sysmocom

sysmocom - s.f.m.c. GmbH

Osmo-GSM-Tester Manual

by Neels Hofmeyr and Pau Espin Pedrol

DRAFT

Copyright © 2017-2020 sysmocom - s.f.m.c. GmbH

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with the Invariant Sections being just 'Foreword', 'Acknowledgements' and 'Preface', with no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

The Asciidoc source code of this manual can be found at https://gitea.osmocom.org/cellular-infrastructure/osmo-gsm-manuals

HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
1	April 13, 2017	Initial version.	NH

Contents

1 WARNING: Work in Progress **1**

2 Introduction **1**

3 Trial: Binaries to be Tested **2**

4 Configuration **3**

4.1 Configuration files and directories . . . 3

4.1.1 *main.conf* . . . 3

4.1.2 *state_dir* . . . 4

4.1.3 *suites_dir* . . . 4

4.1.3.1 *suite.conf* . . . 5

4.1.4 *scenarios_dir* . . . 6

4.1.4.1 *scenario conf file* . . . 6

4.1.5 *resources.conf* . . . 7

4.1.6 *default-suites.conf* . . . 8

4.1.7 *defaults.conf* . . . 8

4.2 Schemas . . . 9

4.2.1 Schema *main config* . . . 9

4.2.2 Schema *resources* . . . 9

4.2.3 Schema *want* . . . 10

4.2.4 Schema *config* . . . 10

4.2.5 Schema *all* . . . 10

4.3 Example Setup . . . 10

4.3.1 Typical Invocations . . . 11

5 Resource Resolution **11**

5.1 Resource Reservation for Concurrent Trials . . . 12

5.2 Understanding config parsing process . . . 12

6 Test API **17**

6.1 Test verdict . . . 18

7 Installation **18**

7.1 Trial Builder . . . 18

7.1.1 Osmocom Build Dependencies . . . 18

7.1.2 Add Build Jobs . . . 18

7.2 Main Unit . . . 20

7.2.1 Create *jenkins* User . . . 20

7.2.2 Install Java on Main Unit . . . 20
7.2.3 Allow SSH Access from Jenkins Master . . . 20
7.2.4 Add Jenkins Slave . . . 21
7.2.5 Add Run Job . . . 22
7.2.6 Install osmo-gsm-tester dependencies . . . 23
7.2.7 User Permissions . . . 24
7.2.7.1 Paths . . . 25
7.2.7.2 Allow DBus Access to ofono . . . 25
7.3 Slave Unit(s) . . . 25
7.3.1 Capture Packets . . . 26
7.3.2 Allow Core Files . . . 26
7.3.3 Allow Realtime Priority . . . 27
7.3.3.1 Allow capabilities: *CAP_NET_RAW*, *CAP_NET_ADMIN*, *CAP_SYS_ADMIN* . . . 27
7.3.4 UHD . . . 27
7.3.5 Log Rotation . . . 28
7.3.6 Install Scripts . . . 28

8 Hardware Choice and Configuration 28
8.1 SysmoBTS . . . 28
8.1.1 IP Address . . . 29
8.1.2 Allow Core Files . . . 29
8.1.3 Reboot . . . 29
8.1.4 SSH Access . . . 29
8.2 Modems . . . 29
8.3 osmo-bts-trx . . . 29

9 Ansible Setup 30

10 Docker Setup 30

11 Debugging 31
11.1 Logging level . . . 31
11.2 python debugger . . . 31
11.3 debug suite . . . 32

12 Troubleshooting 32
12.1 Format: YAML, and its Drawbacks . . . 32
12.2 Osmo-GSM-Tester not running but resources still allocated . . . 32

1 WARNING: Work in Progress

NOTE: Osmo-GSM-Tester is still under heavy development stage: some parts are still incomplete, and details can still change and move around as new features are added and improvements made.

2 Introduction

Osmo-GSM-Tester is a software to run automated tests on real GSM hardware, foremost to verify that ongoing Osmocom software development continues to work with various BTS models, while being flexibly configurable and extendable to work for other technologies, setups and projects. It can be used for instance to test a 3G or 4G network.

Osmo-GSM-Tester (python3 process) runs on a host (general purpose computer) named the *main unit*. It may optionally be connected to any number of *slave units*, which Osmo-GSM-Tester may use to orchestrate processes remotely, usually through SSH.

Hardware devices such as BTS, SDRs, modems, smart plugs, etc. are then connected to either the main unit or slaves units via IP, raw ethernet, USB or any other means·

The modems and BTS instances' RF transceivers are typically wired directly to each other via RF distribution chambers to bypass the air medium and avoid disturbing real production cellular networks. Furthermore, the setup may include adjustable RF attenuators to model various distances between modems and base stations.

Each of these devices, having each a different physical setup and configuration, supported features, attributes, etc., is referred in Osmo-GSM-Tester terminology as a *resource*. Each *resource* is an instance of *resource class*. A *resource_class* may be for instance a *modem* or a *bts*. For instance, an Osmo-GSM-Tester setup may have 2 *modem* instances and 1 *bts* instances. Each of these *resources* are listed and described in configuration files passed to Osmo-GSM-Tester, which maintains a pool of *resources* (available, in use, etc.).

Osmo-GSM-Tester typically receives from a jenkins build service the software or firmware binary packages to be used and tested. Osmo-GSM-Tester then launches a specific set of testsuites which, in turn, contain each a set of python test scripts. Each test uses the *testenv* API provided by Osmo-GSM-Tester to configure, launch and manage the different nodes and processes from the provided binary packages to form a complete ad-hoc GSM network.

Testsuites themselves contain configuration files to list how many resources it requires to run its tests. It also provides means to *filter* which kind of *resources* will be needed based on their attributes. This allows, for instance, asking Osmo-GSM-Tester to provide a *modem* supporting GPRS, or to provide a specific model of *bts* such as a nanoBTS. Testsuites also allow receiving *modifiers*, which overwrite some of the default values that Osmo-GSM-Tester itself or different *resources* use.

Moreover, one may want to run the same testsuite several tiems, each with different set of *resources*. For instance, one may want to run a testsuite with a sysmoBTS and later with a nanoBTS. This is supported by leaving the testsuite configuration generic enough and then passing *scenarios* to it, which allow applying extra *filters* or *modifiers*. Scenarios can also be combined to filter further or to apply further modifications.

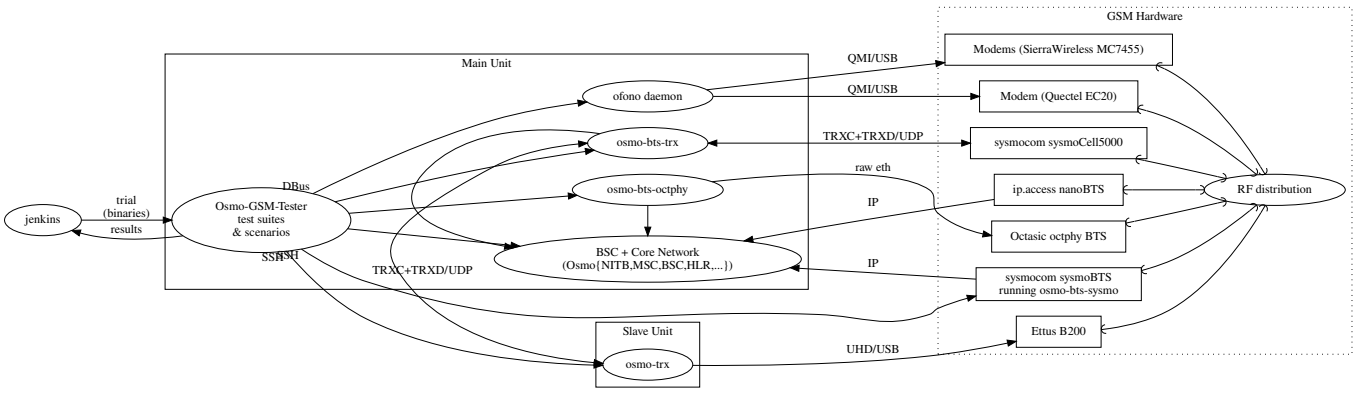

Figure 1: Sample osmo-gsm-tester node 2G setup

3 Trial: Binaries to be Tested

A trial is a set of pre-built sysroot archives to be tested. They are typically built by jenkins using the build scripts found in osmo-gsm-tester's source in the *contrib/* dir, see Section 7.2.4.

A trial comes in the form of a directory containing a number of *<inst-name>.*tgz* tar archives (containing different sysroots) as well as a *checksums.md5* file to verify the tar archives' integrity.

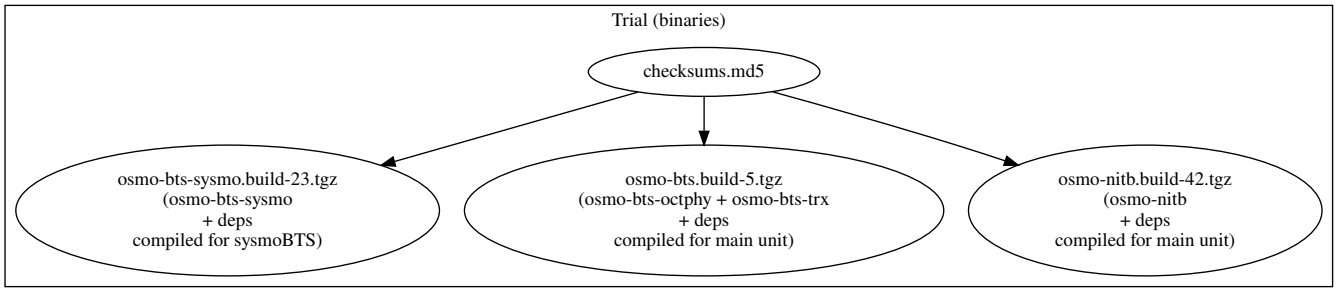

Figure 2: Example of a "trial" containing binaries built by a jenkins job

When the osmo-gsm-tester is invoked to run on such a trial directory, it will create a sub directory named *inst* and unpack the tar archives into it.

For each test run on this trial, a new subdirectory in the trial dir is created, named in the form of *run.<timestamp>*. A symbolic link *last-run* will point at the most recently created run dir. This run dir will accumulate:

- the rendered configuration files used to run the binaries
- stdout and stderr outputs of the binaries
- pcap files for processes doing relevant network communication
- a test log
- jenkins parsable XML (Junit) reports

The script in *contrib/jenkins-run.sh* takes care of related tasks such as

- creating the dir structure,
- generating md5 sums for the various tar.gz containing software builds to be tested,
- cleaning up after the build,
- saving extra logs such as journalctl output from ofonod,
- generating a final .tar.gz file with all the logs and reports to store as jenkins archives.

