Universidad Nacional de Mar del Plata

Facultad de Ingeniería

Departamento de Electrónica

Extensión, flexibilización y mejoras generales de plataforma de testeo de AN-LT para Optical-PHY

Ignacio Goldchluk

David Petruzzi

Trabajo Final de Graduación para acceder al título de Ingeniero Electrónico

Mar del Plata

Universidad Nacional de Mar del Plata

Facultad de Ingeniería

Departamento de Electrónica

Extensión, flexibilización y mejoras generales de plataforma de testeo de AN-LT para Optical-PHY

Ignacio Goldchluk

David Petruzzi

Trabajo Final de Graduación para acceder al título de Ingeniero Electrónico

Mar del Plata

Agradecimientos / Dedicatoria

Resumen

En el marco de mi trabajo en Inphi Argentina, he desarrollado mejoras, optimizaciones y agregado funcionalidades nuevas a una plataforma de testeo automático ya existente pero limitada implementada en Python para el protocolo AN-LT en dispositivos Optical-PHY NRZ y PAM. Entre las técnicas y tecnologías utilizadas se encuentran combinatoria, estadística y programación orientada a objetos. Estas mejoras y funcionalidades permitieron reducir el tiempo de desarrollo de tests, correr mayor cantidad de tests en menor tiempo e identificar con mayor rapidez bugs presentes en los dispositivos.

Python; pytest; code coverage; aseguramiento de la calidad; Auto Negotiation; Link Training; Programación Orientada a Objetos; NRZ; PAM; IEEE;

Abstract

In the context of my work at Inphi Argentina, I have developed improvements, optimizations and added new functionalities to an existing but limited automated testing platform implemented in Python for AN-LT protocol in Optical-PHY NRZ and PAM devices. The techniques and technologies used include combinatorics, statistics and object-oriented programming. These improvements and features allowed to reduce the test development time, run more tests in less time and identify in a faster way bugs present in the devices.

Python; pytest; code coverage; quality assurance; Auto Negotiation; Link Training; Object Oriented Programming; NRZ; PAM; IEEE;

Índice de Contenido

Introducción

En el presente informe se detallan las mejoras y funcionalidades nuevas agregadas a una plataforma de testeo de AN-LT existente implementada en Python.

La plataforma de testeo inicial de AN-LT contaba con varias limitaciones y estaba poco optimizada. No se tenía una métrica de cuanta funcionalidad se estaba testeando y era muy difícil sincronizar y reproducir bugs o escenarios específicos entre el equipo de QA (quality assurance) y los desarrolladores de firmware, puesto que las áreas utilizaban plataformas distintas para testear el dispositivo. Además el testeo de funcionalidad diario (test suite) tomaba demasiado tiempo considerando la cantidad de funcionalidad que se estaba testeando y no era escalable conforme se vaya agregando nuevas funcionalidades a los dispositivos.

En un principio se realizaron optimizaciones de tiempo en la plataforma, reutilizando configuraciones activas para testear mayor cantidad de funcionalidades respetando las limitaciones de tiempo . Una vez reducido el tiempo de test suite se implementó la métrica de *code coverage* y *function coverage* con el objetivo de conocer cuales funcionalidades no se estaban testeando y agregar los tests correspondientes.

Finalmente, luego de consultar a los desarrolladores de firmware, se implementaron y/o mejoraron herramientas para debugging, permitiendo a los desarrolladores obtener mayor visibilidad y utilidad de los resultados de una test suite. Estas mejoras redujeron el tiempo necesario para corregir bugs y defectos en el firmware.

1. Conceptos previos

En el presente capítulo se desarrollan los conceptos requeridos para comprender los fundamentos sobre las limitaciones existentes, mejoras y funcionalidades agregadas.

1.1 Siglas y abreviaturas

- **AN**: Auto Negotiation.
- **LT**: Link Training.
- **LP**: Link Partner.
- **DUT**: Device Under Test.
- **HCD**: Highest Common Denominator.
- **FEC**: Forward Error Correction.
- **IP**: Intellectual property.
- **API:** Application Programming Interface.
- **FIR**: Finite Impulse Response.
- **DSP**: Digital Signal Processing.
- **PAM**: Pulse Amplitude Modulation.
- **NRZ**: Non-Return-to-Zero.
- **SNR:** Signal-to-Noise Ratio.
- **OOP**: Object Oriented Programming.
- **ISP:** Interface Segregation Principle
- **MCU:** Microcontroller Unit, microcontrolador.
- **CRC:** Cyclic Redundancy Check.

1.2 Auto Negotiation

Auto Negotiation (AN) es un procedimiento y mecanismo utilizado en Ethernet en el cual dos dispositivos (Link partners) negocian los parámetros de comunicación tales como velocidad, señalización y código de corrección de errores (FEC). En este proceso ambos Link partners se notifican sus capacidades de transmisión y eligen el común denominador más alto (HCD) que ambos soportan, junto con otros parámetros definidos en las especificaciones del protocolo según IEEE 802.3.

Este proceso se realiza utilizando el modo de comunicación más simple, 10 Gbit/s NRZ, por motivos de backwards compatibility con dispositivos que no soportan AN.

El HCD se obtiene a partir de una tabla de prioridad definida por la IEEE y estándares de Consortium, mientras que el FEC se obtiene según una lista de reglas definidas en diversas cláusulas también por la IEEE y grupos de Consortium. Cabe destacar que tanto el estándar de la IEEE y los grupos Consortium están en constante evolución y actualización y actualmente cubren desde 1 Gbit/s hasta 400 Gbit/s. Todos los dispositivos que implementen AN deben soportar los estándares tanto de la IEEE como los Consortium. El objetivo de AN es permitir a los dispositivos nuevos comunicarse con los antiguos sin inconvenientes.

Una vez realizado el proceso de AN, si los LP tienen al menos un modo común, se puede pasar al procedimiento de Link Training donde se comienza a intercambiar tráfico, o se termina la comunicación si el objetivo era conocer el HCD al cual se comunicarán los dispositivos. El procedimiento en el cual sólo interesa conocer el HCD, sin abrirse al tráfico, se conoce como “AN probe”

A continuación se lista la tabla de prioridad de resolución de AN basada en los estándares de la IEEE y el Ethernet Technology Consortium:

Priority	Technology	Capability	Note
1	400GBASE-KR8/CR8	400 Gb/s, 8 lane	Consortium mode
2	200GBASE-KR4/CR4	200 Gb/s, 4 lane	IEEE mode
3	100GBASE-CR2/KR2	100 Gb/s, 2 lane	IEEE mode
4	100GBASE-CR4	100 Gb/s, 4 lane	IEEE mode
5	100GBASE-KR4	100 Gb/s, 4 lane	IEEE mode
6	100GBASE-KP4	100 Gb/s, 4 lane	IEEE mode
7	100GBASE-CR10	100 Gb/s, 10 lane	IEEE mode
8	50GBASE-KR/CR	50 Gb/s, 1 lane	IEEE mode
9	50GBASE-R2	50 Gb/s, 2 lane	Consortium mode
10	40GBASE-CR4	40 Gb/s, 4 lane	IEEE mode
11	40GBASE-KR4	40 Gb/s, 4 lane	IEEE mode
12	25GBASE-KR/CR	25 Gb/s, 1 lane	IEEE mode
13	25GBASE-KR-S/CR-S	25 Gb/s, 1 lane	IEEE mode
14	25GBASE-R	25 Gb/s, 1 lane	Consortium mode
15	10GBASE-KR-CR	10 Gb/s, 1 lane	IEEE mode

Los casos en los que se utilizará FEC, en orden de resolución:

1. **50GBASE-KR/CR, 100GBASE-KR2/CR2, 200GBASE-KR4/CR4, 400GBASE-KR8/CR8**:
    - FEC es obligatorio.
    - El FEC utilizado es RS544.
2. **10GBASE-KR, 40GBASE-KR4, 40GBASE-CR4, 100GBASE-CR10**:
    - Ambos dispositivos son **capable** de BASE-R y por lo menos un dispositivo realiza **request** de BASE-R.
3. **25GBASE-R, 25GBASE-KR/CR, 25GBASE-KR-S/CR-S, 50GBASE-R2, 100GBASE-KP4, 100GBASE-KR4, 100GBASE-CR4**:
    - Si por lo menos un dispositivo realiza **request** de RS528, se utilizará RS528.
    - Si por lo menos un dispositivo realiza **request** de BASE-R, se utilizará BASE-R.
    - Si se realiza **request** de BASE-R y RS528, se utilizará RS528.
    - Aclaración: **25GBASE-KR-S/CR-S** no soporta RS528, por lo que un **request** de RS528 resulta en BASE-R.
4. **25GBASE-R, 50GBASE-R2** (Consortium):
    - Si por lo menos un dispositivo realiza **request** de RS528 y ambos son **capable**, se utilizará RS528.
    - Si por lo menos un dispositivo realiza **request** de BASE-R y ambos son **capable**, se utilizará BASE-R**.**
5. **100GBASE-KP4, 100GBASE-KR4, 100GBASE-CR4**:
    - FEC es obligatorio.
    - El FEC utilizado es RS528.

1.3 Link Training

Link Training (LT) es el proceso en el cual los Link Partners, luego de haber definido los parámetros de comunicación, envían tráfico y ajustan sus parámetros de transmisión, tales como ecualización, hasta que ambos dispositivos transmiten tráfico con una relación señal a ruido (SNR) que supere el umbral definido antes de comenzar el proceso.

Este proceso tiene un tiempo máximo definido según la IEEE de 500 ms para NRZ y 3000 ms en PAM que debe respetarse, de lo contrario el proceso de Link Training se reiniciará.

No es necesario haber realizado AN antes de LT, puesto que existe la posibilidad de forzar el HCD si se tiene control sobre ambos dispositivos. Tampoco es necesario que los parámetros de transmisión se ajusten durante LT ya que existe la posibilidad de evitar este paso o cargar parámetros pre-seteados definidos por la IEEE, aunque la verificación de SNR se realiza siempre.

1.4 Bundle y followers

Se denomina bundle a un grupo de canales físicos agrupados que comparten tráfico. El canal físico encargado de realizar el proceso de Auto Negotiation se denomina **AN leader** y el resto de los canales se denominan **AN followers**.

Únicamente **AN leader** de cada LP participa en el proceso de AN. Luego el resultado del HCD se aplica sobre todo el bundle (AN leader y AN followers). Existe la posibilidad que algunos AN followers se desactiven luego de negociar el HCD, esto sucede cuando el número de canales en el HCD es menor al número de canales disponible en el bundle. Para el proceso de LT cada canal ajusta sus parámetros de transmisión independientemente y se sigue una lógica de resolución AND/OR:

- **AND**: Es necesario que todos los canales hayan completado el proceso de LT exitosamente para que el proceso de LT del bundle sea exitoso.
- **OR:** Es suficiente con que un canal falle el proceso de LT para que el proceso de LT del bundle falle y se reinicie.

1.5 Chip, intellectual property y testchip

Existen tres tipos de dispositivos distintos que se testean:

1. **Chip**: Circuito integrado de aplicación específica (ASIC) comercial que se le entrega al cliente y que se utiliza en los bancos de prueba.
2. **Intellectual Property**: La intellectual property (IP) corresponde al layout y diseño de un bloque lógico que luego debe instanciarse en un ASIC. En el

caso de la IP, no se comercializa la implementación física sino su diseño, cada cliente decide cómo implementarlo.

3. **Testchip**: ASIC no comercial para uso interno de la empresa que cuenta con instancias de una o varias IP. Los principales objetivos de los testchip son la validación del circuito logico y conexión de las IPs, desarrolo de firmware por parte de la empresa y desarrollo/validación para los clientes, antes que comiencen a instanciar la IP en sus ASICs. Si bien la complejidad de las IP es menor al de un Chip comercial en términos de funcionalidades, el hecho que los ASIC de los clientes y los de uso interno sean distintos puede suponer una mayor complejidad a la hora de testear y reproducir bugs y fallos encontrados en producción.

2. Plataforma de testing

La plataforma se encuentra desarrollada sobre el framework para testing en Python *pytest*.

