distribute.pyconf         mpc.pyconf                 sphinxcontrib_devhelp.pyconf
DOCUMENTATION.pyconf      mpfr.pyconf                sphinxcontrib_htmlhelp.pyconf
docutils.pyconf           msvc.pyconf                sphinxcontrib_jsmath.pyconf
doxygen.pyconf            NETGENPLUGIN.pyconf        sphinxcontrib_napoleon.pyconf
EFICAS.pyconf             netgen.pyconf              sphinxcontrib.pyconf
EFICAS_TOOLS.pyconf       nlopt.pyconf               sphinxcontrib_qthelp.pyconf
eigen.pyconf              numpy.pyconf               sphinxcontrib_serializinghtml.pyconf
embree.pyconf             omniNotify.pyconf          sphinxcontrib_websupport.pyconf
env_scripts               omniORB.pyconf             sphinxintl.pyconf
expat.pyconf              omniORBpy.pyconf           Sphinx.pyconf
f2c.pyconf                openblas.pyconf            sphinx_rtd_theme.pyconf
ffmpeg.pyconf             opencv.pyconf              subprocess32.pyconf
FIELDS.pyconf             openmpi.pyconf             swig.pyconf
freeimage.pyconf          openssl.pyconf             tbb.pyconf
freetype.pyconf           ospray.pyconf              tcl.pyconf
ftgl.pyconf               packaging.pyconf           tcltk.pyconf
functools32.pyconf        ParaViewData.pyconf        TECHOBJ_ROOT.pyconf
gcc.pyconf                ParaView.pyconf            tk.pyconf
GEOM.pyconf               PARAVIS.pyconf             Togl.pyconf
GHS3DPLUGIN.pyconf        ParMetis.pyconf            TRIOCFD_IHM.pyconf
GHS3DPRLPLUGIN.pyconf     patches                    TRIOCFD_PROFILE.pyconf
gl2ps.pyconf              perl.pyconf                TrioCFD.pyconf
glu.pyconf                petsc.pyconf               TRUST.pyconf
gmp.pyconf                Pillow.pyconf              typing.pyconf
GMSHPLUGIN.pyconf         planegcs.pyconf            uranie_win.pyconf
gmsh.pyconf               pockets.pyconf             urllib3.pyconf
graphviz.pyconf           pthreads.pyconf            VISU.pyconf
GUI.pyconf                PY2CPP.pyconf              vtk.pyconf
hdf5.pyconf               pybind11.pyconf            XDATA.pyconf
HELLO.pyconf              PYCALCULATOR.pyconf        YACSGEN.pyconf
HEXABLOCKPLUGIN.pyconf    Pygments.pyconf            YACS.pyconf
HEXABLOCK.pyconf          PyHamcrest.pyconf          zlib.pyconf
HexoticPLUGIN.pyconf      PYHELLO.pyconf
Hexotic.pyconf            pyparsing.pyconf
```

Available product configuration flags

- **name** : the name of the product
- **build_source** : the method to use when getting the sources, possible choices are script/cmake/autotools. If "script" is chosen, a compilation script should be provided with compil_script key
- **compil_script** : to specify a compilation script (in conjunction with build_source set to "script"). The programming language is bash under linux, and bat under windows.
- **get_source** : the mode to get the sources, possible choices are archive/git/svn/cvs
- **depend** : to give SAT the dependencies of the product
- **patches** : provides a list of patches, if required
- **source_dir** : where SAT copies the source
- **build_dir** : where SAT builds the product
- **install_dir** : where SAT installs the product

The following example is the configuration of boost product:

```
default :
{
    name : "boost"
    build_source : "script"
    compil_script :  $name + $VARS.scriptExtension
    get_source : "archive"
    environ :
    {
       env_script : $name + ".py"
    }
    depend : ['Python' ]
    opt_depend : ['openmpi' ]
    patches : [ ]
    source_dir : $APPLICATION.workdir + $VARS.sep + 'SOURCES' + $VARS.sep + $name
    build_dir : $APPLICATION.workdir + $VARS.sep + 'BUILD' + $VARS.sep + $name
    install_dir : 'base'
    properties :
    {
        single_install_dir : "yes"
        incremental : "yes"
    }
}
```

Product properties

Properties are also associated to products. It is possible to list all the properties with the command *./sat config SALOME-9.4.0 –show_properties**

Here are some properties frequently used:

- **single_install_dir** : the product can be installed in a common directory
- **compile_time** : the product is used only at compile time (ex : swig)
- **pip** : the product is managed by pip
- **not_in_package** : the product will not be put in packages
- **is_SALOME_module** : the product is a SALOME module
- **is_distene** : the product requires a DISTENE licence

The product properties allow SAT doing specific choices according to the property. They also allow users filtering products when calling commands. For example it is possible to compile only SALOME modules with the command:

```
# just recompile SALOME modules, not other products
./sat compile SALOME-9.4.0 --properties is_SALOME_module:yes --clean_all
```

Product environment

The product environment is declared in a subsection called environment. It is used by sat at compile time to set up the environment for the compilation of all the products depending upon it. It is also used at run time to set up the application environment.

Two mechanisms are offered to define the environment. The first one is similar to the one used in the application configuration : inside the environ section, we declare variables or paths. A variable appended or prepended by an underscore is treated as a path, to which we prepend or append the valued according to the position of the underscore. In the following example, the value *<install_dir/share/salome/ressources/salome* is prepended to the path SalomeAppConfig.

```
environ :
{
    _SalomeAppConfig : $install_dir + $VARS.sep + "share" + $VARS.sep + "salome" +\
                       $VARS.sep + "resources" + $VARS.sep + "salome"
}
```

But the most common way is to use an environment script, which specifies the environment by using an API provided by sat:

```
# use script qt.py to set up qt environment
environ :
{
   env_script : "qt.py"
}
```

As an example, the environment script for qt is:

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

import os.path
import platform

def set_env(env, prereq_dir, version):
    env.set('QTDIR', prereq_dir)

    version_maj = version.split('.')
    if version_maj[0] == '5':
        env.set('QT5_ROOT_DIR', prereq_dir)
        env.prepend('QT_PLUGIN_PATH', os.path.join(prereq_dir, 'plugins'))
        env.prepend('QT_QPA_PLATFORM_PLUGIN_PATH',
                    os.path.join(prereq_dir, 'plugins'))
        pass
    else:
        env.set('QT4_ROOT_DIR', prereq_dir)
        pass

    env.prepend('PATH', os.path.join(prereq_dir, 'bin'))

    if platform.system() == "Windows" :
        env.prepend('LIB', os.path.join(prereq_dir, 'lib'))
        pass
    else :
        env.prepend('LD_LIBRARY_PATH', os.path.join(prereq_dir, 'lib'))
        pass
```

env is the API provided by SAT, *prereq_dir* is the installation directory, *version* the product version. *env.set* sets a variable, *env.prepend* and *env.append* are used to prepend or append values to a path.

The **setenv** function is used to set the environment at compile time and run time. It is also possible to use **set_env_build** and **set_env_launch** callback functions to set specific compile or run time environment. Finally the function **set_nativ_env** is used for native products.

Product sections

The product configuration file may contain several sections. In addition to the "default" section, it is possible to declare other sections that will be used for specific versions of the product. This allows SAT compiling different versions of a product. To determine which section should be used, SAT has an algorithm that takes into account the version number. Here are some examples of sections that will be taken into account by SAT :

```
# this section will be used for versions between 8.5.0 and 9.2.1
_from_8_5_0_to_9_2_1 :
```

```
{
    ...
}

# this section will only ve used for 9.3.0 version
version_9_3_0 :
{
    ...
}
```

Several version numbering are considered by SAT (not only X.Y.Z) For example V9, v9, 9, 9.0.0, 9_0_0, are accepted.

