IBM Systems and Technology Group

Automated Test Framework (ATF)

Embedded Linux Conference 2012

Overview

- ATF is a Domain Specific Embedded Language (DSEL) for defining tests.
- Intended to be used for Unit Testing through System Testing
- Intended to allow Test Case reuse in simulated, prototype hardware and release hardware environments.
- ATF consists of:
  - A Perl-based Test API that provides:
    - Communication to multiple test target via SSH, telnet, etc.
    - Domain Specific Embedded Language (DSEL) for defining tests
    - A wrapper on top of the Perl Test Harness
  - Test Driver that allows running buckets of tests and summarizing results
  - Test cases that are written in the Test API DSEL and/or Perl

Why ATF?

- Need 'simple-to-write' test cases
- Need to be able to run single tests or buckets of test cases
- Need the ability to drive several systems and firmware components from a single test case:
  - CLI calls
  - RPC calls
  - Packet Injection
- Wanted to avoid making test case writers learn complicated programming languages
- Wanted the lightweight development cycle of a scripting language
- No known existing test suite met all of these requirements

Why Perl?

- Script language
- Code Reuse
  - Perl's Test Harness
  - Tools for handling Test Any Protocol
- Useful for making a DSEL
  - Expressive syntax
  - Reflective
- Powerful enough language to handle unexpected corner cases
- Learned Incrementally

Target Uses

- Unit Testing
  - Used by firmware developers to test good paths
- Build Verification Testing
  - Used by the build to identify bad check ins
- Functional Testing
  - Used by firmware testers to verify component functionality
- Simulation Verification
  - Used by hardware modelers to detect regressions
- System Verification
  - Used by system testers and manufacturing to confirm all components work together
- Automated Regression Testing

Results

- Tests are written by testers & developers
  - Our team has little to no Perl experience
  - Only one of the testers is a programmer
- 1.5 dedicated testers since September 2010
  - Create, run and write additional automation
- Running weekly regression of 6300 tests
- Running daily build verification test of 452 crucial tests

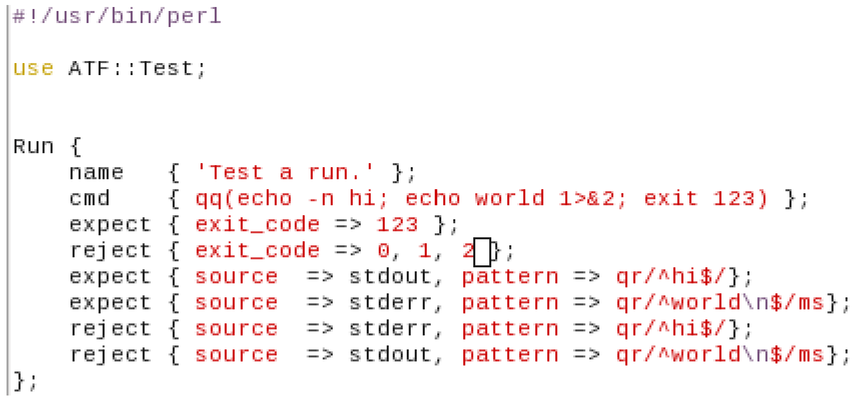
Features

- Supports multiple targets
- HTML based report generation
- Simple parallel execution
- TODO keyword
  - Intended for tests that are written before function is ready
  - Can be used on entire test case, single test, or single expectation
  - Test run, expectations checked, but results flagged as TODO
  - Successes are marked as ‘unexpected successes’
  - Failures are reported, but do not count as failures
- SKIP keyword
  - Intended for tests that temporarily should not be run, because of some detrimental effect (such as preventing other tests in the bucket from running)
  - Can be used on entire test case or single test
  - Unlike TODO, SKIP Will NOT execute the test nor will it check expectations
  - Tests marked as SKIP are reported as skipped, not as failures
- Simplified flow control
  - Test case stages with automatic flow control
  - Variation keyword – allows simple to write looping mechanisms that can be nested

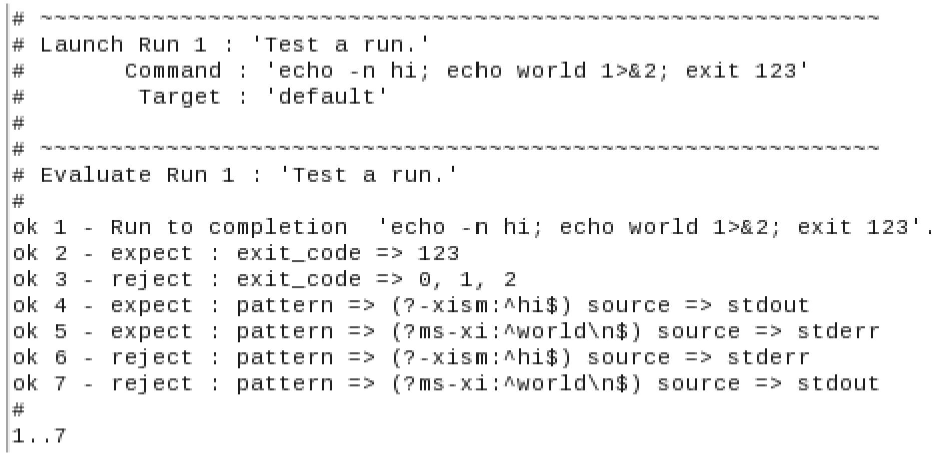
Technical Details

Run Directive

- Declares an action to take
- Declares expectations of the output
- Hides the mechanics of
  - Running the command
  - Gathering output
  - Reporting results
- Results are reported in
  “Test Anything Protocol”



```
#!/usr/bin/perl

use ATF::Test;


Run {
    name    { 'Test a run.' };
    cmd     { qq(echo -n hi; echo world 1>&2; exit 123) };
    expect { exit_code => 123 };
    reject { exit_code => 0, 1, 2};
    expect { source  => stdout, pattern => qr/^hi$/};
    expect { source  => stderr, pattern => qr/^world\n$/ms};
    reject { source  => stderr, pattern => qr/^hi$/};
    reject { source  => stdout, pattern => qr/^world\n$/ms};
};
```



```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Launch Run 1 : 'Test a run.'
#       Command : 'echo -n hi; echo world 1>&2; exit 123'
#        Target : 'default'
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Evaluate Run 1 : 'Test a run.'
#
ok 1 - Run to completion  'echo -n hi; echo world 1>&2; exit 123'.
ok 2 - expect : exit_code => 123
ok 3 - reject : exit_code => 0, 1, 2
ok 4 - expect : pattern => (?-xism:^hi$) source => stdout
ok 5 - expect : pattern => (?ms-xi:^world\n$) source => stderr
ok 6 - reject : pattern => (?-xism:^hi$) source => stderr
ok 7 - reject : pattern => (?ms-xi:^world\n$) source => stdout
#
1..7
```

Run Directive - Expectations

- **Exit code expect/reject**
  - **Declare a list of exit codes to accept or reject**
- **Pattern expect/reject**
  - **Declare a data source**
  - **Declare a pattern**
  - **Optionally capture data**
  - **Optionally mark the test as TODO or SKIP**
- **Assert**
  - **Catches some corner case**
  - **Provide a name**
  - **Declare a Boolean expression that must be true**
- **Ok**
  - **Perl assertion for remaining corner cases**

```
Run {
    cmd     { qq(echo -n hi; echo world 1>&2; exit 123) };

    expect { exit_code => 120..129 };
    reject { exit_code => 0..10 };

    expect { source  => stdout,
             pattern => qr/^(hi|hello|howdy)$/,
             capture => [ greeting => 1 ]};
    reject { source  => stderr,
             pattern => qr/^hi$/ };

    assert { "Check exit code more thoroughly.",
             ($capture{exit_code} >= 120)
                 && ($capture{exit_code} <= 129) };
};

ok($capture{greeting} ne "howdy", "Let's stay formal.");
```