Osmo-GSM-Tester tests create objects to manage the allocated resources during test lifetime. These objects, in turn, usually run and manage processes started from the trial's sysroot binaries. Osmo-GSM-Tester provide APIs for those object classes to discover, unpack and run those binaries. An object class simply needs to request the name of the sysroot it wants to use (for instance *osmo-bsc*), and Osmo-GSM-Tester will take care of preparing everything and providing the sysroot path to it. It's a duty of the resource class to copy over the sysroot to the destination if the intention is to run the binary remotely on another host.

Moreover, Osmo-GSM-Tester supports working with several different versions of a given sysroot by means of storing them in different subdirectories, which are later referenced by an object's class *run_label* attribute named after that subdirectory. This way, for instance, a sysroot can be provided containing binaries linked against libraries present on a CentOS distribution, and

other sysroots with the same name can also be provided which are linked against different versions of CentOS, or a different distro like Debian, or even a different arch like ARM.

When seeking a sysroot of a given name *<inst-name>* in the *inst/* directory, Osmo-GSM-Tester will look for *tgz* files starting with the pattern *<inst-name>*. (up to the first dot). That means, suffixes are available for Osmo-GSM-Tester user to identify the content, for instance having an incrementing version counter or a commit hash. Hence, these example files are considered valid and will be selected by Osmo-GSM-Tester for *osmo-bsc*: *osmo-bsc.tgz*, *osmo-bsc.build-23.tgz*, *osmo-bsc.5f3e0dd2.tgz*, *osmo-bsc.armv7.build-2.tgz*. If either none or more than one valid file is found matching the pattern, an exception will be thrown. If a *run_label=foobar* is provided, Osmo-GSM-Tester will look up for the *tgz* in exactly the same way, but this time in *inst/foobar/* directory instead of *inst/*.

4 Configuration

4.1 Configuration files and directories

Find in below sub-sections all user-defined files and directories used by Osmo-GSM-Tester to run tests on a given setup.

4.1.1 *main.conf*

The main configuration file is basically a placeholder for Osmo-GSM-Tester to find paths to all other files and directories used to operate and run tests.

Osmo-GSM-Tester looks for the main configuration file in various standard paths in this order:

- *./main.conf* (Current Working Directory)
- *$HOME/.config/osmo-gsm-tester/main.conf*
- */usr/local/etc/osmo-gsm-tester/main.conf*
- */etc/osmo-gsm-tester/main.conf*

The config file location can also be set through *-c* command line argument, which then overrides the above locations.

Osmo-GSM-Tester expects to find the following configuration settings in *main.conf*:

- *state_dir*: Path to state_dir directory
- *trial_dir*: Path to trial directory to test against (overridden by cmdline argument)
- *suites_dir*: List of paths to suites_dir directories.
- *scenarios_dir*: List of paths to scenarios_dir directories (optional)
- *default_suites_conf_path*: Path to default-suites.conf file (optional)
- *defaults_conf_path*: Path to defaults.conf file (optional)
- *resource_conf_path*: Path to resources.conf file (optional)

Configuration settings holding a list of paths, such as *suites_dir* or *scenarios_dir*, are used to look up for paths in regular list of order, meaning first paths in list take preference over last ones. As a result, if a suite named *A* is found in several paths, the one on the first path in the list will be used.

These are described in detail in the following sections. If no value is provided for a given setting, sane default paths are used: For *state_dir*, */var/tmp/osmo-gsm-tester/state/* is used. All other files and directories are expected, by default, to be in the same directory as main.conf

Important
Relative paths provided in *main.conf* are parsed as being relative to the directory of that *main.conf* file itself, and not relative to the CWD of the Osmo-GSM-Tester process parsing it.

Sample main.conf file:

```
state_dir: '/var/tmp/osmo-gsm-tester/state'
suites_dir: [ '/usr/local/src/osmo-gsm-tester/suites' ]
scenarios_dir: [ './scenarios' ]
trial_dir: './trial'
default_suites_conf_path: './default-suites.conf'
defaults_conf_path: './defaults.conf'
resource_conf_path: './resources.conf'
```

4.1.2 *state_dir*

It contains global or system-wide state for osmo-gsm-tester. In a typical state dir you can find the following files:

last_used_*.state
: Contains stateful content spanning accross Osmo-GSM-Tester instances and runs. For instance, *last used msisdn number.state* is automatically (and atomically) increased every time osmo-gsm-tester needs to assign a new subscriber in a test, ensuring tests get unique msisdn numbers.

reserved_resources.state
: File containing a set of reserved resources by any number of osmo-gsm-tester instances (aka pool of allocated resources). Each osmo-gsm-tester instance is responsible to clear its resources from the list once it is done using them and are no longer reserved.

lock
: Lock file used to implement a mutual exclusion zone around any state files in the *state_dir*, to prevent race conditions between different Osmo-GSM-Tester instances running in parallel.

This way, several concurrent users of osmo-gsm-tester (ie. several osmo-gsm-tester processes running in parallel) can run without interfering with each other (e.g. using same ARFCN, same IP or same ofono modem path).

If you would like to set up several separate configurations (not typical), note that the *state_dir* is used to reserve resources, which only works when all configurations that share resources also use the same *state_dir*. It's also important to notice that since resources are stored in YAML dictionary form, if same physical device is described differently in several resources.conf files (used by different Osmo-GSM-Tester instances), resource allocation may not work as expected.

4.1.3 *suites_dir*

Suites contain a set of tests which are designed to be run together to test a set of features given a specific set of resources. As a result, resources are allocated per suite and not per test.

Tests for a given suite are located in the form of *.py* python scripts in the same directory where the suite.conf lays.

Tests in the same testsuite willing to use some shared code can do so by putting it eg. in *$suites_dir/$suitename/lib/testlib.py*:

```
#!/usr/bin/env python3
from osmo_gsm_tester.testenv import *

def my_shared_code(foo):
    return foo.bar()
```

and then in the test itself use it this way:

```
#!/usr/bin/env python3
from osmo_gsm_tester.testenv import *

import testlib
suite.test_import_modules_register_for_cleanup(testlib)
from testlib import my_shared_code

bar = my_shared_code(foo)
```

Sample suites_dir directory tree:

```
suites_dir/
|-- suiteA
|   |-- suite.conf
|   '-- testA.py
|-- suiteB
|   |-- testB.py
|   |-- testC.py
|   |-- lib
|   |   '-- testlib.py
|   '-- suite.conf
```

4.1.3.1 *suite.conf*

This file content is parsed using the Want schema.

On the resources section, it provides Osmo-GSM-Tester with the base restrictions (later to be further filtered by scenario files) to apply when allocating resources.

It can also override attributes for the allocated resources through the modifiers section (to be further modified by scenario files later on). Similarly it can do the same for general configuration options (no per-resource) through the config section.

The *schema* section allows defining a suite's own schema used to validate parameters passed to it later on through scenario files (See Section 4.1.4.1), and which can be retrieved by tests using the *tenv.config_suite_specific()* and *tenv.config_test_specific()* APIs. The first one will provide the whole dictionary under schema, while the later will return the dictionary immediatelly inside the former and matching the test name being run. For instance, if *tenv.config_test_specific()* is called from test *a_suite_test_foo.py*, the method will return the contents under dictionary with key *a_suite_test_foo*.

Sample suite.conf file:

```
resources:
  ip_address:
  - times: 9 # msc, bsc, hlr, stp, mgw*2, sgsn, ggsn, iperf3srv
  bts:
  - times: 1
  modem:
  - times: 2
    features:
    - gprs
    - voice
  - times: 2
    features:
    - gprs

config:
  bsc:
    net:
      codec_list:
      - fr1

schema:
```

```
  some_suite_parameter: 'uint'
  a_suite_test_foo:
    one_test_parameter_for_test_foo: 'str'
    another_test_parameter_for_test_foo: ['bool_str']

config:
  suite:
    <suite_name>:
      some_suite_parameter: 3
      a_suite_test_foo:
        one_test_parameter_for_test_foo: 'hello'
        timeout: 30 ❶
```

❶ The per-test *timeout* attribute is implicitly defined for all tests with type *duration*, and will trigger a timeout if test doesn't finish in time specified.

4.1.4 *scenarios_dir*

This dir contains scenario configuration files.

Sample scenarios_dir directory tree:

```
scenarios_dir/
|-- scenarioA.conf
'-- scenarioB.conf
```

4.1.4.1 *scenario conf file*

Scenarios define further constraints to serve the resource requests of a suite.conf, ie. to select specific resources from the general resource pool specified in resources.conf.

If only one resource is specified in the scenario, then the resource allocator assumes the restriction is to be applied to the first resource and that remaining resources have no restrictions to be taken into consideration.

To apply restrictions only on the second resource, the first element can be left emtpy, like:

```
resources:
  bts:
  - {}
  - type: osmo-bts-sysmo
```

On the *osmo_gsm_tester.py* command line and the default_suites.conf, any number of such scenario configurations can be combined in the form:

```
<suite_name>:<scenario>[+<scenario>[+...]]
```

e.g.

```
my_suite:sysmo+tch_f+amr
```

Parametrized scenario conf files:

Furthermore, scenario *.conf* files can be parametrized. The concept is similar to that of systemd's Template Unit Files. That is, an scenario file can be written so that some values inside it can be passed at the time of referencing the scenario name. The idea behind its existence is to re-use the same scenario file for a set of attributes which are changed and that can have a lot of different values. For instance, if a scenario is aimed at setting or filtering some specific attribute holding an integer value, without parametrized scenarios then a separate file would be needed for each value the user wanted to use.

A parametrized scenario file, similar to systemd Template Unit Files, contain the character @ in their file name, ie follow the syntax below:

```
scenario-name@param1,param2,param3,[...],paramN.conf
```

Then, its content can be written this way:

```
$ cat $scenario_dir/my-parametrized-scenario@.conf
resources:
  enb:
  - type: srsenb
    rf_dev_type: ${param1}
modifiers:
  enb:
    - num_prb: ${param2}
```

Finally, it can be referenced during Osmo-GSM-Tester execution this way, for instance when running a suite named *4g*:

```
- 4g:my-parametrized-scenario@uhd,6
```

This way Osmo-GSM-Tester when parsing the scenarios and combining them with the suite will

1. Find out it is parametrized (name contains @).
2. Split the name (*my-parametrized-scenario*) from the parameter list (param1=*uhd*, param2=*6*)
3. Attempt to match a *.conf* file fully matching name and parameters (hence specific content can be set for specific values while still using parameters for general values), and otherwise match only by name.
4. Generate the final scenario content from the template available in the matched *.conf* file.