By default SAT only considers one section : the one determined according to the version number, or the default one. But if the **incremental property** is defined in the default section, and is set to "yes", then SAT enters in the **incremental mode** and merges different sections into one, by proceeding incremental steps. SAT uses the following algorithm to merge the sections:

1. We take the complete "default" section
2. If a "default_win" section is defined, we merge it.
3. If a section name corresponds to the version number, we also merge it.
4. Finally on windows platform if the same section name appended by _win exists, we merge it.

1.3.5 Other configuration sections

The configuration of SAT is split into eight sections : VARS, APPLICATION, PRODUCTS, PROJECTS, PATHS, USER, LOCAL, INTERNAL. These sections are fed by the pyconf files which are loaded by sat: each pyconf file is parsed by SAT and merged into the global configuration. One file can reference variables defined in other files. Files are loaded in this order :

- the internal pyconf (declared inside sat)
- the personal pyconf : *~/.salomeTools/SAT.pyconf*
- the application pyconf
- the products pyconf (for all products declared in the application)

In order to check the configuration and the merge done by sat, it is possible to display the resulting eight section with the command:

```
# display the content of a configuration section
# (VARS, APPLICATION, PRODUCTS, PROJECTS, PATHS, USER, LOCAL, INTERNAL)
SAT/sat config SALOME-9.4.0 -v <section>
```

Note also that if you don't remember the name of a section it is possible to display section names with the automatic completion functionality.

We have already described two of the sections : APPLICATION and PRODUCTS. Let's describe briefly the six others.

VARS section

This section is dynamically created by SAT at run time.
It contains information about the environment: date, time, OS, architecture etc.

```
# to get the current setting
sat config --value VARS
```

USER section

This section is defined by the user configuration file, `~/.salomeTools/SAT.pyconf`.

The `USER` section defines some parameters (not exhaustive):

- **pdf_viewer** : the pdf viewer used to read pdf documentation
- **browser** : The web browser to use (*firefox*).
- **editor** : The editor to use (*vi, pluma*).
- and other user preferences.

```
# to get the current setting
sat config SALOME-xx --value USER

# to edit your personal configuration file
sat config -e
```

Other sections

- **PROJECTS** : This section contains the configuration of the projects loaded in SAT by *sat init –add_project* command.
- **PATHS** : This section contains paths used by sat.
- **LOCAL** : contains information relative to the local installation of SAT.
- **INTERNAL** : contains internal SAT information

1.3.6 Overwriting the configuration

At the end of the process, SAT ends up with a complete global configuration resulting from the parsing of all *.pyconf* files. It may be interesting to overwrite the configuration. SAT offers two overwriting mechanisms to answer these two use cases:

1. Be able to conditionally modify the configuration of an application to take into account specifics and support multi-platform builds
2. Be able to modify the configuration in the command line, to enable or disable some options at run time

Application overwriting

At the end of the application configuration, it is possible to define an overwrite section with the keyword **__overwrite__ :**. It is followed by a list of overwrite sections, that may be conditional (use of the keyword **__condition__ :**). A classical usage of the application overwriting is the change of a prerequisite version for a given platform (when the default version does not compile).

```
__overwrite__ :
[
  {
   # opencv 3 do not compile on old CO6
    __condition__ : "VARS.dist in ['CO6']"
    'APPLICATION.products.opencv' : '2.4.13.5'
  }
]
```

Command line overwriting

Command line overwriting is triggered by sat **-o** option, followed in double quotes by the parameter to overwrite, the = sign and the value in simple quotes. In the following example, we suppose that the application SALOME-9.4.0 has set both flags debug and verbose to "no", and that we want to recompile MEDCOUPLING in debug mode, with cmake verbosity activated. The command to use is:

```
# recompile MEDCOUPLING in debug mode (-g) and with verbosity
./sat -t -o "APPLICATION.verbose='yes'" -o "APPLICATION.debug='yes'" compile\
             SALOME-9.4.0 -p MEDCOUPLING --clean_all
```

CHAPTER

TWO

LIST OF COMMANDS

2.1 Command doc

2.1.1 Description

The **doc** command displays sat documentation.

2.1.2 Usage

- Show (in a web browser) the sat documentation in format xml/html:

  ```
  sat doc --xml
  ```

- Show (in evince, for example) the (same) sat documentation in format pdf:

  ```
  sat doc --pdf
  ```

- Edit and modify in your preference user editor the sources files (rst) of sat documentation:

  ```
  sat doc --edit
  ```

- get information how to compile locally sat documentation (from the sources files):

  ```
  sat doc --compile
  ```

2.1.3 Some useful configuration paths

- **USER**
  - **browser** : The browser used to show the html files (*firefox* for example).
  - **pdf_viewer** : The viewer used to show the pdf files (*evince* for example).
  - **editor** : The editor used to edit ascii text files (*pluma or gedit* for example).

2.2 Command config

2.2.1 Description

The **config** command manages sat configuration. It allows display, manipulation and operation on configuration files

2.2.2 Usage

- Edit the user personal configuration file `$HOME/.salomeTools/SAT.pyconf`. It is used to store the user personal choices, like the favourite editor, browser, pdf viewer:

  ```
  sat config --edit
  ```

- List the available applications (they come from the sat projects defined in `data/local.pyconf`):

  ```
  sat config --list
  ```

- Edit the configuration of an application:

  ```
  sat config <application> --edit
  ```

- Check the system dependencies (if any) used by the application:

  ```
  sat config <application> --check_system
  ```

- Copy an application configuration file into the user personal directory:

  ```
  sat config <application> --copy [new_name]
  ```

- Print the value of a configuration parameter.
  Use the automatic completion to get recursively the parameter names.
  Use *–no_label* option to get *only* the value, *without* label (useful in automatic scripts).
  Examples (with *SALOME-xx* as *SALOME-8.4.0*):

  ```
  # sat config --value <parameter_path>
  sat config --value .          # all the configuration
  sat config --value LOCAL
  sat config --value LOCAL.workdir

  # sat config <application> --value <parameter_path>
  sat config SALOME-xx --value APPLICATION.workdir
  sat config SALOME-xx --no_label --value APPLICATION.workdir
  ```

- Print in one-line-by-value mode the value of a configuration parameter,
  with its source *expression*, if any.
  This is a debug mode, useful for developers.
  Prints the parameter path, the source expression if any, and the final value:

  ```
  sat config SALOME-xx -g USER
  ```

  Note: And so, *not only for fun*, to get **all expressions** of configuration

  ```
  sat config SALOME-xx -g . | grep -e "-->"
  ```

- Print the patches that are applied:

  ```
  sat config SALOME-xx --show_patchs
  ```

- Print the properties available for an application:

  ```
  sat config SALOME-xx show_properties
  ```

- Get information on a product configuration:

```
# sat config <application> --info <product>
sat config SALOME-xx --info KERNEL
sat config SALOME-xx --info qt
```

2.2.3 Some useful configuration paths

Exploring a current configuration.

- **PATHS**: To get list of directories where to find files.
- **USER**: To get user preferences (editor, pdf viewer, web browser, default working dir).

sat commands:

```
sat config SALOME-xx -v PATHS
sat config SALOME-xx -v USERS
```

2.3 Command prepare

2.3.1 Description

The **prepare** command brings the sources of an application in the *sources application directory*, in order to compile them with the compile command.

