JTA Quickstart guide

Cogent Embedded Inc.
http://cogentembedded.com

November 30, 2015

Abstract

This document is the complete reference for JTA test system. It contains installation guide, basic usage guide, step-by-step guides for adding test board and integrating a test. In later chapters it is described how test overlays and pdf test report generation work.

Contents

1 Introduction **4**

2 Installation **5**
2.1 Prerequisites 5
2.2 Running install script 5
2.3 Installing toolchains and sysroots 5
2.3.1 Using meta-jta OE layer for generating toolchain 5
2.4 Configuring tools.sh file 5
2.4.1 Using custom toolchain 6

3 Boards configuration **7**
3.1 Adding a target in Jenkins interface 7
3.2 Writing the board config overlay 7

4 Running tests **9**
4.1 Running single tests 9
4.2 Running a group of tests 9
4.3 Viewing PDF reports 10

5 Adding a sample test **11**
5.1 Adding Test Plan files 11
5.2 Adding spec file 11
5.3 Adding test script 11
5.4 Adding test to frontend 12
5.4.1 Adding plot paramters to `tests.info` 13
5.5 Conclusion 13

6 Base classes and overlays **14**
6.1 Base class format 14
6.2 Overlay format 14
6.3 Overlay and class relationship example 14
6.4 JTA classes and overlays organization 15
6.4.1 Base distribution class 16
6.4.2 Base board class 16

7 Test plans **17**
7.1 Spec file format 17
7.2 Test plan file format 17
7.3 The algorithm 18
7.4 Test plan / test spec relationship example 18

8 Reports **21**

9 Listings **22**

1 Introduction

Testing evaluation boards (and final products based on them) is not easy. There is a number of software products made for that very purpose, the most eminent of which is Lava.

From our point of view, LAVA is hard to extend both in terms of the engine and its frontend. On the other hand, the core of JTA is based on shell scripts that can be trivially extended and uses Jenkins which is well tested and has lots of available plugins to extend the framework with additional features (e.g. e-mail notification of test results).

The JTA framework was designed to provide a core meeting a few points:

- It is usable out of the box: JTA includes 60+ prepackaged tests, including benchmark statistics, plotting and reports generation.
- It is highly custmizable on the front-end side (thanks to the availability of tons of plugins for Jenkins) and also on the backend side, which relies on a simple core written in bash;
- It allows for flexible test configuration using such notions as *test specifications* and *test plans* 7;
- It supports running groups of tests in a batch and generating reports 8);
- It does not impose any demands on boards or distributions;
- It allows easy yet flexible board setup. All you need to do to add a new board is just define some environment variables (block devices/mount points, IP addr, etc.) in a board config file.

As you can see, our goal is to provide a flexible framework with seemless customization and an easy out-of-the-box experience.

2 Installation

2.1 Prerequisites

- Any 64-bit Linux with Docker version 1.8.3 or above.
- Web browser with **javascript** and **CSS** support.

2.2 Running install script

Simply run install.sh script. It will create a docker image with JTA installed. When it's done you should create and start a docker container running the following commands from top directory:

- `jta-host-scripts/docker-create-containter.sh`.
- `jta-host-scripts/docker-start-containter.sh`.

When container is started, JTA web interface will be available on local machine at port 8080.

Please note that board configuration, jenkins `config.xml` file, build logs and toolchains are stored under *userdata* catalogue that is mounted as external volume to docker container under /userdata path. This allows to preserve all configuration when creating a new docker container.

In the rest of the document `/userdata/...` paths denote paths inside docker container.

Some files and catalogues under `/userdata/...` are symlinked inside JTA engine and Jenkins paths. Please see *Init Userdata* section in `Dockerfile` for more details.

2.3 Installing toolchains and sysroots

You need toolchains and sysroots to build tests for different platforms.

2.3.1 Using meta-jta OE layer for generating toolchain

For convenience we provide a yocto layer that includes necessary software for target system: custom layer. You can use this layer to generate toolchain and sysroot (using `bitbake meta-toolchain`) with all libraries and headers needed for building tests. See Poky Documentation for more information.

