CGL Validation Framework

Seminar on High Availability and Timeliness in Linux

Jaakko Kyrö
University of Helsinki, Department of Computer Science
jaakko.kyro@cs.helsinki.fi

4th March 2003

Abstract

The Carrier Grade Linux (CGL) is an effort by the Open Source Development Laboratory (OSDL) to define an enhanced form of Linux that is tailored for the needs of the emerging communications market. This document describes the concepts, components and policies to validate an implementation against the CGL requirements. The purpose of the Validation Framework is to provide open source testing tools for CGL implementors.

1 Introduction

This document describes the OSDL Carrier Grade Linux Validation Framework, as based on the CGL Validation Framework document version 1.1[5]. The following section describes the methodology involved in designing tests. Further chapters cover the design guidelines and development environment followed by a conclusion.

The purpose of the Validation Framework is to establish a standard for testing CGL implementations. The CGL Architecture specification[3] provides merely implementation examples, it doesn't mandate a specific implementation. The framework specification provides the following information:

- The source code directory structure for developers contributing to the validations
- The installed package directory structure for executing tests in the framework, and
- The rules and guidelines for those interested in contributing to the OSDL CGL Validation suite.

1.1 Overview

The Validation Framework consists of a collection of test suites and the rules and guidelines for creating new test suites. The test suites are organized hierarchically, and every test has a dependency to a requirement in the CGL Requirements Specification[4]. The tests associated to a requirement comprise a requirement validation suite, and the collection of requirement validation suites is called the CGL Validation Suite. Figure 1 shows different parts of the framework.

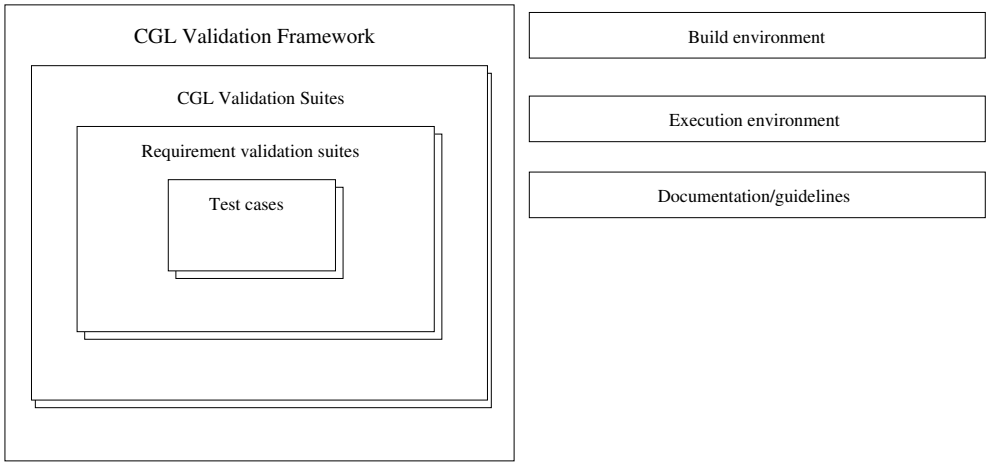

Figure 1: The CGL Validation Framework concepts

1.2 Organization

Making a validation suite involves different human resources. For producing a Validation Suite, following roles and responsibilities have been identified for the validation team.

The source repository has the roles:

Validation Framework Administrator The person who owns the entire OSDL CGL Validation Suite, which includes all validation suites added by Suite Contributors. This role is responsible for integrating new suites into the validation suite.

Suite Maintainer The person responsible for the changes to a specific suite in the full validation framework.

Suite Contributor An individual in the open source community who has provided a validation suite for CGL. Contributions are added in co-operation with the Suite Maintainer.

Trusted Suite Contributor A Suite Contributor can become a Trusted Suite Contributor over time. Can directly add suites into the CGL Validation Suite.

For the usage of the test suite as a whole, additional roles are needed:

Validation Suite Tester Responsible for running the full CGL validation suite and ensuring that each release of the validation suite meets the defined criteria for release.