The sources can be prepared from VCS software (*cvs, svn, git*), an archive or a directory.

Warning: When sat prepares a product, it first removes the existing directory, except if the development mode is activated. When you are working on a product, you need to declare in the application configuration this product in **dev** mode.

2.3.2 Remarks

VCS bases (git, svn, cvs)

The *prepare* command does not manage authentication on the cvs server. For example, to prepare modules from a cvs server, you first need to login once.

To avoid typing a password for each product, you may use a ssh key with passphrase, or store your password (in .cvspass or .gitconfig files). If you have security concerns, it is also possible to use a bash agent and type your password only once.

Dev mode

By default *prepare* uses *export* mode: it creates an image of the sources, corresponding to the tag or branch specified, without any link to the VCS base. To perform a *checkout* (svn, cvs) or a *git clone* (git), you need to declare the product in dev mode in your application configuration: edit the application configuration file (pyconf) and modify the product declaration:

```
sat config <application> -e
# and edit the product section:
#   <product> : {tag : "my_tag", dev : "yes", debug : "yes"}
```

The first time you will execute the *sat prepare* command, your module will be downloaded in *checkout* mode (inside the SOURCES directory of the application). Then, you can develop in this repository, and finally push them in the base when they are ready. If you type during the development process by mistake a *sat prepare* command, the sources in dev mode will not be altered/removed (unless you use -f option).

2.3.3 Usage

- Prepare the sources of a complete application in SOURCES directory (all products):

  ```
  sat prepare <application>
  ```

- Prepare only some modules:

  ```
  sat prepare <application>  --products <product1>,<product2> ...
  ```

- Prepare only some modules with a given property:

  ```
  # prepare only SALOME modules, not prerequisites
  ./sat prepare <application> --properties  is_SALOME_module:yes
  ```

- Use –force to force to prepare the products in development mode (this will remove the sources and do a new clone/checkout):

```
sat prepare <application> --force
```

- Use –force_patch to force to apply patch to the products in development mode (otherwise they are not applied):

```
sat prepare <application> --force_patch
```

- Prepare only products that are not present in SOURCES. This completion mode is used to complete the preparation when it was interrupted, or when the product list was increased:

```
sat prepare <application> --complete
```

2.3.4 Some useful configuration paths

Command *sat prepare* uses the *pyconf file configuration* of each product to know how to get the sources.

Note: to verify configuration of a product, and get name of this *pyconf files configuration*

```
sat config <application> --info <product>
```

- **get_method**: the method to use to prepare the module, possible values are cvs, git, archive, dir.
- **git_info** : (used if get_method = git) information to prepare sources from git.
- **svn_info** : (used if get_method = svn) information to prepare sources from svn.
- **cvs_info** : (used if get_method = cvs) information to prepare sources from cvs.
- **archive_info** : (used if get_method = archive) the path to the archive.
- **dir_info** : (used if get_method = dir) the directory with the sources.

2.4 Command compile

2.4.1 Description

The **compile** command allows compiling the products of a SALOME[1] application.

2.4.2 Usage

- Compile a complete application:

```
sat compile <application>
```

- Compile only some products:

```
sat compile <application> --products <product1>,<product2> ...
```

- Use *sat -t* to duplicate the logs in the terminal (by default the logs are stored and displayed with *sat log* command):

```
sat -t compile <application> --products <product1>
```

- Compile a module and its dependencies:

```
sat compile <application> --products med --with_fathers
```

- Compile a module and the modules depending on it (for example plugins):

```
sat compile <application> --products med --with_children
```

- Force the compilation of a module, even if it is already installed. This option clean the build before compiling:

```
sat compile <application> --products med --force
```

- Update mode, compile only git products which source has changed, including the dependencies. The option is not implemented for svn and cvs, only for git. One has to call sat prepare before, to check if git sources where modified. The mecanism is based upon git log -1 command, and the modification of the source directory date accordingly:

```
# update SALOME sources
./sat prepare <application> --properties  is_SALOME_module:yes

# only compile modules that has to be recompiled.
sat compile <application> --update
```

- Clean the build and install directories before starting compilation:

```
sat compile <application> --products GEOM  --clean_all
```

Note:

a warning will be shown if option *–products* is missing
(as it will clean everything)

- Clean only the install directories before starting compilation:

```
sat compile <application> --clean_install
```

- Add options for make:

[1] http://www.salome-platform.org

```
sat compile <application> --products <product> --make_flags <flags>
```

- Use the *–check* option to execute the unit tests after compilation:

```
sat compile <application> --check
```

- Remove the build directory after successful compilation (some build directory like qt are big):

```
sat compile <application> --products qt --clean_build_after
```

- Stop the compilation as soon as the compilation of a module fails:

```
sat compile <application> --stop_first_fail
```

- Do not compile, just show if products are installed or not, and where is the installation:

```
sat compile <application> --show
```

- Print the recursive list of dependencies of one (or several) products:

```
sat -v5 compile SALOME-master -p GEOM --with_fathers --show
```

2.4.3 Some useful configuration paths

The way to compile a product is defined in the *pyconf* file configuration. The main options are:

- **build_source** : the method used to build the product (cmake/autotools/script)
- **compil_script** : the compilation script if build_source is equal to "script"
- **cmake_options** : additional options for cmake.
- **nb_proc** : number of jobs to use with make for this product.
- **check_install** : allow to specify a list of paths (relative to install directory), that sat will check after installation. This flag allows to check if an installation is complete.
- **install_dir** : allow to change the default install dir. If the value is set to *'base'*, the product will by default be installed in sat base. Unless base was set to 'no' in application pyconf.

2.5 Command launcher

2.5.1 Description

The **launcher** command creates a SALOME launcher, a python script file to start SALOME[2].

2.5.2 Usage

- Create a launcher:

  ```
  sat launcher <application>
  ```

  Generate a launcher in the application directory, i.e `$APPLICATION.workdir`.

- Create a launcher with a given name (default name is `APPLICATION.profile.launcher_name`)

  ```
  sat launcher <application> --name ZeLauncher
  ```

  The launcher will be called *ZeLauncher*.

- Set a launcher which does not initialise the PATH variables:

  ```
  sat launcher <application>  --no_path_init
  ```

  In this case the launcher does not initialise the path variables (the default is to do it only for PATH, not for LD_LIBRARY_PATH, PYTHONPATH, etc).

- Create a generic launcher, which sets the environment (bash or bat) and call the exe given as argument:

  ```
  sat launcher <application> -e INSTALL/SALOME/bin/salome/salome.py -n salome.sh
  ```

  The launcher will be called salome.sh. It will source the environment and call `$APPLICATION.workdir`/INSTALL/SALOME/bin/salome/salome.py. The arguments given to salome.sh are transferred to salome.py.

- Set a specific resources catalog:

  ```
  sat launcher <application>  --catalog  <path of a salome resources catalog>
  ```

  Note that the catalog specified will be copied to the profile directory.

- Generate the catalog for a list of machines:

  ```
  sat launcher <application> --gencat <list of machines>
  ```

  This will create a catalog by querying each machine (memory, number of processors) with ssh.

