Apertis test strategy

Contents

Apertis is an Open Source project which consists of multiple parts that are reflected in the current structure of Apertis Gitlab[1]:

- Packages as the fundamental building blocks of the images
- Infrastructure to provide the tools and automation to build the images
- Tests which ensure that Apertis provides high quality standards

This structure also shows that **tests** are one of the pillars of this distribution.

The QA process takes advantage of the tests to confirm that the behavior of each component is the expected one. During the testing any deviation is reported for further investigation, as described in the Apertis QA page[2].

For a successful QA process a test strategy should be followed in order to:

- Make sure all the relevant parts are tested, with more focus in critical ones
- Provide reliable reports of the status of the different components
- Provide a reliable base to build additional checks

The goal of this document is to provide a strategy to maximize the profits of testing by putting the focus in the components with higher impact in case of an issue.

[1] https://gitlab.apertis.org

[2] https://www.apertis.org/qa/

Real life challenges in embedded Linux projects

Testing is what ties all the pieces together in a project to convert it in a success. Without testing, a project will most probably fail, since the output of one stage won't meet the expectations of a next one. Also, management and risk assessment is not possible for projects where a test strategy does not provide certainty. In the end, a product derived from this type of project will be a failure due to different possible reasons:

- The product might fail to meet the expectations of the consumer
- The budget associated to the project will be overspent
- The times constraints associated to the project will not be met

This is true in general; however, in embedded Linux projects there are specific challenges to take into account. Traditionally, embedded Linux projects are thought as monolithic software, which basically consists in building full images from several pieces of software with product specific customizations on top. While for small projects this usually includes only small customizations and a custom application on top, usually not requiring any special feature, for more complex projects this approach does not scale well.

The reason behind this fact is that on complex projects there are many more variables to be considered:

- Simple projects consist of standard embedded Linux image and an application on top of it, while more complex ones usually require customizations of many different pieces of software.
- Simple projects usually have a local team working on one piece of software, while complex ones tend to have globally distributed teams working in many different pieces of software.
- Simple projects are usually meant to run on well-known and reliable hardware, while complex ones are likely to run in hardware which is also being developed, adding extra uncertainty.
- Simple projects usually are self contained with little interaction with other systems, on the other hand complex are challenged by interactions with other embedded systems or with external services, such as cloud infrastructure.

It is clear that with so many variables involved, a way to decouple and validate changes through testing is vital for the success of a project. With this in mind, the Apertis test strategy is based on the concept of a binary package distribution, from which Apertis inherits its strengths.

Development workflow in binary package distribution

A test strategy is tied to a development workflow since it should provide certainty to the different stages of development. In this context it is important to highlight the development workflow on Apertis, since it is quite different from other embedded Linux projects.

Apertis is a binary package oriented distribution which means that development is based on packages, which are the buildings blocks of images. This approach makes it natural to develop new pieces of software or improving existing ones by changing packages which can be tested isolated from the rest of the system.

With a package centric approach each package is self contained, including:

- Source code
- Unit tests
- Documentation
- Custom patches
- Rules to build and install
- Copyright information
- Custom CI configuration

This isolation helps in different ways:

- Developers can apply changes on a package without being affected by changes in other packages
- Developers can test their changes locally in an well known environment decoupled from external systems
- Changes can be tested in CI in a well known environment before they get merged
- Potential issues are caught earlier during the development process

Additionally, the fact that Apertis supports multiple architectures helps both development and testing as changes in a package can be validated in a different environment. Also, taking into account that Apertis SDK is built using the exact same packages, the development and testing is straightforward. Therefore:

- Developers can build their changes in Apertis SDK
- Developers can test their changes locally in Apertis SDK by only installing the new version of the package
- Changes can be tested in CI using a runner with a different architecture before they get merged

Finally, integration is easily done by installing a custom set of packages to build the desired image. Since packages are prebuilt, integration is a simple process and packages can be reused to build different types of images, providing higher flexibility as well as faster build times for images.

The mentioned characteristics from Apertis overcome the difficulties presented in the traditional monolithic approach of embedded Linux, making it the best option for complex projects.

