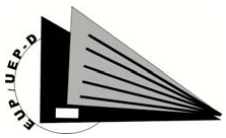
MÁSTER UNIVERSITARIO EN

INGENIERÍA DE SISTEMAS EMPOTRADOS



Escuela Politécnica
Donostia-San Sebastián

VALIDACIÓN Y VERIFICACIÓN DE SOFTWARE CON HERRAMIENTAS AVANZADAS

Alumno: Ibon Fernandez Txurruka

Director: Andoni Arruti Illarramendi

Instructor en la empresa: Ainhoa Aristimuño

Proyecto Fin de Máster, julio de 2016

RESUMEN

Cada vez es más importante que los sistemas hardware en la industria del transporte ferroviario superen estrictas pruebas de validación y verificación y con ello cumplir con las especificaciones internacionales de calidad. El software de los sistemas embebidos en particular, juega un papel crítico en este campo y exige la aplicación de diversos procesos de test que evalúen si el sistema cumple o no con los requisitos propuestos inicialmente.

El rack VEGA contiene varios submódulos en la que se encuentra la CPU basada en una arquitectura *MPC8313*. Uno de los códigos fuentes que se ejecutan en ella contiene el nombre de AX00 y es un software genérico que controla la Unidad de Control de Tracción (TCU) que a su vez contiene la Traction Control Core (TCC). Aplicar un proceso de test es de suma importancia si se quiere mantener un control de calidad sobre el software.

ÍNDICE

LISTA DE ILUSTRACIONES

LISTA DE TABLAS

LISTA DE ECUACIONES

1. INTRODUCIÓN

El presente documento tiene por objeto recopilar el proceso llevado a cabo en el desarrollo de técnicas de validación y verificación con el entorno de LDRA, aplicando a posteriori un ciclo de integración continua en Jenkins.

1.1 Descripción general del proyecto

Este proyecto se ha basado en la aplicación de procesos de análisis estático junto a pruebas unitarias de software embebido para garantizar que un sistema cumpla con las especificaciones de calidad necesarias. Para ello se ha hecho uso de herramientas que implementan motores de análisis estático y dinámico de LDRA con los que se ha comprobado el cumplimiento de los estándares internacionales, reglas personalizadas y métricas de calidad, así como la correcta implementación funcional y estructural del programa.

El código fuente con el que se ha trabajado forma parte del software del núcleo de aplicación de tren dentro del programa de la Unidad de Control de Tracción, desarrollado como software genérico por CAF Power & Automation. Se ha seleccionado la clase *MultiTractControlCore* como principal foco de análisis por su complejidad y gran cantidad de funciones, aunque de manera adicional, se ha trabajado con otras dos unidades secundarias llamadas *TractControlCore* y *AlarmControl* que se han utilizado en el proceso de integración de unidades.

Éste también es un proyecto que ha pretendido implementar diferentes soluciones para el uso industrial del entorno. Por una parte se ha desarrollado una integración continua en Jenkins implementando una serie de scripts que han automatizado varios procesos de validación generado al mismo tiempo los informes necesarios. Esto ha permitido agilizar y optimizar todo el proceso de pruebas diseñado con las herramientas de entorno de LDRA.

Por último se ha implementado una adaptación de todo el entorno de trabajo para que la configuración y la ejecución de los Test sea la más sencilla posible, quedando su uso accesible al personal con conocimientos limitados de las herramientas.

1.2 Propósito del proyecto

La estrategia que se persigue desde ULMA Embedded Solutions S.Coop. se basa en la formación de personal en tecnologías emergentes para el desarrollo de técnicas de testing. En consecuencia el primer punto del proyecto ha englobado un proceso de aprendizaje de conceptos esenciales de validación y verificación de software obteniendo la capacitación para el uso de diferentes entornos de trabajo de LDRA. En el marco de la consultoría y el desarrollo de soluciones tecnológicas de gestión, instalación y administración de sistemas tecnológicos, el proyecto ha buscado analizar un software con una aplicación real dentro de la industria ferroviaria en busca de deficiencias y con miras a plantear posibles mejoras para el cumplimiento del estándar ferroviario EN50128:SIL2 (*ver Tabla 4*).

1.3 Fases del proyecto

Este proyecto contempla siete fases principales:

1- Validación y verificación del código mediante análisis estático en TBvision con Visual Studio 2010.

2- Validación y verificación del código mediante prueba unitaria en TBrun con Visual Studio 2010.

3- Pruebas de integración del código mediante TBrun con Visual Studio 2010.