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Launch Run 1 : 'echo -n hi; echo world 1>&2; exit 123'
#        Target : 'default'
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Evaluate Run 1 : 'echo -n hi; echo world 1>&2; exit 123'
#
ok 1 - Run to completion  'echo -n hi; echo world 1>&2; exit 123'.
ok 2 - expect : exit_code => 120, 121, 122, 123, 124, 125, 126, 127, 128, 129
ok 3 - reject : exit_code => 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
ok 4 - expect : pattern => (?-xism:^(hi|hello|howdy)$) source => stdout
ok 5 - reject : pattern => (?-xism:^hi$) source => stderr
ok 6 - assert : Check exit code more thoroughly.
#
ok 7 - Let's stay formal.
1..7
```

Run Directive – Lowered Expectations

- SKIP
  - Can be used to prevent the execution of:
    - A test file
    - A Run directive
    - An individual pattern test (But don't do this)
- TODO
  - Prevents expect failures from counting as failures
  - Flags unexpected successes.
  - Can mark
    - A test file
    - A Run directive
    - An individual pattern test
    - An exit code test

```
SKIP { "Running these tests will ruin the test environment." };
TODO { "These tests shouldn't work YET." };

Run {
    TODO  { "Don't be surprised if this does not run." };
    SKIP  { "Let's just be safe and not run it." };

    cmd    { "echo hi ; exit 123" };

    expect { exit_code => 120..129,
             TODO => "This always returns zero at the moment."};

    expect { source  => stdout,
             pattern => qr/^(hi|hello|howdy)$/,
             TODO    => "Output doesn't work quite yet." };

    reject { source  => stderr,
             pattern => qr/./,
             TODO    => "Currently there are lot of errors messages." };

};
```

Run Directive - Multi-target

- Support multiple targets
- Default target
  - Defined as local host
  - Can be changed via the command line or via an optional configuration file
- Host is defined as local host
- Other targets are defined in an optional configuration file

```
# This is not really necessary since these target always exist.
RequiredTargets 'default', 'host';

# Skip this test if the default target is the local host.
SKIP { "This test requires the default target to be a remote target"
       } if ref($ATF::Configuration::Target{default}) eq 'ATF::Target::Local';


# Set a local environment variable that we will see locally but not remotely.
$ENV{DUMMY_VAR} = 'DUMMY_VAR';


# Run something locally using 'host'.
Run {
    cmd { "env" };
    target { 'host' };
    expect { source  => stdout, pattern => qr/^DUMMY_VAR=DUMMY_VAR$/m};
};


# Run something remotely using 'default'.
Run {
    cmd { "env" };
    target { 'default' };
    reject { source  => stdout, pattern => qr/^DUMMY_VAR=DUMMY_VAR$/m};
};


# Run something remotely without declaring a target.
Run {
    cmd { "env" };
    reject { source  => stdout, pattern => qr/^DUMMY_VAR=DUMMY_VAR$/m};
};
```

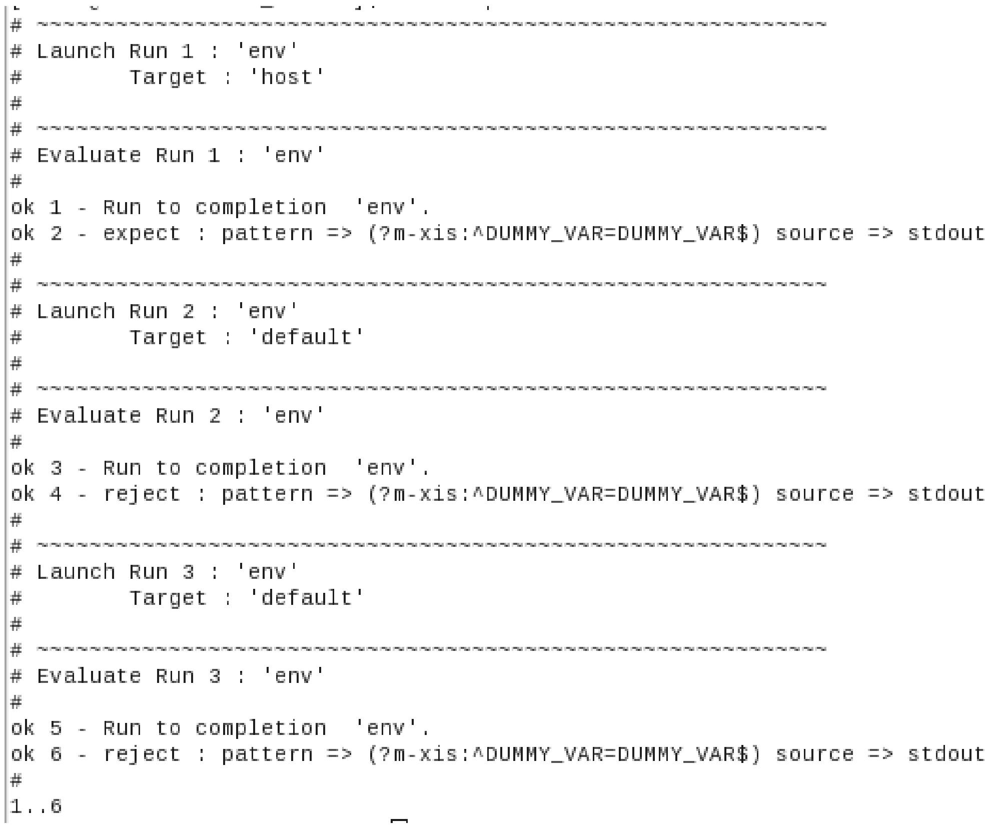
```
# Host OS
DefineTarget(test_sys => 'SSH',
             address  => 192.168.10.10, port => 22,
             user     => 'root',         password => '12345');

# Test MC
DefineTarget(test_mc => 'SSH',
             address => '192.168.10.14', port     => '22',
             user    => 'USERID',         password => 'PASSWORD');