Toolchains and sysroots should generally be stored in `/userdata/toolchains` and installed from within docker container.

2.4 Configuring tools.sh file

`JTA_ENGINE_PATH/scripts/tools.sh` file is used to setup paths and compile flags for each platform.

For poky-generated toolchains one should source envirnoment file and set the following variables:

- SDKROOT - path to rootfs
- PREFIX - gcc prefix, like arm-blabla-linux-gnueabi
- HOST - like PREFIX

Also not code saving original $PATH to $ORIG_PATH since envirnoment script changes it.

See [L. 5] for example.

2.4.1 Using custom toolchain

For using custom toolchain you *additionally* must define the following variables: PATH, PKG_CONFIG_SYSROOT_DIR, PKG_CONFIG_PATH, CC, CXX, CPP, AS, LD, RANLIB, AR, NM, CFLAGS, CXXFLAGS, LDFLAGS, CPPFLAGS, ARCH, CROSS_COMPILE.

You can use environment-setup-core2-32-osv-linux script as reference.

3 Boards configuration

In this document we will use such notions as *targets* and *boards*. Here is what they mean:

Target or Node denotes a front-end Jenkins entity. Jenkins jobs are run on targets.

Board denotes a back-end entity, such as a physical board (specifically, the board or device to run tests on).

Board configuration is stored in `JTA_ENGINE_PATH/overlays/boards/<boardname>.board`, where `<boardname>` is the respective name of the target.

3.1 Adding a target in Jenkins interface

The simplest method of adding a new target is to copy from an existing one. We provide *template-dev* board for that purpose.

1. Open a browser window to the JTA web interface
2. Click on **Target status**
3. Click on **New node**
4. Fill in the *Node name* input field witht he name of the new board
5. Choose *Copy Existing Node*. And enter name of the source node, namely, *template-dev*
6. You will be forwarded to node configuration page. Locate the *Environment variables* section in *Node Properties*. Specify the path to board config file [3.2] using the variable **BOARD_OVERLAY**.

3.2 Writing the board config overlay

Board config file is an overlay (See 6.2) that must inherit `base-board` and include `base-params` base classes (in that order).

The following is the step-by-step description of all mandatory environment variables that should be set by this file:

`TRANSPORT`: defines how JTA should communicate with the board. Currently only `ssh` is supported;

`IPADDR`: IP address or host name of board;

`SSH_PORT`: ssh port number of board;

`LOGIN`: user name for ssh login;

`PASSWORD`: password for ssh login;

`JTA_HOME`: path to the directory on device the tests will run from;

`PLATFORM`: architecture of the board. Currently `ia32`, `arm` and `mips` are supported. Used by some of tests during compilation.

The following variables specify devices and mount points that are used by some file system tests: `SATA_DEV, SATA_MP, USB_DEV, USB_MP, MMC_DEV, MMC_MP`.

4 Running tests

4.1 Running single tests

1. From the main page open *Functional* or *Benchmarks* tab.
2. Click on the test name.
3. Click on *Run test now*.

Here you can set test run parameters. The most relevant are:

Device: Choose a target the test will be run on.

Reboot: If checked target device will be rebooted before running test.

Rebuild: Rebuild[1] the test.

TESTPLAN: (optional) Derive test parameters from test specifications from chose testplan. For testplans see [7]

Press *Run test* button. The test is scheduled for running. If no tests are executed on target it will be run immediately. It will appear in the left frame name *Test run history*. There you can see all this specific test results (including currenlty running one). They can be of a few types:

Solid green circle: The test has been successefully run.

Solid red circle: The test has failed.

Flashing circle: The test is currently running.

If you point mouse over the date of test run the pop-up menu appears from where you can go to *Console output*. There you can view the complete log of test run.

4.2 Running a group of tests

1. From the main page open *Batch runs*.
2. Click on *Run SELECTED tests on SELECTED targets*.
3. Click on *Run test now*.
4. Choose a target[2]
5. Mark tests/benchmarks you would like to run.