- Generate a mesa launcher (if mesa and llvm are parts of the application). Use this option only if you have to use salome through ssh and have problems with ssh X forwarding of OpengGL modules (like Paravis):

  ```
  sat launcher <application> --use_mesa
  ```

2.5.3 Configuration

Some useful configuration paths:

- **APPLICATION.profile**
  - **product** : the name of the profile product (the product in charge of holding the application stuff, like logos, splashscreen)
  - **launcher_name** : the name of the launcher.

[2] http://www.salome-platform.org

2.6 Command log

2.6.1 Description

The **log** command displays sat log in a web browser or in a terminal.

2.6.2 Usage

- Show (in a web browser) the log of the commands corresponding to an application:

  ```
  sat log <application>
  ```

- Show the log for commands that do not use any application:

  ```
  sat log
  ```

- The –terminal (or -t) display the log directly in the terminal, through a CLI[3] interactive menu:

  ```
  sat log <application> --terminal
  ```

- The –last option displays only the last command:

  ```
  sat log <application> --last
  ```

- To access the last compilation log in terminal mode, use –last_compile option:

  ```
  sat log <application> --last_compile
  ```

- The –clean (int) option erases the n older log files and print the number of remaining log files:

  ```
  sat log <application> --clean 50
  ```

2.6.3 Some useful configuration paths

- **USER**
  - **browser** : The browser used to show the log (by default *firefox*).
  - **log_dir** : The directory used to store the log files.

[3] https://en.wikipedia.org/wiki/Command-line_interface

2.7 Command environ

2.7.1 Description

The **environ** command generates the environment files used to run and compile your application (as SALOME[4] is an example).

Note: these files are **not** required, sat sets the environment itself, when compiling. And so does the salome launcher.

These files are useful when someone wants to check the environment. They could be used in debug mode to set the environment for *gdb*.

The configuration part at the end of this page explains how to specify the environment used by sat (at build or run time), and saved in some files by *sat environ* command.

2.7.2 Usage

- Create the shell environment files of the application:

```
sat environ <application>
```

- Create the environment files of the application for a given shell. Options are bash, bat (for windows), tcl, cfg (the configuration format used by SALOME):

```
sat environ <application> --shell [bash|bat|cfg|tcl|all]
```

- Use a different prefix for the files (default is 'env'):

```
# This will create file <prefix>_launch.sh, <prefix>_build.sh
sat environ <application> --prefix <prefix>
```

- Use a different target directory for the files:

```
# This will create file env_launch.sh, env_build.sh
# in the directory corresponding to <path>
sat environ <application> --target <path>
```

- Generate the environment files only with the given products:

```
# This will create the environment files only for the given products
# and their prerequisites.
# It is useful when you want to visualise which environment uses
# sat to compile a given product.
sat environ <application> --product <product1>,<product2>, ...
```

- Generate tcl modules for use with *environment-modules* package.

```
sat environ <application> --shell tcl
```

Use this command to generate tcl modules associated to a module base. The production of a module base is triggered when the flag *base* in the application pyconf is set to a value not equal to *yes*.

```
APPLICATION :
{
    ...
    # trigger the production of a environment module base which name is salome9
    base : 'salome9'
}
```

[4] http://www.salome-platform.org

In this example, the module base will be produced in *BASE/apps/salome9*, and the tcl modules associated in the directory tcl *BASE/apps/modulefiles/salome9*. Later, it will be possible to enable these modules with the shell command *module use –append .../SAT/BASE/modulefiles*.

2.7.3 Configuration

The specification of the environment can be done through several mechanisms.

1. For salome products (the products with the property `is_SALOME_module` as `yes`) the environment is set automatically by sat, in respect with SALOME requirements.
2. For other products, the environment is set with the use of the environ section within the pyconf file of the product. The user has two possibilities, either set directly the environment within the section, or specify a python script which will be used to set the environment programmatically.

Within the section, the user can define environment variables. He can also modify PATH variables, by appending or prepending directories. In the following example, we prepend *<install_dir>/lib* to `LD_LIBRARY_PATH` (note the *left first* underscore), append *<install_dir>/lib* to `PYTHONPATH` (note the *right last* underscore), and set `LAPACK_ROOT_DIR` to *<install_dir>*:

```
environ :
{
  _LD_LIBRARY_PATH : $install_dir + $VARS.sep + "lib"
  PYTHONPATH_ : $install_dir + $VARS.sep + "lib"
  LAPACK_ROOT_DIR : $install_dir
}
```

It is possible to distinguish the build environment from the launch environment: use a subsection called *build* or *launch*. In the example below, `LD_LIBRARY_PATH` and `PYTHONPATH` are only modified at run time, not at compile time:

```
environ :
{
  build :
  {
    LAPACK_ROOT_DIR : $install_dir
  }
  launch :
  {
    LAPACK_ROOT_DIR : $install_dir
    _LD_LIBRARY_PATH : $install_dir + $VARS.sep + "lib"
    PYTHONPATH_ : $install_dir + $VARS.sep + "lib"
  }
}
```

3. The last possibility is to set the environment with a python script. The script should be provided in the *products/env_scripts* directory of the sat project, and its name is specified in the environment section with the key `environ.env_script`:

```
environ :
{
  env_script : 'lapack.py'
}
```

Please note that the two modes are complementary and are both taken into account. Most of the time, the first mode is sufficient.

The second mode can be used when the environment has to be set programmatically. The developer implements a handle (as a python method) which is called by sat to set the environment. Here is an example:

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

```
import os.path
import platform

def set_env(env, prereq_dir, version):
    env.set("TRUST_ROOT_DIR",prereq_dir)
    env.prepend('PATH', os.path.join(prereq_dir, 'bin'))
    env.prepend('PATH', os.path.join(prereq_dir, 'include'))
    env.prepend('LD_LIBRARY_PATH', os.path.join(prereq_dir, 'lib'))
    return
```

sat defines four handles:

- **set_env(env, prereq_dir, version)** : used at build and run time.
- **set_env_launch(env, prereq_dir, version)** : used only at run time (if defined!)
- **set_env_build(env, prereq_dir, version)** : used only at build time (if defined!)
- **set_native_env(env)** : used only for native products, at build and run time.

2.8 Command clean

2.8.1 Description

The **clean** command removes products in the *source, build, or install* directories of an application. Theses directories are usually named `SOURCES, BUILD, INSTALL`.

Use the options to define what directories you want to suppress and to set the list of products.

2.8.2 Usage

- Clean all previously created *build* and *install* directories (example application as *SALOME_xx*):

```
# take care, is long time to restore, sometimes
sat clean SALOME-xx --build --install
```

- Clean previously created *build* and *install* directories, only for products with property *is_SALOME_module*:

```
sat clean SALOME-xxx --build --install \
                     --properties is_SALOME_module:yes
```

2.8.3 Available options

- **–products** : Products to clean.
- **–properties** :

  Filter the products by their properties.
  Syntax: `--properties <property>:<value>`

- **–sources** : Clean the product source directories.
- **–build** : Clean the product build directories.
- **–install** : Clean the product install directories.
- **–generated** : Clean source, build and install directories for generated products.
- **–package** : Clean the application package directory.
- **–all** : Clean the product source, build and install directories.
- **–sources_without_dev** :

  Do not clean the products in development mode,
  (they could have VCS[5] commits pending).

2.8.4 Some useful configuration paths

No specific configuration.

[5] https://en.wikipedia.org/wiki/Version_control

2.9 Command package

2.9.1 Description

The **package** command creates a SALOME archive (usually a compressed Tar[6] file .tgz). This tar file is used later to install SALOME on other remote computer.