4- Validación y verificación del código On-Target mediante análisis estático en TBvision con Integrity.

5- Validación y verificación del código On-Target mediante prueba unitaria en TBrun con Integrity.

6- Pruebas de integración del código On-Target mediante TBrun con Integrity.

7- Integración continua del código en Jenkins con Visual Studio 2010.

1.4 Enmarque del proyecto

Este proyecto fin de máster se ha llevado a cabo en la empresa ULMA Embedded Solutions S.Coop ubicada en Oñati (Gipuzkoa). Esta empresa está situada dentro del grupo ULMA y proporciona servicios de ingeniería en el ámbito de sistemas embebidos.

El presente trabajo se enmarca dentro de la línea de cooperación entre Ulma Embedded Solutions y Liverpool Data Research Associates que proveen a la industria tecnológica de herramientas para análisis, test y requerimientos de trazabilidad del software, así como CAF Power and Automation el cual explora nuevas posibilidades en las relaciones con ULMA Embedded Solutions en el contexto de la validación de software.

Ilustración 1 - Logotipo de Ulma Embedded Solutions

1.4.1 Sobre el grupo ULMA

Hace cincuenta años que se formo el Grupo ULMA, un proyecto nacido de la ilusión así como del carácter emprendedor de unas personas que siguiendo ese sueño cooperativo, decidieron dejar sus anteriores empleos y se embarcaron en aquella aventura.

De aquel esfuerzo y dedicación nacieron diferentes negocios que perduran hasta el día de hoy. En una sucesión de cooperativas que han pasado a formar parte de un importantísimo, por cifra de negocio y número de personas que trabajan en el mismo, grupo empresarial que con ahínco trata de expandirse, internacionalizarse y diversificar su actividad; donde tal vez, esto último conforme su mayor valor o activo. No obstante, sin renunciar al espíritu cooperativo que marca su nacimiento y que todavía está absolutamente presente en el ADN de la entidad, hoy en día se encuentra dentro de un grupo empresarial internacional mayor, la Corporación Mondragón, que es el mayor grupo empresarial cooperativo del mundo y también, por su singularidad, objeto de estudio por importantes Escuelas de Negocios a la vista de los éxitos obtenidos por este modelo.

1.4.2 Sobre ULMA Embedded Solutions

ULMA Embedded Solutions fue creada en 2009 en la incubadora del Grupo ULMA con el objetivo de ofrecer servicios especializados más allá de los prototipos funcionales de sistemas electrónicos, con una respuesta más acorde a las necesidades de la industria.

Hoy en día ofrece servicios especializados de ingeniería a lo largo de todo el ciclo de vida del producto electrónico, desde la conceptualización hasta la fabricación y mantenimiento, incluyendo la especificación de requisitos y el plan de validación, las fases de diseño, desarrollo y test.

Trabajan principalmente en sectores regulados por normativas y estándares como EN 50155, IEC 62304, EN 50128, ISO 26262, IEC 60730, EN 60601, UL 61010, ISO 13849, IEC 61508, etc.

Colaboran con el cliente y sus partners para desarrollar sistemas innovadores, competitivos y escalables. Entre su red de colaboradores se encuentran: Altium, IBM, LDRA, Mondragón Unibertsitatea, National Instruments, NXP, The Reuse Company, Xilinx y XJTAG

2. ESTADO DEL ARTE

2.1 Definiciones

En este apartado se definen algunos conceptos que serán esenciales para la comprensión integra del documento.

2.1.1 Calidad

A la hora de analizar SW, la calidad se basa en la aptitud del sistema para satisfacer las necesidades de la especificación. Para la medición de sus aspectos se miden distintas características:

- **Cumplimiento:** Es la cualidad que define hasta qué punto el sistema ha podido consumar las necesidades de los interesados.
- **Conformidad:** Cumplimiento de los requisitos.
- **Calidad intrínseca:** Mide si el sistema es robusto, fiable, fácil de mantener, etc.
- **Percepción:** La aceptación y reacción del cliente ante el producto final.

2.1.2 Verificación

El proceso que se realiza para revisar si una determinada cosa está cumpliendo con los requisitos y normas previstos**.**

2.1.3 Validación

Proceso de revisión al que se somete un programa informático para comprobar que cumple con sus especificaciones.