Validation Suite Customer A person or organization that will use an official release of the CGL validation suite to analyze a given CGL implementation.

2 Methodology

Test suites added to the framework contain a number of functional an non-functional tests. Functional testing ensures that the functionality and interfaces of a component meet their requirements. Non-functional testing is for checking certain characteristics over and above the correct functionality.

2.1 Functional testing

Functional testing is carried out on a single component. The purpose is to ensure that a component meets the CGL requirements imposed upon it. For example, components are tested for normal operation, error conditions, standards conformance etc.

The requirements usually explicitly state the desired behavior of a component, or they contain references to other standards or Application Programming Interface (API) definitions. A requirement may also give no reference to the desired behavior at all. The Validation Framework gives guidelines for writing test for all of the cases above.

It is often not trivial to directly derive test cases from a given requirement even if the requirement states the behavior of a component somehow. Usually a requirement needs to be refined and split into sub-requirements in order to have something testable.

The guidelines for testing concerned with different kinds of requirements are described below:

Standards conformance Some CGL requirements refer to existing standard specifications (for example the POSIX Timer support). When possible, existing standards test suites are added to the validation framework. If no such test suites exist, an open-source project will be created for the standard in question.

API conformance APIs are the basic unique functional unit s that make up a function, component or a requirement. The majority of the functional software testing required for a CGL implementation consists of API testing. The composition and characteristics of each API must be determined, and individual test cases to cover the API must be created.

Items to test for each API are the following:

- Parameter selection: Minimum, maximum, normal and error conditions for each range of values.
- Parameter combination: Predictions for each parameter combination, and verification of results.
- Call sequence: The ordering of API calls, if there are such restrictions.

It might not be practical to test every individual API. In this case, a risk analysis should be performed and documented as well as the result.

There should also be conformance tests for APIs specified by the CGL architecture. In this case, the requirements documentation should describe the API accurately enough.

No functionality reference The functionality of the component in question should be determined by other means. One method is to examine all existing documentation about the component in question. An analysis of an implementation can also be done.

2.2 Non-functional testing

Non-functional testing doesn't directly cover the functionality of a component. The goal is to confirm a component's correct behavior with respect to conditions external to the component itself. Examples of non-functional testing:

Integration Verifying the correct co-operation of multiple components. The components can be hardware or software components, in this document, only software components are discussed. Involves testing the operation of the component and the system, when a component is introduced into the system.

Reliability Testing the system's behavior under given conditions for a specified period of time. The given conditions may be for example heavy CPU load, heavy memory load, etc. There are different types of reliability tests:

- Continuous operation, to ensure that the system can run for prolonged periods of time during use.
- Volume load, sending large amounts of data into the component or system to detect errors.
- Stress tests, to determine the maximum levels of load before the performance of the component or the system degrades below acceptable limits.

Performance Testing that determine the optimal conditions under which the component or system will operate. Involves measuring response times or throughput and locating bottlenecks.

The CGL Requirements Definition includes some non-functional requirements. Test cases for these requirements should be made following similar strategies as in functional testing described in section 2.1. In addition, to the explicitly stated requirements, there are some implicit non-functional requirements, which are necessary to provide an acceptable product to users. Some approaches for designing tests for the implicit requirements below:

Use cases A set of use cases should be generated for the system. A simulated environment could then be created using the details defined in the scenarios. Some scenarios can be present in the architecture description, or they can be defined by the customer or the tester.

Load-testing This approach involves running load tests beyond the physical capabilities of the system to ascertain the limits of the overall system. The data gathered should present the actual limits of the system in terms of performance, availability and system parameters (memory footprint, CPU usage).

Fault injection Hardware failures are introduced during operation, for example detaching communication cables and disks.

The data generated by the tests needs to be analyzed. The standards by which the data is measured might not be present in the case of implicit requirements. In this case, external data can be used in the analysis, for example benchmarks of similar systems.

3 Development guidelines

For development of the validation suites, the Validation Framework defines some practical guidelines. These involve source code organization, required files and naming of directories. The guidelines must be followed to take advantage of the build framework.