Depending on the selected options, the archive includes sources and binaries of SALOME products and prerequisites.

Usually utility *sat* is included in the archive.

Note: By default the package includes the sources of prerequisites and products. To select a subset, use the *–without_property* or *–with_vcs* options.

2.9.2 Usage

- Create a package for a product (example as *SALOME_xx*):

  ```
  sat package SALOME_xx
  ```

  This command will create an archive named `SALOME_xx.tgz` in the working directory (`USER.workDir`). If the archive already exists, do nothing.

- Create a package with a specific name:

  ```
  sat package SALOME_xx --name YourSpecificName
  ```

Note: By default, the archive is created in the working directory of the user (`USER.workDir`).

If the option *–name* is used with a path (relative or absolute) it will be used.

If the option *–name* is not used and binaries (prerequisites and products) are included in the package, the OS[7] architecture will be appended to the name (example: `SALOME_xx-CO7.tgz`).

Examples:

```
# Creates SALOME_xx.tgz in $USER.workDir
sat package SALOME_xx

# Creates SALOME_xx_<arch>.tgz in $USER.workDir
sat package SALOME_xx --binaries

# Creates MySpecificName.tgz in $USER.workDir
sat package SALOME_xx --name MySpecificName
```

- Force the creation of the archive (if it already exists):

  ```
  sat package SALOME_xx --force
  ```

- Include the binaries in the archive (products and prerequisites):

  ```
  sat package SALOME_xx --binaries
  ```

  This command will create an archive named `SALOME_xx _<arch>.tgz` where <arch> is the OS architecture of the machine.

[6] https://en.wikipedia.org/wiki/Tar_(computing)
[7] https://en.wikipedia.org/wiki/Operating_system

- Do not delete Version Control System (VCS[8]) information from the configuration files of the embedded sat:

  ```
  sat package SALOME_xx --with_vcs
  ```

  The version control systems known by this option are CVS[9], SVN[10] and Git[11].

2.9.3 Some useful configuration paths

No specific configuration.

[8] https://en.wikipedia.org/wiki/Version_control
[9] https://fr.wikipedia.org/wiki/Concurrent_versions_system
[10] https://en.wikipedia.org/wiki/Apache_Subversion
[11] https://git-scm.com

2.10 Command generate

2.10.1 Description

The **generate** command generates and compiles SALOME modules from cpp modules using YACSGEN.

Note: This command uses YACSGEN to generate the module. It needs to be specified with *–yacsgen* option, or defined in the product or by the environment variable `$YACSGEN_ROOT_DIR`.

2.10.2 Remarks

- This command will only apply on the CPP modules of the application, those who have both properties:

  ```
  cpp : "yes"
  generate : "yes"
  ```

- The cpp modules are usually computational components, and the generated module brings the CORBA layer which allows distributing the component on remote machines. cpp modules should conform to YACSGEN/hxx2salome requirements (please refer to YACSGEN documentation).

2.10.3 Usage

- Generate all the modules of a product:

  ```
  sat generate <application>
  ```

- Generate only specific modules:

  ```
  sat generate <application> --products <list_of_products>
  ```

  Remark: modules which don't have the *generate* property are ignored.

- Use a specific version of YACSGEN:

  ```
  sat generate <application> --yacsgen <path_to_yacsgen>
  ```

2.11 Command init

2.11.1 Description

The **init** command manages the sat local configuration (which is stored in the data/local.pyconf file). It allows to initialise the content of this file.

2.11.2 Usage

- A sat project provides all the pyconf files relative to a project (salome for example). Use the *–add_project* command to add a sat project locally, in data/local.pyconf (by default sat comes without any project). It is possible to add as many projects as required.

```
sat init --add_project <path/to/a/sat/project/project.pyconf>
```

- If you need to remove a sat project from the local configuration, use the *–reset_projects* command to remove all projects and then add the next ones with *–add_project*:

```
sat init --reset_projects
sat init --add_project <path/to/a/new/sat/project/project.pyconf>
```

- By default the product archives are stored locally within the directory containing sat, in a subdirectory called ARCHIVES. If you want to change the default, use the *–archive_dir* option:

```
sat init --archive_dir  <local/path/where/to/store/product/archives>
```

- sat enables a **base** mode, which allows to mutualize product builds between several applications. By default, the mutualized builds are stored locally within the directory containing sat, in a subdirectory called BASE. To change the default, use the *–base* option:

```
sat init --base <local/path/where/to/store/product/mutualised/product/builds>
```

- In the same way, you can use the *–workdir* and *–log_dir* commands to change the default directories used to store the application builds, and sat logs:

```
sat init --workdir <local/path/where/to/store/applications>
sat init --log_dir <local/path/where/to/store/sat/logs>
```

2.11.3 Some useful configuration paths

All the sat init commands update the local pyconf sat file data/local.pyconf. The same result can be achieved by editing the file directly. The content of data/local.pyconf is dumped into two sat configuration variables:

- **LOCAL**: Contains notably all the default paths in the fields archive_dir, base, log_dir and workdir.
- **PROJECTS**: The field project_file_paths contains all the project files that have been included with –add_project option.

sat commands:

```
sat config -v LOCAL
sat config -v PROJECTS
```

2.12 Command template

2.12.1 Description

The **template** command generates the sources of a SALOME module out of a template. SAT provides two templates for SALOME 9 :

- PythonComponent : a complete template for a SALOME module implemented in python (with data model and GUI).
- CppComponent : a template for a SALOME component implemented in C++, with a code coupling API.

2.12.2 Usage

- Create a python SALOME module:

```
sat template --name <product_name> --template PythonComponent\
             --target <my_directory>
```

Create in *my_directory* a ready to use SALOME module implemented in python. The generated module can then be adapted to the needs, and pushed in a git repository.

2.13 Command application

2.13.1 Description

The **application** command is used to create a virtual SALOME application. This command use appli_gen tool to generate the virtual application. It uses symbolic links and do not work on windows platform.

2.13.2 Usage

- Create an application:

  ```
  sat application <application>
  ```

  Create the virtual application directory in the salomeTool application directory `$APPLICATION.workdir`.

- Give a name to the application:

  ```
  sat application <application> --name <my_application_name>
  ```

  Remark: this option overrides the name given in the virtual_app section of the configuration file `$APPLICATION.virtual_app.name`.

- Change the directory where the application is created:

  ```
  sat application <application> --target <my_application_directory>
  ```

- Set a specific SALOME resources catalog (it will be used for the distribution of components on distant machines):

  ```
  sat application <application> --catalog <path_to_catalog>
  ```

  Note that the catalog specified will be copied to the application directory.

- Generate the catalog for a list of machines:

  ```
  sat application <application> --gencat machine1,machine2,machine3
  ```

  This will create a catalog by querying each machine through ssh protocol (memory, number of processor) with ssh.

- Generate a mesa application (if mesa and llvm are parts of the application). Use this option only if you have to use salome through ssh and have problems with ssh X forwarding of OpengGL modules (like Paravis):

  ```
  sat application <application> --use_mesa
  ```

2.13.3 Some useful configuration paths

The virtual application can be configured with the virtual_app section of the configuration file.

- **APPLICATION.virtual_app**
  - **name** : name of the launcher (to replace the default runAppli).
  - **application_name** : (optional) the name of the virtual application directory, if missing the default value is `$name + _appli`.