Scenario to set suite/test parameters:

First, the suite needs to define its schema in its suite.conf file. Check Section 4.1.3.1 on how to do so.

For instance, for a suite named *mysuite* containing a test *a_suite_test_foo.py*, and containing this schema in its suite.conf file:

```
schema:
  some_suite_parameter: 'uint'
  a_suite_test_foo:
    one_test_parameter_for_test_foo: 'str'
    another_test_parameter_for_test_foo: ['bool_str']
```

One could define a parametrized scenario *myparamscenario@.conf* like this:

```
config:
  suite:
    mysuite:
      some_suite_parameter: ${param1}
      a_suite_test_foo:
        one_test_parameter_for_test_foo: ${param2}
        another_test_parameter_for_test_foo: ['true', 'false', 'false', 'true']
```

And use it in Osmo-GSM-Tester this way:

```
mysuite:myparamscenario@4,hello.conf
```

4.1.5 *resources.conf*

The *resources.conf* file defines which hardware is connected to the main unit, as well as which limited configuration items (like IP addresses or ARFCNs) should be used.

A *resources.conf* is validated by the resources schema. That means it is structured as a list of items for each resource type, where each item has one or more attributes — looking for an example, see Osmo-GSM-Tester subdirectory *doc/examples*.

Side note: at first sight it might make sense to the reader to rather structure e.g. the *ip_address* or *arfcn* configuration as *"arfcn: GSM-1800: [512, 514, . . .]"*,
but the more verbose format is chosen in general to stay consistent with the general structure of resource configurations, which the resource allocation algorithm uses to resolve required resources according to their traits. These configurations look cumbersome because they exhibit only one trait / a trait that is repeated numerous times. No special notation for these cases is available (yet).

4.1.6 *default-suites.conf*

The *default-suites.conf* file contains a YAML list of *suite:scenario+scenario+. . .* combination strings as defined by the *osmo-gsm-tester.py -s* commandline option. If invoking the *osmo-gsm-tester.py* without any suite definitions, the *-s* arguments are taken from this file instead. Each of these suite + scenario combinations is run in sequence.

A suite name must match the name of a directory in the suites_dir/ as defined by main.conf.

A scenario name must match the name of a configuration file in the scnearios_dir/ as defined by main.conf (optionally without the *.conf* suffix).

Sample default-suites.conf file:

```
- sms:sysmo
- voice:sysmo+tch_f
- voice:sysmo+tch_h
- voice:sysmo+dyn_ts
- sms:trx
- voice:trx+tch_f
- voice:trx+tch_h
- voice:trx+dyn_ts
```

4.1.7 *defaults.conf*

In Osmo-GSM-Tester object instances requested by the test and created by the suite relate to a specific allocated resource. That's not always the case, and even if it the case the information stored in resources.conf for that resource may not contain tons of attributes which the object class needs to manage the resource.

For this exact reason, the *defaults.conf* file exist. It contains a set of default attributes and values (in YAML format) that object classes can use to fill in the missing gaps, or to provide values which can easily be changed or overwritten by suite.conf or scenario.conf files through modifiers.

Each binary run by osmo-gsm-tester, e.g. *osmo-nitb* or *osmo-bts-sysmo*, typically has a configuration file template that is populated with values for a trial run. Hence, a suite.conf, scenario.conf or a resources.conf providing a similar setting always has precedence over the values given in a *defaults.conf*

Sample defaults.conf file:

```
nitb:
  net:
    mcc: 901
    mnc: 70
    short_name: osmo-gsm-tester-nitb
    long_name: osmo-gsm-tester-nitb
    auth_policy: closed
    encryption: a5_0

bsc:
  net:
    mcc: 901
    mnc: 70
    short_name: osmo-gsm-tester-msc
    long_name: osmo-gsm-tester-msc
    auth_policy: closed
    encryption: a5_0
```

```
    authentication: optional

msc:
  net:
    mcc: 901
    mnc: 70
    short_name: osmo-gsm-tester-msc
    long_name: osmo-gsm-tester-msc
    auth_policy: closed
    encryption: a5_0
    authentication: optional

bsc_bts:
  location_area_code: 23
  base_station_id_code: 63
  stream_id: 255
  osmobsc_bts_type: osmo-bts
  trx_list:
  - nominal_power: 23
    max_power_red: 0
    arfcn: 868
    timeslot_list:
    - phys_chan_config: CCCH+SDCCH4
    - phys_chan_config: SDCCH8
    - phys_chan_config: TCH/F_TCH/H_PDCH
    - phys_chan_config: TCH/F_TCH/H_PDCH
    - phys_chan_config: TCH/F_TCH/H_PDCH
    - phys_chan_config: TCH/F_TCH/H_PDCH
    - phys_chan_config: TCH/F_TCH/H_PDCH
    - phys_chan_config: TCH/F_TCH/H_PDCH
```

4.2 Schemas

All configuration attributes in Osmo-GSM-Tester are stored and provided as YAML files, which are handled internally mostly as sets of dictionaries, lists and scalars. Each of these configurations have a known format (set of keys and values), which is called *schema*. Each provided configuration is validated against its *schema* at parse time. Hence, *schemas* can be seen as a namespace containing a structured tree of configuration attributes. Each attribute has a schema type assigned which constrains the type of value it can hold.

There are several well-known schemas used across Osmo-GSM-Tester, and they are described in following sub-sections.

4.2.1 Schema *main config*

This schema defines all the attributes available in Osmo-GSM-Tester the main configuration file main.conf, and it is used to validate it.

4.2.2 Schema *resources*

This schema defines all the attributes which can be assigned to a *resource*, and it is used to validate the resources.conf file. Hence, the resources.conf contains a list of elements for each resource type. This schema is also used and extended by the want schema.

It is important to understand that the content in this schema refers to a list of resources for each resource class. Since a list is ordered by definition, it clearly identifies specific resources by order. This is important when applying filters or modifiers, since they are applied per-resource in the list. One can for instance apply attribute A to first resource of class C, while not applying it or applying another attribute B to second resources of the same class. As a result, complex forms can be used to filter and modify a list of resources required by a testsuite.

On the other hand, it's also important to note that lists for simple or scalar types are currently being treated as unordered sets, which mean combination of filters or modifiers apply differently. In the future, it may be possible to have both behaviors for scalar/simple types by using also the YAML *set* type in Osmo-GSM-Tester.

4.2.3 Schema *want*

This schema is basically the same as the resources one, but with an extra *times* attribute for each resource item. All *times* attributes are expanded before matching. For example, if a *suite.conf* requests two BTS, one may enforce that both BTS should be of type *osmo-bts-sysmo* in these ways:

```
resources:
  bts:
  - type: osmo-bts-sysmo
  - type: osmo-bts-sysmo
```

or alternatively,

```
resources:
  bts:
  - times: 2
    type: osmo-bts-sysmo
```

4.2.4 Schema *config*

This schema defines all the attributes which can be used by object classes or tests during test execution. The main difference between this schema and the resources schema is that the former contains configuration to be applied globally for all objects being used, while the later applies attributes to a specific object in the list of allocated resources. This schema hence allows setting attributes for objects which are not allocated as resources and hence not directly accessible through scenarios, like a BSC or an iperf3 client.

This schema is built dynamically at runtime from content registered by: - object classes registering their own attributes - test suite registering their own attributes through suite.conf and tests being able to later retrieve them through *testenv* API.

4.2.5 Schema *all*

This schema is basically an aggregated namespace for want schema and config schema, and is the one used by suite.conf and scenario.conf files. It contains these main element sections:::

- Section *resources*: Contains a set of elements validated with want schema. In suite.conf it is used to construct the list of requested resources. In scenario.conf, it is used to inject attributes to the initial suite.conf *resources* section and hence further restrain it.
- Section *modifiers*: Both in suite.conf and scenario.conf, values presented in here are injected into the content of the resources section after *resource* allocation, hereby overwriting attributes passed to the object class instance managing the specific *resource* (matches by resource type and list position). Since it is combined with the content of resources section, it is clear that the want schema is used to validate this content.
- Section *config*: Contains configuration attributes for Osmo-GSM-Tester object classes which are not *resources*, and hence cannot be configured with modifiers. They can overwrite values provided in the defaults.conf file. Content in this section follows the config schema.

4.3 Example Setup

Osmo-GSM-Tester comes with an example official setup which is the one used to run Osmocom's setup. There are actually two different available setups: a production one and an RnD one, used to develop Osmo-GSM-Tester itself. These two set ups share mostly all configuration, main difference being the resources.conf file being used.

All Osmo-GSM-Tester related configuration for that environment is publicly available in *osmo-gsm-tester.git* itself:

- main.conf: Available Available under *sysmocom/*, with its paths already configured to take required bits from inside the git repository directory.
- suites_dir: Available under *sysmocom/suites/*
- scenarios_dir: Available under *sysmocom/scenarios/*
- resources.conf: Available under *sysmocom/* as *resources.conf.prod* for Production setup and as *resources.conf.rnd* for the RnD setup. One must use a symbolic link to have it available as *resources.conf*.

There are also small sample setups under the *doc/examples/* directory to showcase how to set up different types of networks.

4.3.1 Typical Invocations

Each invocation of osmo-gsm-tester deploys a set of pre-compiled binaries for the Osmocom core network as well as for the Osmocom based BTS models. To create such a set of binaries, see Section 3.

Examples for launching test trials:

- Run the default suites (see default_suites.conf) on a given set of binaries from *path/to/my-trial* with main.conf available under a standard path:

```
osmo-gsm-tester.py path/to/my-trial
```

- Same as above, but run an explicit choice of *suite:scenario* combinations:

```
osmo-gsm-tester.py path/to/my-trial -s sms:sysmo -s sms:trx -s sms:nanobts
```

- Same as above, but run one *suite:scenario1+scenario2* combination, setting log level to *debug* and enabling logging of full python tracebacks, and also only run just the *mo_mt_sms.py* test from the suite, e.g. to investigate a test failure:

```
osmo-gsm-tester.py path/to/my-trial -s sms:sysmo+foobar -l dbg -T -t mo_mt
```

- Same as above, but tell Osmo-GSM-Tester to read the *main.conf* in specific directory *path/to/my/main.conf*:

```
osmo-gsm-tester.py -c path/to/my/main.conf path/to/my-trial -s sms:sysmo+foobar -l dbg -T - ↩
    t mo_mt
```

A test script may also be run step-by-step in a python debugger, see Section 11.