La comunicación con los dispositivos se realiza a través de protocolos propietarios y la API utilizada por los clientes se desarrolla en el lenguaje C. Es necesario, por lo tanto, generar *wrappers* en Python tanto para el protocolo propietario como para la API. Se utiliza Python para desarrollar tests debido al bajo tiempo de desarrollo. En ocasiones se requiere agregar tests para cientos de configuraciones del dispositivo en el mismo día que se agrega la funcionalidad. Además, el hecho que Python sea *dynamically-typed, interpreted*, *multi-paradigm,* etc. lo convierte en el lenguaje de elección para desarrollar la plataforma de testing.

Se decidio utilizar este framework por las siguientes razones:

1. **Estabilidad y documentación**: pytest es un proyecto maduro, estable, con extensa documentación y alto nivel de adopción. Esto significa que la posibilidad de bugs en el framework es muy baja y es fácil de aprender e implementar gracias a su documentación y comunidad.
2. **Soporta diferentes versiones de Python**: Permite utilizar la misma versión de pytest para proyectos que se desarrollaron en Python 2 y proyectos nuevos que se desarrollan en Python 3.

3. **Plugins externos**: Debido al alto nivel de adopción y su comunidad, se encuentran desarrollados varios plugins útiles, de los cuales varios se utilizan en la plataforma.
4. **Fácil parametrización**: Debido a la gran cantidad de configuraciones válidas que deben verificarse para cada dispositivo, uno de los requisitos más importantes es la parametrización de tests. El framework pytest soporta múltiples formas de parametrización y generación de combinaciones. Es posible parametrizar tests directamente, indirectamente, agrupar parametrizaciones, etc. lo cual reduce el tiempo de desarrollo de tests y permite verificar todas las configuraciones válidas para una determinada funcionalidad.
5. **Fixtures**: Los fixtures son funciones que se ejecutan antes o después de un test. Permiten mantener vivas instancias de objetos durante toda la test suite, reutilizar código, etc. lo cual es imprescindible en la plataforma, ya que las instancias de los dispositivos deben existir y ser las mismas durante toda la test suite, no se puede perder la comunicación, etc.

2.1 Fixtures

Un *fixture* es una función que se ejecuta antes o después de un test. Las ventajas que tienen por sobre la funcionalidad *setup/teardown* de otros frameworks son las siguientes:

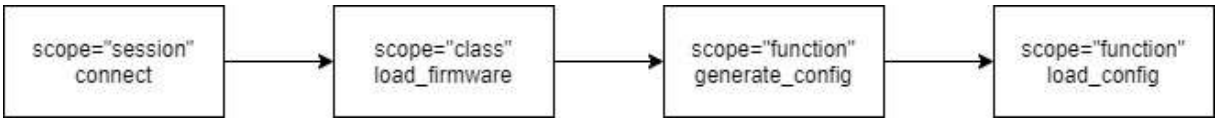
1. Poseen nombres explícitos y se activan declarándose como argumento de un test.
2. Son modulares, esto significa que un fixture puede ser llamado por otro fixture.
3. Son escalables y permiten ser parametrizados, esto significa que un test que utiliza un fixture parametrizado automáticamente se parametriza para ejecutarse con diferentes configuraciones.
4. La lógica en la que se realiza el teardown es simple y permite realizar *cleanup* de forma segura. De esta forma el equipo de QA puede enfocarse en el cuerpo del test, sabiendo que el setup y teardown son manejados de forma correcta.

5. Permiten especificar diferentes *scopes*. Esto significa que un fixture con *scope* “function” se ejecuta para cada test, un *scope* “*session”* se ejecuta una única vez, etc. lo que reduce el tiempo de la test suite. Cabe aclarar que el valor de retorno de un fixture es persistente, utiliza la keyword de Python “*yield*” en lugar de un *return*, lo que significa que un fixture de *scope “session”* se ejecuta una única vez, guarda su valor de retorno, y cada vez que un test u otro fixture lo llame simplemente vuelve a devolver ese valor por referencia.

La secuencia de *setup* utilizada en la mayoría de los tests consiste en:

1. Conectarse a todos los dispositivos. Esto implica activar el protocolo propietario para comunicarse con las placas, crear las instancias de los dispositivos a utilizar, procesos, etc. que se requieren durante toda la test suite.
2. Cargar el firmware en los dispositivos que sea necesario.
3. Generar la configuración de cada dispositivo a partir de los valores de parametrización.
4. Cargar las configuraciones en los dispositivos y ejecutar el test.

Por lo tanto, una forma conveniente de crear fixtures para la secuencia sería la siguiente:



Que se traduce en el siguiente código:

```python
@pytest.fixture(scope="session")
def connect(*args, **kwargs):
    # Conectarse a la placa, crear instancias de dispositivos, etc.
    # ...
    yield devices

@pytest.fixture(scope="class")
def load_firmware(connect, *args, **kwargs):
    devices = connect
    # Cargar el firmware en todos los dispositivos, hacer teardown, etc.
    # ...
    yield devices

@pytest.fixture(scope="function")
def generate_config(load_firmware, parameters, *args, **kwargs):
    devices = load_firmware
    # Generar toda la configuracion para cada dispositivo
    # ...
    yield devices, configs

@pytest.fixture(scope="function")
def load_config(generate_config, *args, **kwargs):
    devices, configs = generate_config
    # Cargar las configuraciones
    # ...
    yield devices
```

Luego, cada test ejecuta los pasos necesarios. Por ejemplo, el TestAnIt carga la configuración de AN-LT dentro del fixture ya que la secuencia de carga de configuración es siempre la misma, el TestRegisterAccess únicamente requiere conectarse a la placa, ya que verifica la lectura/escritura de registros sin utilizar el firmware, etc. por lo que la forma de especificarlo en código sería la siguiente

```python
class TestAnlt(object):
    def test_anlt(self, load_config, *args, **kwargs):
        devices = load_config
        # 1- Obtener el expected status y HCD segun la configuracion
        # 2- Verificar que ambos LP negocien a expected status con HCD
        # 3- Verificar estabilidad, SNR, etc.


class TestRegisterAccess(object):
    def test_register_access(self, connect, *args, **kwargs):
        devices = connect
        # 1- Escribir a multiples registros
        # 2- Verificar que el valor escrito y leido coincidan
```

2.2 Dispositivo

Todos los dispositivos concretos (que se pueden instanciar) son subclases de un dispositivo base llamado *BaseDevice* debido a que comparten funcionalidades en común, independientemente de la aplicación específica de cada dispositivo, tales como cargar el firmware, reiniciar el MCU, cargar una configuración, etc. Sin embargo, las primeras implementaciones de dispositivos concretos contenían una gran cantidad de código repetido e inconsistencias entre ellos. Estas limitaciones e ineficiencias imposibilitaban testear placas con ciertas combinaciones de dispositivos sin tener que realizar adaptaciones, demorando el tiempo de *bring up* de cada dispositivo o funcionalidad. Esto se debía principalmente al poco tiempo disponible para realizar la primera implementación de la plataforma, el cual no fue suficiente para analizar y desarrollar una arquitectura útil, a la falta de experiencia en general del equipo de QA en OOP y a la falta de conocimiento en general tanto de QA como de los desarrolladores en los proyectos futuros que debian ser soportados.

Dentro de la refactorización de la plataforma general y la utilización de *submodules*, se implementaron diferentes técnicas de OOP y patrones de diseño de software sobre los objetos que conforman a un dispositivo concreto debido a que se encontraron las siguientes limitaciones o ineficiencias:

1. Ciertos métodos que, en un principio se esperaba que fueran distintos entre dispositivos, tenían una implementación concreta igual o lo suficientemente

parecida, lo que permitía moverlos al *BaseDevice*, eliminandolas de los dispositivos concretos y reduciendo la **duplicación de código**.

2. Existen dispositivos que se desarrollan con el fin de ser versiones de bajo costo, bajo consumo o simplemente *next gen* con nuevas funcionalidades de dispositivos existentes que ya se encuentran en producción. Con el objetivo que la transición de los clientes que actualizan su dispositivo sea lo más "suave" posible, estos dispositivos nuevos comparten la mayoría de la API con la generación anterior. Idealmente un dispositivo *DeviceGen4* debería tener como *parent class* a *DeviceGen3* e implementar únicamente las funcionalidades nuevas que no sean compatibles. Esto no era posible debido a la implementación de la secuencia de inicializacion de los dispositivos concretos, lo que obligaba a duplicar código y mantener/actualizar varios dispositivos en lugar de uno.
3. En el caso de métodos que contenían una secuencia determinada de llamadas a otros métodos, estos no se encontraban implementados en el *BaseDevice*, por lo que cada dispositivo tenía su propia secuencia, por ejemplo, *teardown* del dispositivo, *constructor methods*, etc.. En muchos casos estas secuencias no eran compatibles o consistentes entre dispositivos, lo que imposibilitaba realizar abstracciones a nivel de plataforma de los dispositivos que se utilizaban.
4. El *BaseDevice* obligaba a implementar toda la funcionalidad posible de los dispositivos, por lo que un dispositivo concreto que no tenga soporte para ANLT igual debía implementar los métodos (vacíos) relacionados a la funcionalidad.

2.2.1 Composition y factory

Con el fin de generalizar parte de los métodos *constructors*, facilitar *inheritance* y que cada dispositivo deba implementar únicamente las interfaces que necesita, se separó parte de la funcionalidad del dispositivo en objetos, los cuales se instancian a través de otro objeto llamado *Factory*. Este patrón se conoce como *Factory Design Pattern*, la clase *Factory* recibe información del dispositivo y crea los objetos correspondientes según corresponda.

El dispositivo concreto se encarga de instanciar la clase *Factory* y crear los objetos cuando sea necesario. Cabe aclarar que si bien el dispositivo se separó en bloques, estos corresponden a bloques **funcionales** y no a bloques **físicos**. Por ejemplo, existen objetos que configuran el tráfico de PRBS, monitores de SNR, funcionalidad de ANLT, etc. en lugar de objetos PLL, Analog Front-End y otros bloques lógicos.

De esta forma, cada dispositivo implementa los bloques funcionales que posee, y solo está obligado a implementar los métodos comunes declarados en el *BaseDevice* que son necesarios para utilizar la plataforma.

```python
class DeviceFactory(object):
    def __init__(self, device_info):
        self.device_info = device_info

    def create_snr(self):
        # Logica que asigna el bloque de SNR
        # 1- Obtener el DeviceSnr correspondiente
        # 2- Instanciarlo con el device_info y devolverlo
        return DeviceSnr(self.device_info)

    def create_prbs(self):
        # Logica que asigna el bloque de PRBS, igual a SNR
        return DevicePrbs(self.device_info)

    def create_anlt(self):
        # Logica que asigna el bloque de ANLT, los dispositivos
        # que no tienen la funcionalidad no llaman a este metodo
        return DeviceAnlt(self.device_info)

    # Mas metodos ...
```

2.2.2 Template para secuencias

Para estandarizar las secuencias determinadas de los dispositivos, permitir realizar abstracciones en el código de la plataforma respecto del dispositivo y posibilitar la implementación de *inheritance* para dispositivos *next gen*, se decidió utilizar el *Template Design Pattern*. Este patrón de diseño consiste en definir el “esqueleto” o secuencia de un algoritmo en el *BaseDevice* y que los dispositivos concretos implementen los pasos de la secuencia, **sin modificar el orden de la secuencia**.

Por ejemplo, el método *startup* se utiliza en tests que requieren un reinicio total de los dispositivos y consiste en los siguientes pasos:

1. Reiniciar la fuente de alimentación de la placa de pruebas.
2. Reiniciar los dispositivos individualmente.
3. Limpiar los estados internos del dispositivo, utilizados en los tests.
4. Cargar el firmware.
5. Poner a los dispositivos en un estado conocido.