CHAPTER

THREE

RELEASE NOTES

3.1 SAT version 5.8.0

3.1.1 Release Notes, May 2021

New features and improvements

New key git_options

This new key was introduced in order to be able to use specific *git clone* options when getting sources with sat. This was motivated by paraview, which sometimes requires the option –recursive. This git_options key should be added in the git_info section of the product:

```
git_info:
{
    repo : "https://gitlab.kitware.com/paraview/paraview-superbuild"
    repo_dev : $repo
    git_options: ' --recursive '
}
```

Completion of system_info section with system specifics

This development allows defining more precisely the system prerequisites that are required (the name of some packages change from one linux distribution to the other, with this development it is now possible to specialise system prerequisites for each distribution). In addition a new product called *salome_system* was added, which includes all *implicit* system prerequisites (prerequisites that are not specifically managed by sat, and that should be installed on the user machine). The command *sat config –check_system* is now quite exhaustive. The syntax of this new section is:

```
system_info :
{
    rpm : ["dbus-libs", "gmp", ...  "zlib"]
    rpm_dev : ["openssl-devel", "tbb-devel", ... "libXft-devel"]
    apt : ["libbsd0", ...   "zlib1g"]
    apt_dev : ["libssl-dev", "gcc", ...]

    # specific to some plateform(s)
    "CO7" :
    {
        rpm_dev : ["perl"]
    }
    "CO8-FD30-FD32" :
    {
        rpm_dev : ["perl-interpreter"]
    }
}
```

Change log

This chapter does not provide the complete set of changes included, only the most significant changes are listed.

Artifact	Description
sat #24027	Print a warning for users executing .sh environment scripts outside bash (zsh,...)
spns #20662	For sat compile –update : do not change the date of source directory for tags (only for branches)
sat #18868	Implement the recompilation of archives produced with BASE mode
sat #18867	Implement the recompilation of an archive produced with git links –with_vcs)
sat #20601	bug fix the case where the the name of a product differ from the pyconf name
sat #20061	Use a new optimized algorithm for the calculation of dependencies (much faster)
sat #20089	bug fix for python 2.6
sat #20490	suppress false positive return of sat prepare if tag doesn't exist
sat #20391	Update Exception API message method to conform with python3
sat #20460	debug sat config –check_system on debian system

3.2 SAT version 5.7.0

3.2.1 Release Notes, November 2020

New features and improvements

New field build_depend used in product configuration files

In order to improve the setting of the environment at run-time and compile-time, a new field was introduced in the product configuration files : *build_depend*. This field allows the user to specify which products are required for the build (use the field *build_depend*) , and which one are used at runtime (use the former field *depend*). If a product is used at both build and runtime it is only declared (like before) in the *depend* field (it is the case for example of graphviz which is used at build-time by doxygen, and at run-time by YACS).

These two fields are used by sat accordingly to the context for the dependencies evaluation. Here is the example of med prerequisites (medfile.pyconf), which depends at runtime on hdf5 and python, and requires cmake for the compilation:

```
...
depend : ["hdf5", "Python"]
build_depend : ["cmake"]
```

New option –update for sat compile

The time spent to compile salome and its 60 prerequisites is regularly increasing... and can exceed ten hours on slow computers! It is therefore problematic and expensive in term of resources to recompile completely salome everyday. The **–update** option was introduced to allow compiling only the products which source code has changed. This option is **only implemented for git** (not for svn and cvs). To use the option, one has to call *sat prepare* before. this call will get new sources, and will allow sat checking if the source code was modified since the last compilation. The mechanism is based upon git *log -1* command, and the modification of the source directory date accordingly:

```
# update SALOME sources and set the date of the source directories of git
# products accordingly: to the last commit
./sat prepare <application> --properties  is_SALOME_module:yes

# only compile products that has to be recompiled.
sat compile <application> --update
```

This option can also be mixed with *–proterties* option, to avoid recompiling salome prerequisites:

```
# only compile SALOME products which source code has changed
sat compile <application> --update --properties is_SALOME_module:yes
```

sat do not reinitialise PATH, LD_LIBRARY_PATH and PYTHONPATH variables anymore

The last versions of sat were reinitialising the PATH, LD_LIBRARY_PATH and PYTHONPATH variables before the compilation. The objective was to avoid bad interaction with the user environment, and ensure that sat environment was correctly set for build. Alas this policy causes difficulties, notably on cluster where people sometimes need to use an alternate compiler and have to set it through *module load* command. It was therefore decided to suppress this policy.

Please note that apart from this use case (set the environment of a specific compiler) it is strongly advised to use sat with a clean environment! Note also that it is possible to manage with sat a compiler as a product, and therefore delegate the setting of this compiler to sat. When you have the choice it is a better option.

Change log

This chapter does not provide the complete set of changes included, only the most significant changes are listed.

Artifact	Description
sat #19888	suppress at compile time the reinit if PATH, LD_LIBRARY_PATH and PYTHONPATH
sat #19894	use the product configuration file to assert if a product was compiled or not. (before sat was using the product directory, which was in some cases error prone)

3.3 SAT version 5.6.0

3.3.1 Release Notes, July 2020

New features and improvements

Checking of system dependencies

SALOME depends upon some system prerequisites. Recent examples are tbb and openssl. For these products SALOME made the choice not to embed the prerequisite, but to rely on the system version. SAT has now the capacity to check for the system dependencies in two ways:

- **sat prepare** command will return an error if system prerequisites are not installed.
- **sat config** has now an option **–check_system** that list all the system prerequisites with their status.

Removing build dependencies from binary archives

SALOME archive are getting fat. In order to reduce the size of binary archives, the management by sat of the build prerequisites was modified. build prerequisites declared with the property **compile_time : "yes"** are not included anymore in binary archives.

New option -f –force for sat compile

This option can be used to **force** the recompilation of products. It is an alternative to –clean_all, which do not work properly with the single_dir mode (it will erase the complete PRODUCTS directory, which is usually not expected!

Change log

This chapter does not provide the complete set of changes included, only the most significant changes are listed.

Artifact	Description
sat #18501	bad management of rm_products functionality in archives
sat #18546	for products installed in BASE, replace in directory name / by _ to avoid the creation if a directory
sat #19109	more robust choice of the package manager to check system dependencies
sat #18720	add option –use-mesa in automatic completion
sat #19192	don't remove PRODUCTS dir when compilation fails
sat #19234	remove build products from bin archives, better management of their environment
sat #19218	correct out_dir_Path substitution for appended variables
sat #18350	-f option for sat compile : force the recompilation
sat #18831	sat compile –clean_all : do all the cleaning, then compile (bug correction with single_dir mode)
sat #18653	replace platform.linux_distribution by distro.linux_distribution for python 3.8+

3.4 SAT version 5.5.0

3.4.1 Release Notes, November 2019

New features and improvements

pip mode for python modules

This new mode was introduced in order to simplify the management of the python products (which number is constantly raising years after years...). It is triggered by two properties within the application configuration file:

```
pip : 'yes'
pip_install_dir : 'python'
```

The first property activates the use of pip for all the products which have themselves the pip property activated (it concerns twenty products). The second property specifies that the products should be installed directly in python directory, and not in their own specific directory. This second property is useful on windows platform to reduce the length of python path.

After several tests and iterations, the following management was adopted:

- sat prepare <application> does nothing for pip products (because at prepare time we don't have python compiled, and the use of native pip may not be compatible).
- sat compile <application> use the pip module installed in python to get pip archives (wheels), store them in local archive directory, and install them either in python directory, or in the product directory (in accordance to pip_install_dir property).

single directory mode

This new mode was introduced in order to get shorter path on windows platform. It is triggered by the property **single_install_dir** within the application configuration file:

```
single_install_dir : "yes"
```

When activated, all the products which have themselves the property **single_install_dir** are installed in a common directory, called PRODUCTS.