[1] Some tests require rebuilding if their parameters were changed.
[2] Reports generation is implemented only for the first target selected and only for Run SELECTED tests on SELECTED batch run

6. Enter test plan to `TESTPLAN` variable. Test plan is mandatory for batch runs.

7. Click on *Run test* button.

The batch test run is scheduled for running on target. On the test page there is a list of running tests (their statuses are the same as in [4.1]).

When batch run is finished you can view generated PDF report.

4.3 Viewing PDF reports

1. From the batch test run page click on *Workspace link*

2. Click on pdf_reports folder.

3. Click the bottommost pdf file.
   It has `<target>.<date>.<testrun>.json.xml.pdf` format.

5 Adding a sample test

This section describes how to integrate tests to OSV. We will add a simple test that calls bc computing a value passed through spec parameter.

5.1 Adding Test Plan files

Create `JTA_ENGINE_PATH/overlays/testplans/testplan_bc_exp1.json`[L. 3] and `testplan_bc_exp2.json`[L. 4] files.

As you can see we've created two testplan files which reference two specs. Testplan can reference multiple specs for different tests, so for example we could run all filesystem tests with specific block device.

5.2 Adding spec file

Create `JTA_ENGINE_PATH/overlays/test_specs/Benchmark.bc.spec`[L. 6] file.

This spec file contatins two cases: `bc-exp1` that generates `EXPR1`, `EXPR2` variable and assignes it "`2*2`", "`3*3`" values [3] and `bc-spec-exp2` that does the same but with "`2+2`" and "`3+3`" values. These variables is inteded to be used inside test script for controlling different test cases. And we will use it as a parameter to `bc-device.sh` script.

You don't usually need more than one spec files, because all different cases can be listed in one file.

5.3 Adding test script

Test script is the bash file that runs when test is executed on target. Create it[L. 7] with the path `JTA_ENGINE_PATH/tests/Benchmark.bc/bc-script.sh`. This file should meet a strict format with following definitions:

`tarball` name of the tarball;

`test_build` should contain test build commands;

`test_deploy` should contain commands that deploy test to device; *put* command is usually used;

`test_run` should contain all steps for actual test execution.

Generic benchmark/test script can be sourced if test meets common patterns. In this particular example `benchmark.sh` is sourced that will execute these steps (and some other like overlay prolog file and reports generation).

For testing purposes we will use a simple script that is executed on device. It accepts two parameters, calls *bc* with them and produces an output. Create `bc-script.tar.gz` tarball containing a folder with `bc-device.sh`[L. 8] file.

[3] Any variable defined in board config file[**??**] or in (inherited) base file[3.2] can be used. For example `$MINNOW_SATA_DEV`

When benchmark is finished results parsing phase is started. **Each** benchmark (not Functional test) should provide a special python parsing script called `parser.py` that defines how to parse results. All you should do is to fill a `cur_dict` dictionary with `{subtest: value}` pairs and call `plib.process_data` with respecitve arguments:

`ref_section_pat` : regexp that describes the format of threshold expressions

`cur_dict:` : dictionary containing `{subtest: value}` pairs with test results;

`m:` plot type. 's' - single, 'm' - multiple

`label:` axis label

See [L. 9] for a simple script that parses two *bc* outputs.

Core script `common.py` checks the values to agree with *reference values* that should be in `reference.log` file in the directory where main test script resides. See [L. 10] for sample `reference.log` file asserts both results must be greater that 0.

Test integration is complete. Now you should be able to locate test under *Benchmarks* tab in main page.

5.4 Adding test to frontend

The simpliest way to add a benchmark in frontend is using one as a template.

1. From main page click on *New Test*;
2. Fill in *Test name* input field;
3. Choose *Copy existing Test* combo box;
4. Enter test name to the *Copy from* text field. For example, *Benchmark.bonnie*;
5. Press *OK* button.

You will be forwarded to the test configuration page. There are a lot of parameters there, but you only need to set up a few of them:

Description: Textual description of the test;

TESTPLAN: (a string parameter) path to the test plan. Not mandatory. But we will use one for that sake of demonstration. Put `testplans/testplan_bc_exp1.json` there.

Execute shell: bash script that will be executed when test is run.
Put `source ../tests/$JOB_NAME/bc-script.sh` there.