2.1.4 Lista de abreviaturas y definiciones

API	Interfaz de Programación de Aplicaciones
SW	Software
HW	Hardware
TCU	Traction Control Unit (Unidad de Control de Cracción)
TCC	Traction Control Core (Núcleo del Control de Tracción)
VEGA	Vehicle Electronics for Generic Application
ON-TARGET	Testear la aplicación sobre TARGET
LDRA	Asociación de Investigación de Datos de Liverpool
CAF	Construcciones y Auxiliar de Ferrocarriles, S.A.
IDE	Entorno de Desarrollo Integrado
BUG	Un fallo de Software
RTOS	Sistema Operativo en Tiempo Real
MICRO	Circuito Integrado Programable
Host OS	Sistema Operativo instalado en un PC
PC	Ordenador Personal
TCMS	Traction Control and Monitoring System
LCSAJ	Linear Code Sequence and Jump
CPU	Central Processing Unit
DSP	Digital Signal Processor

2.2 Concepto de Testing

El proceso de test es algo que tiene que aplicarse desde el primer instante del ciclo de vida del proyecto y es a menudo el último recurso en la búsqueda de errores. Previamente se pasa por una fase de especificación de requisitos y la realización de un diseño que a su vez, se debe poner en práctica. Es entonces cuando una sucesión de pruebas verifica el diseño sobre los requisitos establecidos, validando así tanto el diseño como los propios requisitos.

Aun así el testing no deja de ser algo imperfecto, más aun si no se lleva a cabo de forma correcta. Cuanto más rigurosos sean los test, mayor será la cantidad de averías detectadas y mayor será el coste de enmendarlas. Este sobreesfuerzo aumenta de manera exponencial a medida que se realicen pruebas más rigurosas, por lo que la solución pasa por encontrar un equilibrio que consiga combinar al mismo tiempo distintas técnicas de verificación en el proceso previo a las pruebas de validación.

Es importante trazar una estrategia para enfocar este esfuerzo de manera más eficiente. La cadena ERROR → FALLO → AVERIA establece que la corrección de fallos antes de que estos se conviertan en averías es más efectiva que subsanarlos a posteriori. De la misma forma, encontrar errores es más óptimo que corregir los fallos.

A veces la revisión del código queda en manos de los propios programadores, donde encontrar un error puede evitar que se cumpla la cadena de fallos y averías. Por otra parte, el análisis estático es usado para la detección de fallos antes de que estos se conviertan en averías, pero no sirve para evaluar el código y tampoco para validarlo. De esta amanera, la ejecución de test es la que consigue detectar estas averías del sistema. Estos son únicamente efectivos si parten de una completa y consistente especificación del comportamiento esperado que no permita ningún resquicio de ambigüedad.

La combinación de diferentes técnicas para conseguir un resultado óptimo se refleja en el modelo *Swiss Chesse* (*ver ilustración 2*) donde se aplica una sucesión de capas que se diferencian durante el proceso de validación y verificación del código. Primero se efectúa una **revisión** sobre criterios básicos de programación que conlleva tres pasos esenciales:

- **Encapsulación:** Ocultación de la información, haciendo uso de atributos de tipo prívate, métodos públicos, limitar el uso de expresiones a la menor cantidad de métodos posible, etc.

- **Cohesión:** Consiste en que cada clase sea responsable de una tarea bien definida, definición del nombre de la clase haciendo que éste tenga relación directa con el contenido, fácil entendimiento del código siendo este reutilizable, etc.

- **Acoplamiento:** El nivel de dependencia que tiene una clase sobre otra, ocultación de los detalles de implementación (funcionamiento interno o cómo se ejecutan las tareas) para que un cambio en una clase no exija la modificación de otra.

La siguiente capa del modelo *Swiss Chesse* se basaría en el **análisis estático** del código (*ver capítulo 2.3*) que detecta prácticas incorrectas en el software, al igual que errores de control y flujo de datos. Por último el proceso de **test** basado en la prueba unitaria (*ver capítulo 2.7*) verifica el comportamiento y la robustez del sistema.

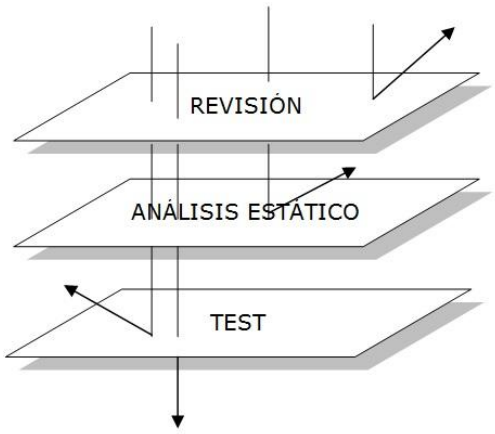


Ilustración 2 - Modelo Swiss Cheese

2.3 Análisis estático

En análisis estático es aquél que se fundamenta en la verificación del software sin que éste tenga que ser ejecutado. En este capítulo se expone la información necesaria para comprender en qué subconjuntos se divide y qué opciones ofrecen las herramientas de LDRA para su puesta en marcha.