Esta secuencia se traduce en el siguiente código, donde se observa dos implementaciones distintas del método *state_init*, sin modificar la secuencia.

```
class BaseDevice(object):
    # Otros metodos
    # ...
    def startup(self, *args, **kwargs):
        self.board_restart()
        self.soft_reset()
        self.internal_state_cleanup()
        self.firmare_load()
        self.state_init()

    def state_init(self):
        raise NotImplementedError()

class ConcreteDevice(BaseDevice):

    def state_init(self):
        self.api.set_state(self.api.FW_STATE_APPLICATION)


class OtherConcreteDevice(BaseDevice):

    def state_init(self):
        self.api.init()
```

A su vez, el *constructor* (métodos *__new__* e *__init__* en Python), tanto de los dispositivos como de los bloques, se modificó para también utilizar el *Template Pattern*, separando la secuencia en distintos pasos. Esto permite realizar la implementación de subclases (*inheritance).*

2.2.3 Interfaces

Con el objetivo de eliminar incompatibilidades y permitir abstraer a la plataforma de las implementaciones concretas de los dispositivos, se optó por implementar los objetos que componen a la funcionalidad de un dispositivo a través de interfaces.

Debido a que Python es un lenguaje weakly-typed, es decir, el tipo de dato de variables se verifica en runtime, las funciones y métodos pueden tomar cualquier valor. No es posible forzar e implementar el concepto de software interface en Python, sin embargo es posible documentar las interfaces para que el usuario sepa cómo la plataforma espera interactuar con las mismas.

Es común que en la plataforma existan métodos que tomen argumentos que puedan tener múltiples tipos de datos. Por ejemplo, un argumento “channels” puede ser del tipo *list*, *tuple* o *set*, si bien todos estos tipos de datos comparten funcionalidad y propiedades, no son equivalentes, por lo que el usuario debía revisar la plataforma y asegurarse de no realizar operaciones que no sean válidas en el tipo de dato que recibe el método.

Al crear interfaces con sus respectivas docstrings, se elimina cualquier ambigüedad en argumentos o valores de retorno, como se observa en el código a continuación

```python
class SnrInterface(object):
    # ...
    def read(self, intf, channels):
        """
        Reads the SNR for a given physical interface and returns the values
        in mdB as a dict of {channel: snr}

        :param intf: the interface
        :type intf: int
        :param channels: the list of channels
        :type channels: list
        :return: dict of SNR values with channel as keys
        :rtype: dict
        """
        raise NotImplementedError()


class DeviceSnr(SnrInterface):
    # ...
    def read(self, intf, channels):
        # Como sabemos que es una lista, podemos aplicar sort
        channels.sort()
        return {ch: self.device.api.snr_get(intf, ch) for ch in channels}
```

2.3 Queries y Stressors

Debido a que la verificación del funcionamiento de un dispositivo y las formas de generar perturbaciones en el mismo suelen ser comunes para todos los proyectos, existen clases llamadas *Queries* y *Stressors*.

Una *Query* es una clase que obtiene información sobre el estado de un dispositivo y genera los mensajes de error o fallos correspondientes. Por ejemplo la clase *SnrQuery* lee los valores de SNR del dispositivo y genera mensajes de error si cualquier valor se encuentra por fuera del rango esperado. La clase AnltQuery verifica que ambos dispositivos realicen el proceso de AN-LT de forma esperada, obteniendo el HCD, FEC, status final, etc.

Un *Stressor* es una clase que genera una perturbación sobre un dispositivo, o desde un dispositivo hacia otro. Únicamente se encarga de verificar que se haya podido generar la perturbación en el tráfico sin realizar verificaciones, puesto que estas le corresponden a las *Queries*. Por ejemplo la clase *ReconfigStress* vuelve a cargar una o varias configuraciones repetidas veces y verifica que los dispositivos se hayan

configurado correctamente, sin realizar verificaciones de SNR, BER, etc. La clase *SquelchStress* apaga uno o varios canales, verifica que se haya perdido la conexión en el TX y RX correspondientes, los vuelve a encender y finalmente verifica que la conexión se haya establecido nuevamente.

Las *Queries* y *Stressors* se programan en base a las *Interfaces* definidas anteriormente, por lo tanto, como los dispositivos implementan la misma interface, las *Queries* y *Stressors* son independientes a la implementación del dispositivo. La abstracción respecto de la implementación concreta permite utilizar las clases para testear múltiples dispositivos, reduciendo el tiempo de bring up.

A modo de ejemplo se muestra el código de la clase *SnrQuery*:

```
class SnrQuery(BaseQuery):
    def __init__(self, device, intf, channels):
        self.device = device
        self.intf = intf
        self.channels = channels

    def query(self):
        min_snr = self.device.min_snr
        snr_values = self.device.snr.read(self.intf, self.channels)
        for channel, value in snr_values.items():
            self.check_for_error(
                condition= value<min_snr,
                msg="SNR ({read_snr}) lower than minimum ({min_snr}) on {device} {intf} {ch}".format(
                    read_snr=value, min_snr=min_snr, device=device.name, intf=intf, ch=channel
                )
            )
```

Existe también, una clase cuyo objetivo es simplificar la combinación de *Queries* y *Stressors*, puesto que es común el tener que verificar información del tráfico luego de generar una perturbación. Esta clase se implementa como un *iterator*, puesto que recibe como parámetros la lista de *Queries*, *Stressors*, el orden y cantidad de iteraciones. También permite iterar de a un paso o ejecutar todas las iteraciones juntas, acumulando la lista de mensajes de errores.

3. Optimizaciones de tiempo de ejecución

La principal limitación que presentaba la plataforma inicial consistía en el tiempo que tomaba ejecutar una test suite completa. En un principio se estaba testeando la funcionalidad básica de AN-LT con aproximadamente 600 combinaciones en 6 horas.

Conforme se agregaran más funcionalidades a testear se haría imposible cubrirlas en una test suite. Esto se debe a las siguientes razones:

1. Con el objetivo de soportar integración continua, se testea automáticamente una versión nueva del firmware cada noche. Se encuentra programada para comenzar a la 1:00 de Argentina luego que finaliza el día laboral de los desarrolladores, a 4 horas de diferencia en la zona horaria.
2. Automáticamente al finalizar el build de la versión de firmware se comienza a correr la test suite diaria.
3. El equipo de QA comienza su jornada laboral a las 8:00. El equipo analiza los resultados de la test suite y utiliza los bancos de prueba para detectar fallos y reportarlos a los desarrolladores.

Por estos motivos, se establece un límite de test suite de 7 horas.

En la siguiente tabla se presentan los resultados luego de haber aplicado las mejoras a desarrollar en el presente capítulo

Estado	Cantidad de tests	Duración
Antes	~600	6 horas
Despues	~1700	7 horas

3.1 Testeo incremental

La secuencia de testeo de múltiples funcionalidades sobre la misma configuración era la siguiente:

1. Secuencia de inicialización y proceso de AN-LT.
   1.1. Carga de firmware en todos los dispositivos, cleanup y teardown general.
   1.2. Generación de configuraciones específicas.
   1.3. Carga de configuraciones en los dispositivos.
   1.4. Verificación del proceso de AN-LT.
2. Testeo de funcionalidad **A**.
3. Repetir 1.
4. Testeo de funcionalidad **B**.
5. Repetir 1

La secuencia de inicialización y proceso de AN-LT se repetía para cada funcionalidad que se agregaba, por lo que no era escalable y se perdía tiempo, ya que el paso 1. demora aproximadamente 20s. Se decidió modificar el método evitando repetir el paso 1, reordenando y optimizando la secuencia por la siguiente:

1. Secuencia de inicialización y proceso de AN-LT.
2. Testeo de funcionalidad **A**.
3. Testeo de funcionalidad **B**.
4. Testeo de funcionalidad **C**.

Este nuevo método denominado *testeo incremental* permitió reducir el tiempo de test suite inicial de 6 a 4 horas aproximadamente. Si bien el tiempo de ejecución se reduce considerablemente, posee las siguientes desventajas:

1. La secuencia se trunca cuando falla cualquier paso, y todas las funcionalidades posteriores que no fueron testeadas se consideran como fallos.
2. No es posible ejecutar una única funcionalidad. Todas las funcionalidades deben ejecutarse juntas.

3.1.1 Implementación

En primer lugar, fue necesario crear un nuevo fixture que permita cargar una configuración con *scope "class"* ya que el fixture existente se ejecuta para cada función y no es posible modificar el scope de un fixture en *runtime*. Para evitar duplicar código, se movió toda la funcionalidad del fixture a una función y los fixtures pasan a ser *wrappers* de la función.

```python
@pytest.fixture(scope="function")
def load_config(generate_config, *args, **kwargs):
    yield load_config_func(generate_config, *args, **kwargs)


@pytest.fixture(scope="class")
def load_config_class(generate_config, *args, **kwargs):
    yield load_config_func(generate_config, *args, **kwargs)

# Esta funcion no es un fixture
def load_config_func(generate_config, *args, **kwargs):
    return load_config_func(generate_config, *args, **kwargs)
```

Adicionalmente, se modificaron los argumentos de los tests para que ejecuten el nuevo fixture *load_config_class*, y se agregó una verificación extra que marca el test como fallo si el paso anterior falló.

```python
class TestAnlt(object):

    def test_anlt_A(self, load_config_class, *args, **kwargs):
        # ...
        assert ...

    def test_anlt_B(self, load_config_class, *args, **kwargs):
        assert kwargs['runtime_stats']['previous_test']['status'] == 'PASSED'
        # ...

    def test_anlt_C(self, load_config_class, *args, **kwargs):
        assert kwargs['runtime_stats']['previous_test']['status'] == 'PASSED'
        # ...
```

3.2 No recarga del firmware

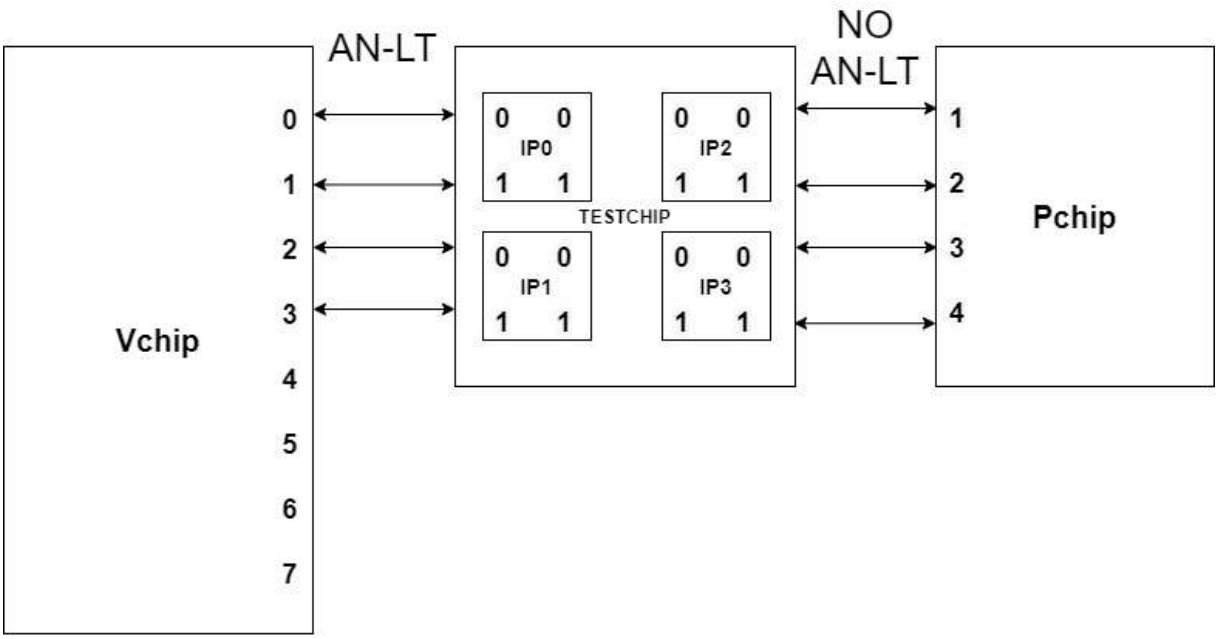
Tras analizar los cuellos de botella de tiempo, se identificó que la carga del firmware de los dispositivos del banco de pruebas es uno de los factores que más aporta a la duración de cada test. Limitar la carga del firmware era una funcionalidad fácil de implementar en la plataforma. Las siguientes implementaciones redujeron significativamente el tiempo de test suite:

1. Inicialmente el firmware se cargaba para todos los dispositivos del banco de pruebas, incluso aquellos que no se utilizaban en el proceso de AN-LT. Se

modificó la plataforma para que únicamente cargue el firmware en los dispositivos que se utilizan en cada test.