5.4.1 Adding plot paramters to `tests.info`

Plot plugin needs to know which parameters it should display. It uses `tests.info` file for that purpose. Open `JTA_ENGINE_PATH/logs/tests.info` and add the following line: `"bc":["result1","result2"]`. Make sure you meet json syntax this file uses.

This line says to draw *result1* and *result2* values on the plot.

5.5 Conclusion

So, below is the list of all components our benchmark uses.

spec file `Benchmark.bc.spec`[L. 6] that contain list of various options that generate variables for testing;

testplan files `testplan_bc_exp{1,2}.json` [L. 3], [L. 4] that contain lists of specifications should be used for test(s);

test script `bc-script.sh`[L. 7] that runs all top-level commands;

tarball with `bc-device.sh`[L. 8] file that does actual testing on device;

parser.py [L. 9] that parses the results and gives them to core parsing component that prepares data for plots and reports;

reference.log [L. 10] that contains reference values then benchmark results are checked against;

tests.info should be modified to include values should be drawn on plot.

6 Base classes and overlays

This section describes base classes and overlays concept, how to write ones and mechanism implementing them.

6.1 Base class format

Base class is a special file similar to shell script with definitions of basic parameters. It has special fields `OF.NAME` and `OF.DESCRIPTION` that set base class name and description respectively. You can have as many base classes as you want. We provide base classes for *boards* and *distributions*.

6.2 Overlay format

Overlay file has simple format, similar to that of the bash shell. It has two extra syntactic constructs:

`inherit` is used to read and inherit the base class config file.
Example: `inherit "base-file"`. It is possible to override functions and variables of base class.

`include` is used to include all contents of base class config file. No variables and functions overriding is permited.

`override override-func` are used for overriding base variables and functions.
Syntax:

```
override-func ov_rootfs_logread() {
    # commands
}

override VARIABLE new_value
```

6.3 Overlay and class relationship example

Simple class and overlay relationship is shown in [F. 1]. *base-distrib* class defines a function

`ov_rootfs_logread`

and two variables:

`LOGGER_VAR`,

`BASE_VAR`.

nologger.dist overlay redefines `ov_rootfs_logread` function and `LOGGER_VAR` variable. In the end *prolog.sh* contains overriedden function, overridden `LOGGER_VAR` variable and vanilla `BASE_VAR` variable.