5 Resource Resolution

- A global configuration resources.conf defines which hardware is plugged to the Osmo-GSM-Tester setup, be it the main unit or any slave unit. This list becomes the *resource pool*.
- Each suite contains a number of test scripts. The amount of resources a test may use is defined by the test suite's suite.conf.
- Which specific modems, BTS models, NITB IP addresses etc. are made available to a test run is typically determined by suite.conf and a combination of scenario configurations — or picked automatically if not.

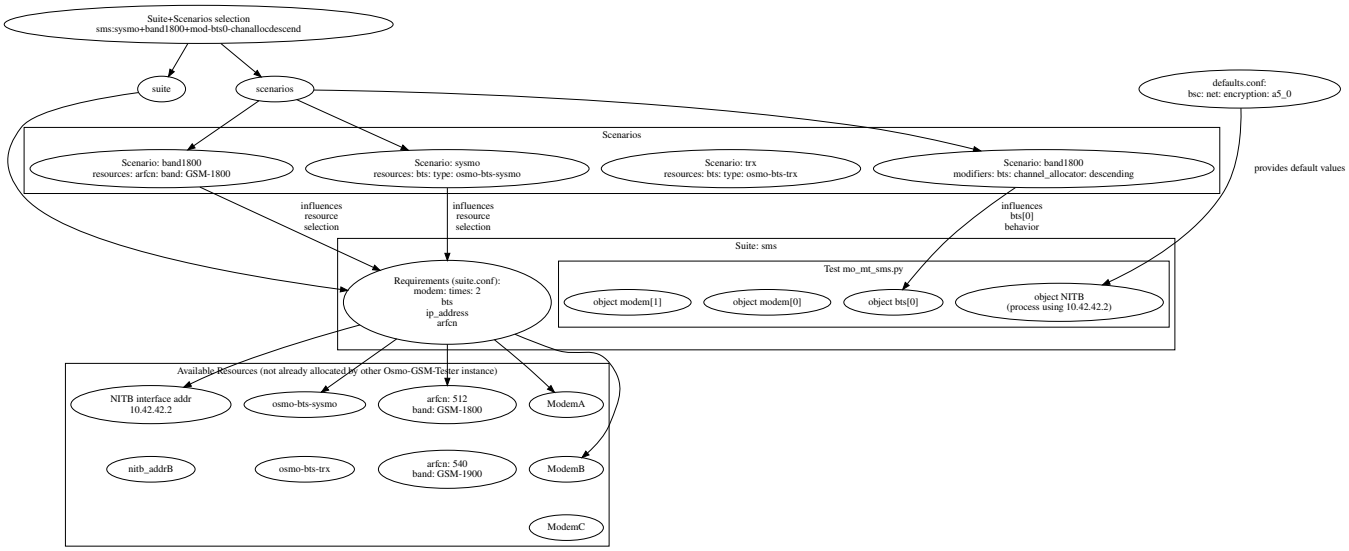

Figure 3: Example of how to select resources and configurations: scenarios may pick specific resources (here BTS and ARFCN), remaining requirements are picked as available (here two modems and a NITB interface)

5.1 Resource Reservation for Concurrent Trials

While a test suite runs, the used resources are noted in a global state directory in a reserved-resources file. This way, any number of trials may be run consecutively without resource conflicts. Any test trial will only use resources that are currently not reserved by any other test suite. The reservation state is human readable.

The global state directory is protected by a file lock to allow access by separate processes.

Also, the binaries from a trial are never installed system-wide, but are run with a specific *LD_LIBRARY_PATH* pointing at the trial's inst, so that several trials can run consecutively without conflicting binary versions. For some specific binaries which require extra permissions (such as osmo-bts-octphy requiring *CAP_NET_RAW*), *patchelf* program is used to modify the binary *RPATH* field instead because the OS dynamic linker skips *LD_LIBRARY_PATH* for binaries with special permissions.

Once a test suite run is complete, all its reserved resources are torn down (if the test scripts have not done so already), and the reservations are released automatically.

If required resources are unavailable, the test trial fails. For consecutive test trials, a test run needs to either wait for resources to become available, or test suites need to be scheduled to make sense. (← **TODO**)

5.2 Understanding config parsing process

- Data processing can be understood as operations on sets (Venn diagram)

```
$ src/osmo-gsm-tester.py "$TRIAL_JOB_DIR" -s 4g:srsenb-rftype@zmq+srsue-rftype@zmq+mod-enb- ↩
    nprb@6 -t ping.py
```

- First Suite and scenarios dictionaries (*resources*, *modifiers*, *config*) are combined (set union operation):

Table 1: Manual replica of *suite.py resource_requirements()* and *resource_modifiers()* methods

Table 1: (continued)

File	original	after *times* replication	python syntax	combined

4g	`run_node:`
	`- times: 1`
	`enb:`
	`- times: 1`
	`type: srsenb`
	`modem:`
	`- times: 2`
	`type: srsue`
		`run_node:`
		`- {}`
		`enb:`
		`- type: srsenb`
		`modem:`
		`- type: srsue`
		`- type: srsue`
			`'resources': {`
			`'run_node': [{}],`
			`'enb': [{'type': 'srsenb'}],`
			`'modem': [{'type: 'srsue'},`
			`{'type: 'srsue'}]`
			`}`
			`'modifiers': {}`
				`'resources': {`
				`'run_node': [{}],`
				`'enb': [{'type': 'srsenb'}],`
				`'modem': [{'type': 'srsue'},`
				`{'type': 'srsue'}]`
				`}`
				`'modifiers': {}`

srsenb-rftype@zmq	`resources:`
	`enb:`
	`- type: srsenb`
	`rf_dev_type: zmq`
		`resources:`
		`enb:`
		`- type: srsenb`
		`rf_dev_type: zmq`
			`'resources': {`
			`'enb': [{'type': 'srsenb',`
			`'rf_dev_type': 'zmq'}]`
			`}`
			`'modifiers': {}`
				`'resources': {`
				`'run_node': [{}],`
				`'enb': [{'type': 'srsenb',`
				`'rf_dev_type': 'zmq'}],`
				`'modem': [{'type': 'srsue'},`
				`{'type': 'srsue'}]`
				`}`
				`'modifiers': {}`

srsue-rftype@zmq	`resources:`
	`modem:`
	`- type: srsue`
	`rf_dev_type: zmq`
	`times: 1`
		`resources:`
		`modem:`
		`- type: srsue`
		`rf_dev_type: zmq`
			`'resources': {`
			`'modem': [{'type': 'srsue',`
			`'rf_dev_type': 'zmq'}]`
			`}`
			`'modifiers': {}`
				`'resources': {`
				`'run_node': [{}],`
				`'enb': [{'type': 'srsenb',`
				`'rf_dev_type': 'zmq'}],`
				`'modem': [{'type': 'srsue',`
				`'rf_dev_type': 'zmq'},`
				`{'type': 'srsue'}]`
				`}`
				`'modifiers': {}`

Table 1: (continued)

mod-enb-nprb@6	modifiers:
	enb:
	- ↩
	num_prb: 6
	times: ↩
	1
		modifiers:
		enb:
		- ↩
		num_prb: 6
			'resources': {}
			'modifiers': {
			'enb': [{'num_prb': ↩
			6}]
			}
				'resources': {
				'run_node': [{}],
				'enb': [{'type': ' ↩
				srsenb',
				'rf_dev_type ↩
				': 'zmq'}],
				'modem': [{'type': ' ↩
				srsue',
				' ↩
				rf_dev_type': 'zmq'},
				{'type': ' ↩
				srsue'}]
				}
				'modifiers': {
				'enb': [{'num_prb': ↩
				6}]
				}

- Second, the resulting *resources* set is used to match a set of resources from *resources.list* in order to allocate them (intersection of sets):

Table 2: Manual replica of *resource.py reserve()* method

resources.conf	*resources* filters	matched

Table 2: (continued)

run_node:
- run_type: ssh
run_addr: ↩
10.12.1.195
ssh_user: ↩
jenkins
ssh_addr: ↩
10.12.1.195
enb:
- label: srsENB- ↩
zmq
type: srsenb
rf_dev_type: zmq
remote_user: ↩
jenkins
addr: ↩
10.12.1.206
- label: srsENB- ↩
B200
type: srsenb
rf_dev_type: uhd
rf_dev_args: " ↩
type=b200,serial=317B9FE"
remote_user: ↩
jenkins
addr: ↩
10.12.1.206
modem:
- label: srsUE- ↩
zmq_1
type: srsue
rf_dev_type: zmq
remote_user: ↩
jenkins
addr: ↩
10.12.1.195
imsi: ↩
'001010123456789'
ki: '001123'
- label: srsUE- ↩
zq_2
type: srsue
rf_dev_type: zmq
remote_user: ↩
jenkins
addr: ↩
10.12.1.180
imsi: ↩
'001010123456781'
ki: '001124'
	'resources': {
	'run_node': ↩
	[{}],
	'enb': [{'type': ↩
	'srsenb',
	' ↩
	'rf_dev_type': 'zmq'}],
	'modem': [
	{'type ↩
	': 'srsue',
	' ↩
	'rf_dev_type': 'zmq'},
	{'type ↩
	': 'srsue'}
	]
	}
		'resources': {
		'run_node': [{'run_type': 'ssh',
		'run_addr': ↩
		'10.12.1.195',
		'ssh_user': 'jenkins',
		'ssh_addr': ↩
		'10.12.1.195'}],
		'enb': [{'label': 'srsENB-zmq',
		'type': 'srsenb',
		'rf_dev_type': 'zmq',
		'remote_user': 'jenkins',
		'addr': 10.12.1.206}],
		'modem': [
		{'label': 'srsUE-zmq_1',
		'type': 'srsue',
		'remote_user': jenkins,
		'addr': '10.12.1.195',
		'imsi': '001010123456789'
		'ki': '001123',
		'rf_dev_type': 'zmq'},
		{'label': 'srsUE-zmq_2',
		'type': 'srsue',
		'remote_user': jenkins,
		'addr': '10.12.1.180',
		'imsi': '001010123456781'
		'ki': '001124'}
		]
		}