# Redefine the default target
DefaultTarget('test_mc');
```

Run Directive – Parallel Execution

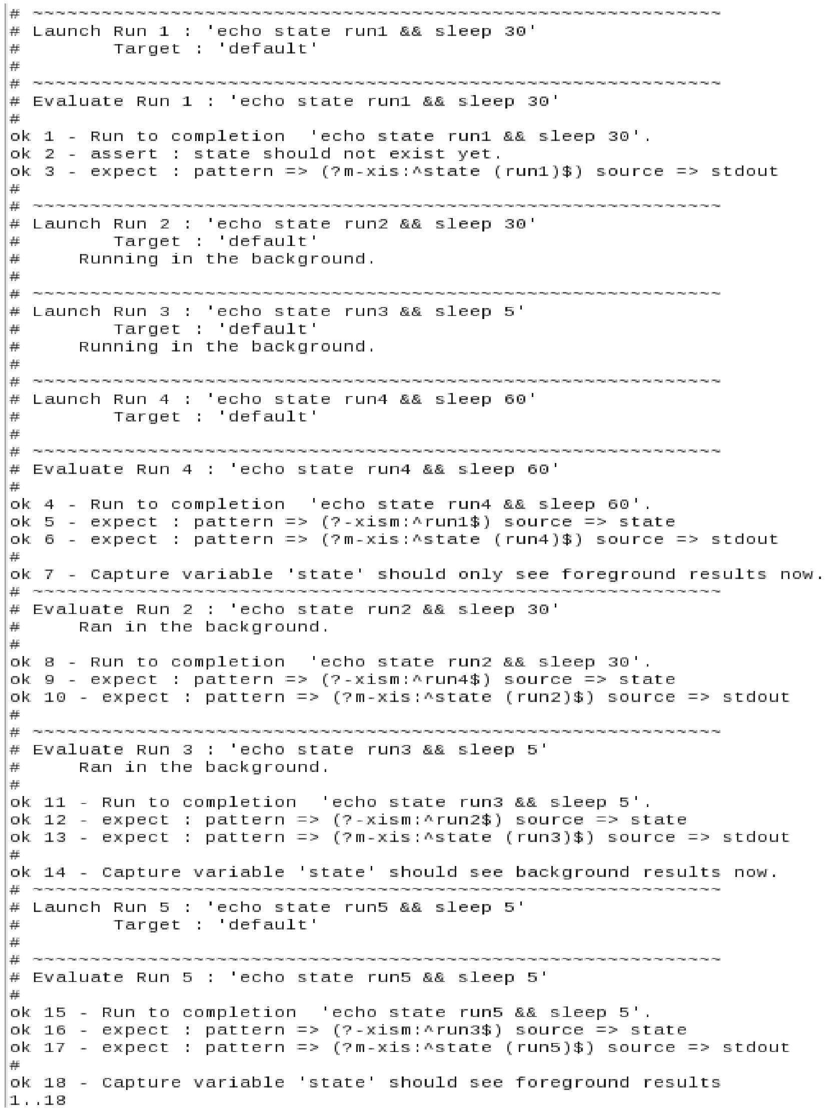
- Simplified parallel execution
- Run time is not deterministic
- Maximum run time is known
- Visibility of captured values is deterministic
  - Each Run sees captured values available when they are launched
  - Foreground Run's results are evaluated immediately at completion
  - Background Runs are evaluated at the end of the the enclosing Parallel block in the order they were launched.

```
# Set the default timeout.
DefaultTimeout(120);

# Start a Parallel block for the fence
Parallel {
    Run { # Run 1: a long running normal command.
        cmd  {"echo state run1 && sleep 30"};
        assert {"state should not exist yet.", !exists($capture{state})};
        expect { source  => stdout, pattern => qr/^state (run1)$/m,
                 capture => [state => 1] };
    };


    Run { # Run 2: a long running background command.
        cmd  {"echo state run2 && sleep 30"};
        expect { source  => "state", pattern => qr/^run4$/ };
        expect { source  => stdout,  pattern => qr/^state (run2)$/m,
                 capture => [state => 1] };
        background;
    };

    Run { # Run 3: a quick background Run.
        cmd  {"echo state run3 && sleep 5"};
        expect { source  => "state", pattern => qr/^run2$/ };
        expect { source  => stdout,  pattern => qr/^state (run3)$/m,
                 capture => [state => 1] };
        background;
    };

    Run { # Run 4: a longer running blocking Run.
        cmd  {"echo state run4 && sleep 60"};
        expect { source  => "state", pattern => qr/^run1$/ };
        expect { source  => stdout,  pattern => qr/^state (run4)$/m,
                 capture => [state => 1] };
    };

    ok($capture{state} eq 'run4',
       "Capture variable 'state' should only see foreground results now.");
};

ok($capture{state} eq 'run3',
   "Capture variable 'state' should see background results now.");


Run { # Run 5: a quick blocking Run.
    cmd  {"echo state run5 && sleep 5"};
    expect { source  => "state", pattern => qr/^run3$/ };
    expect { source  => stdout,  pattern => qr/^state (run5)$/m,
             capture => [state => 1] };
};

ok($capture{state} eq 'run5',
   "Capture variable 'state' should see foreground results");
```

Run Directive – Previous Results

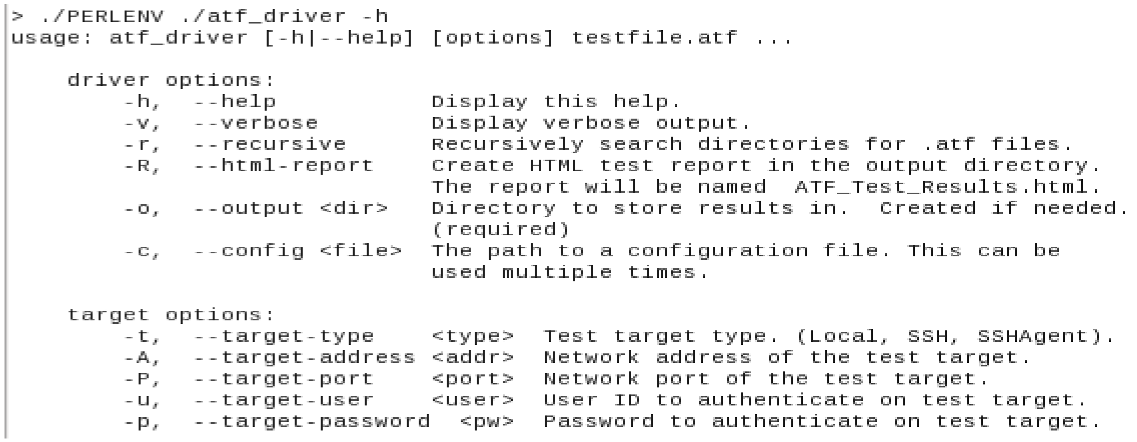


```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Launch Run 1 : 'env'
#        Target : 'host'
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Evaluate Run 1 : 'env'
#
ok 1 - Run to completion  'env'.
ok 2 - expect : pattern => (?m-xis:^DUMMY_VAR=DUMMY_VAR$) source => stdout
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Launch Run 2 : 'env'
#        Target : 'default'
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Evaluate Run 2 : 'env'
#
ok 3 - Run to completion  'env'.
ok 4 - reject : pattern => (?m-xis:^DUMMY_VAR=DUMMY_VAR$) source => stdout
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Launch Run 3 : 'env'
#        Target : 'default'
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Evaluate Run 3 : 'env'
#
ok 5 - Run to completion  'env'.
ok 6 - reject : pattern => (?m-xis:^DUMMY_VAR=DUMMY_VAR$) source => stdout
#
1..6
```

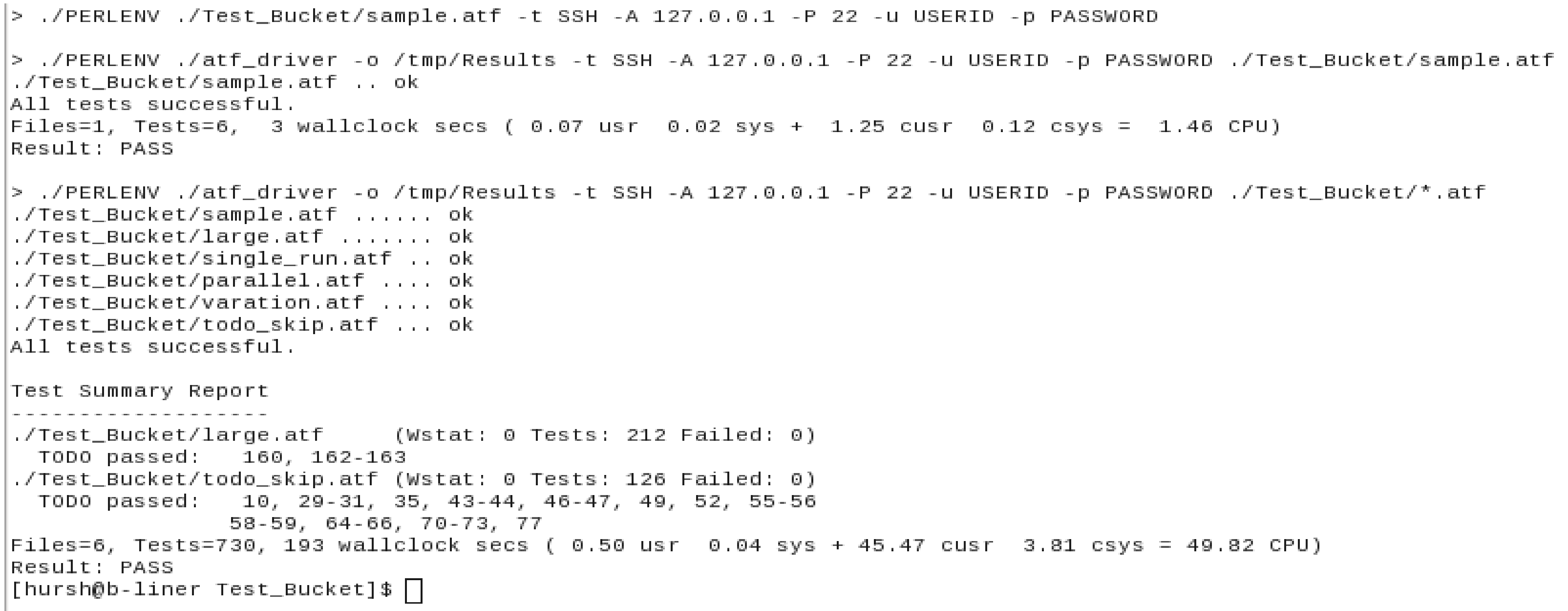