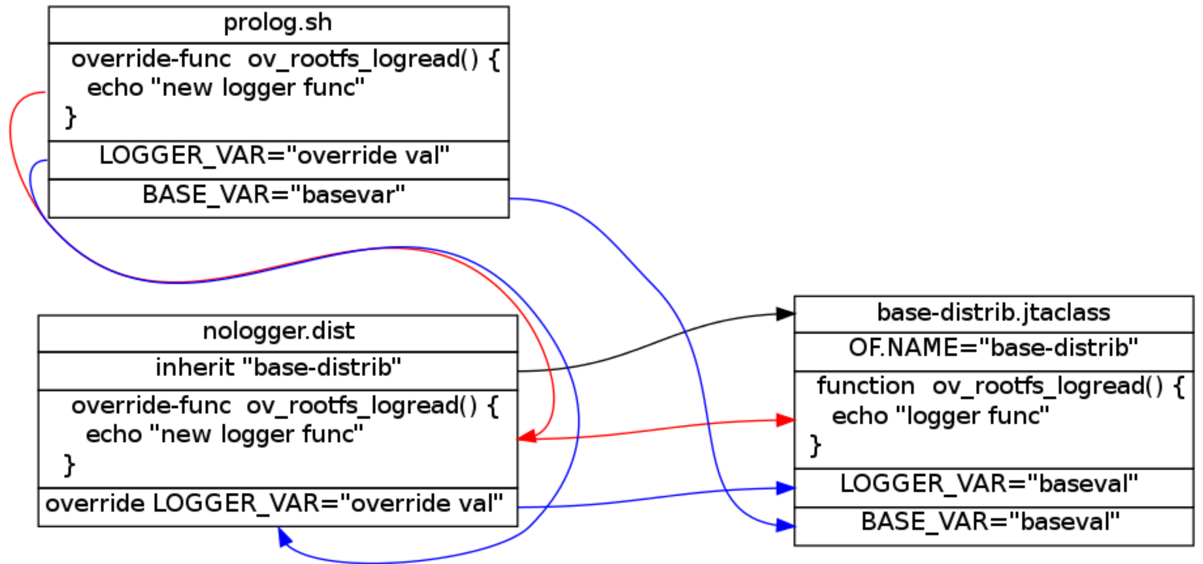

Figure 1: Simple base class and overlay example

6.4 JTA classes and overlays organization

`ovgen.py` script takes a number of base class and overlay files and produces `prolog.sh` script file that is executed before each test is run. There are two conceptes implemented using the scheme:

1. Distribution - defines commands for basic actions on device
2. Board - specifies how to communicate with the device

[F. 2] displays this scheme from top-level perspective.

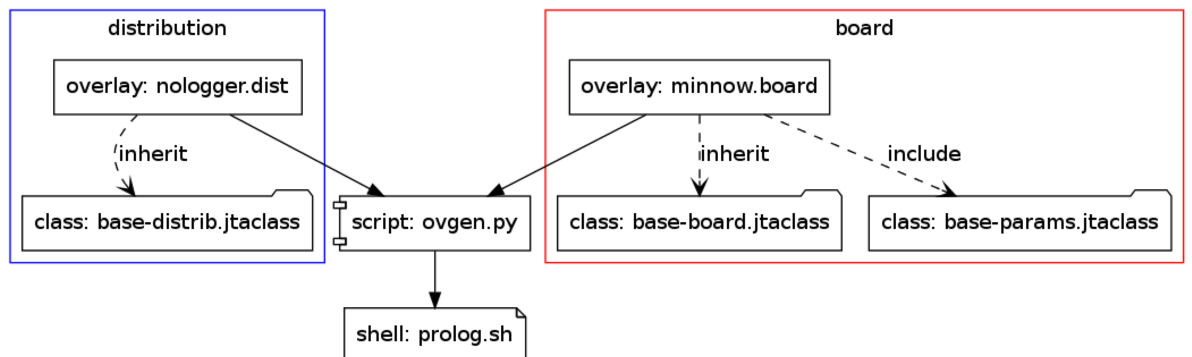

Figure 2: JTA Base classes and overlays from toplevel perspective

6.4.1 Base distribution class

base-distrib is the base class (See 6.1) that defines functions necessary for working with system. It is located in `overlays/base/base-distrib.jtaclass` file.

It defines the following functions:

`ov_get_firmware:` get kernel version;

`ov_rootfs_reboot:` reboot system;

`ov_rootfs_state:` get uptime, memory usage, mounetd devices, etc;

`ov_logger:` put string to syslog;

`ov_rootfs_sync:` sync filesystem;

`ov_rootfs_drop_caches:` drop FS caches;

`ov_rootfs_oom:` adjust oom;

`ov_rootfs_kill:` kill specified processes;

`ov_rootfs_logread:` get syslogs.

You can redefine these functions in your distib overlay file that inherits *base-distrib* class. Default distrib overlay `base.dist` just inherits base class with no modifications.

6.4.2 Base board class

base-board is the base class that defines functions necessary for working with system. It is located in `overlays/base/base-distrib.jtaclass` file.

`ov_transport_get:` get specified file from board;

`ov_transport_put:` copy specified file to the board;

`ov_transport_cmd:` run command on board;

You can redefine these in your board overlay for non-standard methods for communicating with device.

7 Test plans

Test plans is the core feature of JTA. They provide the very flexibility in configuring tests to be run on different boards and scenarios. This section describes how test plans work and their implementation.