Testing in binary package distribution

To take advantage of the benefits of a binary package distribution source packages needs to be designed to be self contained in terms of functionality and testing. This means that a source packages should include not only the functionality it is meant to provide, but also a way to validate it. Providing the test functionality could be very challenging in some scenarios, but the benefits of it are worth the price, since it allows scaling in complex projects.

The following guidelines allow source packages to provide the test functionality:

- From source packages several packages can be built, which could potentially include:
    - Binary packages meant to be used in target devices
    - Alternative binary packages with limited functionality based on architecture that can be used in development to test basic functionality
    - Alternative binary packages meant to be used in development which provide functionality to emulate the interaction with other systems
    - Alternative binary packages meant to be used in development with additional monitoring and diagnostic functionalities
- During development, the use of alternative packages allows testing the core of the source code
- On building, unit tests should be run to ensure basic functionality
- During review, both the main and the test functionality should be checked to provide as much coverage as possible
- Before integrating changes into main branches, basic automated integration tests on different hardware should be performed.
- After integrating changes into main branches, integration tests need to be run to ensure no regressions are found.

Classifications

The first step towards solving a problem is to understand and describe it. This section aims to do that by describing how different components are classified and the criteria used for the classifications.

Components

For the purpose of this document the term **component** is used to refer to an item to be tested. A component can match a package or a set of packages that work together to provide a certain functionality. A component can be further divided in sub-components if it is necessary to improve the testing of some specific functionality.

Component metrics

The level of testing required in each case should be determined taking into account different aspects:

- **Component source**: One of the key elements to understand the level of test required is the source of the component. Under this category we can find different cases:
  - Upstream components, for example **systemd**.
  - Upstream components with significant Apertis-specific changes, like the **Linux kernel**.
  - Apertis-specific components, such as the **Apertis Update Manager**.
- **Upstream activity**: Another key element to evaluate is how much a component is actively developed:
  - High upstream activity, as an example the mainline **Linux kernel**, **systemd**, **rust-coreutils**.
  - Medium upstream activity, like **OpenSSL** or **GnuPG**.
  - No or minimal upstream activity, like the tool **lqa**.
- **Component commonality**: Some components are more common than others, depending on the functionality they provide, thus having them used by a wider range of users:
  - High: Components under this category are common to any Apertis image. A good example of this is **systemd**.
  - Normal: Components that are common to an important set of use cases, such as **Docker**.
  - Low: This component has a very specific use case, like the **Maynard graphical shell** (the reference shell).
- **Component criticality**: Some components are more critical than others, depending on the functionality they provide and the use case. Since Apertis is an Open Source distribution the criticality is evaluated from a general perspective. However, product teams and Apertis derivatives in general are encouraged to adjust this metric according to their specific needs/use cases. The different criticalities used by Apertis are:
  - High: Components under this category provide a critical functionality which is essential for the system. A good example of this is the **Linux kernel**.
  - Normal: Components under this class provide a functionality that is not critical for the system, but still required. For instance, **tracker**.
  - Low: This group provides functionality desirable but not required for the system. An example for this category is **cups**.
- **Component target**: We use this category to differentiate components

based on their target environment:

  - Target: Components aimed to be shipped on target devices.
  - Development: Components specific to development environment.

Loops and types

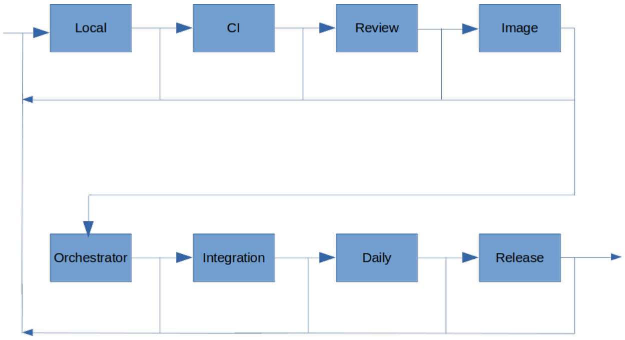
There are different stages of testing in the QA process which help to define different level of loops. The purpose of testing is to spot deviations from the expected behavior which is the first part in the loop. The second part is to correct such deviations to provide the desired behavior. The iterations in each loop are run until the result of the tests show the expected behavior. It is important to note that every loop includes the previous ones as prerequisites, making sure that any change in the code is evaluated in all the defined loops. Additionally, it is interesting to note that, inner loops have less impact, since they affect smaller groups.