```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Launch Run 1 : 'echo state run1 && sleep 30'
#        Target : 'default'
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Evaluate Run 1 : 'echo state run1 && sleep 30'
#
ok 1 - Run to completion  'echo state run1 && sleep 30'.
ok 2 - assert : state should not exist yet.
ok 3 - expect : pattern => (?m-xis:^state (run1)$) source => stdout
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Launch Run 2 : 'echo state run2 && sleep 30'
#        Target : 'default'
#     Running in the background.
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Launch Run 3 : 'echo state run3 && sleep 5'
#        Target : 'default'
#     Running in the background.
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Launch Run 4 : 'echo state run4 && sleep 60'
#        Target : 'default'
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Evaluate Run 4 : 'echo state run4 && sleep 60'
#
ok 4 - Run to completion  'echo state run4 && sleep 60'.
ok 5 - expect : pattern => (?-xism:^run1$) source => state
ok 6 - expect : pattern => (?m-xis:^state (run4)$) source => stdout
#
ok 7 - Capture variable 'state' should only see foreground results now.
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Evaluate Run 2 : 'echo state run2 && sleep 30'
#     Ran in the background.
#
ok 8 - Run to completion  'echo state run2 && sleep 30'.
ok 9 - expect : pattern => (?-xism:^run4$) source => state
ok 10 - expect : pattern => (?m-xis:^state (run2)$) source => stdout
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Evaluate Run 3 : 'echo state run3 && sleep 5'
#     Ran in the background.
#
ok 11 - Run to completion  'echo state run3 && sleep 5'.
ok 12 - expect : pattern => (?-xism:^run2$) source => state
ok 13 - expect : pattern => (?m-xis:^state (run3)$) source => stdout
#
ok 14 - Capture variable 'state' should see background results now.
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Launch Run 5 : 'echo state run5 && sleep 5'
#        Target : 'default'
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Evaluate Run 5 : 'echo state run5 && sleep 5'
#
ok 15 - Run to completion  'echo state run5 && sleep 5'.
ok 16 - expect : pattern => (?-xism:^run3$) source => state
ok 17 - expect : pattern => (?m-xis:^state (run5)$) source => stdout
#
ok 18 - Capture variable 'state' should see foreground results
1..18
```

Running Tests

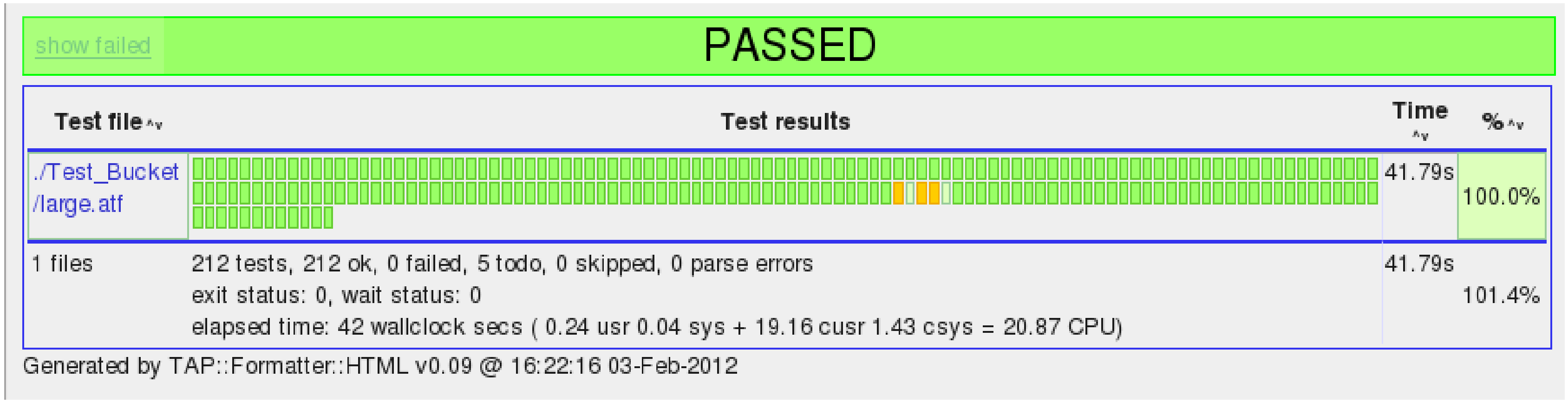


```
> ./PERLENV ./atf_driver -h
usage: atf_driver [-h|--help] [options] testfile.atf ...

    driver options:
        -h,  --help             Display this help.
        -v,  --verbose          Display verbose output.
        -r,  --recursive        Recursively search directories for .atf files.
        -R,  --html-report      Create HTML test report in the output directory.
                                The report will be named  ATF_Test_Results.html.
        -o,  --output <dir>     Directory to store results in.  Created if needed.
                                (required)
        -c,  --config <file>    The path to a configuration file. This can be
                                used multiple times.

    target options:
        -t,  --target-type      <type>  Test target type. (Local, SSH, SSHAgent).
        -A,  --target-address <addr>  Network address of the test target.
        -P,  --target-port      <port>  Network port of the test target.
        -u,  --target-user      <user>  User ID to authenticate on test target.
        -p,  --target-password  <pw>  Password to authenticate on test target.
```



```
> ./PERLENV ./Test_Bucket/sample.atf -t SSH -A 127.0.0.1 -P 22 -u USERID -p PASSWORD

> ./PERLENV ./atf_driver -o /tmp/Results -t SSH -A 127.0.0.1 -P 22 -u USERID -p PASSWORD ./Test_Bucket/sample.atf
./Test_Bucket/sample.atf .. ok
All tests successful.
Files=1, Tests=6,  3 wallclock secs ( 0.07 usr  0.02 sys +  1.25 cusr  0.12 csys =  1.46 CPU)
Result: PASS

> ./PERLENV ./atf_driver -o /tmp/Results -t SSH -A 127.0.0.1 -P 22 -u USERID -p PASSWORD ./Test_Bucket/*.atf
./Test_Bucket/sample.atf ...... ok
./Test_Bucket/large.atf ....... ok
./Test_Bucket/single_run.atf .. ok
./Test_Bucket/parallel.atf .... ok
./Test_Bucket/varation.atf .... ok
./Test_Bucket/todo_skip.atf ... ok
All tests successful.