2.3.1 Subconjuntos del análisis estático

Principalmente se divide en cuatro partes que son la coincidencia de patrones, análisis semántico, ejecución simbólica e interpretación abstracta.

- **Análisis de patrones:** Es una técnica que tiende a buscar expresiones inadecuadas y entre otras cosas, el cumplimiento de diferentes estilos a la hora de escribir código, regulando nombres de variables, etc.

- **Análisis semántico:** Hace uso de la primera etapa de un proceso de compilación para construir un árbol de sintaxis abstracta. A su vez a este árbol se le añade información semántica adicional para una búsqueda exhaustiva de violaciones de reglas en lenguajes programación. El objetivo es analizar el código fuente en busca de estructuras peligrosas del lenguaje.

- **Ejecución simbólica:** Lleva a cabo un análisis del flujo de datos. De esta manera se fija en cómo los datos son creados, usados y eliminados. Esta tarea sustituye el programa con valores simbólicos y considera el sistema como una sucesión de sentencias ejecutables, consiguiendo averiguar si pueden ocurrir errores en el código y cuando pueden hacerlo. Son muchos los fallos que se consiguen captar con esta técnica, tales como datos no inicializados, funciones que no se llegan a usar, alcance de los problemas, puntos muertos, mal uso de punteros, etc.

- **Interpretación abstracta:** Hace uso de modelos matemáticos para predecir errores en tiempo de ejecución. Funciona haciendo uso del código como un modelo abstracto y ejecutando posteriormente dicho modelo en busca de fallos. Cuanto más se asemeje el modelo al código original, menos recursos consumirá su análisis. Esto es algo a tomar en cuenta ya que es un proceso que consume muchos medios.

2.3.2 Opciones de análisis estático en LDRA

LDRA ofrece una gran variedad de opciones para ejecutar el análisis estático. Algunas son necesarias para que en un momento posterior se tengan que efectuar pruebas unitarias. Su aplicación añade criterios más estrictos a la hora de analizar la calidad del código.

Main Static Analysis

El "Main Static Analysis" da comienzo al análisis del código fuente. Reformatea una copia del archivo de origen, que facilita un análisis más detallado de todo el motor de LDRA (*LDRA Testbed*). También ejecuta una búsqueda sobre el código fuente de todos los estándares de programación, violaciones del código y las LCSAJ que puedan estar presentes. Finalmente el "Main Static Analysis" asigna la estructura de los procedimientos de código fuente para producir el *Static Callgraph* (*ver ilustración 33*). El "Main Static Analysis" es el requisito previo para todas las demás instalaciones de *LDRA Testbed*, y por lo tanto tiene que ser ejecutado antes de proceder con las otras opciones de análisis estáticas o dinámicas.

Complexity Analysis

Es el siguiente paso después de la opción "Main Static Analysis" y es un requisito previo al "Static Data Flow Analysis". El "Complexity Analysis" informa sobre la estructura subyacente del código fuente, calculando una variedad de métricas que pueden ser utilizadas para hacer cumplir diversas normas sobre la complejidad del programa. En el caso de C / C ++ el "Complexity Analysis" analiza el código en una función a los procedimientos existentes. La extensión del análisis es configurable en el sentido de que el propio usuario puede decidir si obtener todos los detalles o ninguno, en algunos de los procedimientos o en todos ellos. Se produce siempre un informe de síntesis para cada proceso.

Static Data Flow Analysis

Este análisis produce tres tipos de información sobre el código fuente: La "Procedure Call Information" enumera cada método, indicando todas sus llamadas a otras funciones dentro de la propia estructura del procedimiento (incluyendo llamadas recursivas), la detección de "Data Flow Anomaly" que analiza las acciones sobre las variables en el archivo de origen e informa de cualquier posible problema con su uso y por último el "Procedure Interface Analysis" que informa de cada procedimiento y de los usos de todos los parámetros y las variables globales utilizadas en el mismo. "Static Data Flow Analysis" recorre el código fuente en busca de varios tipos de usos anómalos de las variables, generando un amplio informe que pone de relieve toda una gama de posibles defectos de programa.