The current defined loops are:

- Local loop: This loop is the closer one to development, which includes developer testing and unit tests that are used during development. Based on the results and after several iterations, the developer improves the quality of the changes he is preparing before submitting a Merge Request. As result of this loop a Merge Request is submitted.
- CI loop: This loop includes the previous one and goes a step beyond, taking advantage of the Gitlab CI and its OBS integration. The proposed changes in a Merge Request are tested with linters, license scanners and built in OBS, which includes running its unit tests. Additionally simple integration tests can be run to confirm the changes will not introduce any regression. As result of this loop all the pipelines associated to the Merge Request pass.
- Review loop: The review process is a key element in the Open Source culture, which allows developers to receive feedback of the proposed changes. During this process the reviewer can suggest small changes/fixes or even a complete different approach to reach the same goal. The feedback needs to be addressed and any change will trigger additional iterations in this loop. As a result the Merge Request is then merged or discarded/replaced.
- Image loop: This next loop focus in the image generation and initial integration testing, and is the first loop going beyond component isolation. Here, some aspects of integration are evaluated, like the installation, package dependencies availability check, as well as license compliance checks. As a result a set of reference images are made available for all the supported architectures.
- Orchestrator loop: One step further, a new loop is formed by the daily orchestrator runs, which builds the different types of images used by Apertis,

including Docker images used for development and CI, toolchains, flatpak runtimes, apart from the standard Apertis images.

- Integration tests, automated (LAVA) or manual: This next loop also includes automated tests that are run in LAVA on actual devices of different architectures to confirm some behavior works as expected. Manual integration tests aid to the process to cover for some functionality that cannot be tested automatically for example. In this loop platform specific tests can be added to validate hardware specific functionality.
- General use of daily images: An important loop is the general use by the community, developers and downstream distributions of daily/development images, which can fill the gap in case deviations from the expected behavior are detected and reported. Daily images use `-security` and `-updates` repositories which provide newer versions of the available packages. The distinction between these two types of repositories is important, since `-security` is used to publish high critical updates that should be applied without delay, while `-updates` is used for non-critical ones. The recommendation for production is to use the base repository plus `-security` to provide a reliable platform, while development can take advantage of the newer features already available in `-updates` which will include in the next release.
- Common use of release images: Similar to the previous loop, this one takes advantage of the common use of release images. The main difference here is the audience, since release images are the recommended ones in general and thus have a bigger userbase. During a release, the folding is applied, which consists in merging the changes from `-updates` and `-security` into the main branch, used as a starting point for the next release.



During these loops different types of tests are performed:

- Functional: Tests aimed to confirm that the desired functionality behaves as expected.
- Performance: Tests meant to verify the performance parameters are within a defined range.

- Security: Tests used to confirm that no known security vulnerability is found.

The way of implementing these types of tests is tied to the component under analysis, but all the aspects described above should be taken into account. As a result, guideline tests should be able to mimic real life usage as much as possible including very unlikely ones. It is a common issue that a component is tested only taking into account the functional aspect of it, but later on when it is tested under real life conditions and stress, performance parameters do not comply with the expectations.

The main purpose of testing is to spot deviations from the expected behavior which is the first step in any loop. The second step is to correct such deviation to provide the desired behavior.

Priorities

The testing process will result in either a success or a failure, in the last case, a bug report should be filled and triaged in order to prioritize the more critical issues:

- Critical: Deviations from the expected behavior in critical components that are unacceptable for a release use this priority. This type of issue is considered a release blocker and should be addressed with the highest priority. A good example of such issue would be when an image fails to boot.
- High: Deviations from the expected behavior in critical components that are not considered release blockers are triaged under this category. One example of such issue would be a crash in a critical component that only happens in a very specific scenario.
- Normal: Deviations from the expected behavior in non-critical components are triaged under this category. An issue in the language support could be a good example of this type of issue.
- Low: Deviations that do not affect the expected behavior fall into this category. As an example a log entry not expected or a minimal visual deviation.