Test Summary Report
-------------------
./Test_Bucket/large.atf     (Wstat: 0 Tests: 212 Failed: 0)
  TODO passed:   160, 162-163
./Test_Bucket/todo_skip.atf (Wstat: 0 Tests: 126 Failed: 0)
  TODO passed:   10, 29-31, 35, 43-44, 46-47, 49, 52, 55-56
                58-59, 64-66, 70-73, 77
Files=6, Tests=730, 193 wallclock secs ( 0.50 usr  0.04 sys + 45.47 cusr  3.81 csys = 49.82 CPU)
Result: PASS
[hursh@b-liner Test_Bucket]$ 
```

Test Output – HTML Report

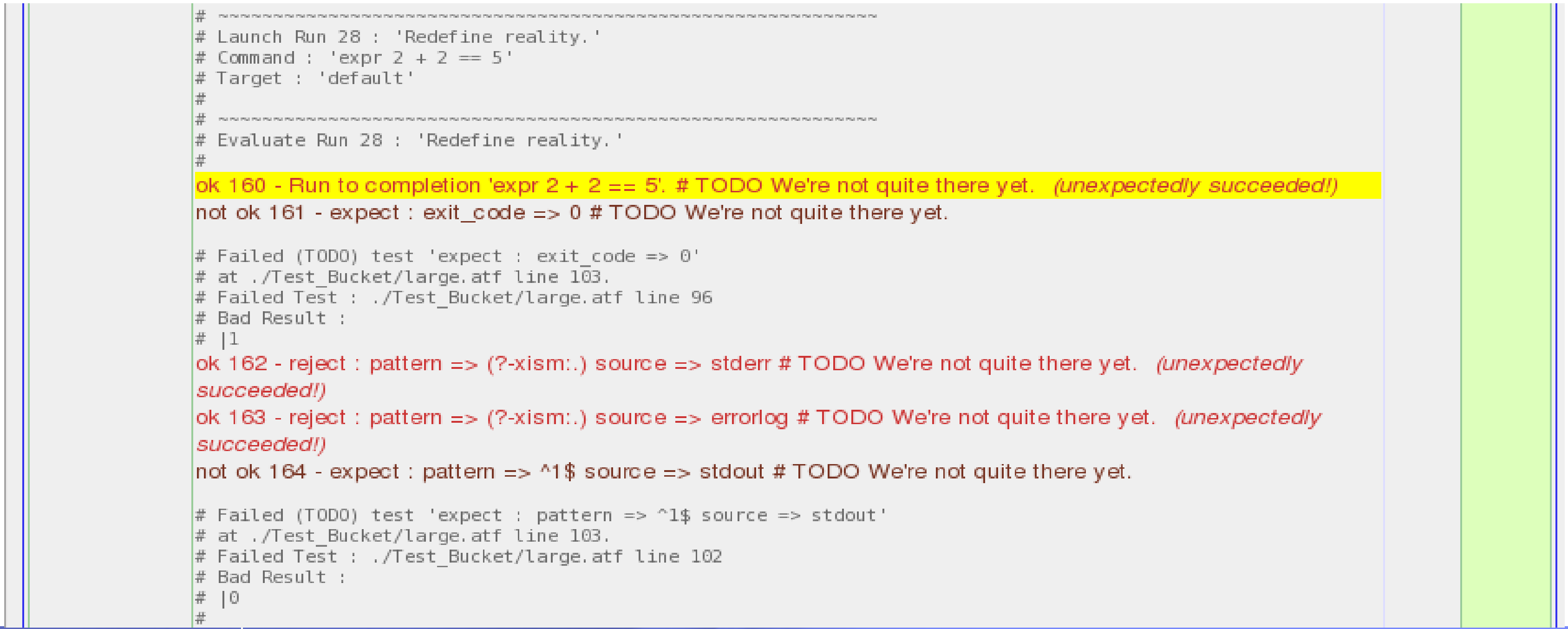


show failed

PASSED

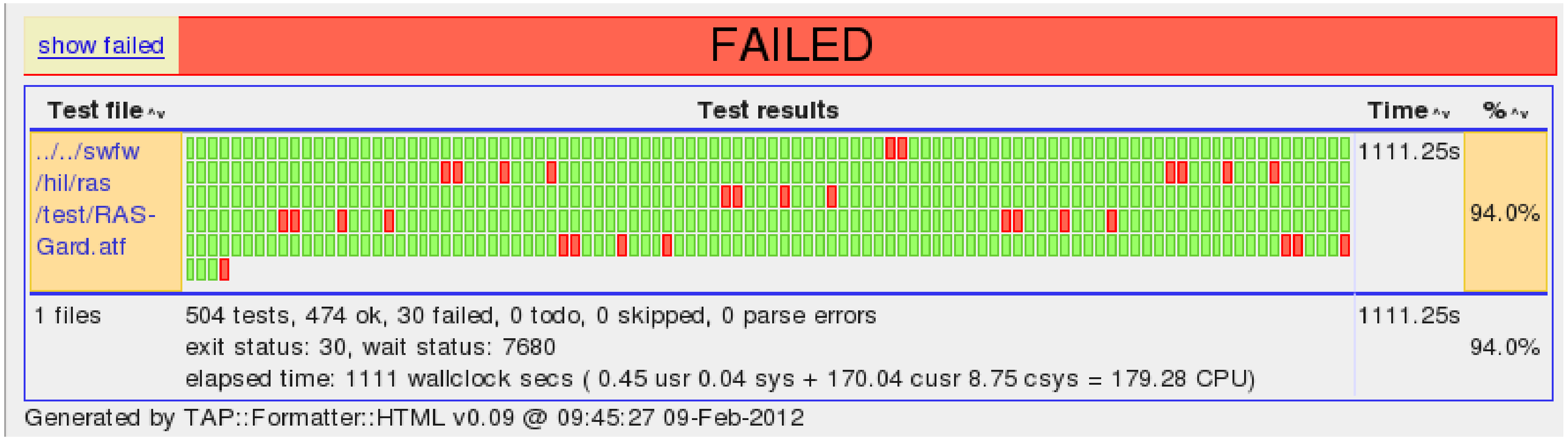
Test file ^v	Test results	Time ^v	% ^v
./Test_Bucket /large.atf		41.79s	100.0%
1 files	212 tests, 212 ok, 0 failed, 5 todo, 0 skipped, 0 parse errors exit status: 0, wait status: 0 elapsed time: 42 wallclock secs (0.24 usr 0.04 sys + 19.16 cusr 1.43 csys = 20.87 CPU)	41.79s	101.4%

Generated by TAP::Formatter::HTML v0.09 @ 16:22:16 03-Feb-2012



```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Launch Run 28 : 'Redefine reality.'
# Command : 'expr 2 + 2 == 5'
# Target : 'default'
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Evaluate Run 28 : 'Redefine reality.'
#
ok 160 - Run to completion 'expr 2 + 2 == 5'. # TODO We're not quite there yet.  (unexpectedly succeeded!)
not ok 161 - expect : exit_code => 0 # TODO We're not quite there yet.