Cross Reference

LDRA Testbed es capaz de producir una referencia cruzada de todos los elementos de datos (parámetros globales y locales) utilizados en el código fuente, estableciendo también su uso. "Cross Reference" es especialmente importante cuando se requiere la documentación del código fuente.

Information Flow

Este tipo de análisis, también conocido como *Variable Dependency Analysis*, es un estudio de las interdependencias de las variables del código fuente. Este módulo es un instrumento muy importante de cara a la documentación y también un buen detector de fallos.

Data Object Analysis

Este procedimiento permite al usuario filtrar la información del "Static Data Flow Analysis", "Cross Reference" y el análisis del "Information Flow" para crear un informe sobre variables específicas. Esto es esencial para el seguimiento correcto de las variables que se basan en datos y constantes, al mismo tiempo que es útil para realizar pruebas.

MCDC Test Case Planner

Las pruebas MCDC o *Modified condition decision coverage* se generan en el dominio estático y están diseñados para proporcionar a los usuarios una primera medida de las condiciones booleanas dentro del código fuente bajo prueba. Proporciona una información sobre el esfuerzo de la prueba potencial y recursos que pueden necesitar la aplicación. En análisis se basa en verificar que haya al menos una situación de prueba en el que el resultado sea TRUE y al menos otra situación en el que el resultado sea FALSE (*ver tabla 1*).

a	&&	b	&&	(c	||	(d	&&	e))
F		-		-		-		-
T		F		-		-		-
T		T		F		F		-
T		T		F		T		F
T		T		F		T		T
T		T		T		-		-

Tabla 1 – Ejemplo MCDC

2.3.3 Métricas

En el campo de la ingeniería del software una métrica es generalmente un recurso usado para realizar comparativas o para la planificación del proyecto en desarrollo. Por ello engloba cualquier medida o conjunto de medidas destinadas a conocer o estimar el tamaño u otra característica de un software o un sistema de información.

Las métricas son muy importantes para tener conocimiento de características de ingeniería de nuestro sistema. Pueden ayudarnos a saber cuál es la calidad de nuestro producto, cuánto tardaría en concluir una ejecución, cuántos recursos necesitaría, etc. El

problema del software es su nivel de abstracción y la dificultad de monitorizar el progreso de la ejecución, ya que a veces un proceso puede llegar a ser cíclico y no-linear.

Por ello las métricas proporcionan un mecanismo que facilita un contexto de "dónde estamos y dónde vamos a estar en el futuro". Se pretende medir un aspecto concreto del software de manera cuantitativa, proporcionando información administrativa en distintos puntos críticos del código.

El uso de métricas se justifica por tres razones fundamentales:

- **Predicción:** Planificación de tareas y recursos de acuerdo a la escala y la complejidad adecuada.
- **Prevención:** Reducción de riesgos futuros mediante estadísticas de problemas pasados.
- **Corrección:** Evaluación de lo que funciona y lo que no y las mejoras posibles.

Por otra parte, en relación a su función, el tipo de métrica se clasifica en cuatro grupos principales:

- **Métricas estructurales:** Son las que sirven para medir el tamaño y la complejidad del problema y pueden ayudar a predecir tanto el esfuerzo como el riesgo del proyecto.
- **Métricas de gestión de proyectos:** Miden aspectos como recursos usados, costes y en general los riesgos presentes para el propio proyecto.
- **Métricas de la madurez del proceso:** Se fijan en el desgaste y la degradación sufrida por el sistema, a la vez que sugiere las mejoras que se pueden aplicar.
- **Métricas de calidad:** Proporcionan una idea de hasta qué punto el producto cumple con los requisitos de calidad.

2.3.4 Estándares de codificación

Un código fuente completo debe reflejar un estilo armonioso, como si un único programador hubiera escrito todo el software de una sola vez. El establecer un estándar de codificación es muy importante para asegurarse de que todos los desarrolladores del proyecto trabajen de forma coordinada. Esto no solo es importante al comenzar un nuevo proyecto. Cuando se incorpora código fuente previo o cuando se realiza el mantenimiento de un sistema de software creado por anterioridad (*Legacy Code*) facilita mucho la tarea el tener una serie de directrices a seguir.

Usar técnicas de codificación sólidas y llevar a cabo buenas prácticas de programación con vistas a generar un código robusto es de gran importancia para la calidad del software y para obtener un buen rendimiento en su ejecución. Además, si se aplica de forma continuada un estándar de codificación bien definido, se utilizan técnicas de programación apropiadas y posteriormente, se efectúan revisiones del código de rutinas, el proyecto se convierte en un sistema de software fácil de comprender y de mantener.