2. Aplicando el mismo método del punto 1. a nivel de testchip, se modificó la plataforma para que únicamente cargue el firmware en las IPs que se utilizan en cada test.

Tómese el siguiente banco de pruebas como ejemplo:



La funcionalidad de AN-LT se testea entre *Vchip* y las IP {0,1} de TESTCHIP, mientras que la funcionalidad de no AN-LT se testea entre Pchip y las IP {2,3} de TESTCHIP respectivamente. En la secuencia de inicialización, los tests de AN-LT cargan el firmware en *Vchip* y las IP {0,1} de TESTCHIP únicamente, para un total de 3 cargas de firmware, comparado con 6 cargas de firmware en la implementación original.

La carga de cada firmware puede tardar entre 1 y 4 segundos según el tamaño del mismo, por lo que con 1700 tests, esta optimización redujo el tiempo de test suite en aproximadamente 1 hora.

3.2.1 Implementación

El framework *pytest* actualiza una *environment variable* PYTEST_CURRENT_TEST que contiene el nombre del test. A su vez, los nombres de los tests son lo suficientemente descriptivos ya que indican unívocamente la combinación de AN-LT

que se está ejecutando sobre el DUT. Por lo que, a partir del nombre del test se puede obtener que IPs se van a utilizar y cargar únicamente el firmware para esas IPs. Con respecto a las SRC, se puede obtener cuales se utilizan según las IPs del DUT a partir de los datapaths.

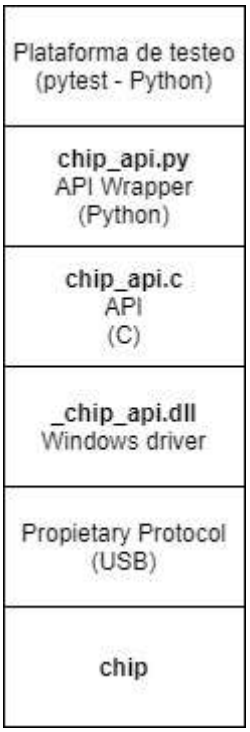
4. Code coverage y function coverage

A continuación se detalla el estado de **code coverage** de la API para los dispositivos al comienzo y el estado actual

Proyecto	Inicio de testeo	Code coverage inicial	Code coverage actual
SIP	Noviembre 2019	32%	70%
CIP	Diciembre 2019	28%	65%

4.1 Software stack

Cabe aclarar que si bien lo que se testea es el firmware de los dispositivos, la configuración, lectura de registros y datos, etc. se realiza a través de una Application Programming Interface (API) tanto por los clientes como por el equipo de QA. En la siguiente figura se presenta el software stack desde la plataforma en Python hasta la comunicación con el dispositivo.



Para comunicar al dispositivo (chip) desde una PC, existe desarrollado un protocolo propietario sobre USB, el cual únicamente realiza lectura y escritura de registros.

La API se encuentra escrita en C, ésta generalmente se compila con las funciones de lectura y escritura de registros definida. Sin embargo, cuando se desea generar el wrapper de Python para poder comunicarse con la plataforma de testeo, se modifican las funciones para que reciban *callbacks*. De esta forma, se puede decidir que función de lectura y escritura de registros utilizar desde la plataforma, ya sean funciones implementadas en Python o compiladas en otra librería.

4.2 Code coverage

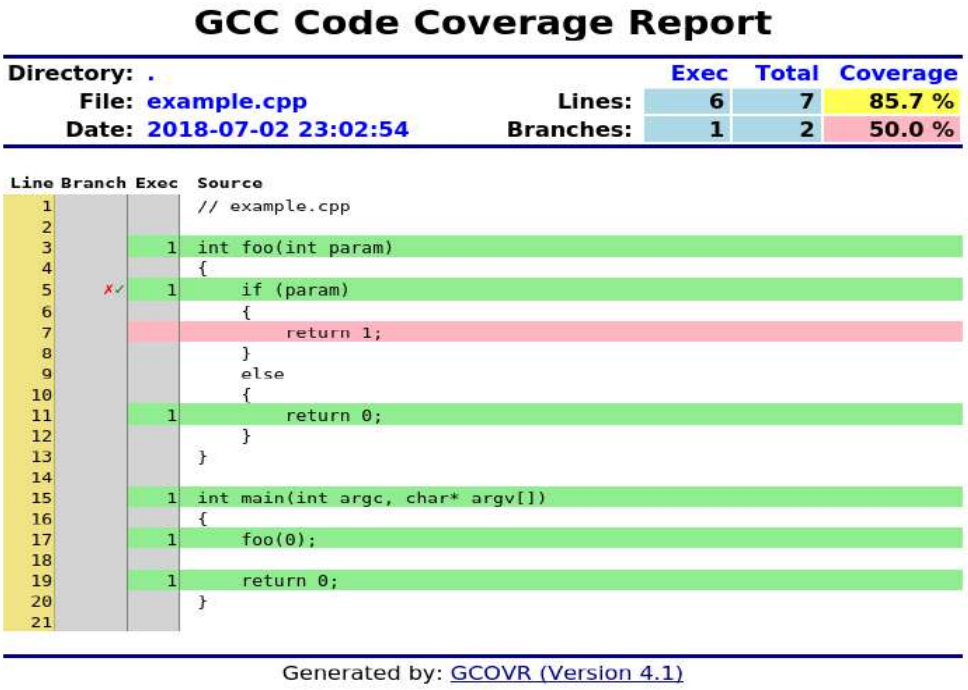
La métrica que se decidió utilizar es *code coverage*, que indica qué líneas de un programa se ejecutan luego de una test suite. La herramienta elegida fue *GCOV* puesto que viene incluida con el compilador *GCC*, ya utilizado para compilar la API en C.

Según el *software stack* introducido en la sección 3.1, se puede observar que el *code coverage* se aplicará sobre la API en C. Si bien la plataforma de testeo utiliza el wrapper en Python, este es un archivo generado automáticamente simplemente se comunica con la librería compilada.

Para poder obtener la métrica de *code coverage* solo es necesario compilar la API utilizando *GCC* con las opciones correspondientes, generar el wrapper para *Python* y correr una test suite.

Cuando se compila un archivo con *GCOV* se genera un archivo *.gcno* y a medida que se ejecutan las líneas de ese archivo se actualiza un archivo *.gcda*. Estos archivos no son legibles por humanos, por lo que se debe utilizar una herramienta extra para generar reportes que sean de utilidad y permitan extender el *code coverage*. En un principio se intentó utilizar la herramienta *lcov*, sin embargo al ser incompatible con Windows se optó por el módulo *gcovr* de *Python* debido a su fácil instalación, uso y funcionalidades similares a *lcov*. El propósito de la herramienta es no solo convertir los archivos en formato legible para humanos, sino combinar múltiples archivos .gcda.

Inicialmente se generó un archivo HTML como el de la siguiente imagen:



GCC Code Coverage Report

		Exec	Total	Coverage
Directory: .				
File: example.cpp	Lines:	6	7	85.7 %
Date: 2018-07-02 23:02:54	Branches:	1	2	50.0 %

Line	Branch	Exec	Source
1			// example.cpp
2			
3		1	int foo(int param)
4			{
5	✗✓	1	if (param)
6			{
7			return 1;
8			}
9			else
10			{
11		1	return 0;
12			}
13			}
14			
15		1	int main(int argc, char* argv[])
16			{
17		1	foo(0);
18			
19		1	return 0;
20			}
21			

Generated by: GCOVR (Version 4.1)

Sin embargo, el archivo *chip_api.c* contiene cerca de 80000 líneas dependiendo del dispositivo, por lo que no resultaba útil generar un HTML ya que debía leerse una gran cantidad de líneas de código manualmente.

4.3 Function coverage

Debido a la longitud del archivo HTML resultante, fue necesario procesarlo y generar otro archivo que permita identificar rápidamente la falta de coverage en la funcionalidad/APIs y agregarlas a la plataforma de testeo. Se optó por generar el siguiente formato de tabla:

Nombre de la función	Líneas ejecutadas/total	Porcentaje ejecutadas/total	Presente en el manual de usuario?

Dado que *gcovr* no cuenta con la posibilidad de analizar funciones individualmente fue necesario generar primero el HTML y luego utilizar técnicas de procesamiento de texto tanto en el HTML generado como en el manual de usuario de la API del dispositivo para generar el formato de tabla deseado. Las columnas fueron elegidas por los siguientes motivos:

1. **Nombre de la función**: Permite saber qué función no se está llamando, referenciarla en el reporte HTML, analizar qué líneas no se ejecutan y agregar tests para cubrir la funcionalidad sin testear.
2. **Líneas ejecutadas/total**: Permite dar prioridad a funciones con mayor cantidad de líneas. Enfocando el testeo a las APIs con mayor funcionalidad y cantidad de líneas permite incrementar el *code coverage* más rápido y cubrir mayor funcionalidad en menor tiempo que enfocando el testeo a APIs con menor cantidad de líneas.
3. **Porcentaje ejecutadas/total**: Permite dar prioridad a funciones con menor porcentaje de líneas ejecutadas, por la misma razón que el punto 2.
4. **Presente en el manual de usuario**: Si una función no se encuentra en el manual de usuario significa que es de uso interno por otras funciones. El equipo de QA no debería enfocarse en agregar tests para estas funciones.

En la siguiente imagen se puede observar parte de un archivo generado automáticamente en una test suite que midió el *code coverage*

```
111 - prefix_pwrup_bias_qpmp_ldo_pwrup :                    64/67 95.52%
112 - prefix_is_fw_running_ok :                             39/41 95.12%
113 - prefix_anlt_recenter_tx_fifo :                        19/20 95.00%
114 - prefix_anlt_open_tx :                                 19/20 95.00%
115 - prefix_pwrup_eru_pwrup :                              30/32 93.75% - NOT IN USER GUIDE
116 - prefix_prbs_chk_status :                             94/101 93.07%
117 - prefix_channel_wait_for_ack_clear :                   26/28 92.86% - NOT IN USER GUIDE
118 - prefix_init :                                         26/28 92.86%
119 - prefix_rx_dsp_eyemon_query_dump :                     12/13 92.31%
120 - prefix_channel_wait_for_ack_set :                     23/25 92.00% - NOT IN USER GUIDE
121 - prefix_anlt_wait_for_ack_clear :                      22/24 91.67% - NOT IN USER GUIDE
122 - prefix_rx_dsp_eyemon_query :                          32/35 91.43%
123 - prefix_mse_binary_search :                            10/11 90.91% - NOT IN USER GUIDE
124 - prefix_anlt_wait_for_ack_set :                        19/21 90.48% - NOT IN USER GUIDE
125 - prefix_pwrup_bias_pwrup :                             19/21 90.48% - NOT IN USER GUIDE
126 - prefix_mcu_msg_push_message :                         36/40 90.00% - NOT IN USER GUIDE
127 - prefix_snr_fixp_to_mse :                               9/10 90.00%
128 - prefix_prbs_chk_is_enabled :                          16/18 88.89%
129 - prefix_prbs_chk_ber :                                  8/9 88.89%
130 - prefix_rx_dsp_dfe_coefficients_print :                 8/9 88.89%
131 - prefix_mcu_download_firmware_from_file :              82/93 88.17%
132 - prefix_init_tx :                                      28/32 87.50%
133 - prefix_init_rx :                                      28/32 87.50%
134 - prefix_is_link_ready :                                13/15 86.67%
135 - prefix_prbs_chk_config :                              75/87 86.21%
136 - prefix_mcu_pif_read :                                 31/36 86.11%
137 - prefix_mcu_direct_download_image_impl :               49/57 85.96% - NOT IN USER GUIDE
138 - prefix_prbs_chk_status_print :                        24/28 85.71% - NOT IN USER GUIDE
139 - prefix_pwrup_rules_dump :                             18/21 85.71%
140 - prefix_rx_dsp_snr_format :                             6/7 85.71%
141 - prefix_mcu_status_query :                             28/33 84.85%
142 - prefix_rx_dsp_snr_format_fixp :                       11/13 84.62%
```

5. Refactorización de la plataforma de AN-LT

En este capítulo se desarrolla la refactorización en general de la plataforma de AN-LT que se realizó luego de haber realizado una primera iteración optimización de tiempo. La refactorización se realizó en el marco de la refactorización general de la plataforma de testeo AN-LT y no AN-LT, que involucra todos los dispositivos.