Each test consist of a number of test cases. Each test case must trace back to a requirement it is validating. The traceability is inherently present in the source code organization guidelines, described in the following section.

3.1 Source code organization in the Validation Suite

Each validation suite must follow a standard naming convention and directory structure:

- The full OSDL CGL Validation Suite will be stored as a single component, entitled `cglvalidation`, in the OSDL CGL development environment.
- Within the `cglvalidation` component directory there will be one directory per OSDL CGL requirement category. The requirement category is one of the requirement categories defined in the OSDL CGL Requirements Definition (availability, serviceability, etc.).
- Within each category directory will be the directories for each requirement validation suite listed according to the requirement name. The requirement name is the name following the requirement ID number (such as remotebootsupport for requirement 2.2 Remote Boot Support or eventlog for requirement 1.3 Event Logging POSIX IEEE 1003.25).
- Within the directory for each requirement, there must be a `bin/`, `src/`, and `doc/` directory as well as a `Makefile` and a `.spec` file. The contents of the directories and files follow:
  - `bin/` – This is where any binaries necessary for execution of the given validation suite are stored.
  - `src/` – This directory contains the source files and scripts associated with the validation suite.
  - `doc/` – This directory is where any documentation for the validation suite is stored
  - `.spec` – This file contains the information needed to make an RPM (Red Hat Package Manager) package out of the validation suite.
  - `Makefile` This file is used by the build system to make the RPM for the validation suite. It must have sections to "make all", "make clean", and "make rpm."

3.2 Source file directory structure

The following syntax defines the general scheme for naming the source directories:

```
cglvalidation/<CGL requirement category>/<CGL requirement name>/bin
cglvalidation/<CGL requirement category>/<CGL requirement name>/src
    ...wrap.sh
    .../<CGL subrequirement 1>_<feature 1 name>/*.c,*.h,etc.
    .../<CGL subrequirement 1>_<feature 2 name>/*.c,*.h,etc.
    .../<CGL subrequirement 2>_<feature 1 name>/*.c,*.h,etc.
    .../<CGL subrequirement 2>_<feature 2 name>/*.c,*.h,etc.
cglvalidation/<CGL requirement category>/<CGL requirement name>/doc
```

The requirement category refers to the categories present in the CGL Requirements Definition, i.e. Standards, Availability, Platform etc. The requirement name also refers to the name of the requirement in the CGL Requirements. The sub-requirement means a subsection of a larger requirement, for example the Hot Swap Support requirement has three sub-requirements, Hot Insert, Hot Remove and Hot Device Identity.

If a requirement has no sub-requirements, the requirement name should be repeated instead of the sub-requirement name.

3.3 Required files

All files necessary to test a requirement need to be placed in the proper place in the directory structure outlined above. The files listed below are the minimum required files for any given requirement.

- The main directory should contain a copy of the validation suite license, the `Makefile` and the `.spec` file.
- The `bin/` directory should contain all the executable files to run the test. This includes post-processing tools and utility programs.
- The `src/` directory should contain all source files needed to execute the tests and a wrapper script called `wrap.sh`
- The `doc/` directory should have at least a HTML file that can be used as an introduction, named `index.html` and a README file that should contain general documentation for the test, i.e. the requirement, execution instructions, dependencies, and contact information.

The wrapper script is responsible for executing all the tests and storing results. The script can be of any common shell language (e.g. Bash, Perl). The results should be stored inside the `<CGL requirement>/results` directory. The format of the results is also defined.

3.4 Test results

The results provided by the tests should follow certain conventions. This way the results for the individual tests can be aggregated and analyzed in a unified and automated fashion. The results are written into a file called `results.txt` in the directory mentioned in section 3.3.

The results file format is XML. The most important elements are described below:

Requirement Identifies the requirement under test.

Requirement description A brief description of the requirement.

Requirement ID The requirement identifier which must match with the CGL Requirements Definition.

Subrequirement Identifies a sub-requirement by a name and a number.