En el contexto de software embebido, los conjuntos de recomendaciones y directrices marcan las pautas a la hora de proveer portabilidad, seguridad y fiabilidad al código fuente. En ese aspecto el estándar para C/C++ MISRA (*The Motor Intustry Software Reliability Association*) regula el uso de buenas prácticas en sectores como el automovilístico, ferroviario, aeroespacial, telecomunicaciones, equipos médicos y otros.

2.3.5 Calidad del código

Debido al creciente tamaño y complejidad de las aplicaciones de hoy en día las inspecciones manuales de código son cada vez menos viables. En lugar de ello, los desarrolladores tienen que usar herramientas automatizadas para evaluar de forma rápida y efectiva tres puntos fundamentas que definen la calidad que contiene un sistema software:

- **Claridad:** Mide aspectos como hasta qué punto el código puede entenderse de manera fácil y sencilla, si es auto-descriptivo y está bien comentado. También puede medir si hay elementos que se encarguen de una sola tarea o multitud de ellas, si los métodos pueden entenderse en un primer vistazo o si por el contrario son demasiado largos y requieren un tiempo determinado para ser analizados.

- **Mantenibilidad:** Se refiere a la simplicidad estructural del diseño y evalúa la facilidad para modificar el código. Determina si todos los elementos están empaquetados de forma modular, si tiene una jerarquía hereditaria muy profunda, la cantidad de rutas de ejecución independientes que existen en la definición del método (complejidad ciclomática), cuanto código duplicado existe, etc.

- **Testabilidad:** Es la cualidad que mide propiedades como que el código fuente se aproveche de manera óptima, qué cantidad de la aplicación pueden llegar a ejecutarse, los tipos de prueba posibles (unitarias, integración, escenario, etc.) y la calidad de los casos de prueba.

Todas estas características son fundamentales en la valoración global de la calidad del código, y el proceso para la determinación de las mismas requiere herramientas avanzadas como las de LDRA.

2.4 Análisis dinámico

El análisis dinámico de software, a diferencia del método estático, es un tipo de análisis del sistema que supone la ejecución del programa y observar su comportamiento. Para que el análisis dinámico resulte efectivo el programa que se quiera analizar se debe ejecutar con los suficientes casos de prueba como para producir un comportamiento interesante.

En este proyecto, el análisis dinámico del código se ha utilizado en el periodo de formación de la las distintas herramientas, haciendo uso del proyecto de la caja registradora de LDRA. Sin embargo, no se ha implementado en la etapa de verificación, ya que el código fuente utilizado no estaba preparado para ello.

2.5 Opciones adicionales de análisis en LDRA

Más allá de las opciones de análisis estáticos disponibles, los entornos de LDRA proporcionan la opción de llevar a cabo otro tipo de operaciones que complementan los anteriores. Si bien no se han ejecutado en la herramienta de visualización de informes TBvision (aun disponiendo de esa opción, *ver ilustración 24*), algunas de estas tareas se han efectuado posteriormente en la etapa de prueba unitaria con TBrun de manera automática. En este apartado se procede a explicar dichos métodos analíticos.

- **Dynamic Coverage Analysis:** Mide cuál eficaz han sido las ejecuciones de test sobre el código instrumentado. El "Dynamic Coverage Analysis" monitorea el historial de ejecución generado en *run-time* para producir una variedad de resultados y genera nuevos informes en modo gráfico y textual.

- **Dynamic Dataset Analysis:** Se utiliza para descubrir qué conjuntos de datos han ejecutado una línea específica, o un grupo de líneas. Esta información es útil para las pruebas de regresión después de que el código fuente haya sido alterado. En esta situación "Dynamic Dataset Analysis" permite al usuario determinar fácilmente qué conjuntos de datos han sido alterados para volver a ejecutar el código correspondiente al área modificada.

- **Profile Analysis:** Este análisis identifica los conjuntos de datos de prueba redundantes. Por ejemplo, durante el curso de un "Dynamic Coverage Analysis" el procedimiento consiste en aplicar conjuntos de test sobre datos. En este caso el usuario se ve obligado a generar gran cantidad de información de test con el fin de alcanzar una parte del código no ejecutada previamente. A menudo, algunos de los datos de prueba utilizados al principio del proceso se vuelven redundantes en el sentido de que los estados, ramas y *LCSAJs* que están cubiertos por estos elementos también se cubren por los posteriores casos de test. Por lo tanto, si las ejecuciones de pruebas se repiten utilizando datos de prueba no redundantes, los valores de las métricas de cobertura pueden llegar a tener que modificarse. El objetivo de "Profile Analysis" es detectar estos conjuntos de datos de prueba redundantes e informar al usuario.