- Finally, modifiers are applied on top of the combined configuration before being passed to the python class managing it:

Table 3: Also done by *resource.py reserve()* method after matching resources

Matched resources	modifiers	Result

'resources': {
'run_node': [{' ↩
run_type': 'ssh',
' ↩
run_addr': '10.12.1.195',
' ↩
ssh_user': 'jenkins',
' ↩
ssh_addr': '10.12.1.195'}],
'enb': [{'label ↩
': 'srsENB-zmq',
'type': ↩
'srsenb',
' ↩
rf_dev_type': 'zmq',
' ↩
remote_user': 'jenkins',
'addr': ↩
10.12.1.206}],
'modem': [
{' ↩
label': 'srsUE-zmq_1',
'type ↩
': 'srsue',
' ↩
remote_user': jenkins,
'addr ↩
': '10.12.1.195',
'imsi ↩
': '001010123456789'
'ki': ↩
'001123',
' ↩
rf_dev_type': 'zmq'},
{' ↩
label': 'srsUE-zmq_2',
'type ↩
': 'srsue',
' ↩
remote_user': jenkins,
'addr ↩
': '10.12.1.180',
'imsi ↩
': '001010123456781'
'ki': ↩
'001124'}
]
}
	'modifiers': {
	'enb': [{' ↩
	num_prb': 6}]
	}
		'resources': {
		'run_node': [{'run_type': 'ssh',
		'run_addr': ↩
		'10.12.1.195',
		'ssh_user': 'jenkins',
		'ssh_addr': ↩
		'10.12.1.195'}],
		'enb': [{'label': 'srsENB-zmq',
		'type': 'srsenb',
		'rf_dev_type': 'zmq',
		'remote_user': 'jenkins',
		'addr': '10.12.1.206',
		'num_prb': 6}],
		'modem': [
		{'label': 'srsUE-zmq_1',
		'type': 'srsue',
		'remote_user': jenkins,
		'addr': 10.12.1.195,
		'imsi': '001010123456789'
		'ki': '001123',
		'rf_dev_type': 'zmq'},
		{'label': 'srsUE-zmq_2',
		'type': 'srsue',
		'remote_user': jenkins,
		'addr': 10.12.1.180,
		'imsi': '001010123456781'
		'ki': '001124'}
		]
		}

Warning
Right now algorithms based on lists of scalar/simple types being unordered vs complex types (dictionaries, list) being ordered. Other ways can be supported by explicitly using *set* type from yaml in lists of scalars.

6 Test API

All tests run by Osmo-GSM-Tester are python script files. On top of usual python standard features, Osmo-GSM-Tester provides a set of public APIs and tools that these tests can use in order to interact with the core of Osmo-GSM-Tester, like creating object classes, run processes synchronously or asynchronously, wait for events, retrieve specific configuration, etc. This section aims at documenting the most relevant tools available for tests.

First of all, it is important to avoid blocking out of the core's main event loop in the test, since doing that will prevent Osmo-GSM-Tester core functionalities to work properly, such as control of asynchronous processes.

To get access to those functionalities, a test must import a test environment previously prepared by Osmo-GSM-Tester before the test was started:

```
#!/usr/bin/env python3
from osmo_gsm_tester.testenv import *
```

After the test environment is imported, aome usual functionalities are available directly under the global scope. Specially noticeable is the existence of object *tenv*, which provides access to most of the functionalities.

The test can simply ask Osmo-GSM-Tester to sleep some time, while giving control back to Osmo-GSM-Tester core's mainloop:

```
sleep(3) # sleep for 3 seconds
```

One can also wait for events in the background, for instance launch a child process locally in the same host and wait for its termination:

```
proc = process.Process('process_description_name', working_dir_to_store_logs, 'sleep 4') # ❶
tenv.remember_to_stop(proc) # ❷
proc.launch() # ❸
proc.wait() # ❹
```

❶ Create process object. This line doesn't yet runs it.

❷ Make sure the core will kill the process if this test fails

❸ Start process asynchronously

❹ wait until process is done. One could waiting generically here too: *wait(proc.terminated)*

If running asynchronously is not needed, one can run synchronously in an easy way:

```
proc = process.Process('process_description_name', working_dir_to_store_logs, 'sleep 4')
proc.launch_sync()
```

One can also log output using either the regular *print* function from python, or using Osmo-GSM-Tester specific functions available:

```
log('this is a regular log message')
dbg('this is a dbg message, only printed on outputs where dbg is enabled')
err('outputs log message for non-expected events')
print('this is the same as log()')
```

The test also gains access to suite and/or test specific configuration through different APIs:

```
test_config = tenv.config_test_specific()
threshold = int(test_config.get('threshold', 2))
suite_config = tenv.config_suite_specific()
foobar = suite_config['foobar']
```

A test requiring a really specific config file for an object class it is going to run can provide its own template files by overlaying an own directory containing them on top of the usual default directory where object class templates are (*osmo-gsm-tester.git/src/osmo_gsm_tester/obj/templates/*):

```
tenv.set_overlay_template_dir(os.path.join(os.path.dirname(__file__), 'mytemplatedir'))
```

Several tests in a suite can also share code by using some APIs provided by subdirectory in the suite directory where the test belongs to.

```
# File containing function foobar() available under ${suite_dir}/lib/testlib.py:
import testlib
tenv.test_import_modules_register_for_cleanup(testlib)
from testlib import foobar
```

For a complete set of features and how to use them, one can have a look at real examples present in Osmo-GSM-Tester git repository under the *sysmocom/* directory. Besides those, have a look too a *testenv.py* file, which implements the *tenv* object available to tests.

6.1 Test verdict

In general, a test reaching the end of the file and returning control to Osmo-GSM-Tester core will be flagged as a successful test (PASS).

If an exception is thrown from within the test file and propagated to Osmo-GSM-Tester, the test will be considered as failed and Osmo-GSM-Tester will store all failure related information from the caught exception.

7 Installation

7.1 Trial Builder

The Trial Builder is the jenkins build slave (host) building all sysroot binary packages used later by Osmo-GSM-Tester to run the tests. It's purpose is to build the which the Osmo-GSM-Tester runner job can fetch.

7.1.1 Osmocom Build Dependencies

Each of the jenkins builds requires individual dependencies. This is generally the same as for building the software outside of osmo-gsm-tester and will not be detailed here. For the Osmocom projects, refer to http://osmocom.org/projects/cellular-infrastructure/wiki/Build_from_Source . Be aware of specific requirements for BTS hardware: for example, the osmo-bts-sysmo build needs the sysmoBTS SDK installed on the build slave, which should match the installed sysmoBTS firmware.

7.1.2 Add Build Jobs

There are various jenkins-build-* scripts in osmo-gsm-tester/contrib/, which can be called as jenkins build jobs to build and bundle binaries as artifacts, to be run on the osmo-gsm-tester main unit and/or BTS hardware.

Be aware of the dependencies, as hinted at in Section 7.1.1.

While the various binaries could technically be built on the osmo-gsm-tester main unit, it is recommended to use a separate build slave, to take load off of the main unit.

Please note nowadays we set up all the osmocom jenkins jobs (including Osmo-GSM-Tester ones) using *jenkins-job-builder*. You can find all the configuration's in Osmocom's *osmo-ci.git* files 'jobs/osmo-gsm-tester-*.yml. Explanation below on how to set up jobs manually is left as a reference for other projects.

On your jenkins master, set up build jobs to call these scripts — typically one build job per script. Look in contrib/ and create one build job for each of the BTS types you would like to test, as well as one for the *build-osmo-nitb*.

These are generic steps to configure a jenkins build job for each of these build scripts, by example of the jenkins-build-osmo-nitb.sh script; all that differs to the other scripts is the "osmo-nitb" part:

- *Project name*: "osmo-gsm-tester_build-osmo-nitb"
  (Replace *osmo-nitb* according to which build script this is for)
- *Discard old builds*
  Configure this to taste, for example:
  - *Max # of build to keep*: "20"
- *Restrict where this project can be run*: Choose a build slave label that matches the main unit's architecture and distribution, typically a Debian system, e.g.: "linux_amd64_debian8"
- *Source Code Management*:
  - *Git*
    - *Repository URL*: "https://gitea.osmocom.org/cellular-infrastructure/osmo-gsm-tester"
    - *Branch Specifier*: "*/master"
    - *Additional Behaviors*
      - *Check out to a sub-directory*: "osmo-gsm-tester"
- *Build Triggers*
  The decision on when to build is complex. Here are some examples:
  - Once per day:
    Build periodically: "H H * * *"
  - For the Osmocom project, the purpose is to verify our software changes. Hence we would like to test every time our code has changed:
    - We could add various git repositories to watch, and enable *Poll SCM*.
    - On jenkins.osmocom.org, we have various jobs that build the master branches of their respective git repositories when a new change was merged. Here, we can thus trigger e.g. an osmo-nitb build for osmo-gsm-tester everytime the master build has run:
      Build after other projects are built: "OpenBSC"
    - Note that most of the Osmocom projects also need to be re-tested when their dependencies like libosmo* have changed. Triggering on all those changes typically causes more jenkins runs than necessary: for example, it rebuilds once per each dependency that has rebuilt due to one libosmocore change. There is so far no trivial way known to avoid this. It is indeed safest to rebuild more often.
- *Build*
  - *Execute Shell*

    ```
    #!/bin/sh
    set -e -x
    ./osmo-gsm-tester/contrib/jenkins-build-osmo-nitb.sh
    ```

    (Replace *osmo-nitb* according to which build script this is for)
- *Post-build Actions*
  - *Archive the artifacts*: "*.tgz, *.md5"
    (This step is important to be able to use the built binaries in the run job below.)

Tip
When you've created one build job, it is convenient to create further build jobs by copying the first one and, e.g., simply replacing all "osmo-nitb" with "osmo-bts-trx".

7.2 Main Unit

The main unit is a general purpose computer that orchestrates the tests. It runs the core network components, controls the modems and so on. This can be anything from a dedicated production rack unit to your laptop at home.

This manual will assume that tests are run from a jenkins build slave, by a user named *jenkins* that belongs to group *osmo-gsm-tester*. The user configuration for manual test runs and/or a different user name is identical, simply replace the user name or group.