# Failed (TODO) test 'expect : exit_code => 0'
# at ./Test_Bucket/large.atf line 103.
# Failed Test : ./Test_Bucket/large.atf line 96
# Bad Result :
# |1
ok 162 - reject : pattern => (?-xism:.) source => stderr # TODO We're not quite there yet.  (unexpectedly
succeeded!)
ok 163 - reject : pattern => (?-xism:.) source => errorlog # TODO We're not quite there yet.  (unexpectedly
succeeded!)
not ok 164 - expect : pattern => ^1$ source => stdout # TODO We're not quite there yet.

# Failed (TODO) test 'expect : pattern => ^1$ source => stdout'
# at ./Test_Bucket/large.atf line 103.
# Failed Test : ./Test_Bucket/large.atf line 102
# Bad Result :
# |0
#
```

Test Output – HTML Report



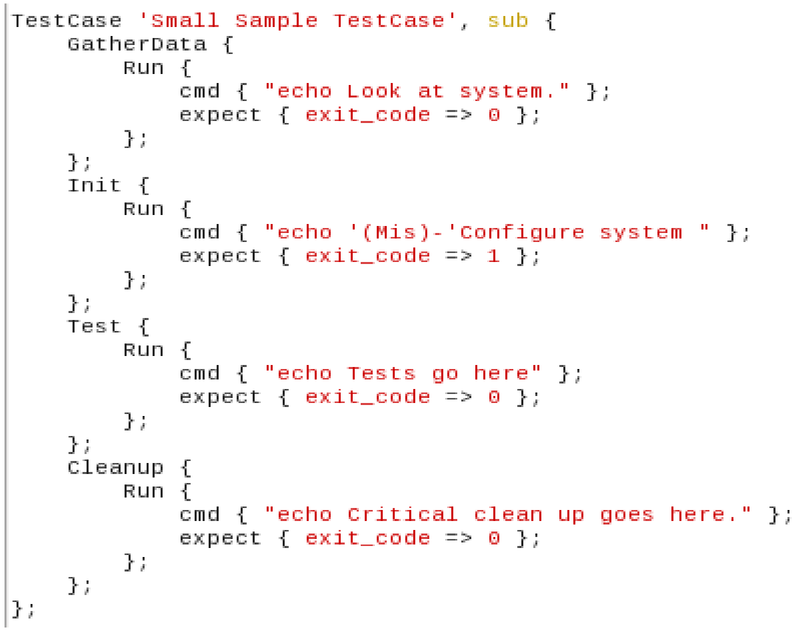
show failed — FAILED

Test file	Test results	Time	%
../../swfw/hil/ras/test/RAS-Gard.atf		1111.25s	94.0%
1 files	504 tests, 474 ok, 30 failed, 0 todo, 0 skipped, 0 parse errors exit status: 30, wait status: 7680 elapsed time: 1111 wallclock secs (0.45 usr 0.04 sys + 170.04 cusr 8.75 csys = 179.28 CPU)	1111.25s	94.0%

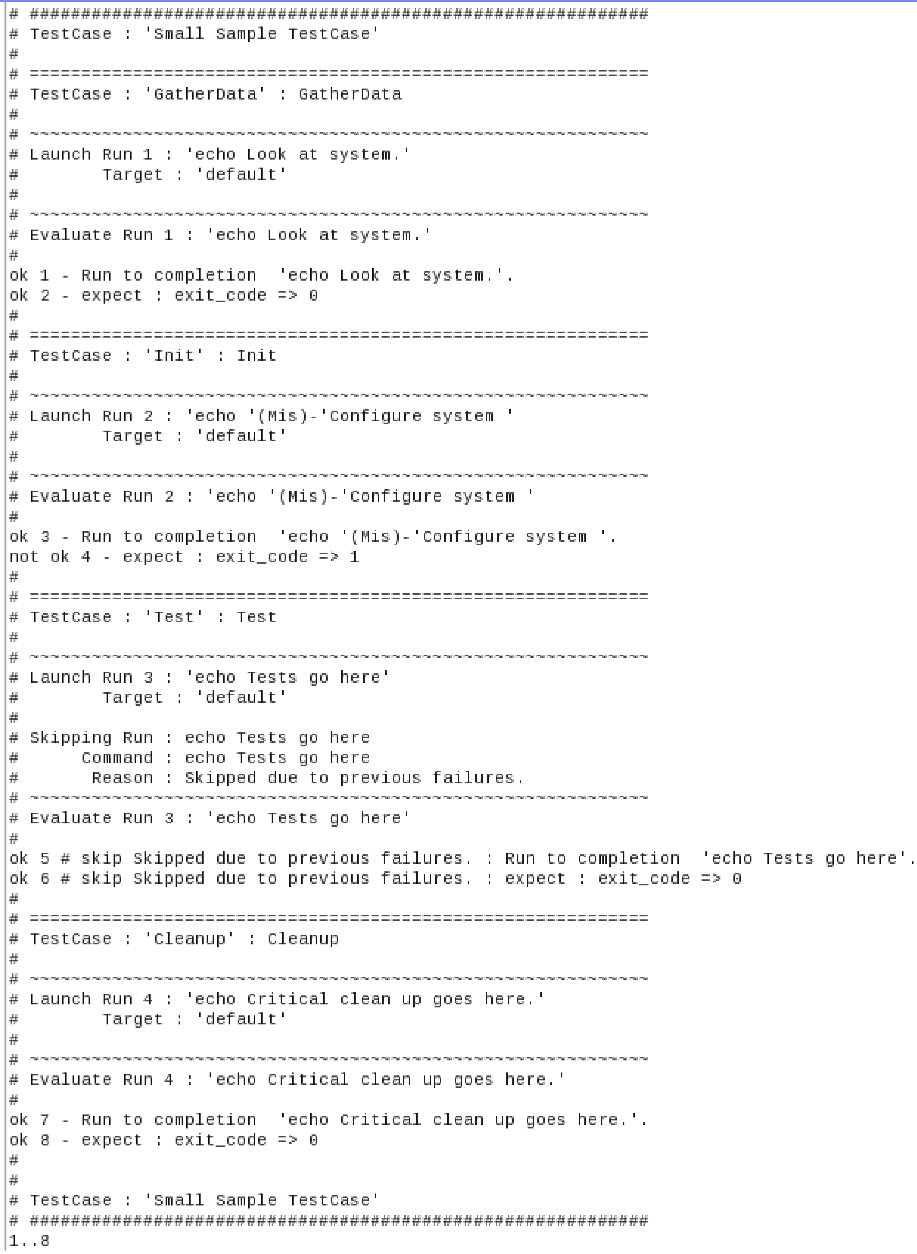
Generated by TAP::Formatter::HTML v0.09 @ 09:45:27 09-Feb-2012

Flow Control - TestCase

- High level conditional test execution
- Multiple TestCase declarations are allowed in a single test file
- TestCase declarations cannot be nested
- TestCase sections are implicit Parallel blocks.