2.6 Instrumentación

La instrumentación de un módulo permite el aislamiento del programa en un entorno simulado. El objetivo de este proceso es que la unidad que se quiera validar sea completamente autónoma y no tenga que realizar llamadas a otros módulos. En este apartado se desarrollará brevemente cómo se ha llevado a cabo esta tarea.

2.6.1 Opción de instrumentación en LDRA

Generate Instrumented Program(s)

La Instrumentación en LDRA copia los archivo de origen en la clase a validar y añade un prefijo **inszt_** al nombre de archivo original por defecto. Por ejemplo, en el caso de este proyecto el nombre generado ha sido **inszt_multitractcontrolcore.cpp** (*ver ilustración 3*). A continuación, *LDRA Testbed* añade estructuras de control en puntos estratégicos que monitorizan el flujo de control en tiempo de ejecución. Después de ser compilado este archivo instrumentado se ejecuta durante la fase de prueba unitaria en modo caja blanca (*ver apartado 5*).

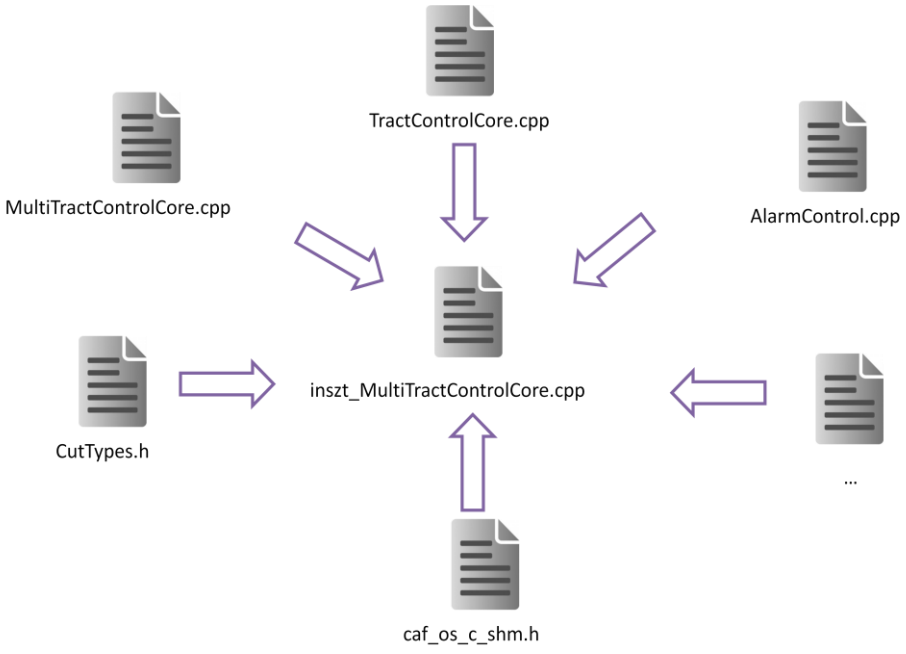


Ilustración 3 – Instrumentación de MultiTractControlCore

LDRA inserta estructuras de datos para añadir puntos estratégicos en el código recién creado como:

- Break Points
- Saltos de flujo de control
- Etiquetas del programa
- El comienzo y el final de los procedimientos

Estas pruebas de ejecución son simples llamadas a funciones que se ejecutan principalmente en tres pasos:

- Crear y abrir el fichero de historial de ejecución
- Escribir información sobre el programa en ejecución a través del flujo de salida
- Cerrar el fichero

Estas llamadas son representaciones numéricas, los cuales controlan el flujo de saltos que han sido establecidas por los datos de prueba. El proceso es similar, en principio, al proceso que realiza un programador al insertar escrituras (*print*) para conocer el estado de una sección en particular. Este historial de ejecución se analiza a continuación por el módulo de análisis de cobertura dinámica para producir la información de cobertura.

2.7 Prueba Unitaria

Una prueba unitaria sirve para comprobar el correcto funcionamiento de un módulo de código que en un principio estaría integrado dentro de un sistema mayor. En este capítulo se profundiza en los diferentes métodos de aplicación de las pruebas, así como el proceso de creación de casos de test.

2.7.1 Objetivo de la prueba unitaria

La finalidad de este tipo de prueba sería asegurarse de que cada uno de los módulos funcione correctamente por separado y posteriormente, con las pruebas de integración, verificar el correcto desempeño del sistema o subsistema en cuestión.