Please, note installation steps and dependencies needed will depend on lots of factors, like your distribution, your specific setup, which hardware you plan to support, etc.

This section aims at being one place to document the rationale behind certain configurations being done in one way or another. For an up to date step by step detailed way to install and maintain the Osmocom Osmo-GSM-Tester setup, one will want to look at the ansible scripts section.

7.2.1 Create *jenkins* User

On the main unit, create a jenkins user:

```
useradd -m jenkins
```

7.2.2 Install Java on Main Unit

To be able to launch the Jenkins build slave, a Java RE must be available on the main unit. For example:

```
apt-get install default-jdk
```

7.2.3 Allow SSH Access from Jenkins Master

Create an SSH keypair to be used for login on the osmo-gsm-tester. This may be entered on the jenkins web UI; alternatively, use the jenkins server's shell:

Login on the main jenkins server shell and create an SSH keypair, for example:

```
# su jenkins
$ mkdir -p /usr/local/jenkins/keys
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/jenkins/.ssh/id_rsa): /usr/local/jenkins/keys/ ↩
    osmo-gsm-tester-rnd
Enter passphrase (empty for no passphrase): <enter a passphrase>
Enter same passphrase again: <enter a passphrase>
Your identification has been saved in /usr/local/jenkins/keys/osmo-gsm-tester-rnd
Your public key has been saved in /usr/local/jenkins/keys/osmo-gsm-tester-rnd.pub.
The key fingerprint is:
...
```

Copy the public key to the main unit, e.g. copy-paste:

```
cat /usr/local/jenkins/keys/osmo-gsm-tester-rnd.pub
# copy this public key
```

On the main unit:

```
mkdir ~jenkins/.ssh
cat > ~jenkins/.ssh/authorized_keys
# paste above public key and hit Ctrl-D
chown -R jenkins: ~jenkins/.ssh
```

Make sure that the user running the jenkins master accepts the main unit's host identification. There must be an actual RSA host key available in the known_hosts file for the jenkins master to be able to log in. Simply calling ssh and accepting the host key as usual is not enough. Jenkins may continue to say "Host key verification failed".

To place an RSA host key in the jenkins' known_hosts file, you may do:

On the Jenkins master:

```
main_unit_ip=10.9.8.7
ssh-keyscan -H $main_unit_ip >> ~jenkins/.ssh/known_hosts
chown jenkins: ~jenkins/.ssh/known_hosts
```

Verify that the jenkins user on the Jenkins master has SSH access to the main unit:

```
su jenkins
main_unit_ip=10.9.8.7
ssh -i /usr/local/jenkins/keys/osmo-gsm-tester-rnd jenkins@$main_unit_ip
exit
```

7.2.4 Add Jenkins Slave

In the jenkins web UI, add a new build slave for the osmo-gsm-tester:

- *Manage Jenkins*
  - *Manage Nodes*
    - *New Node*
      - Enter a node name, e.g. "osmo-gsm-tester-1"
        (the "-1" is just some identification in case you'd like to add another setup later).
      - *Permanent Agent*

Configure the node as:

- *# of executors*: 1
- *Remote root directory*: "/home/jenkins"
- *Labels*: "osmo-gsm-tester"
  (This is a general label common to all osmo-gsm-tester build slaves you may set up in the future.)
- *Usage*: *Only build jobs with label expressions matching this node*
- *Launch method*: *Launch slave agents via SSH*
  - *Host*: your main unit's IP address
  - *Credentials*: choose *Add / Jenkins*
    - *Domain*: *Global credentials (unrestricted)*
    - *Kind*: *SSH Username with private key*
    - *Scope*: *Global*
    - *Username*: "jenkins"
      (as created on the main unit above)
    - *Private Key*: *From a file on Jenkins master*
      - *File*: "/usr/local/jenkins/keys/osmo-gsm-tester-rnd"
    - *Passphrase*: enter same passphrase as above
    - *ID*: "osmo-gsm-tester-1"
    - *Name*: "jenkins for SSH to osmo-gsm-tester-1"

The build slave should be able to start now.

7.2.5 Add Run Job

This is the jenkins job that runs the tests on the GSM hardware:

- It sources the artifacts from jenkins' build jobs.
- It runs on the osmo-gsm-tester main unit.

Sample script to run Osmo-GSM-Tester as a jenkins job can be found in *osmo-gsm-tester.git* file *contrib/jenkins-run.sh*.

Please note nowadays we set up all the osmocom jenkins jobs (including Osmo-GSM-Tester ones) using *jenkins-job-builder*. You can find all the configuration's in Osmocom's *osmo-ci.git* files 'jobs/osmo-gsm-tester-*.yml. Explanation below on how to set up jobs manually is left as a reference for other projects.

Here is the configuration for the run job:

- *Project name*: "osmo-gsm-tester_run"
- *Discard old builds*
  Configure this to taste, for example:
  - *Max # of build to keep*: "20"
- *Restrict where this project can be run*: "osmo-gsm-tester"
  (to match the *Label* configured in Section 7.2.4).
- *Source Code Management*:
  - *Git*
    - *Repository URL*: "https://gitea.osmocom.org/cellular-infrastructure/osmo-gsm-tester"
    - *Branch Specifier*: "*/master"
    - *Additional Behaviors*
      - *Check out to a sub-directory*: "osmo-gsm-tester"
      - *Clean before checkout*
- *Build Triggers*
  The decision on when to build is complex. For this run job, it is suggested to rebuild:
  - after each of above build jobs that produced new artifacts:
    Build after other projects are built: "osmo-gsm-tester_build-osmo-nitb, osmo-gsm-tester_build-osmo-bts-sysmo, osmo-gsm-tester_build-osmo-bts-trx"
    (Add each build job name you configured above)
  - as well as once per day:
    Build periodically: "H H * * *"
  - and, in addition, whenever the osmo-gsm-tester scripts have been modified:
    Poll SCM: "H/5 * * * *"
    (i.e. look every five minutes whether the upstream git has changed)
- *Build*
  - Copy artifacts from each build job you have set up:
    - *Copy artifacts from another project*
      - *Project name*: "osmo-gsm-tester_build-osmo-nitb"
      - *Which build*: *Latest successful build*
      - enable *Stable build only*
      - *Artifacts to copy*: "*.tgz, *.md5"
    - Add a separate similar *Copy artifacts...* section for each build job you have set up.

- *Execute Shell*

```
#!/bin/sh
set -e -x

# debug: provoke a failure
#export OSMO_GSM_TESTER_OPTS="-s debug -t fail"

PATH="$PWD/osmo-gsm-tester/src:$PATH" \
  ./osmo-gsm-tester/contrib/jenkins-run.sh
```

  Details:

  * The *jenkins-run.sh* script assumes to find the *osmo-gsm-tester.py* in the *$PATH*. To use the most recent osmo-gsm-tester code here, we direct *$PATH* to the actual workspace checkout. This could also run from a sytem wide install, in which case you could omit the explicit PATH to "$PWD/osmo-gsm-tester/src".
  * This assumes that there are configuration files for osmo-gsm-tester placed on the system (see Section 4.1.1).
  * If you'd like to check the behavior of test failures, you can uncomment the line below "# debug" to produce a build failure on every run. Note that this test typically produces a quite empty run result, since it launches no NITB nor BTS.

- *Post-build Actions*
  - *Archive the artifacts*
    * *Files to archive*: "*-run.tgz, *-bin.tgz"
      This stores the complete test report with config files, logs, stdout/stderr output, pcaps as well as the binaries used for the test run in artifacts. This allows analysis of older builds, instead of only the most recent build (which cleans up the jenkins workspace every time). The *trial-N-run.tgz* and *trial-N-bin.tgz* archives are produced by the *jenkins-run.sh* script, both for successful and failing runs.

7.2.6 Install osmo-gsm-tester dependencies

This assumes you have already created the jenkins user (see Section 7.2.1).

Dependencies needed will depend on lots of factors, like your distribution, your specific setup, which hardware you plan to support, etc.

On a Debian/Ubuntu based system, these commands install the mandatory packages needed to run the osmo-gsm-tester.py code, i.e. install these on your main unit:

```
apt-get install \
        python3 \
        python3-yaml \
        python3-mako \
        python3-gi \
        python3-watchdog \
        locales
```

If one plans to use the 2G ESME (*esme.py*), following extra dependencies shall be installed:

```
apt-get install python3-setuptools python3-pip
pip3 install "git+https://github.com/podshumok/python-smpplib.git@master#egg=smpplib"
```

If one plans to use the 2G OsmoHLR (*hlr_osmo.py*), following extra dependencies shall be installed:

```
apt-get install sqlite3
```

If one plans to use SISPM power supply hardware (*powersupply_sispm.py*), following extra dependencies shall be installed:

```
apt-get install python3-setuptools python3-pip
pip3 install \
        pyusb \
        pysispm
```

If one plans to use software-based RF emulation on Amarisoft ENB implemented through its CTRL interface (*rfemu_amarisoftctrl.py*), following extra dependencies shall be installed:

```
apt-get install python3-websocket
```

If one plans to use srsLTE UE metrics subsystems (*ms_srs.py*), following extra dependencies shall be installed:

```
apt-get install python3-numpy
```

If one plans to use ofono modems (*ms_ofono.py*), following extra dependencies shall be installed:

```
apt-get install \
        dbus \
        python3 \
        ofono \
        python3-pip \
        udhcpc
pip3 install \
        pydbus
```

If one plans to use Open5GS EPC, pymongo modules to interact against MongoDB (*epc_open5gs.py*) shall be installed:

```
pip3 install \
        pymongo
```

Important
ofono may need to be installed from source to contain the most recent fixes needed to operate your modems. This depends on the modem hardware used and the tests run. Please see Section 8.2.

Finally, these programs are usually required by osmo-gsm-tester on the Slave Unit to run and manage processes:

```
apt-get install \
        tcpdump \
        patchelf \
        sudo \
        libcap2-bin \
        iperf3
```

7.2.7 User Permissions

On the main unit, create a group for all users that should be allowed to use the osmo-gsm-tester, and add users (here *jenkins*) to this group.

```
groupadd osmo-gsm-tester
gpasswd -a jenkins osmo-gsm-tester
```

Note
you may also need to add users to the *usrp* group, see Section 7.3.4.

A user added to a group needs to re-login for the group permissions to take effect.