Generalization of sat launcher command

sat launcher command was extended to generate launchers based on an executable given as argument with **-e** option:

```
sat launcher <application> -n salome.sh -e INSTALL/SALOME/bin/salome.py
```

The command generates a launcher called salome.sh, which sets the environment, and launches the INSTALL/SALOME/bin/salome.py.

optimization of sat compile

For a complete compilation of salome, sat compile command was spending more than three minutes to calculate the dependencies and the order in which the products should be compiled. The algorithm used was clumsy, and confused. It was therefore completely rewritten using a topological sorting. The products order calculation takes now less than one second.

new management of sections in product configuration files

The sections defined in products are used to specify the variations in the way products are built. Depending upon the tag or version of the product, sat chooses one of these sections and sets the product definition according to it. With time, the number of sections increased a lot. And it is not easy to visualise the differences between these sections, as they often are identical, except few variations. With the windows version, new sections are introduced to manage windows specifics.

Therefore the need of a new mode for managing sections arises, that would be simpler, more concise, and help the comprehension. This new mode is called **incremental**, and is triggered by the property **incremental** within the default section of the product:

```
default:
{
    ....
    properties:
    {
        incremental : "yes"
    }
    ...
}
```

When this mode is defined, the definition of the product is defined incrementally, by taking into account the reference (the default section) and applying to it corrections defined in the other incremental sections. Depending upon the case, several sections may be taken into account, in a predefined order:

- the default section, which contains the reference definition
- on windows platform, the default_win section if it exists
- the section corresponding to the tag. the algorithm to determine this section remains unchanged (what changes is that in incremental mode the section only define deltas, not the complete definition)
- on windows platform, if it exists the same section postfixed with "_win".

Here is as an example the incremental definition used for boost products. For version 1.49 of boost, we extend the definition because we need to apply a patch:

```
default :
{
    name : "boost"
    build_source : "script"
    compil_script :  $name + $VARS.scriptExtension
    get_source : "archive"
    environ :
    {
       env_script : $name + ".py"
    }
    depend : ['Python' ]
    opt_depend : ['openmpi' ]
    patches : [ ]
    source_dir : $APPLICATION.workdir + $VARS.sep + 'SOURCES' + $VARS.sep + $name
    build_dir : $APPLICATION.workdir + $VARS.sep + 'BUILD' + $VARS.sep + $name
    install_dir : 'base'
    properties :
    {
        single_install_dir : "yes"
        incremental : "yes"
    }
}
```

```
version_1_49_0:
{
    patches : [ "boost-1.49.0.patch" ]
}
```

Suppression of the global "no_base" flag in application configuration

no_base : "no" is not interpreted anymore in application pyconf. One has to use the **base** flag. The possible values are:

- **yes** : all the products go into the base
- **no** : no product goes into the base

The complete usage rule of bases is explained in the documentation.

Change log

This chapter does not provide the complete set of changes included, only the most significant changes are listed.

Artifact	Description
spns #8544	The documentation has been improved!
spns #16894	clean the temp directory at the end of sat package
sat #12965	optimisation of sat compile : better, simpler and faster algo for dependencies!
sat #17206	Use pip to manage python modules
sat #17137	check_install functionality improvement : uses linux expending shell rules and interprets environment variables
sat #8544	Update and improvement of documentation
sat # 8547	Generalisation of sat launcher command (new option –exe to specify which exe should be launched after setting the environment
sat #17357	New field "rm_products" to blacklist products in overwrite section of appli pyconf
sat #17194	Parametrisation of the value of INSTALL and BINARIES directories (in src/internal_config/salomeTools.pyconf)
sat #17639	Warning when sat is launcher with python3
sat #17359	New incremental mode for the definition of products
sat #17766 sat #17848	The environment of products is now loaded in the order of product dependencies. To treat correctly dependencies in the environment
sat #17955	No unit tests for native products
	SAT_DEBUG and SAT_VERBOSE environment variables are now available in the compilation, which can now forward the information and do the job!
sat #18392	Bug, binaries archives do not work when products are in base

3.5 SAT version 9.4.0

3.5.1 Release Notes, April, 2019

This version of sat was used to produce SALOME 9.3.0

New features and improvments

sat package

The sat package command has been completed and finalised, in order to manage standalone packages of sat, with or without an embedded project. Options **–ftp** and **–with_vcs** have been added, in order to reduce the size of salome project packages (without these options, the archive of the sat salome project is huge, as it includes all the prerequisites archives. The **–ftp** option allows pointing directly to salome ftp site, which provides the prerequisites archives. These are therefore not included. With the same approach, **–with_vcs** option specify an archive that

points directly to the git bases of SALOME. Sources of SALOME modules are therefore not embedded in the archive, reducing the size.

```
# produce a standalone archive of sat
sat package -t

# produce a HUGE standalone archive of sat with the salome project embedded.
sat package -t -p salome

# produce a small archive with sat and embedded salome project,
# with direct links to ftp server and git repos
sat package -t -p salome --ftp --with_vcs
```

repo_dev property

This new application property **repo_dev** was introduced to trigger the use of the development git repositories for all the git bases of an application. Before, the only way to use the development git repositories was to declare all products in dev mode. This was problematic, for example one had to use *–force_patch* option to apply patches, or to use *–force* option to reinstall sources.

The use of the development git repository is now triggered by declaring this new **repo_dev** property in the application. And products are declared in dev mode only if we develop them.

```
# add this section in an application to force the use of the development git bases
# (from Tuleap)
properties :
{
    repo_dev : "yes"
}
```

windows compatibility

The compatibility to windows platform has been improved. The calls to lsb_release linux command have been replaced by the use of python platform module. Also the module med has been renamed medfile, and module Homard has been renamed homard_bin, in order to avoid lower/upper case conflicts.

Change log

This chapter does not provide the complete set of changes included, only the most significant changes are listed.

Artifact	Description
sat #12099	Add a new field called check_install to verify the correct installation
sat #8607	Suppression of sat profile command, replaced by sat template command (AppModule)
69d6a69f43	Introduction of a new property called repo_dev to trigger the use of the dev git repository.
scs #13187	Update of PythonComponent template
sat #16728	Replace call to lsb_release by platform module
sat #13318 sat #16713	command sat package -t -p salome –ftp –with_vcs debug of sat packages containing sat and embedded projects
sat #16787	Rename product med by medfile and Homard by homard_bin

3.6 SAT version 5.3.0

3.6.1 Release Notes, February, 2019

New features and improvments

sat init

The command *sat init* has been finalized, with the addition of options **–add_project** and **–reset_projects**. It is now able to manage projects after an initial git clone of sat. The capacity is used by users installing sat from the git repositories:

```
# get sources of sat
git clone https://codev-tuleap.cea.fr/plugins/git/spns/SAT.git sat

# get SAT_SALOME project (the sat project that contains the configuration of SALOME)
git clone https://codev-tuleap.cea.fr/plugins/git/spns/SAT_SALOME.git

# initialise sat with this project
sat init --add_project $(pwd)/SAT_SALOME/salome.pyconf
```

It is possible to initialise sat with several projects by calling several times *sat init –add_project*

sat prepare : git retry functionality

With large git repositories (>1GB) *git clone* command may fail. To decrease the risk, sat prepare will now retry three times the *git clone* function in case of failure.