7.1 Spec file format

Spec file format uses json syntax. It uses the following format:

```
{
    "testName": "name of test",
    "specs": [
        # spec entries
        {
          "name":"spec name",
          "variable1":"value1",
          "variable2":"value2",
          ...
          "variableN":"valueN",
        },
        ...
    ]
```

Listing 1: Spec file format

Each spec file contatins test name and number of spec entries for this test. Each spec entry has a name and a list of variable/value pairs that become `TESTNAME_VARIABLE=”VALUE”` in `prolog.sh` whenever this spec is chosen in *test plan*. `VALUE` could be bash variable reference as well, since it will be expanded during runtime. For example it could reference block device (e.g. `$SATA_DEV`) from board config file.

7.2 Test plan file format

Test plan file format uses json syntax. It uses the following format:

```
{
    "testPlanName": "name of test plan",
    "tests": [
        #test spec entries
        {
            "testName": "name of test",
            "spec": "spec name"
        },
     ....
     ]
}
```

Listing 2: Test plan file format

Each test plan file contains a number of test spec entries, each specyfing which spec should be used with given test. Testplans are usecase oriented, e.g there could be test plan for number of tests to for running on sata device (defined in board file).

Test plan (as for now) does not denote which tests *will* be run, rather it specifies which environment variables should be *generated* in `prolog.sh`. This is useful in *batch runs* (TODO: reference here) when multiple tests are run with same `prolog.sh` file.

7.3 The algorithm

These are the steps taken by `ovgen.py` script with regard to test plan processing:

1. parse *spec files* in `overlays/specs` directory;
2. parse *test plan file* that is specified via `TEST_PLAN` environment variable;
3. For each test entry *TE* in testplan:
   (a) Locate the specified test spec *SP* among all test specs;
   (b) Generate all `VARIABLE="VALUE"` from *SP* to `prolog.sh`

See [F. 3.

7.4 Test plan / test spec relationship example

Below is the simple example of test plan generation. See [F. 4]

1. User specifies `testplan_sata.json` in `TEST_PLAN` envirnoment variable before running test;
2. ovgen.py script reads all test spec files from `overlays/specs` as well as specified test plan file;
3. reads *Benchmark.Bonnie* entry where *sata* spec is specified;
4. reads *sata* spec from inside `Benchmark.bonnie.spec` file;
5. generates `BENCHMARK_BONNIE_MOUNT_BLOCKDEV` and `BENCHMARK_BONNIE_MOUNT_POINT` variable definitions and writes them to prolog.sh file;