```
TestCase 'Small Sample TestCase', sub {
    GatherData {
        Run {
            cmd { "echo Look at system." };
            expect { exit_code => 0 };
        };
    };
    Init {
        Run {
            cmd { "echo '(Mis)-'Configure system " };
            expect { exit_code => 1 };
        };
    };
    Test {
        Run {
            cmd { "echo Tests go here" };
            expect { exit_code => 0 };
        };
    };
    Cleanup {
        Run {
            cmd { "echo Critical clean up goes here." };
            expect { exit_code => 0 };
        };
    };
};
```



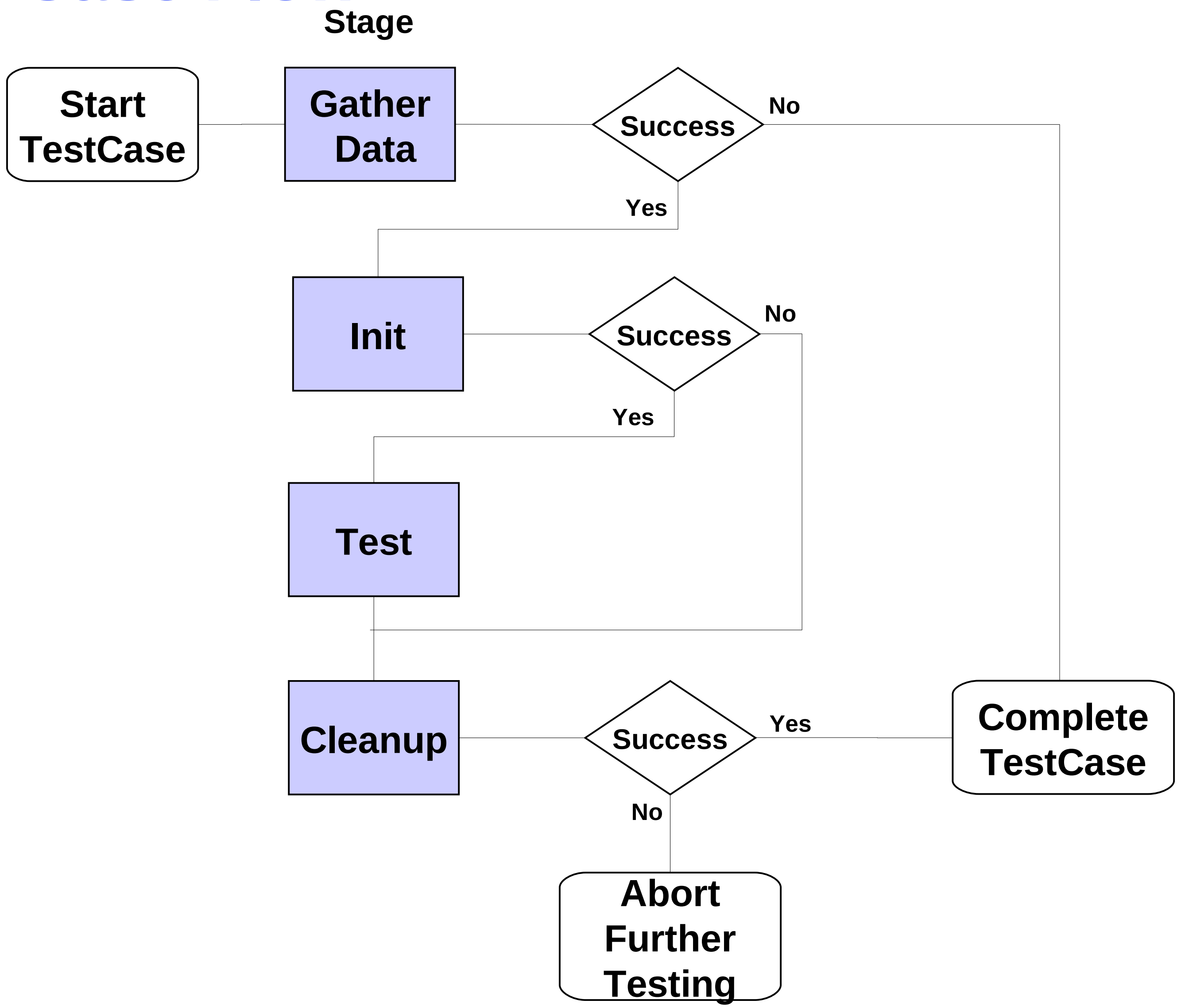
```
# ###########################################################
# TestCase : 'Small Sample TestCase'
#
# ===========================================================
# TestCase : 'GatherData' : GatherData
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Launch Run 1 : 'echo Look at system.'
#        Target : 'default'
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Evaluate Run 1 : 'echo Look at system.'
#
ok 1 - Run to completion  'echo Look at system.'.
ok 2 - expect : exit_code => 0
#
# ===========================================================
# TestCase : 'Init' : Init
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Launch Run 2 : 'echo '(Mis)-'Configure system '
#        Target : 'default'
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Evaluate Run 2 : 'echo '(Mis)-'Configure system '
#
ok 3 - Run to completion  'echo '(Mis)-'Configure system '.
not ok 4 - expect : exit_code => 1
#
# ===========================================================
# TestCase : 'Test' : Test
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Launch Run 3 : 'echo Tests go here'
#        Target : 'default'
#
# Skipping Run : echo Tests go here
#      Command : echo Tests go here
#       Reason : Skipped due to previous failures.
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Evaluate Run 3 : 'echo Tests go here'
#
ok 5 # skip Skipped due to previous failures. : Run to completion  'echo Tests go here'.
ok 6 # skip Skipped due to previous failures. : expect : exit_code => 0
#
# ===========================================================
# TestCase : 'Cleanup' : Cleanup
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Launch Run 4 : 'echo Critical clean up goes here.'
#        Target : 'default'
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Evaluate Run 4 : 'echo Critical clean up goes here.'
#
ok 7 - Run to completion  'echo Critical clean up goes here.'.
ok 8 - expect : exit_code => 0
#
#
# TestCase : 'Small Sample TestCase'
# ###########################################################
1..8
```

Test Case Sections and Flow Control

Section	Purpose	Result on Failure
GatherData	Inspect Environment	Skip Init, Test and Cleanup
Init	Setup for testing	Skip Test and run Cleanup
Test	Feature testing	None
Cleanup	Undo Changes Made	Abort further testing

- **GatherData - For 'look but don't touch' inspection of the test environment prior to modifying the system state or testing anything. This data can be used to restore the original state in the 'Cleanup' stage**
- **Init - For modifying the test environment in preparation for the actual test to be performed in the Test section, such as creating error conditions or initializing resources to be used in the test**
- **Test - Perform the actual tests and check the results**
- **Cleanup - Perform any necessary cleanup before exiting the test case**

Test Case Flow



Flow Control - Variation

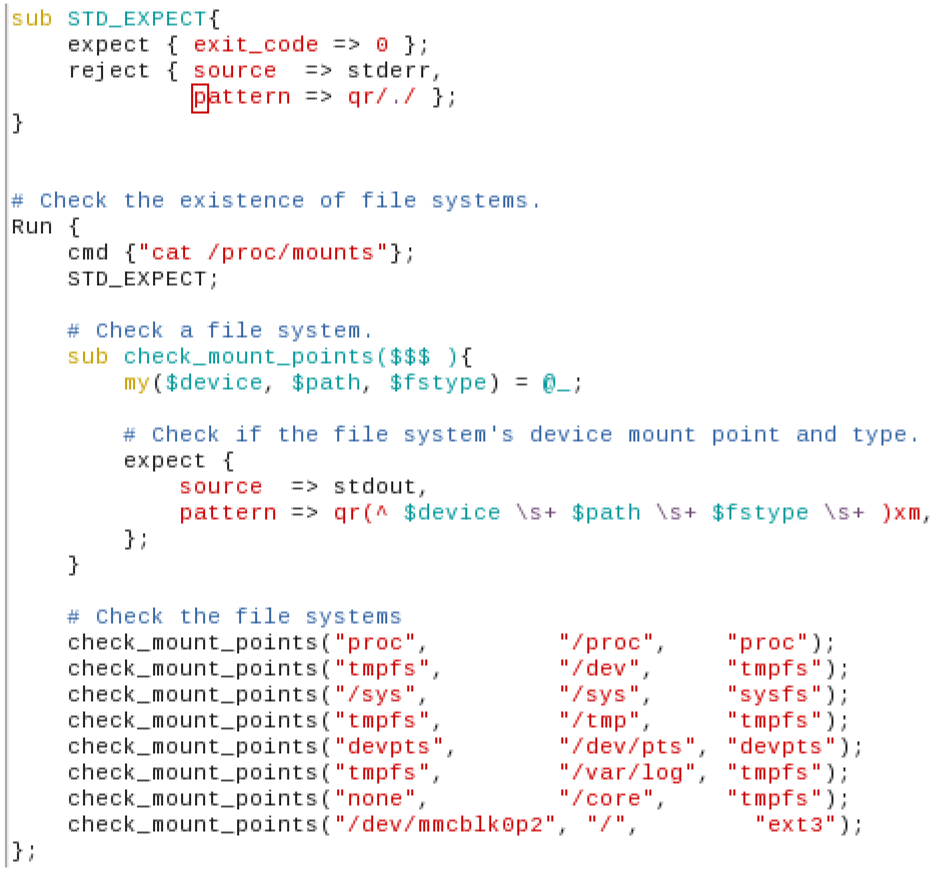
- Looping construct
- Can loop across multiple variables
- Annotates test output with active variation state
- Preserves captures for each iteration