Feature Identifies the feature to be tested. Should match the directory naming conventions. Each Feature element in the file contains the bulk of the test results information. The following attributes are defined:

Name The textual name of the feature

Total The total number of test cases

Pass, Fail The number of tests that passed or failed, respectively.

Time Time to run all tests

MemUse The memory sage during execution

CPUUse The CPU usage during execution

Transactions The number of transactions that occurred during the execution.

Data Each feature is likely to have log data, which should be contained in this element

Graphs Optional graph data produced by the test.

3.5 Test maintenance

The test suites are undergoing constant development. Existing suites are revised, enhancements added etc. The Suite Maintainer is the key role in maintaining the existing test suites. Coordination between maintainers of different suites is also crucial. The set of test suites and the framework are versioned so that new baselines can be established.

Test maintenance is categorized as follows:

Corrective maintenance This type of maintenance means fixing defects in existing test suites.

Adaptive maintenance It may be necessary to port test suites to other system architectures. Enhancements to the test execution utilities for distributed testing also fall in this category.

Perfective maintenance This category stands for various optimizations. The wrapper scripts or controlling components may be enhanced to allow more efficient execution. Tests may also be enhanced for better scalability or more thorough testing.

All patches submitted to a suite maintainer must contain information about the type of maintenance performed, problem description and change resolution information. Changes must also be reviewed by the maintainer and tested against a known version of a CGL build. The maintainer is responsible for accepting or rejecting the patch.

Porting tests to other system architectures require a maintainer, whose sole purpose is to coordinate the porting process. A branch of the baseline source is created for the port. Once the port is finished, the build tools will have the capacity to selectively create binary packages for the specified architecture.

4 Validation suite usage

The Validation Suite, when used to test an existing CGL implementation, imposes some requirements on the system to be tested and to the deployment mechanisms of the suite.

4.1 Test environment

The test suite must be capable of running in a native CGL environment. The necessary dependencies of the test suite must be installed on the system. Therefore, the system under test (SUT) must be capable of receiving binaries or source from the test suite repository. In addition, the SUT must be capable of downloading additional software packages from the Internet for satisfying support application dependencies.

2 GB of disk space should be reserved for executing the tests. Some tests may require an additional host, which should be similarly configured.

For deployment, all the dependencies of each test suite should be packaged as a part of the Validation Framework. The executable tests will be contained in a single downloadable image. Once downloaded and unpacked, all of the support utilities and test cases will be located under a common specified directory for easy cleanup.

4.2 Environment tools

General utilities are created to support the execution, data collection and report analysis. These utilities are kept separate of the actual tests. The goal is, that the utilities were shared among other test suites in the framework.

As the dependencies of a test suite are identified, a description of all tools, utilities, and test harnesses should be included with the test suite (A `README` file). Each test suite is intended to be self-contained with a full description of the environmental dependencies needed to execute the test suite in question.

The usage of test harnesses is not restricted. In the case of such requirement, the test harness should be included in the test suite. If this is not possible, a descriptive file providing instructions on how to locate, download, configure and use such a test harness should be included.

5 Support mechanisms

To aid in the development of test suites, a build framework and an execution framework have been set up by the OSDL CGL Validation Subgroup[1]. The build framework provides test suite developers with a central source repository, a collection of build automation tools and Web-based feedback on build and test results as well as test planning facilities. The execution framework has a number of Proof of Concept CGL builds, under which the test suites can be executed and validated.

The structure of each validation suite enables the implementation of the automated features. For example, the directory structure and the file naming conventions enable a tool to generate a web page representing a map of requirements to test cases.

5.1 Build framework

There is a build framework established by the OSDL CGL Validation Subgroup. The framework consists of a source repository, automated build system, automated basic acceptance test system and web-based feedback and test planning systems. The purpose of the build framework is to assist the development of the validation suites.

The source repository includes the source code of the test suites submitted to the framework. Binary and source packages of the validation suites are made from this repository. Thus, the source code submitted to the repository must be compilable and perform properly.