5.1 Generalización de la plataforma de testeo

La plataforma inicial de AN-LT se había diseñado para un layout específico de banco de pruebas, con dos dispositivos con conexiones fijas. Esto no era escalable ya que agregar un nuevo layout o dispositivo implicaba repetir la configuración y tests manualmente. Por estas razones el tiempo de *bring up* aumentaba considerablemente.

Se implementaron distintas funcionalidades y estrategias con el objetivo de generalizar la plataforma de testeo. Luego de implementar estas funcionalidades, la

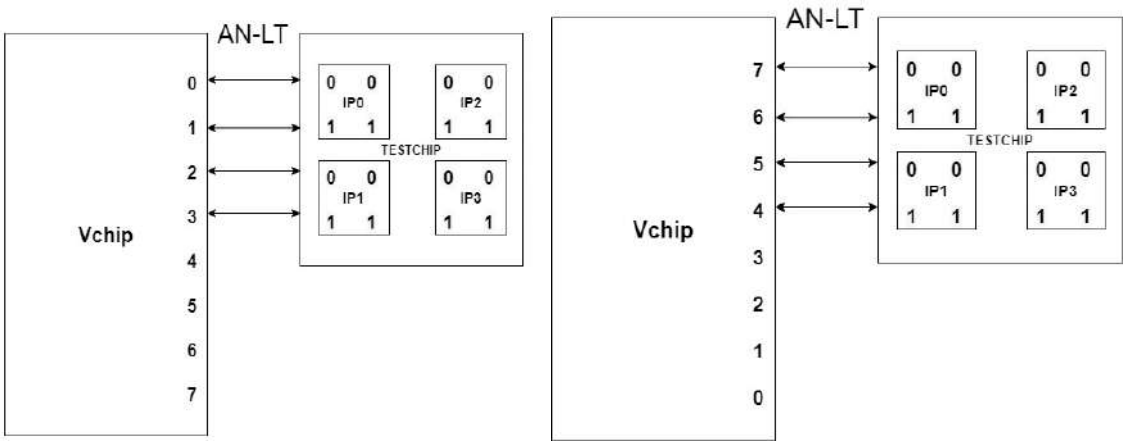
plataforma de testeo se independizó de la conexión de los bancos de prueba y para dar soporte a un nuevo dispositivo únicamente se requiere implementar los métodos especificados en la interfaz de AN-LT de la plataforma. De esta forma, el tiempo de *bring up* de un banco de prueba se redujo de días a unas pocas horas o incluso minutos, lo que permite comenzar a testear un dispositivo o banco de prueba con mayor rapidez.

5.1.1 Datapath class

Dentro de las mejoras y funcionalidades introducidas en la plataforma, tanto AN-LT como no AN-LT se encuentran los **datapath**: clases que contienen información sobre las conexiones del banco de pruebas y diversos métodos para obtener esta información.

Los **datapath** permiten generalizar la etapa de generación configuraciones plataforma de testeo de las conexiones particulares de cada banco de prueba, puesto que otorgan la posibilidad de inferir la configuración física del LP a partir de la configuración del DUT.

Tómese como ejemplo los siguientes banco de pruebas:



Tomando el TestChip como DUT al que se le aplica la configuración principal, configurando un bundle simple de un par $TX = IP0,\ lane0$, $RX = IP1\ lane1$:

- El **Vchip** del primer banco de pruebas genera su configuración sobre el par TX3 RX0.
- El **Vchip** del segundo banco de pruebas genera su configuración sobre el par TX4 RX7.

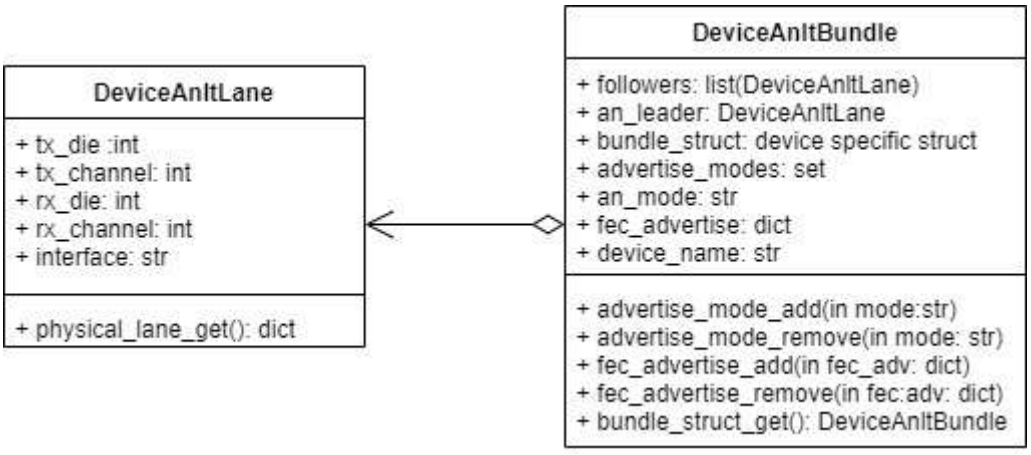
Ya no es necesario contemplar cada banco de pruebas en una situación caso por caso, acelerando el proceso de *bring up* de los bancos de prueba.

5.1.2 Bundle y lanes universales

Se presentan las siguientes incompatibilidades en los dispositivos respecto a la configuraciòn de bundles en AN-LT:

1. No todos los dispositivos cuentan con una estructura de bundle en su API. En algunos dispositivos el bundle de AN-LT se declara implícitamente como un conjunto de canales TX/RX activos y un canal RX de referencia que corresponde al an leader, mientras que otros dispositivos cuentan con estructuras específicas tanto para el bundle como para cada lane que lo compone.
2. Cuando se realiza la configuración de AN-LT en un TestChip, los lanes se referencian como IP (die) + IP lane (tx/rx). Por ejemplo *die0_tx0_die1_rx1* corresponde a un lane con su TX en el canal 0 de la IP 0 y su RX en el canal 1 de la IP 1, sin embargo, el dispositivo que se encuentre conectado debe recibir las conexiones físicas de los pines, que serían TX0 RX3. Esta referencia IP + IP lane solo aplica a AN-LT en TestChip, tanto Chips como configuraciones de no AN-LT en TestChip se referencian con el lane físico.

Por estos motivos, se crearon *lanes* y *bundes* de ANLT universales:



Respecto al ***DeviceAnItLane***, los dispositivos del tipo Chip pueden utilizarla directamente ya que los canales físicos y lanes de ANLT son equivalentes, se puede entender como una única IP con todos los canales, por lo que sus lanes siempre se

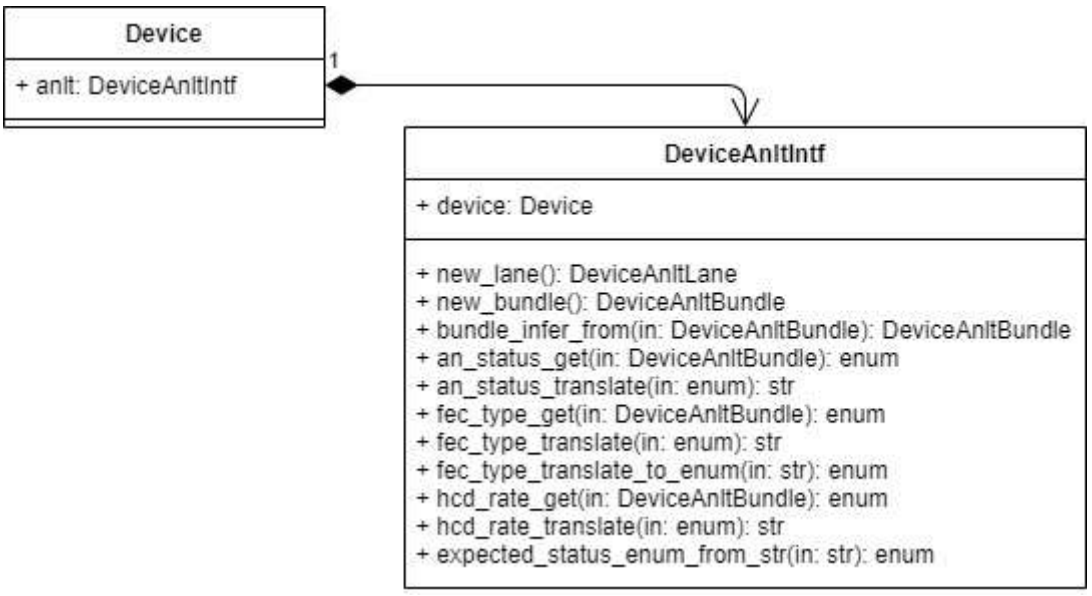
referencian como *die0_tx#_die0_rx#*. Los dispositivos TestChip deberán re-implementar el método con *physical_lane_get()* especificando su mapeo correspondiente.

Con respecto a ***DeviceAnItBundle***, únicamente los dispositivos que contengan en su API estructuras para representar un bundle deberán re-implementar el método *bundle_struct_get()*, mientras que el resto de los dispositivos puede utilizar la clase directamente, ya que el método devuelve una referencia a la misma instancia de la clase.

Se puede observar que las clases siguen el **principio de segregacion de interfaces (ISP)** ya que ningún dispositivo se encuentra obligado a implementar métodos que no utilizan. A su vez, se refactorizaron las secciones de la plataforma de testeo que accedían a estructuras específicas de bundles para que accedan a objetos del tipo ***DeviceAnItBundle***, eliminando la dependencia de implementaciones específicas.

5.3.1 Interface de AN-LT

En un principio no existía una interfaz común para métodos de AN-LT, por lo que existían métodos con distintos nombres en las clases de los dispositivos que cumplían la misma funcionalidad. Se decidió por implementar una interface común para la funcionalidad de AN-LT requerida por la plataforma de testeo como se observa en la siguiente imagen:



Como se explicó anteriormente, la secuencia de AN-LT verifica el status final, el HCD y FEC. Si bien el formato de algunos valores es distinto entre dispositivos (por ejemplo algunos dispositivos tienen status “AN.FAIL” y otros “AN_FAIL”) la plataforma de AN-LT impone un formato específico que debe respetarse a la hora de agregar nuevos dispositivos a la plataforma. De esta forma, las diferencias entre dispositivos quedan encapsuladas y la plataforma no requiere ser modificada/extendida constantemente para contemplar casos especiales.

5.2 Generación automática de combinaciones

Una combinación de AN-LT se corresponde a un testcase, la misma consiste de:

- Configuración específica: Habilitar AN, LT, setear el DSP mode, reglas especiales, etc.
- Advertised modes para cada bundle: 25GBASE-KR-S/CR-S, 100GBASE-CR4, 50GBASE_KR/CR, etc.
- FEC capable/request para cada advertised mode.
- Configuración de lane + bundles.

La notación utilizada es *NxMxMODE*, donde:

1. *N* es la cantidad de bundles.
2. *M* es la cantidad de canales por bundle.
3. *MODE* es el modo de señalización de **un canal**.

Por lo tanto, una notación *2x2x50gkr* se refiere a 2 bundles de 2 canales de 50gkr cada uno, es decir, 2 bundles de 100GBASE-KR2.

Inicialmente cada combinación se agregaba manualmente debido a que el primer dispositivo que utilizó AN-LT contenía limitaciones de hardware y debía verificarse la funcionalidad de cada caso. Este método no era escalable debido a la cantidad de combinaciones totales que se pueden generar.

Tomando como ejemplo un dispositivo de 4 canales físicos, existen 280 combinaciones únicamente de lane + bundle, si se agrega la variable de advertise modes y FEC la cantidad de combinaciones puede sobrepasar las 2000 por cada mode. Si bien esta es una lista exhaustiva de combinaciones, ejecutarlas todas requeriría más de 5 horas por cada mode.