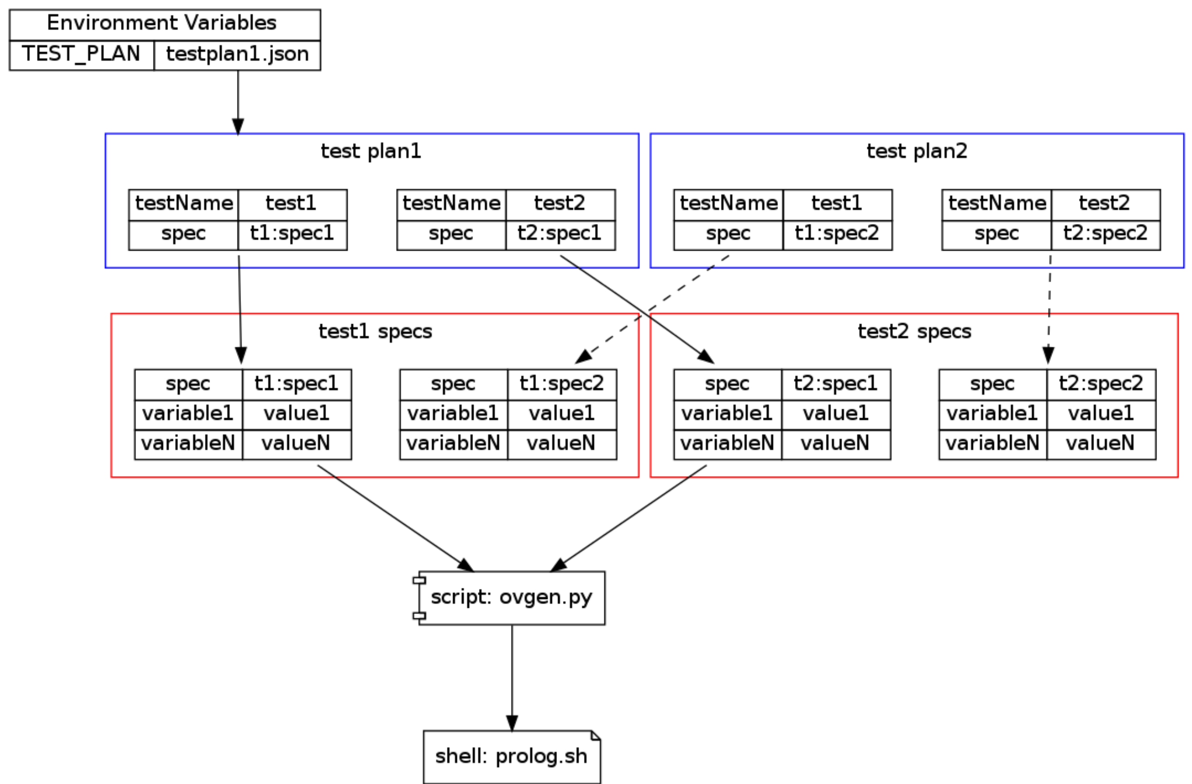

Figure 3: Testplans top level scheme

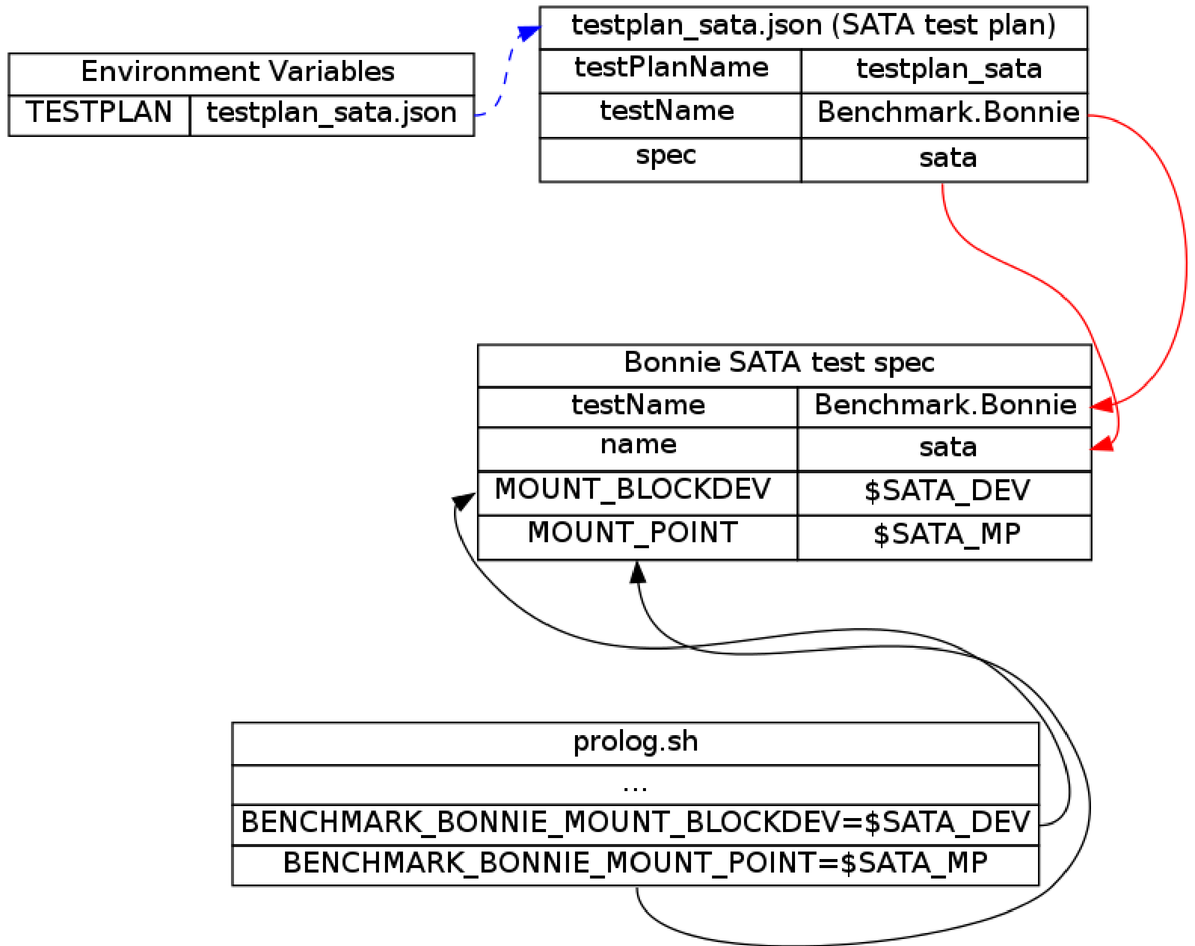

Figure 4: Simple testplan example

8 Reports

This section describes how reports are implemented.

9 Listings

```
{
    "testPlanName": "testplan_bc",
    "tests": [
        {
            "testName": "Benchmark.bc",
            "spec": "bc-exp1"
        }    ]
}
```

Listing 3: testplan_bc_exp1.json file

```
{
    "testPlanName": "testplan_bc",
    "tests": [
        {
            "testName": "Benchmark.bc",
            "spec": "bc-exp2"
        }    ]
}
```

Listing 4: testplan_bc_exp2.json file

```
if [ "${PLATFORM}" = "intel-minnow" ];
then
        SDKROOT=$JTA_ENGINE_PATH/tools/intel-minnow/
           sysroots/core2-32-osv-linux/
        # environment script changes PATH in the way
           that python uses libs from sysroot which is
            not what we want, so save it and use later
        ORIG_PATH=$PATH
        PREFIX=i586-osv-linux
        source $JTA_ENGINE_PATH/tools/intel-minnow/
           environment-setup-core2-32-osv-linux

        HOST=arm-osv-linux-gnueabi

        unset PYTHONHOME
        env -u PYTHONHOME
```

Listing 5: intel minnow tools section

```
  {
    "testName": "Benchmark.bc",
    "specs":
```

```
    [
        {
            "name":"bc-exp1",
            "EXPR1":"2*2",
            "EXPR2":"3*3"