Two kinds of releases are made from the repository: A “stable” release and a “development” release. The development packages contains the latest build of the current versions of the test suites. These typically provide the latest bug fixes to existing tests as well as newly developed tests, but are more likely to have bugs. The stable packages have undergone extensive verification to assure correct behavior. These packages in turn are typically used to validate a CGL implementation in its final stages, and to test compliance with the CGL Requirements.

5.2 Execution framework

The OSDL provides an execution framework for testing the Validation Suite. The execution framework provides automation for test bench configuration, test execution and results.

The execution framework utilizes the Scalable Test Platform (STP)[2]. STP is a system for automating the QA testing process for the Linux Kernel, as well as automating benchmarking and regression testing on diverse hardware systems. At the OSDL, this application is used to

provide a public STP service for testing of the Linux kernel. The Validation Framework utilizes the STP for testing the validation suites using the OSDL Proof of Concept CGL build.

Figure 2 displays an overall view of the Scalable Test Platform (STP). The PLM stands for Patch Lifecycle Manager, it is a tool for tracking the patches submitted by kernel developers against the official Linux kernel source code. The PLM can track dependencies between patches, so that the kernel can be automatically configured.

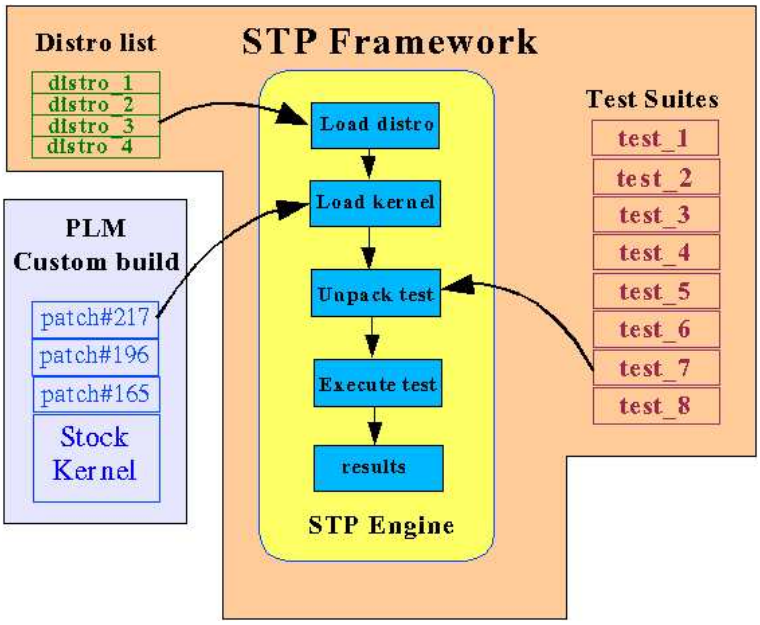

Figure 2: The Scalable Test Platform

The STP has a number of hardware platforms, one of which is chosen as the test platform according to the tester's requirements. The host is then cleared of all code and a new installation of a Linux distribution of choice is performed for each test run. Afterwards the results are collected and an email notification is sent to the tester.

6 Conclusion

The OSDL Carrier Grade Linux Validation Subgroup has defined the OSDL Carrier Grade Linux Validation Framework. The framework contains validation suites for testing CGL implementations. The development of the validation suites is done in an open source manner, and the CGL Validation Subgroup provides collaboration for developers. The CGL Validation Framework Document provides guidelines and rules for contributing to the development of the test suites.

References

[1] The CGL development website. http://developer.osdl.org/.

[2] The Scalable Test Platform website. http://stp.sourceforge.net/.

[3] T. Anderson et al. Carrier Grade Linux Architecture Specification. Technical report, Open Source Development Laboratory, inc., 2002.

[4] T. Anderson et al. Carrier Grade Linux Requirements Specification. Technical report, Open Source Development Laboratory, inc., 2002.

[5] R. Lynch, S. Glass, C. Thomas, and J Fleischer. Carrier Grade Linux Validation Framework. Technical report, Open Source Development Laboratory, inc., 2002.