Constraints

To develop a sustainable test strategy the constraints for testing also need to be taken into account, in order to provide the best possible trade off. Having this in mind, the following list describes the possible constraints.

- Environment availability: Tests require some type of environment to be executed, depending on the type of test this can include:
    - Development computer.
    - Server/Virtual Machine: e.g. Gitlab runners.

  - External service: Gitlab, LAVA.
  - Reference boards: iMX6 Sabrelite, RenesRenesas R-Car M3, UP Squared 6000. From the previous list, the availability of reference boards to run a test is the most challenging one, since it implies having boards of different types, models and architectures, in order to be able to confirm the expected behavior of each test.

- Time availability: Even with the right environment time is always a challenge, due to different reasons:
  - Environment shared among different projects.
  - Test periodicity, since some tests are meant to be run regularly.

- Maintenance costs: The number of tests and supported boards/environments have a direct impact in the maintenance costs of both software and hardware.

Strategy

Since both the number of components and possible tests is huge, plus the constraints involved, it is not possible to test everything, be it functionality, behavior or component. Based on this, the test strategy should provide a guidance to where to put the focus on in order to maximize the cost-benefit.

Additionally the test strategy should provide a reference to triage any issue found during the testing.

The strategy should also take advantage of the loops previously defined in order to spot any issue in the loop nearest to the local one in order to reduce its impact.

Classify components

To help selecting what tests to include or to support, the strategy suggests classifying each component or component group as follows:

Source

- 1: Apertis specific components
- 2: Upstream components with significant Apertis specific changes
- 3: Upstream components that are unmodified or with minimal changes

Upstream activity

- 1: Component with no or with minimal upstream activity
- 2: Component with medium upstream activity
- 3: Component with high upstream activity

Commonality

- 1: Component has high commonality

- 2: Component has normal commonality
- 3: Component has low commonality

Criticality

- 1: Component has high criticality
- 2: Component has normal criticality
- 3: Component has low criticality

Target

- 1: Component in meant to be used on target devices
- 2: Component is specific to development environment

The following table uses a set of components as example to illustrate the approach:

Component	Source	Activity	Commonality	Criticality	Target
Linux	3	3	1	1	1
Linux UML	3	1	1	1	1
AUM	1	1	1	1	1
OSTree	3	3	1	1	1
connman	3	3	1	1	1
rust-coreutils	2	2	1	1	1
dnsmasq	3	3	1	3	2
QA Report App	1	1	1	2	2
Pipewire	3	3	1	1	1
Bluez	3	3	2	1	1
Flatpak	3	2	2	1	1

Define tests required for each component

Based on the previous evaluation the recommended tests for each component needs to be evaluated using a clear guideline.

- Local loop
  - Developer tests
    - Required: all components
  - Unit tests
    - Required: components which support unit tests
    - Encouraged: components that are under development, typically this is the case of Apertis specific components
- CI loop
  - Linters
    - Encouraged: components that are under development, typically this is the case of Apertis specific components
  - License scan

        - Required: all components included in target images
    - OBS build
        - Required: all components
    - Small integration tests
        - Required: components with low community activity and high commonality/criticality that are under development, typically this is the case of Apertis specific components
        - Encouraged: components with high commonality/criticality
        - Desired: components with high commonality/criticality
- Review loop
    - Required: all components
- Image loop
    - Installation
        - Required: components with normal or high commonality/criticality
        - Desired: all components
    - License compliance
        - Required: all components included in target images
- Orchestrator loop
    - Installation
        - Required: components with normal or high commonality/criticality
        - Desired: all components
- Integration tests automated(LAVA) or manual
    - Functional tests
        - Required: all normal or high commonality/criticality
        - Desired: all components
    - Performance tests
        - Required: Apertis specific components
- Common use of daily images
    - Desired: User to test all components
- Common use of released images
    - Desired: User to test all components

Current status and gaps

The following table summarizes for each component in the sample the status according to the guidelines previously presented:

- 0: Some requirements are not meet for this loop
- 1: All the required tests are performed, additional tests should be encouraged
- 2: All the encouraged tests are performed, additional improvements can be done
- 3: All the desired tests are run