7.2.7.1 Paths

Assuming that you are using the example config, prepare a system wide state location in */var/tmp*:

```
mkdir -p /var/tmp/osmo-gsm-tester/state
chown -R :osmo-gsm-tester /var/tmp/osmo-gsm-tester
chmod -R g+rwxs /var/tmp/osmo-gsm-tester
setfacl -d -m group:osmo-gsm-tester:rwx /var/tmp/osmo-gsm-tester/state
```

Important
the state directory needs to be shared between all users potentially running the osmo-gsm-tester to resolve resource allocations. Above *setfacl* command sets the access control to keep all created files group writable.

With the jenkins build as described here, the trials will live in the build slave's workspace. Other modes of operation (a daemon scheduling concurrent runs, **TODO**) may use a system wide directory to manage trials to run:

```
mkdir -p /var/tmp/osmo-gsm-tester/trials
chown -R :osmo-gsm-tester /var/tmp/osmo-gsm-tester
chmod -R g+rwxs /var/tmp/osmo-gsm-tester
```

7.2.7.2 Allow DBus Access to ofono

Put a DBus configuration file in place that allows the *osmo-gsm-tester* group to access the org.ofono DBus path:

```
# cat > /etc/dbus-1/system.d/osmo-gsm-tester.conf <<END
<!-- Additional rules for the osmo-gsm-tester to access org.ofono from user
     land -->

<!DOCTYPE busconfig PUBLIC "-//freedesktop//DTD D-BUS Bus Configuration 1.0//EN"
 "http://www.freedesktop.org/standards/dbus/1.0/busconfig.dtd">
<busconfig>

  <policy group="osmo-gsm-tester">
    <allow send_destination="org.ofono"/>
  </policy>

</busconfig>
END
```

(No restart of dbus nor ofono necessary.)

7.3 Slave Unit(s)

The slave units are the hosts used by Osmo-GSM-Tester to run proceses on. It may be the Main Unit itself and processes will be run locally, or it may be a remote host were processes are run usually through SSH.

This guide assumes slaves unit(s) use same configuration as the Main Unit, that is, it runs under *jenkins* user which is a member of the *osmo-gsm-tester* user group. In order to do so, follow the instruction under the Main Unit section above. Keep in mind the *jenkins* user on the Main Unit will need to be able to log in through SSH as the slave unit *jenkins* user to run the processes. No direct access from Jenkins Master node is required here.

7.3.1 Capture Packets

In order to allow collecting pcap traces of the network communication for later reference, allow the osmo-gsm-tester group to capture packets using the *tcpdump* program:

```
chgrp osmo-gsm-tester /usr/sbin/tcpdump
chmod 750 /usr/sbin/tcpdump
setcap cap_net_raw,cap_net_admin=eip /usr/sbin/tcpdump
```

Put *tcpdump* in the *$PATH* — assuming that *tcpdump* is available for root:

```
ln -s `which tcpdump` /usr/local/bin/tcpdump
```

Tip
Why a symlink in */usr/local/bin*? On Debian, *tcpdump* lives in */usr/sbin*, which is not part of the *$PATH* for non-root users. To avoid hardcoding non-portable paths in the osmo-gsm-tester source, *tcpdump* must be available in the *$PATH*. There are various trivial ways to modify *$PATH* for login shells, but the jenkins build slave typically runs in a **non-login** shell; modifying non-login shell enviroments is not trivially possible without also interfering with files installed from debian packages. Probably the easiest way to allow all users and all shells to find the *tcpdump* binary is to actually place a symbolic link in a directory that is already part of the non-login shell's *$PATH*. Above example places such in */usr/local/bin*.

Verify that a non-login shell can find *tcpdump*:

```
su jenkins -c 'which tcpdump'
# should print: "/usr/local/bin/tcpdump"
```

Warning
When logged in via SSH on your main unit, running *tcpdump* to capture packets may result in a feedback loop: SSH activity to send tcpdump's output to your terminal is in turn is picked up in the tcpdump trace, and so forth. When testing *tcpdump* access, make sure to have proper filter expressions in place.

7.3.2 Allow Core Files

In case a binary run for the test crashes, a core file of the crash should be written. This requires a limit rule. Create a file with the required rule:

```
sudo -s
echo "@osmo-gsm-tester - core unlimited" > /etc/security/limits.d/osmo-gsm-tester_allow- ↩
    core.conf
```

Re-login the user to make these changes take effect.

Set the **kernel.core_pattern** sysctl to **core** (usually the default). For each binary run by osmo-gsm-tester, a core file will then appear in the same dir that contains stdout and stderr for that process (because this dir is set as CWD).

```
sysctl -w kernel.core_pattern=core
```

Tip
Files required to be installed under */etc/security/limits.d/* can be found under *osmo-gsm-tester.git/utils/limits.d/*, so one can simply cp them from there.

7.3.3 Allow Realtime Priority

Certain binaries should be run with real-time priority, like *osmo-bts-trx*. Add this permission on the main unit:

```
sudo -s
echo "@osmo-gsm-tester - rtprio 99" > /etc/security/limits.d/osmo-gsm-tester_allow-rtprio. ↩
    conf
```

Re-login the user to make these changes take effect.

Tip
Files required to be installed under */etc/security/limits.d/* can be found under *osmo-gsm-tester.git/utils/limits.d/*, so one can simply cp them from there.

7.3.3.1 Allow capabilities: *CAP_NET_RAW*, *CAP_NET_ADMIN*, *CAP_SYS_ADMIN*

Certain binaries require *CAP_NET_RAW* to be set, like *osmo-bts-octphy* as it uses a *AF_PACKET* socket. Similarly, others (like osmo-ggsn) require *CAP_NET_ADMIN* to be able to create tun devices, and so on.

To be able to set the following capability without being root, osmo-gsm-tester uses sudo to gain permissions to set the capability.

This is the script that osmo-gsm-tester expects on the host running the process:

```
echo /usr/local/bin/osmo-gsm-tester_setcap_net_raw.sh <<EOF
#!/bin/bash
/sbin/setcap cap_net_raw+ep $1
EOF
chmod +x /usr/local/bin/osmo-gsm-tester_setcap_net_raw.sh
```

Now, again on the same host, we need to provide sudo access to this script for osmo-gsm-tester:

```
echo "%osmo-gsm-tester ALL=(root) NOPASSWD: /usr/local/bin/osmo-gsm-tester_setcap_net_raw. ↩
    sh" > /etc/sudoers.d/osmo-gsm-tester_setcap_net_raw
chmod 0440 /etc/sudoers.d/osmo-gsm-tester_setcap_net_raw
```

The script file name *osmo-gsm-tester_setcap_net_raw.sh* is important, as osmo-gsm-tester expects to find a script with this name in *$PATH* at run time.

Tip
Files required to be installed under */etc/sudoers.d/* can be found under *osmo-gsm-tester.git/utils/sudoers.d/*, so one can simply cp them from there.

Tip
Files required to be installed under */usr/local/bin/* can be found under *osmo-gsm-tester.git/utils/bin/*, so one can simply cp them from there.

7.3.4 UHD

Grant permission to use the UHD driver to run USRP devices for osmo-bts-trx, by adding the jenkins user to the *usrp* group:

```
gpasswd -a jenkins usrp
```

To run osmo-bts-trx with a USRP attached, you may need to install a UHD driver. Please refer to http://osmocom.org/projects/-osmotrx/wiki/OsmoTRX#UHD for details; the following is an example for the B200 family USRP devices:

```
apt-get install libuhd-dev uhd-host
/usr/lib/uhd/utils/uhd_images_downloader.py
```

7.3.5 Log Rotation

To avoid clogging up /var/log, it makes sense to choose a sane maximum log size:

```
echo maxsize 10M > /etc/logrotate.d/maxsize
```

7.3.6 Install Scripts

Important
When using the jenkins build slave as configured above, **there is no need to install the osmo-gsm-tester sources on the main unit**. The jenkins job will do so implicitly by checking out the latest osmo-gsm-tester sources in the workspace for every run. If you're using only the jenkins build slave, you may skip this section.

If you prefer to use a fixed installation of the osmo-gsm-tester sources instead of the jenkins workspace, you can:

1. From the run job configured above, remove the line that says

   ```
   PATH="$PWD/osmo-gsm-tester/src:$PATH" \
   ```

   so that this uses a system wide installation instead.
2. Install the sources e.g. in */usr/local/src* as indicated below.

On the main unit, to install the latest in */usr/local/src*:

```
apt-get install git
mkdir -p /usr/local/src
cd /usr/local/src
git clone https://gitea.osmocom.org/cellular-infrastructure/osmo-gsm-tester
```

To allow all users to run *osmo-gsm-tester.py*, from login as well as non-login shells, the easiest solution is to place a symlink in */usr/local/bin*:

```
ln -s /usr/local/src/osmo-gsm-tester/src/osmo-gsm-tester.py /usr/local/bin/
```

(See also the tip in Section 7.3.1 for a more detailed explanation.)

The example configuration provided in the source is suitable for running as-is, **if** your hardware setup matches (you could technically use that directly by a symlink e.g. from */usr/local/etc/osmo-gsm-tester* to the *example* dir). If in doubt, rather copy the example, point *paths.conf* at the *suites* dir, and adjust your own configuration as needed. For example:

```
cd /etc
cp -R /usr/local/src/osmo-gsm-tester/example osmo-gsm-tester
sed -i 's#\./suites#/usr/local/src/osmo-gsm-tester/suites#' osmo-gsm-tester/paths.conf
```

Note
The configuration will be looked up in various places, see Section 4.1.1.

8 Hardware Choice and Configuration

8.1 SysmoBTS

To use the SysmoBTS in the osmo-gsm-tester, the following systemd services must be disabled:

```
systemctl mask osmo-nitb osmo-bts-sysmo osmo-pcu sysmobts-mgr
```

This stops the stock setup keeping the BTS in operation and hence allows the osmo-gsm-tester to install and launch its own versions of the SysmoBTS software.

8.1.1 IP Address

To ensure that the SysmoBTS is always reachable at a fixed known IP address, configure the eth0 to use a static IP address:

Adjust */etc/network/interfaces* and replace the line

```
iface eth0 inet dhcp
```

with

```
iface eth0 inet static
  address 10.42.42.114
  netmask 255.255.255.0
  gateway 10.42.42.1
```

You may set the name server in */etc/resolve.conf* (most likely to the IP of the gateway), but this is not really needed by the osmo-gsm-tester.