Con el fin de poder tener la flexibilidad suficiente de poder correr cualquier combinación sin que el tiempo de la test suite exceda el permitido se realizó lo siguiente:

1. Se generan todas las combinaciones de lane+bundle para 1x1, 1x2, 2x1, 2x2, 1x4 y 4x1.
2. Cada bundle hace advertise de un solo mode y los casos de múltiples bundles hacen advertise del mismo mode.
3. No se modifican los valores de FEC.

Si bien se generan todas las combinaciones, en la test suite diaria solo se ejecutan los casos 1xN y una cantidad fija de combinaciones al azar NxM (N>1) por limitaciones de tiempo. Para combinaciones especiales (LT standalone, Presets, etc.) se genera una combinación al azar por cada modo. Las combinaciones al azar están determinadas por una semilla que se basa en la fecha de comienzo de la test suite, asegurando que cada noche se ejecute una test suite distinta. Sin embargo, es posible pasarle una semilla determinada a la plataforma cuando se desea repetir las mismas combinaciones de un día en particular.

Cabe aclarar que la resolución de AN es independiente al advertise de FEC, por ejemplo 25GBASE-KR/CR siempre tiene prioridad sobre 25GBASE-KR-S/CR-S, incluso si el advertise de 25GBASE-KR/CR se realice sin FEC y el de 25GBASE-KR-S/CR-S con FEC, por lo que es posible generar grupos de tests para AN resolution y FEC resolution por separado.

5.2.1 Generación de combinaciones de AN múltiple advertise

Existen 15 advertise modes posibles en cada dispositivo. Cada link partner puede setear el bit de advertise o no, obteniéndose un total de 2^{30}combinaciones posibles, por lo que no es posible generar todas las combinaciones posibles. Se decidió por el siguiente enfoque:

1) Se generan todas las combinaciones posibles de 25G haciendo advertise de al menos 2 modos (prioridad 12 a 14 en la tabla) ya que los bundles de 1 lane y 25Gbit NRZ son los más utilizados por los clientes y de los que más comúnmente se suele hacer múltiple advertise. Si bien la tabla de prioridad indica solo 3 modos, pueden existir múltiples modos por cada uno con igual

prioridad, por ejemplo 25GBASE-R (prioridad 14) contiene 25GBASE-KR1-CONSORTIUM y 25GBASE-CR1-CONSORTIUM.

2) Para el resto de los modos posibles, se hace advertise del modo + todos los de menor prioridad. Por ejemplo, el test de múltiple advertise de 40GBASE-KR4 realiza advertise de 10GBASE-KR/CR, 25GBASE-R, 25GBASE-KR-S/CR-S, 25GBASE-KR/CR y 40GBASE-KR4.

5.2.2 Generación de combinaciones para FEC resolution

Debido a que la resolución de AN se realiza independientemente de los valores de FEC advertise, se puede generar un test de FEC resolution por cada mode. Sin embargo existen 6 bits de FEC (advertise/capable para FC, RS528 y RS544) en cada link partner, obteniéndose un total de 2^{12} combinaciones por cada mode, por lo que no es posible generar todas las combinaciones posibles de FEC.

Se optó por implementar únicamente las combinaciones válidas para cada mode

Señalización de lane	**FEC soportado**	**Combinaciones**
10Gbit/s	BASE-R (FC)	2^4
25Gbit/s	FC, RS528*	2^8
50Gbit/s	RS544	2^4

*Si bien 25GBASE-KR-S/CR-S no soporta RS528, se contemplan los bits asignados al FEC al momento de realizar la FEC resolution.

Se genera únicamente 1 test de FEC resolution por mode, realizándose las reconfiguraciones con diferente FEC advertise, debido a que generar un test por cada FEC advertise por mode agregaría miles de tests a la plataforma, generando ruido en los resultados.

5.3 Modularización y reutilización de tests

En cuanto a los tests, debido a que gran parte de la funcionalidad de AN-LT tiene una secuencia de testeo similar, se aplicaron ventajas de OOP como inheritance,

polymorphism, etc. con el fin de reducir el código duplicado y acelerar el testeo de funcionalidades nuevas.

A grandes rasgos, la secuencia de testeo AN-LT es la siguiente:

1. Generar la configuración automática que se envía a los dispositivos.
2. Configurar los dispositivos.
3. Verificar que los dispositivos alcancen el estado esperado según los procedimientos a ejecutar en un tiempo determinado.
4. Una vez alcanzado el estado y dependiendo cual sea, se realiza una verificación de estabilidad, es decir, que el estado no cambie durante un tiempo determinado en los dispositivos.
5. Verificar que el HCD y el FEC negociado se corresponda al esperado.
6. Verificar que la SNR y la BER tengan valores aceptables.
7. Tirar abajo un canal en un dispositivo, levantarlo y verificar que los dispositivos abran la configuración nuevamente, verificando SNR y BER.

La secuencia se implementó en una clase TestAnlt base, separado parte de la funcionalidad y verificación en distintos métodos. Esto permite agregar tests para nueva funcionalidad con relativa facilidad, reutilizando gran parte del código y únicamente agregando verificación extra.

Se presentan dos ejemplos concretos:

1. **LT standalone**: El caso de LT sin realizar AN es especial, puesto que los dispositivos no tienen HCD ni FEC (lo elige el usuario) y el estado esperado es distinto a AN-LT (LT_COMPLETE en LT standalone, COMPLETE en AN-LT). Fue suficiente con extender la clase base (inheritance) para que en caso que la configuración sea LT sin AN contemple las situaciones mencionadas anteriormente. La clase encargada de testear LT standalone tiene como parent class la clase base y únicamente se encarga de generar las combinaciones que se desean ejecutar y la configuración para LT sin AN.
2. **AN-LT con preset de TX:** Como se había mencionado en el capítulo de introducción, existe la posibilidad que un dispositivo le indique a otro que setee sus parámetros de transmisión según presets estandarizados por la IEEE. En este caso, similar al anterior, la clase encargada de testear el preset de TX genera las combinaciones a ejecutar, la configuración de preset y luego

del paso 5 se verifica que el dispositivo que recibió la solicitud de preset haya seteado sus parámetros de transmisión según el estándar de la IEEE.

A continuación se muestra el ejemplo de la clase para verificar la funcionalidad de **AN-LT con preset de TX**

```python
class TestAnltPreset(TestAnlt):
    def test_anlt_A(self, load_config_class, *args, **kwargs):
        # Llamar al test de la parent class
        super(TestAnltPreset, self).test_anlt_A(load_config_class, *args, **kwargs)
        # Chequear presets
        status = self.verify_presets(load_config_class, *args, **kwargs)
        # ...

    def verify_presets(self, load_config_class, *args, **kwargs):
        # Verificar los presets, devolver status y lista de errores
        # ...
        return status, errors
```

5.4 Reutilización de código entre proyectos

Si bien la plataforma de testeo se encuentra en un único *repositorio* en un sistema de control de versiones, cada proyecto se encuentra en una *branch* independiente. Esta organización dificulta el mantenimiento y actualización de los distintos proyectos, ya que si se realiza una mejora en cualquiera de los proyectos, esta debe ser migrada manualmente al resto, lo que conlleva una pérdida de tiempo para el equipo de QA.

Se contemplaron las distintas soluciones:

1. *Branch* única: Esta solución elimina completamente la necesidad de migrar cambios y mejoras entre proyectos, sin embargo, si se requiere un cambio o mejora para un proyecto habría que esperar a testear el cambio en todos los proyectos, incrementando significativamente el tiempo de desarrollo de los proyectos.
2. *Git subtree*: Un git subtree consiste en un *repositorio* que se encuentra dentro de otro *repositorio* de git. Tanto los archivos como el *commit history* se encuentran disponibles dentro de la carpeta del *repositorio* clonado. La principal ventaja es que no requiere acceso al *repositorio* remoto y el *subtree* pasa a considerarse como parte del repositorio, por lo que el usuario no sabe que la carpeta pertenece a otro *repositorio* y cualquier cambio hecho en el

subtree se considera como un cambio hecho en el *repositorio* principal. SIn embargo, actualizar el *subtree* con nuevos cambios en el remoto es considerablemente difícil, ya que al momento de clonar el *subtree*, el *repositorio* principal lo considera como una carpeta más y no como otro *repositorio*.

3. *Git submodule*: Un submodule consiste de un pointer a un *commit* específico de un *repositorio* externo. El *repositorio* principal contiene únicamente la metadata del *submodule*, por lo que el usuario es responsable de hacer *pull* de los archivos y actualizar el *commit* al que apunta el *repositorio*. A diferencia del *subtree*, el *submodule* se trackea independientemente al *repositorio* principal, por lo que es posible subir cambios al *submodule* desde cualquier *repositorio* que lo contenga. Las principales desventajas son que se requiere acceso al *remote*, ya que el *repositorio* original solo contiene la *metadata* del *submodule*, y es más difícil de mantener ya que al considerarse como un *repositorio* independiente dentro de otro *repositorio*, el usuario es responsable del *submodule.* Las ventajas son que, al quedar fijo el *commit* del *submodule*, se puede hacer *push* de cambios para un proyecto específico sin riesgo a agregar bugs en los otros proyectos, ya que cada proyecto puede actualizar su versión del *submodule* cuando crea conveniente.

Una vez analizadas las tres alternativas, se optó por implementar *Git submodule* en los proyectos. Luego de ser migrados todos los archivos comunes, con la refactorización y modularización requerida, se adoptó el siguiente flujo de trabajo y actualización del submodule:

1. Los proyectos únicamente tienen permitido trackear al *master* del submodule.
2. Para implementar una funcionalidad, fix o agregar soporte para un proyecto en el submodule, se crea una branch de desarrollo.
3. Se testean los proyectos necesarios con las funcionalidades y cambios agregados, cualquier fix se agrega en la *branch* de desarrollo y no sobre el master.
4. Una vez testeados los proyectos, se hace un *freeze* de la branch de desarrollo, se actualizan los proyectos que requieren los cambios agregados y se realiza un *merge* de la *frozen branch* de release hacia el master con el tag correspondiente.

5. Los proyectos que no hayan testeado los cambios pueden seguir utilizando versiones anteriores del submodule, y actualizarse cuando sea posible.

6. Funcionalidades agregadas a pedido de los desarrolladores

En el presente capítulo se desarrollan las funcionalidades que fueron agregadas a la plataforma por pedido de los desarrolladores con los objetivos de acelerar el proceso de debugging, lograr sincronizar la plataforma de testeo en Python con el testeo utilizado por los desarrolladores y lograr mejorar la curva de aprendizaje para que los desarrolladores puedan utilizar la plataforma eficientemente.

6.1 Sobrescritura de configuración

La plataforma de testeo genera todas las combinaciones de AN-LT sin ninguna configuración o regla especial. Por este motivo, se agregó una funcionalidad de sobrescritura de configuración. En caso que se desee ejecutar una combinación con reglas de configuración especiales que no se encuentran en ningún test existente se agregó la posibilidad de especificar en un archivo de configuración YAML los campos que se desean sobrescribir y sus valores. En el momento de ejecutar el test con la configuración extra simplemente se especifica el nombre de la misma.

Por ejemplo, la SNR mínima para concluir exitosamente el proceso de LT se especifica en el campo “rules.lt.snr_threshold” en mdB y tiene un valor por defecto de 24.7 dB, si se quisiera ejecutar un test con el objetivo de debuggear con un valor de 29 dB el archivo YAML tendría el siguiente formato

```
"rules":
        "lt":
                "snr_threshold": 29000
```

Esta funcionalidad otorga flexibilidad total a la configuración de los dispositivos, ya que permite ejecutar cualquier combinación con cualquier configuración y verificar el funcionamiento de configuraciones nuevas de forma rápida antes de agregar los tests correspondientes. En caso que alguna funcionalidad nueva contenga bugs se le puede notificar rápidamente a los desarrolladores, logrando que se trabaje en

paralelo, los desarrolladores solucionan los bugs mientras el equipo de QA agrega los tests.