Component	Local	CI	Image	Orchestrator	Integration
Linux	3	1	3	3	2
Linux UML	1	0	3	3	2
AUM	2	0	3	3	3
OSTree	3	1	3	3	3
connman	2	1	3	3	3
rust-coreutils	2	0	3	3	3
dnsmasq	2	1	3	3	2
QA Report App	2	1	-	-	3
Pipewire	2	1	3	3	3
Bluez	2	1	3	3	3
Flatpak	2	1	3	3	3

Considerations for product teams

The previous sections provide general concepts around the Apertis test strategy from a general distribution perspective. However, since Apertis is used to build products these concepts needs to be applied in a way that supports the development process of such products.

The flexibility given by Apertis is vital to create an efficient workflow, and to make that happen some guidelines should be followed:

- Development should follow the Apertis workflow to enforce self containment and isolation, adding unit tests and Gitlab CI customizations to packages
- Components under development require special attention, so all the loops mentioned need to be used to maximize the benefits.
- Components under development need to include unit tests which exercise different aspects of the software.
- Components under development should include CI tests that are run in LAVA in the target hardware(s) where applicable. These tests should be run before changes are merged to confirm that certain functionality and/or performance of the new version are according to expectations.
- Since multiple teams work on the same product, it is important that tests are designed also based on other teams'expectations on functionality and performance.
- Close iteration between teams is needed when a team spots a regression introduced by other team. In that regard LAVA provides a single reference point to share tests, results and logs.
- Experience gained during development should be used not to only improve the component itself but also to improve the tests around it. This is a good way to avoid having the same issue in the future.
- Integration tests on target systems needs to be run, either automated or manual to validate the resultant image.

Apertis provides the infrastructure to support these guidelines and already implements them. However, each product team needs to decide how to implement them since each project has its own restrictions, requirements and scope.

As an example, a product team working in IOT project scenario[3] can use Apertis Fixed Function image recipe as reference and add the additional packages to build its reference image. In such case, the product takes advantage of an already well proven base reference, but needs to follow the above guidelines to make sure that no regressions and to extend the test coverage to include the new features.

In such scenario, new packages to provide application specific logic should:

- Include unit tests[4]
- Include CI tests running in LAVA[5] if possible
- Run Apertis test[6] for the still valid functionality
- Run additional either automate or manual tests[7] to check the new functionality

Follow up tasks

As this strategy provides general guidelines to avoid gaps between expectations and actual results some follow up tasks are suggested:

- Identify testing gaps: This document provides metrics for components and sets expectations regarding the test loops that should be in place. Based on these statements a more elaborated list of components and testing gaps needs to be built.

- Provide CI integration tests to run before merging: As described, components under development are the ones that could add instability to the development of a product. To minimize this risk, CI should run different types of tests before merging changes. The support for these kind of test is described in Apertis package centric tests[8] which takes into account the following considerations:

  - Should be based in pre-hooks to make it easier to extend to add additional checks, such as new linters.
  - Should include tests on LAVA on the target hardware when applicable.
  - Should be configurable to be able to avoid the overhead of building and testing components if it is not necessary, for instance during the folding.

[3] https://www.apertis.org/concepts/overview/#industrial-iot-scenario
[4] https://www.apertis.org/guides/unit_testing/
[5] https://www.apertis.org/guides/apertis-packages-testing/
[6] https://www.apertis.org/qa/test-data-reporting/
[7] https://www.apertis.org/qa/test_cases_guidelines/
[8] https://www.apertis.org/guides/apertis-packages-testing/

- Provide guidelines for developers to run local tests on different architectures using emulation, such as QEMU virtual machines or docker images.

- Provide a way to work with an interactive remote hardware environment for developers for debugging. LAVA is not meant to run in an interactive session with developers, however, a low level service could be implemented to allow developers to share the hardware and run debugging sessions.

- Provide a way to run visual regression tests. This type of test is very useful when developing applications that provide user interfaces since it allows catching unexpected changes earlier. An initial task should be to identify tools to be used to provide this types of tests and provide a sample test for a package.

- Robot Framework integration with LAVA has been planned, from which an implementation phase should be started.