```
use ATF::Test;
use strict;

our($Service, $Command);
Variation Service => [ 'sshd', 'ntpd' ],
          Command => [ 'restart', 'status' ], sub{
□
     Run{
         name    { "Running $Command on $Service" };
         cmd     { "/etc/init.d/$Service $Command" };
         expect { exit_code => 0    };
     };

     Run{
         cmd { "echo Running $Command on $Service" };
         expect { source => stdout,
                  pattern => qr/(.*)/,
                  capture => [saw => 1] };
     };
};

DumpCapture();
```

```
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Launch Run 5 : 'Running restart on ntpd'
#       Command : '/etc/init.d/ntpd restart'
#        Target : 'default'
#     Variation : [Service => 'ntpd', Command => 'restart']
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Evaluate Run 5 : 'Running restart on ntpd'
#     Variation : [Service => 'ntpd', Command => 'restart']
#
ok 9 - Run to completion  '/etc/init.d/ntpd restart'.
ok 10 - expect : exit_code => 0
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Launch Run 6 : 'echo Running restart on ntpd'
#        Target : 'default'
#     Variation : [Service => 'ntpd', Command => 'restart']
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Evaluate Run 6 : 'echo Running restart on ntpd'
#     Variation : [Service => 'ntpd', Command => 'restart']
#
ok 11 - Run to completion  'echo Running restart on ntpd'.
ok 12 - expect : pattern => (?-xism:(.*)) source => stdout
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Launch Run 7 : 'Running status on ntpd'
#       Command : '/etc/init.d/ntpd status'
#        Target : 'default'
#     Variation : [Service => 'ntpd', Command => 'status']
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Evaluate Run 7 : 'Running status on ntpd'
#     Variation : [Service => 'ntpd', Command => 'status']
#
ok 13 - Run to completion  '/etc/init.d/ntpd status'.
ok 14 - expect : exit_code => 0
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Launch Run 8 : 'echo Running status on ntpd'
#        Target : 'default'
#     Variation : [Service => 'ntpd', Command => 'status']
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Evaluate Run 8 : 'echo Running status on ntpd'
#     Variation : [Service => 'ntpd', Command => 'status']
#
ok 15 - Run to completion  'echo Running status on ntpd'.
ok 16 - expect : pattern => (?-xism:(.*)) source => stdout
#
# $capture = {
#                'ntpd' => {
#                            'restart' => {
#                                           'saw' => 'Running restart on ntpd'
#                                         },
#                            'status' => {
#                                          'saw' => 'Running status on ntpd'
#                                        }
#                          },
#                'saw' => 'Running status on ntpd',
#                'sshd' => {
#                            'restart' => {
#                                           'saw' => 'Running restart on sshd'
#                                         },
#                            'status' => {
#                                          'saw' => 'Running status on sshd'
#                                        }
#                          }
#              };
# $current = $capture;
1..16
```

Native Perl

- Can handle corner cases
  - Supports POSIX
  - Support for system calls
- To reduce repetition
  - Subroutines are easy to use
  - The .. operator helps generate lists
  - A test group can easily make reusable libraries of common functions



```perl
sub STD_EXPECT{
    expect { exit_code => 0 };
    reject { source  => stderr,
             pattern => qr/./ };
}


# Check the existence of file systems.
Run {
    cmd {"cat /proc/mounts"};
    STD_EXPECT;

    # Check a file system.
    sub check_mount_points($$$ ){
        my($device, $path, $fstype) = @_;

        # Check if the file system's device mount point and type.
        expect {
            source  => stdout,
            pattern => qr(^ $device \s+ $path \s+ $fstype \s+ )xm,
        };
    }

    # Check the file systems
    check_mount_points("proc",         "/proc",    "proc");
    check_mount_points("tmpfs",        "/dev",     "tmpfs");
    check_mount_points("/sys",         "/sys",     "sysfs");
    check_mount_points("tmpfs",        "/tmp",     "tmpfs");
    check_mount_points("devpts",       "/dev/pts", "devpts");
    check_mount_points("tmpfs",        "/var/log", "tmpfs");
    check_mount_points("none",         "/core",    "tmpfs");
    check_mount_points("/dev/mmcblk0p2", "/",        "ext3");
};
```

Future Plans – Packet Injection

- Currently using a prototype inhouse
- For internal data channels
- Could be adapted to TCP or other data stream

```
Parallel{
    port_listen {
        port    { "bar1", "bar2", ..., "barN" };
        reject  { Packet::Match(SA => '123CDE') };
        timeout { 2 };
        #TODO { "Not done yet." };
        #SKIP { "Don't try yet." };
        background;
    };

    Parallel{
        port_listen {
            port    { "foo1", "foo2" };
            expect  { Packet::Match(SA => '123CDE') };
            capture { seq_num => Packet::Field('seq') };
            timeout { 2 };
            background;
        };

        port_inject {
            port { "baz" }; # One at a time
            send { TcpPacket::Build(SA => '123CDE') };
        };
    }; # wait here at most 2 seconds

    # Test Packet
    my %packet = (SA         => "ABC123",
                  DA         => "FFFFFFFFFFFF", # broadcast (?)
                  tcpPayload => ("XX" x 100));  # 100 bytes of random payload

    # Listen for broadcast.
    Variation port => ["bar1", "bar2", ..., "barN"], sub{
        port_listen {
            port    { $port };
            expect  { Packet::Match(%packet) };
            timeout { 5 };
        };
    };

    port_inject {
        port { "baz" }; # One at a time
        send { Packet::Build(%packet) };
    };

}; # wait here at most 5 seconds
```

Future Plans – Improvements

- Remote Session Reuse
- Output Issues in HTML Report
- More Data Capture (Logs, Files)
- Store Results
- Wrap more existing features in Perl's test harness
- Packaging and Dependency Bundling
- Interactive Mode (Pause, Runtime Input)
- Random Variation Subsets
- Other Protocols (CIM?)

To Conclude...

- Testing is important at all levels of development.
- Tests must be easy to write or they won't be written.
- Tests must be easy to run or they won't be used.
- Use or make a Domain Specific Embedded Language
  - For ease of use.
  - For flexibility to handle unexpected corner cases.

Questions?

- Contact
  Daniel Hursh
  hursh@us.ibm.com

- ATF Homepage
  https://sourceforge.net/projects/atf-test

- ATF Source
  git clone git://git.code.sf.net/p/atf-test/code atf-test-code