6.2 Conversión de configuración final a C

Debido a las capas de abstracción que presenta la plataforma de testeo, uno de los problemas que se presentaba era el desconocimiento de cada campo específico de una configuración, lo que dificultaba a los desarrolladores replicar la configuración con la cual se ejecutó un test desde su plataforma en C. Por este motivo se agregó una funcionalidad que convierte la estructura de configuración final de Python a C, permitiéndole a los desarrolladores referirse a la configuración de la plataforma de testeo, copiarla a su plataforma y replicar los fallos encontrados por el equipo de QA.

```
rules.pll.pll_ignore_lol = False
rules.pll.pll_settings_disable = False
rules.pll.spll_mode = 0
rules.pll.tmon_cal_disable = False
rules.pll.tmon_cal_force = 0

rules.rx.ac_coupling_bypass = True

rules.rx.adv.ffe_dc_adapt_force = False
rules.rx.adv.ffe_fir_adapt_force = False
rules.rx.adv.ffe_gain_adapt_force = False
rules.rx.adv.ffe_leakage_eta = 7
rules.rx.adv.skip_dsp_fine_tune = False
rules.rx.afe_trim = 6
rules.rx.baud_rate = 28125000
rules.rx.cmi_mode = 1
rules.rx.ctle = 0
rules.rx.ctle1 = 0
rules.rx.ctle2 = 0
rules.rx.ctle_manual_control = False
rules.rx.dfe_precoder_en = False
rules.rx.dsp_mode = 2
rules.rx.dtl_mode = 0
rules.rx.enable = True
rules.rx.gray_mapping = True
rules.rx.ieee_demap = True
rules.rx.invert_chan = False
rules.rx.mlsd_en = False
rules.rx.pga_att_en = False
rules.rx.prbs_chk_en = False
rules.rx.preamp_bias_ctrl = 0

rules.rx.rx_qc.data_mode_dis = False
rules.rx.rx_qc.data_mode_hist_dis = True
rules.rx.rx_qc.data_mode_mse_min_threshold = 112
rules.rx.rx_qc.data_mode_retry_fail_max = 5
rules.rx.rx_qc.dis = True
rules.rx.rx_qc.hist_dis = True
rules.rx.rx_qc.mse_min_threshold = 91
rules.rx.rx_qc.retry_pass_max = 1
rules.rx.signalling = 0
rules.rx.src = 0
rules.rx.subrate_ratio = 0
rules.rx.vga_tracking = False

rules.tx.baud_rate = 28125000
rules.tx.driver_mode = 0
rules.tx.enable = True
```

Esta funcionalidad logró mejorar la sincronización entre desarrolladores y equipo de QA, reduciendo el tiempo que se requería para notificar a los desarrolladores de una configuración que contenía bugs o fallos y permitiendo proveer la configuración para replicar dicho fallo directamente.

6.3 Clustering de fallos

Si bien la plataforma de testeo ejecuta una test suite diariamente en un servidor de automatización (Jenkins) que tiene soporte para procesar los resultados de cada test suite y generar reportes, se presenta la siguiente situación:

1. Se debe generar una nueva versión del firmware para distribuir a los clientes.
2. Se corre una test suite en el servidor de automatización que generalmente contiene entre 500 y 2500 tests.
3. Una vez finalizada, se genera automáticamente un reporte con el status de cada testcase, su output y mensajes de error (si es que falló).
4. Dependiendo del estado de madurez del firmware del dispositivo o algún error introducido en el desarrollo del firmware puede haber más de 100 fallos presentes.

El equipo de QA debía analizar manualmente uno por uno los errores, identificar la causa de los fallos y reportar a los desarrolladores la cantidad de tests que falló por cada causa de error.

Este proceso significaba una pérdida de tiempo de horas para el equipo de QA y los desarrolladores no podían comenzar a trabajar en los bugs. Además, debido a la diferencia horaria del equipo de QA y los desarrolladores, en algunos casos se requería esperar al día siguiente para obtener un reporte manual del estado de la nueva versión de firmware.

Se buscó automatizar el proceso de agrupamiento de errores similares y generar un reporte extra, con el objetivo de liberar tiempo del equipo de QA y permitir a los desarrolladores identificar las causas de fallos más comunes sin tener que esperar al equipo de QA.

A continuación se nombran las ventajas y desventajas de las posibles soluciones:

1. **Regular expressions**:

a. Ventajas:
    i. Permite generar una lista {regular expression: causa de fallo}.
    ii. Se tiene control total sobre todas las causas de fallo y mensaje de error.
b. Desventajas:
    i. Los mensajes de error son específicos para cada proyecto, por lo que cada proyecto debe generar y mantener su propia lista de regular expressions.
    ii. A medida que se encuentran nuevos errores debe agregarse la regular expression.
    iii. La plataforma genera gran parte de cada mensaje de error en runtime, por lo que el patrón de la regular expression no es trivial.
    iv. Existen causas de errores iguales con mensajes de error ligeramente distintos, por ejemplo “Time out waiting for link ready on Device” y “Waiting for link ready on Device failed” corresponden al mismo error.

2. **Error codes**:
    a. Ventajas:
        i. Cada causa de error queda unívocamente referenciada por un código de error, por ejemplo “SNR too low” = 1, “link not ready” = 2. La solución queda independiente al formato de error de mensaje, siempre que se especifique el código de error.
    b. Desventajas:
        i. Aplicar esta solución implicaría revisar toda la codebase y agregar el mensaje de error a cada fallo.
        ii. Sería necesario actualizar la lista de códigos de error a medida que se encuentren nuevos errores.
        iii. Algunos errores se detectan en runtime y es difícil o imposible asignar un código de error en el código.
3. **Clustering**: Clustering es la tarea de agrupar un conjunto de objetos (en este caso fallos) de tal forma que objetos del mismo grupo son más similares a aquellos en otros grupos.
    a. Ventajas:

i. Es directamente aplicable a todos los proyectos ya que únicamente requiere el nombre del test y el mensaje de error, a los cuales se tiene acceso.
ii. No se requiere conocimiento previo de los posibles errores, como se generan los mensajes de errores, etc.

b. Desventajas:
i. Es necesario encontrar y entrenar un modelo estadístico.

Luego de analizar las soluciones, se optó por utilizar **Clustering**, el proceso y modelo estadístico sigue los siguientes pasos

1. Leer el reporte generado automáticamente, tomar los tests que fallaron con sus respectivos mensajes de error.
2. Procesar los mensajes de errores para eliminar similitud o disimilitud no deseada.
3. Generar una matriz de correlación de tests “A” dada por la siguiente ecuación:

$$A[i,j] = corr(test_i, test_j) = f(test_i.error\_message, test_j.error\_message)$$

4. Generar la matriz de disimilitud “B” a partir de “A” considerando que:

$$disimilitud(a,b) = x \in R: x \in [0,1]$$

$$disimilitud(a,b) = f(corr(a,b))$$

$$Si\ corr = 1,\ disimilitud = 0$$

$$Si\ corr = 0,\ disimilitud = 1$$

5. A partir de B, generar clusters utilizando el algoritmo de distancia apropiado.

Una vez decidido del modelo a utilizar, los siguientes parámetros debieron ser elegidos y ajustados cuidadosamente:

1. Función de correlación en base a mensajes de error.
2. Función de disimilitud en base a la correlación.
3. Método y métrica de clustering para generar la matriz intermedia jerárquica.
4. Método, límite y criterio para formar la matriz de clustering plana a partir de la matriz intermedia jerárquica.

En la siguiente imagen presenta el resultado final en una test suite de AN-LT, se puede observar que los errores similares quedan agrupados en el mismo cluster

cluster	name	status	message
4	test_anlt_squelch[rate_1x2x25gcr_con_die0_tx0_die1_rx0_die1_tx1_die1_rx1_split]	failed	['V[redacted] is not locked after squelching/unsquelching']
4	test_anlt_squelch[rate_1x2x25gcr_con_die1_tx1_die0_rx0_die0_tx1_die0_rx1_split_DFE1_RC_DFE2_LT_STANDALONE]	failed	['V[redacted] is not locked after squelching/unsquelching']
4	test_anlt_squelch[rate_1x2x25gkr_die0_tx0_die1_rx0_die1_tx1_die1_rx1_split]	failed	['V[redacted] is not locked after squelching/unsquelching']
4	test_anlt_squelch[rate_1x2x25gkr_die0_tx1_die0_rx0_die1_tx0_die0_rx1_split]	failed	['V[redacted] is not locked after squelching/unsquelching']
4	test_anlt_squelch[rate_1x2x25gkr_die1_tx0_die0_rx0_die1_tx1_die1_rx1_split]	failed	['S[redacted] is not locked after squelching/unsquelching', 'V[redacted] is not locked after squelching/unsquelching']
4	test_anlt_squelch[rate_1x2x25gkr_die1_tx0_die0_rx1_die1_tx1_die1_rx1_split]	failed	['S[redacted] is not locked after squelching/unsquelching']
4	test_anlt_squelch[rate_1x2x25gkr_die1_tx0_die1_rx0_die1_tx1_die1_rx1]	failed	['S[redacted] is not locked after squelching/unsquelching', 'V[redacted] is not locked after squelching/unsquelching']
4	test_anlt_squelch[rate_1x4x25gcr_die0_tx1_die0_rx0_die1_tx1_die0_rx1_die1_tx0_die1_rx0_die0_tx0_die1_rx1_split]	failed	['V[redacted] is not locked after squelching/unsquelching']
4	test_anlt_squelch[rate_25gcr_con_die1_tx1_die1_rx1]	failed	['V[redacted] is not locked after squelching/unsquelching']
4	test_anlt_squelch[rate_25gcr_con_die1_tx1_die1_rx1_BYP_FIR_WALK]	failed	['V[redacted] is not locked after squelching/unsquelching']
4	test_anlt_squelch[rate_25gkr_con_die1_tx1_die1_rx1]	failed	['V[redacted] is not locked after squelching/unsquelching']
4	test_anlt_squelch[rate_2x2x25gkr_die1_tx1_die0_rx0_die0_tx1_die0_rx1_die0_tx0_die1_rx0_die1_tx0_die1_rx1_split]	failed	['V[redacted] is not locked after squelching/unsquelching']
1	test_an_probe[rate_100gkr4_die1_tx0_die1_rx1_split]	failed	['AN timed out, DUT ([redacted]) HCD = 100GBASE_KR4, DUT status = COMPLETE, LP ([redacted]) HCD = NOT SUPPORTED, LP status = BUSY', 'AN/LT failed, DUT ([redacted]) HCD = 100GBASE_KR4, DUT status = COMPLETE, LP ([redacted]) HCD = NOT SUPPORTED, LP status = BUSY', "DUT (100GBASE_KR4) and LP (NOT SUPPORTED) didn't negotiate to the same rate"]
1	test_an_probe[rate_25gcr_con_die0_tx1_die1_rx1_split]	failed	['AN timed out, DUT ([redacted]) HCD = NOT SUPPORTED, DUT status = BUSY, LP ([redacted]) HCD = NOT SUPPORTED, LP status = BUSY', 'AN/LT failed, DUT ([redacted]) HCD = NOT SUPPORTED, DUT status = BUSY, LP ([redacted]) HCD = NOT SUPPORTED, LP status = BUSY', 'DUT ([redacted]) did not negotiate for 25GBASE_CR1_CONS']

Esta funcionalidad fue aplicada a todos los proyectos, ya que los algoritmos utilizados son independientes a la funcionalidad de cada proyecto. Se lograron los resultados deseados inicialmente, el equipo de QA puede identificar causas de fallos y reportar errores con mayor rapidez, los desarrolladores pueden analizar al reporte generado sin necesitar del equipo de QA y priorizar aquellos bugs que afectan a la mayor cantidad de tests, es decir, los clusters que contienen más elementos.