        },
        {
            "name":"bc-exp2",
            "EXPR":"2+2",
            "EXPR2":"3+3"
        }
    ]
}
```

Listing 6: `Benchmark.bc.spec` file

```
#!/bin/bash

tarball=bc-script.tar.gz

function test_build {
    echo "test compiling (should be here)"
}

function test_deploy {
   put bc-device.sh  $JTA_HOME/jta.$TESTDIR/
}

function test_run {
    assert_define BENCHMARK_BC_EXPR1
    assert_define BENCHMARK_BC_EXPR2
    report "cd $JTA_HOME/jta.$TESTDIR; ./bc-device.sh
       $BENCHMARK_BC_EXPR1 $BENCHMARK_BC_EXPR1"
}
. ../scripts/benchmark.sh
```

Listing 7: `bc-script.sh` file

```
#!/bin/bash

BC_EXPR1=$1
BC_EXPR2=$1

BC1=`echo $BC_EXPR1 | bc`
BC2=`echo $BC_EXPR2 | bc`
echo "$BC1,$BC2"
```

Listing 8: `bc-device.sh` file

```
#!/bin/python

import os, re, sys, json

sys.path.insert(0, '/home/jenkins/scripts/parser')
import common as plib

cur_dict = {}
cur_file = open(plib.CUR_LOG,'r')
print "Reading current values from " + plib.CUR_LOG +
    "\n"

ref_section_pat = "^\[[\w_ .]+.[gle]{2}\]"

raw_values = cur_file.readlines()
results = raw_values[-1].rstrip("\n").split(",")
cur_file.close()

cur_dict["result1"] = results[0]
cur_dict["result2"] = results[1]

sys.exit(plib.process_data(ref_section_pat, cur_dict,
    's', 'value'))
```

Listing 9: `parser.py` file

```
[result1|ge]
0
[result2|ge]
0
```

Listing 10: `reference.log` file