Reset of LD_LIBRARY_PATH and PYTHONPATH before setting the environment

Every year, a lot of problems occur, due to users (bad) environment. This is most of the time caused by the presence (out-of-date) .bashrc files. To prevent these (time-consuming) problems, sat now reset LD_LIBRARY_PATH and PYTHONPATH variables before setting the environment thus avoiding side effects. Users who wish anayway to start SALOME with a non empty LD_LIBRARY_PATH or PYTHONPATH may comment the reset in salome launcher or in env_launch.sh file.

New option –complete for sat prepare

This option is used when an installation is interrupted or incomplete. It allows downloading only the sources of missing products

```
# only get sources of missing products (i.e products not present in INSTALL dir)
git prepare SALOME-master -c
```

** New option –packages for sat clean**

SALOME packages are big... It is usefull to be able to clean them whith this new option.

```
# remove packages present in PACKAGES directory of SALOME-master
git clean SALOME-master --packages
```

Global configuration keys “debug”, “verbose” and “dev” in applications

These new keys can be defined in applications in order to trigger the debug, verbose and dev mode for all products. In the following example, the SALOME-master application will be compiled in debug mode (use of **-g** flag), but with no verbosity. Its products are not in development mode.

```
APPLICATION :
{
    name : 'SALOME-master'
    workdir : $LOCAL.workdir + $VARS.sep + $APPLICATION.name + '-' + $VARS.dist
    tag : 'master'
    dev : 'no'
    verbose :'no'
    debug : 'yes'
    ...
}
```

Change log

This chapter does not provide the complete set of changes included, only the most significant changes are listed.

Artifact	Description
sat #16548 sat #8566	Finalisation of sat init command (options -add_project and –reset_projects)
sat #12994	new git retry functionality for sat prepare : give three trials in case of failures
sat #8581	traceability : save tag of sat and its projects
sat #8588	reset LD_LIBRARY_PATH and PYTHONPATH before launching SALOME
sat #9575	Improvement of the DISTENE licences management (notably for packages)
sat #8597	Implementation of option sat prepare -c (–complete) for preparing only the sources that are not yet installed
sat #8655	implementation of option sat clean –packages
sat #8532 sat #8594	sat log : rename option –last_terminal in –last_compile Extension of sat log –last_compile to the logs of make check
sat #13271	hpc mode triggered by product “hpc” key in state of MPI_ROOT variable
sat #8606	sat generate clean old directories before a new generation
sat #12952	Add global keys “debug”, “verbose” and “dev” to manage globally these modes for all the products of an application
sat #8523	protection of call to ssh on windows platform

3.7 SAT version 5.2.0

3.7.1 Release Notes, December, 2018

This version of salomeTool was used to produce SALOME 9.2.0

New features and improvments

Generalisation of –properties option

Wherever the –product option was available (to select products), an option **–properties** has been implemented, to offer a alternative way to select products, based on theur property. For example

```
# get only the sources of SALOME modules, not the prerequisites
sat prepare SALOME-9.2.0 --properties is_SALOME_module:yes
```

Compatibility with python 3

salomeTool is still meant to run under python2. But it magages now the build of applications runninfg under python3. It includes: * the generation of python3 launcher, * the testing of applications under python 3 (*sat test* command).

New syntax for the naming of sections in product pyconf

The old syntax is still supported for compatibility, but the new one, more explicit, is recommended.

```
# all tags from 8.5.0 to 9.2.1, with variants (8, 8_5_0, 8.5, V8, v8.6, etc)
_from_8_5_0_to_9_2_1
{
    ....
```

mesa launcher

When salome is used on a remote machine, the use of openGL 3 is not compatible with X11 forwarding (ssh -X). This cause segmentation faults when the 3D viewers are used. For people who have no other choice and need to use ssh (it may be useful for testing SALOME on a client remote machine), we provide in the packages a mesa launcher mesa_salome. It will avoid the segmentation faults, at the price of poor performance : it should only be used in this case! If performance is required, a solution based on the use of VirtualGL and TurboVNC/x2go would be recommended. But this requires some configuration of the tools to be done as root. To activate the production of the mesa launcher, use the application property **mesa_launcher_in_package**:

```
# activate the production of a launcher using mesa library
properties :
{
    mesa_launcher_in_package : "yes"
}
```

Change log

This chapter does not provide the complete set of changes included, only the most significant changes are listed.

Artifact	Description
sat #8577	Add a –properties option everywhere useful (whenever there is a –product option)
sat #8471	Windows portage necessary to produce SALOME 8.2.0 on Windows
sat #13031	Python 3 compatibility
sat #8561	New syntax for sections names in products pyconf files
sat #11056	New application property mesa_launcher_in_package to activate the production of a mesa launcher in the package

3.8 SAT version 5.1.0

3.8.1 Release Notes, June, 2018

This version of sat was used to produce SALOME 8.5.0

New features and improvements

sat compile : management of a verbose and debug option

The verbose and debug option for cmake products is activated through two new keys introduced in application configuration files : **debug** and **verbose**. **debug** option will trigger the tranmmission of *-DCMAKE_VERBOSE_MAKEFILE=ON* to cmake, while **verbose** option will transmit *-DCMAKE_VERBOSE_MAKEFILE=ON*. The new options can be activated for a selected products (within the option dictionnary associated to the products):

```
# for KERNEL compilation : specify to cmake a debug compilation with verbosity
KERNEL : {tag : "V7_8_0", base : "yes", debug : "yes", verbose : "yes"}
```

These two options can also be activated globally, for all products, through golbal keys:

```
specify to cmake a debug compilation with verbosity for all products
APPLICATION :
{
    name : 'SALOME-master'
    workdir : $LOCAL.workdir + $VARS.sep + $APPLICATION.name + '-' + $VARS.dist
    tag : 'master'
    verbose :'yes'
    debug : 'yes'
    ...
}
```

Implementation of salome test functionality with sat launcher

sat launcher is now able to launch salome tests (before the development, only virtual applications where able to launch salome tests). SALOME module was adapted to hold the tests (through links to SALOME module test directories). Notably, the results and logs of the test are stored in *INSTALL/SALOME/bin/salome/test*.

```
# display help for salome test command
salome test -h

# show available tests (without running them)
salome test -N

# run tests
salome test
```

Change log

This chapter does not provide the complete set of changes included, only the most significant changes are listed.

Artifact	Description
sat #8908	sat compile : management of a verbose and debug options
sat #8560	Define handles set_env_build and set_env_launch to be able to specialise env
sat #8638	Improve information printed by –show option of sat compile
sat #8911	Implementation of salome test with sat launcher in connection with SALOME module
sat #11056	Generation of new salome launcher with mesa with sat launcher command
sat #11028	Use of a new property “configure_dependency” to manage the dependency of all salome modules to CONFIGURATION module
sat #10569	Debug and improvement of products filters in sat commands
sat #8576 sat #8646 sat #8605 sat #8646 sat #8576	Improve if messages displayed by sat compile command Improve management of errors

3.9 SAT version 5.0.0

3.9.1 Release Notes, January 2018

This version of sat was used to produce SALOME 8.4.0

New features and improvments

Complete re-engineering

Separation of the tool and the configurations files

Clarification of main use cases

Simplification and uniformisation of tha API

Local prerequisite base

Use of properties associated to products

1. facilitate the development of services (example has_unit_tests property)
2. help the user to select specific products with a given property

New type of packages

robust and easy to install!