6.4 Análisis de datos a través de distintas versiones de firmware

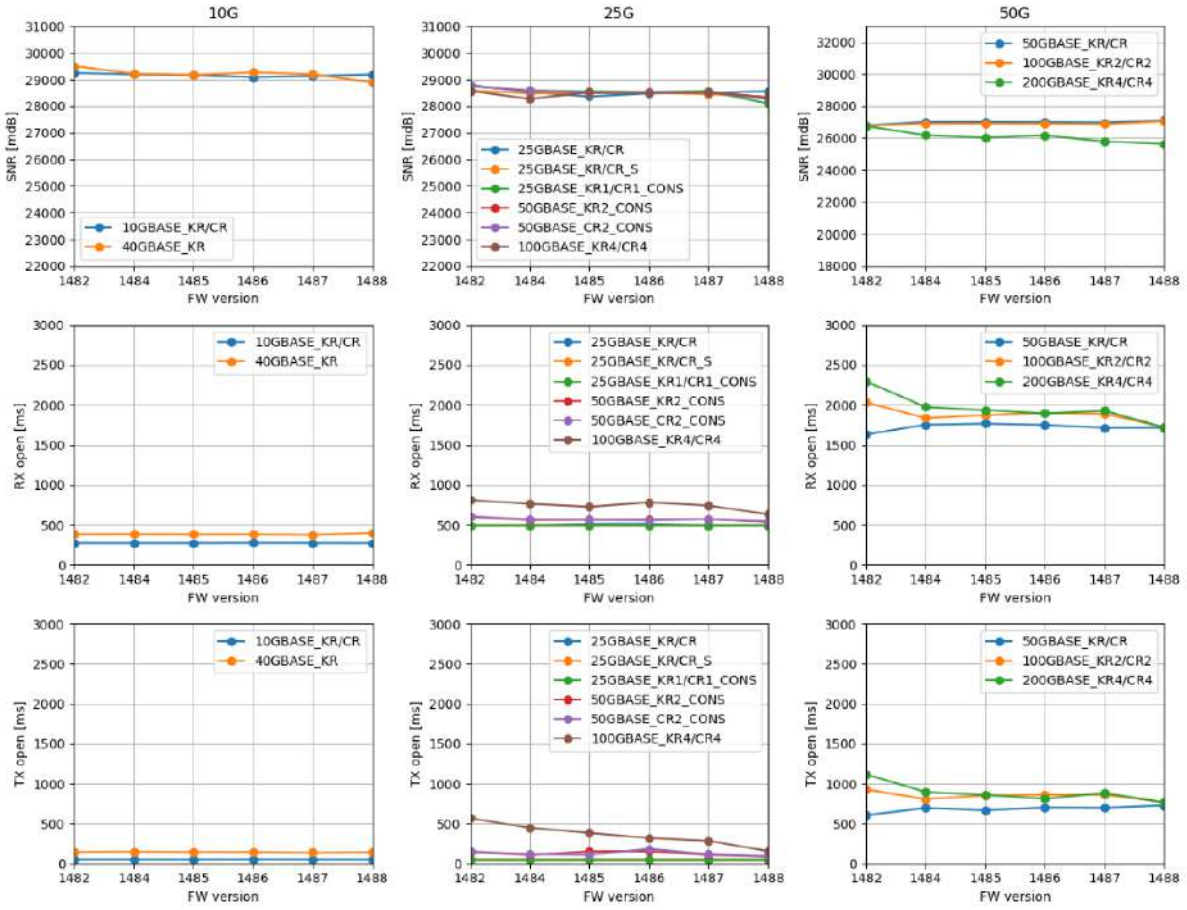
La plataforma de testeo no tiene memoria de valores medidos anteriormente y únicamente puede comparar valores actuales con rangos esperados. A modo de ejemplo, la siguiente situación no estaba contemplada en la plataforma:

1. El rango de SNR para NRZ en las últimas versiones del firmware se encuentra entre 28 y 30 dB.
2. El rango esperado de SNR en la plataforma se especifica entre 20 y 35 dB.
3. Se agrega una nueva funcionalidad a la versión de firmware X que resulta en una degradación SNR en 6dB. Al seguir estando dentro del rango esperado, la plataforma no detecta el cambio.
4. Se agrega otra nueva funcionalidad a la versión de firmware X+10 que resulta en una degradación de la SNR en 3dB. Al encontrarse por debajo del rango esperado la plataforma reporta fallos.

5. Se arregla la funcionalidad introducida en X+10, pero la funcionalidad introducida en X que resulta en una degradación mayor de la SNR nunca se detectó.

Con el fin de poder detectar cambios en mediciones significativos que no son identificados por la plataforma de testeo se decidió por guardar las mediciones relevantes en una base de datos para cada test suite ejecutada e implementar una plataforma extra de análisis de datos que permite generar gráficos, tablas y detectar cambios significativos en las distintas métricas de calidad de los dispositivos y firmware.

En la siguiente imagen se observa un ejemplo de parámetros relevantes para AN-LT a través de distintas versiones del firmware.



Estos gráficos se generan automáticamente y se agregan a los reportes generados por cada test suite. Tanto el equipo de QA como los desarrolladores pueden referir al reporte e identificar degradaciones o cambios significativos en los diferentes parámetros de calidad observando gráficos, sin necesidad de navegar por el historial de test suite y comparar mediciones manualmente. En caso de haberse producido

una degradación por un cambio introducido en el firmware se logra detectar y corregir con mayor rapidez para la siguiente versión.

6.5 Robustness testing

La plataforma de testeo y el entorno real de los dispositivos no eran iguales. Mientras que la plataforma de testeo recarga el firmware y reinicia los dispositivos para cada test con el objetivo de asegurar que configuraciones anteriores no interfieran con las actuales, los clientes cargan el firmware únicamente cuando hay una versión nueva (cada 1 mes aprox.), se realizan múltiples reconfiguraciones de cada dispositivo y se presentan factores externos constantes que estresan los dispositivos como caída de comunicación, disrupción de tráfico, etc. que en la plataforma se simulan solo una vez en cada test.

Estas diferencias llevaron a múltiples situaciones en las que los clientes reportaron fallos estadísticos que no eran detectados en la plataforma de testeo.

Se requería, por lo tanto, testear los dispositivos bajo las mismas condiciones de uso que los clientes. Se agregó a la plataforma la funcionalidad denominada ***Robustness testing***, que busca generar una cantidad de estrés en los dispositivos similares a los que se presentan en el entorno de aplicación real. Por cada iteración (disrupción de tráfico, caída de canal/comunicación, etc.) se realizan mediciones de parámetros de calidad. Al finalizar una test suite de robustness se archiva la *raw data* generada, que luego puede ser procesada por los desarrolladores o el equipo de QA para generar reportes, o ser entregada como prueba de fidelidad a los clientes.

6.7 Regression de fallos previos

En la plataforma existen fallos no consistentes, son tests que al ejecutarse múltiples veces bajo las mismas condiciones muestran un comportamiento no determinista, es decir, fallan y pasan sin patrón aparente.

Con el objetivo de permitir a los desarrolladores discriminar entre fallos consistentes y no consistentes, se decidió implementar funcionalidad para correr únicamente los tests que fallaron en una test suite previa. Esta funcionalidad no asegura que los

fallos no consistentes pasen en la segunda iteración, pero permite descartar rápidamente gran cantidad de los mismos, y el equipo de QA o desarrolladores debe correr manualmente unos pocos casos.

Cabe aclarar que el equipo de QA había implementado funcionalidad para correr una secuencia de tests especificada en un archivo con el formato correspondiente, por lo que los pasos y requisitos restantes eran los siguientes:

1. Generar un archivo con el nombre de los tests que fallaron. Este paso ya se ejecutaba para la implementación de *Error Clustering*.
2. Generar un segundo archivo con el formato requerido por *pytest* a partir del primero. Es necesario en este paso agregar los tests incrementales previos al fallo, es decir, si test_anlt_C falló el script debe agregar automáticamente test_anlt_A y test_anlt_B.

El archivo especificado en 2. se archiva en la herramienta de CI/CD y se agregó soporte para que la herramienta de CI/CD pueda recibir un archivo, completando la funcionalidad requerida. De esta forma los desarrolladores o equipo de QA únicamente debe descargar el archivo correspondiente de la regresión de interés y cargarlo en la herramienta de CI/CD.

Anteriormente cada test se debía correr manualmente para determinar si era consistente o no, por lo que esta funcionalidad reduce significativamente el tiempo desperdiciado por QA y los desarrolladores.

Bibliografía

Anexos

a. Combinaciones de AN-LT para un dispositivo de 4 canales

Consideraciones:

1. Un dispositivo de 4 canales tiene 4 TX y 4 RX debido a que el TX y RX no necesariamente se encuentran en el mismo canal físico.
2. Los modos de comunicación son de 1, 2 y 4 canales por bundle.
3. Por convención, el *an leader* se corresponde siempre al RX de menor canal físico, de lo contrario a cada total de combinaciones de múltiples canales habría que multiplicarla por la cantidad de canales.
4. Para casos con múltiples bundles, únicamente se generan combinaciones de igual cantidad de canales por bundle. No se ejecutan casos de 1 bundle de 1 canal y un bundle de 2 canales activos al mismo tiempo. La razón por la que no se generan es que 2 bundles de 2 canales generan un mayor estrés en el MCU que un bundle de 1 canal y uno de 2, y el mayor estrés se genera para 4 bundles de 1 canal cada uno.
5. La nomenclatura de agrupamiento de canales en un bundle se toma con el índice de cada set. Por ejemplo {0,1,2,3} es un caso de 4 bundles de un canal cada uno, {0,0} es un bundle de 2 canales, etc.
6. La nomenclatura para una combinación se toma NxM donde N es el número de bundles y M la cantidad de canales por bundle.

Casos por canal:

1. Para 1 canal hay 16 combinaciones, 4 RX y 4 TX, $4 * 4 = 16$. No hay diferentes combinaciones de bundles.
2. Para 2 canales se debe considerar que el leader es siempre el RX de menor valor, por lo tanto:
    a. El primer RX siempre puede elegir entre 4 TX para conectarse y el segundo entre 3 TX, dando 12 por cada par posible de RX.
    b. Solo se pueden tomar 3 RX, dado que si se toma el canal RX3 no queda un RX de mayor denominación y no se respeta la convención de an leader.

c. Si el primer RX0 => segundo RX $\epsilon\{1, 2, 3\}$ da un total de 3 combinaciones, de la misma se obtienen 2 para RX1 y 1 para RX3, dando un total de 6 combinaciones de pares RX.
d. Hay 2 formas de conectar 2 canales, {0,1} (2x1) o {0,0} (1x2).
e. El total es entonces $12 * 6 * 2 = 144$

3. Para 4 canales hay 24 combinaciones de canales. Se puede obtener el resultado asumiendo que los RX quedan fijos activos como {0,1,2,3} y los TX cambian su conexión con los RX, por lo que el resultado sería las permutaciones de 4 canales tomados de a 4, $4! = 24$. Estas 24 combinaciones deben multiplicarse por las 5 de formas de tomar 4 canales en bundles enteros de igual número que se detallan a continuación, dando un total de 120:
    a. {0,0,0,0} (1x4)
    b. {0,0,1,1}, {0,1,0,1}, {1,0,0,1} (2x2)
    c. {0,1,2,3} (4x1)

b. Codigos de correccion de errores en AN-LT

Como se mencionó en la sección *1.2*, existen dos códigos de corrección de errores que se pueden indicar en el proceso de AN-LT: FireCode y Reed-Solomon. Si bien los códigos *Reed-Solomon* tienen diversas aplicaciones y bibliografía, *FireCode* únicamente se utiliza en comunicaciones digitales

b.1 FireCode (FC)

FireCode es un CRC(2112, 2080), lo que indica que cada bloque de FEC tiene una longitud de 2112 bits, donde 2080 son información y 32 son bits de paridad. El código garantiza una corrección de hasta 11 bits por bloque.

El bloque de FEC contiene 32 filas de 65 bits cada una, 64 bits de información y 1 bit de *transcoding overhead* (T-bit). Al final de cada bloque se agregan 32 bits de paridad. Esto da el total de $32 * 65 + 32 = 2112$ bits por bloque.

El código no decrementa el *symbol rate* de la capa física ni aumenta la frecuencia de señalización. Se mapean los 66 bits de palabra (64 de datos y 2 de sincronización) en 64 bits de datos y 1 de *transcoding overhead*, lo que permite agregar 32 bits de paridad cada 32 palabras de 64 bits y mantener tanto la frecuencia como el symbol rate.