8.1.2 Allow Core Files

In case a binary run for the test crashes, a core file of the crash should be written. This requires a limits rule. Append a line to /etc/limits like:

```
ssh root@10.42.42.114
echo "* C16384" >> /etc/limits
```

8.1.3 Reboot

Reboot the BTS and make sure that the IP address for eth0 is now indeed 10.42.42.114, and that no osmo* programs are running.

```
ip a
ps w | grep osmo
```

8.1.4 SSH Access

Make sure that the jenkins user on the main unit is able to login on the sysmoBTS, possibly erasing outdated host keys after a new rootfs was loaded:

On the main unit, for example do:

```
su - jenkins
ssh root@10.42.42.114
```

Fix any problems until you get a login on the sysmoBTS.

8.2 Modems

TODO: describe modem choices and how to run ofono

8.3 osmo-bts-trx

TODO: describe B200 family

9 Ansible Setup

Since the set of steps to set up a full Osmo-GSM-Tester environment can be quite long and tedious, nowadays the Osmocom RnD and Production Osmo-GSM-Tester setups are installed and maintained using Ansible scripts. The set of ansible scripts is available in Osmocom's git repository osmo-ci.git under *ansible* subdirectory, with the rest of ansible scripts to set jenkins slaves, etc.

Since these set of scripts is mainly aimed at Osmocom's own setup, and debian is used there, so far only debian hosts are supported officially, though patches to support other distributions are welcome.

In there, the *setup-gsm-tester.yml* file is responsible of doing all required steps to set up a host to become either a Main Unit or a a Slave Unit. The ansible file can be run as follows:

```
$ ansible-playbook -i hosts setup-gsm-tester.yml
```

You will need root-alike access in the remote host in order to let ansible install everything Osmo-GSM-Tester, however, no root-specific user is required as long as your remote user has sudo access on that host. If that's your case, add the following parameters to *ansible-playbook*:

```
$ ansible-playbook -i hosts -b -K -u your_remote_user setup-gsm-tester.yml
```

The *setup-gsm-tester.yml* file is mostly an aggregator of tasks. Most Osmo-GSM-Tester related tasks can be found under subdirectory *roles/gsm-tester-**.

Since different (for instance Production vs RnD) can have different characteristics, some per-host variables can be found under directory host_vars/, specifying for instance the number of expected modems attached to the Main Unit, the DHCP server static leasing for devices, etc.

The different tasks usually have tags to differentiate which kind of Osmo-GSM-Tester host they are required by. They are also set to differentiate sets of tasks required if a specific feature is being used in the host (for instance, willing to manage modems with ofono). This allows playing with the *-t* and *--skip-tags* when running *ansible-playbooks* in order to run specific set of tasks on each host.

For instance, to run tasks required to set up a Slave Unit, one can run:

```
$ ansible-playbook -i hosts setup-gsm-tester.yml -t osmo-gsm-tester-proc
```

To run all modem-related tasks:

```
$ ansible-playbook -i hosts setup-gsm-tester.yml -t modem
```

Don't forget to read all README.md files available in different subdirectories to find out more detailed information on how to run the scripts.

10 Docker Setup

A sample Osmo-GSM-Tester setup based on docker containers and maintained by the Osmocom community is available in Osmocom's git repository docker-playground.git, under *osmo-gsm-tester* subdirectory. In there, one can find:

- A *Dockerfile* file can be found which builds a docker image which can be used both to run as an osmo-gsm-tester Main Unit or as a Slave Unit. The main difference to use it as one or the other is whether osmo-gsm-tester.py is run on it (Main Unit) or otherwise sshd (Slave Unit). A convenience script is provided in the same directory to start the processes just explained (*osmo-gsm-tester-{master,slave}.sh*).
- A *jenkins.sh* file is provided which handles all the magic to start a Main Unit and a Slave Unit on the same docker private network so they can interact. It also takes care on running the docker containers with all the required permissions, mount all virtual filesystem bindings, etc.
- A sample resources.conf file is provided which provides some virtual resources configured to be run on the Slave Unit.

The *jenkins.sh* script expects the trial directory to be in */tmp/trial*, and will bind that directory to the docker Main Unit instance so osmo-gsm-tester uses it. Hence, one must place a the trial to be run in there before running the setup. There is yet no specific docker container available to build trials, but one can re-use an Osmocom jenkins slave container available to in *docker-playground.git* in order to build them using the scripts in *osmo-gsm-tester.git/contrib/jenkins-build-*.sh*. Alternatively, the quick way is to get them from any of the Osmocom's Osmo-GSM-Tester jenkins jobs, which store them as artifacts.

When running the whole setup through the *jenkins.sh* script, standard out (*stdout*) and standard error (*stderr*) outputs for each docker container are made available to the host running the script, under */tmp/logs* directory. Results generated by Osmo-GSM-Tester's last run can be found as usual under the trial directory (*/tmp/trial/last_run*).

The Osmo-GSM-Tester git revision being checked out to build and run inside the docker containers can be selected by setting the *OSMO_GSM_TESTER_BRANCH* environment variable. For instance, to install and run branch *mybranch* in *osmo-gsm-tester.git*, one can use:

```
export OSMO_GSM_TESTER_BRANCH=mybranch
./jenkins.sh
```

Specific command line parameters to be passed to Osmo-GSM-Tester process inside the Main Unit docker container instance can be set with the *OSMO_GSM_TESTER_OPTS* environment variable. For instance, to run suite *4g* with debug logging level:

```
export OSMO_GSM_TESTER_OPTS="-s 4g -l dbg"
./jenkins.sh
```

11 Debugging

Osmo-GSM-Tester is a complex program which at the same time orchestrates sets of other complex programs to form a network of nodes. As such, it can be sometimes challenging to find out what is going on during a trial run. This section aims at providing some tips on how to debug possible issues.

11.1 Logging level

Osmo-GSM-Tester runs by default under *info* log level. As a first debugging step, it is always a good idea to increase log verbosity. By switching to debug level (command line argument *-l dbg*), a lot more information and events are displayed which can give a much better idea to understand possible misconfigurations or wrong steps.

In any case, Osmo-GSM-Tester usually provides several log files of interest. In general, both a *log* and a *log_brief* are stored directly under the trial's run directory, the first containing output up to debug level included, while the second contains output up to info level included. Furthermore, Osmo-GSM-Tester writes a debug level log file per test case under each test's run directory.

It is also in general useful to enable the *-T* command line argument. By using it, it will instruct Osmo-GSM-Tester to write the full backtrace to the log output when something wrong happens, such an unexpected exception.

11.2 python debugger

Osmo-GSM-Tester can be further debugged using python's debugger *pdb*. Easiest way to use it is to modify the python code were you want to break and add this code:

```
import pdb; pdb.set_trace()
```

When Osmo-GSM-Tester runs over that code, it will pause and provide a debugging interactive shell, where one can inspect variables, execute code, etc.

Tip
Remember Osmo-GSM-Tester is managed by its internal main loop, meaning if you jump into a debugger console you will still need to give back control to the main loop for events to be processed and checks done. That can be done for instance by calling the *MainLoop.sleep(log_obj, secs)* internal API in general or 'sleep(secs)' under test context.

11.3 debug suite

Sometimes, however, one may be interested in debugging the behavior of the software under test by Osmo-GSM-Tester rather than Osmo-GSM-Tester itself. For instance, one may simply want to set up a full running network of nodes and keep it up until some manual tests are done, or one may want Osmo-GSM-Tester to do so at a given point of time.

To fulfill this kind of scenarios, Osmo-GSM-Tester provides some code available for tests to gain access to a high-level interactive console which is fully integrated with Osmo-GSM-Tester's own main loop. So the approach here is usually to write a regular test (with its corresponding suite.conf) to set up and run all required processes and then allow it to jump into the interactive console instance. Then the test pulls received commands from it and it is responsible for parsing and implementing them. One command can for instance ask a modem to send an sms to another. Another command can for instance jump into a debugger console.

The interactive console is available to tests through the *prompt* method, and its implementation can be found under *method input_polling* in *util.py*.

An interactive console example as explained in this section can be found under the *debug/interactive.py* test in osmo-gsm-tester.git.

12 Troubleshooting

12.1 Format: YAML, and its Drawbacks

The general configuration format used is YAML. The stock python YAML parser does have several drawbacks: too many complex possibilities and alternative ways of formatting a configuration, but at the time of writing seems to be the only widely used configuration format that offers a simple and human readable formatting as well as nested structuring. It is recommended to use only the exact YAML subset seen in this manual in case the osmo-gsm-tester should move to a less bloated parser in the future.

Careful: if a configuration item consists of digits and starts with a zero, you need to quote it, or it may be interpreted as an octal notation integer! Please avoid using the octal notation on purpose, it is not provided intentionally.

12.2 Osmo-GSM-Tester not running but resources still allocated

The reserved_resources.state is used to keep shared state of the the resources allocated by any Osmo-GSM-Tester instance. Each Osmo-GSM-Tester instance being run is responsible to de-allocate the used resources before exiting. In general, upon receiving a shutdown action (ie. *CTRL+C*, *SIGINT*, python exception, etc.), Osmo-GSM-Tester is able to handle properly the situation and de-allocate the resources before the process exits. Similarly, Osmo-GSM-Tester also takes care of terminating all its children processes being managed before exiting itself.

However, under some circumstances, Osmo-GSM-Tester will be unable to de-allocate the resources and they will remain allocated for subsequent Osmo-GSM-Tester instances which try to use them. That situation is usually reached when someone terminates Osmo-GSM-Tester in a hard way. Main reasons are Osmo-GSM-Tester process receiving a *SIGKILL* signal (*kill -9 $pid*) which cannot be caught, or due to the entire host being shut down in a non proper way.

As a noticeable example, SIGKILL is known to be sent to Osmo-GSM-Tester when it runs under a jenkins shell script and any of the two following things happen:

- User presses the red cross icon in the Jenkins UI to terminate the running job.
- Connection between Jenkins master (UI) and Jenkins slave running the job is lost.

Once this situation is reached, one needs to follow 2 steps:

- Gain console access to the Main Unit and manually clean or completely remove the *reserved_resources.state* in the state_dir. In general it's a good idea to make sure no Osmo-GSM-Tester instance is running at all and then remove completely all files in state_dir, since Osmo-GSM-Tester could theoretically have been killed while writing some file and it may have ended up with corrupt content.
- Gain console access to the Main Unit and each of the Slave Units and kill any hanging long-termed processes in there which may have been started by Osmo-GSM-Tester. Some popular processes in this list include *tcpdump*, *osmo-**, *srs**, etc.