2.7.2 Proceso de creación de test

El proceso de creación del test consiste en tres puntos fundamentales: la definición de los procedimientos que se quieran validar, designación de los casos de test y la identificación de las condiciones.

Por lo tanto, a la hora de establecer las condiciones de los test nos centraremos en qué es lo que queremos testear y qué aspectos del sistema son relevantes. Existe un amplio abanico de condiciones de test válidos para cada requisito de la especificación. Un simple requisito puede producir múltiples condiciones que deben de ser verificadas. Al mismo tiemplo, una condición puede ser definida por dos o más requisitos.

Un caso de test no deja de ser una especificación para un escenario operativo que describe el qué se quiere verificar y el cómo se quiere ejecutar. Por lo tanto se espera tener una cantidad de entradas y salidas conocidas, antes de que se realice el propio test.

2.7.3 Black Box Testing

El test de caja negra (*ver ilustración 4*) traza pruebas sin el conocimiento exacto de cómo está implementado el código. Un sistema formado por módulos que cumplen las características de caja negra será más fácil de entender ya que permitirá dar una visión más clara del conjunto. El sistema también será más robusto y fácil de mantener. En caso de ocurrir un fallo, éste podrá ser aislado y abordado de manera más sencilla.

La unidad a la que se le aplica el test se simula a través de las interfaces de la misma. Se pueden diseñar pruebas que demuestren que dicha función están bien realizada para la cual es necesario implementar una descripción completa de las interfaces y el comportamiento que se espera que tenga la unidad. Dichas pruebas son llevadas a cabo sobre la interfaz del software, es decir, sobre de la función actuando en ella como una caja negra, proporcionando unas entradas y estudiando las salidas para ver si concuerdan con los datos esperados.

Ilustración 4 - Black Box Testing

2.7.4 White Box Testing

Las pruebas de caja blanca (también conocidas como pruebas de caja de cristal o pruebas estructurales) se centran en los detalles procedimentales del software, por lo que a diferencia de las pruebas de caja negra, su diseño está fuertemente ligado al código fuente. Se trata de escoger distintos valores de entrada para examinar cada uno de los posibles flujos de ejecución del programa y cerciorarse de que se devuelven los valores de salida adecuados.

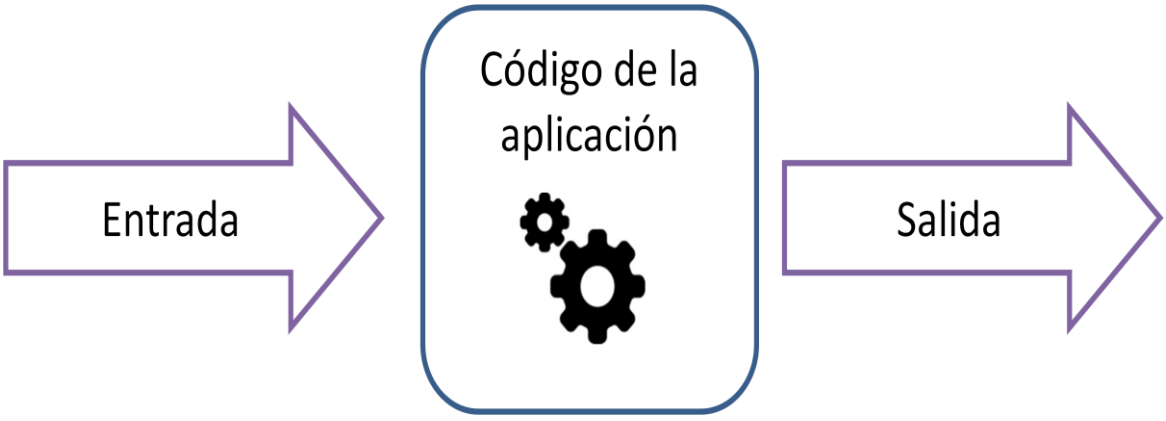


Ilustración 5 - White Box Testing

Una característica de las pruebas de caja blanca es que están basadas en una implementación concreta del programa. Por ello, si el código se modifica también será necesario que las pruebas se tengan que rediseñar.

Aunque este tipo de test es aplicable a varios niveles (unidad, integración y sistema), habitualmente se aplican a las unidades de software. Su cometido es comprobar los flujos de ejecución dentro de cada unidad (función, clase, módulo, etc.) pero también pueden testear los flujos entre unidades durante la integración, e incluso entre subsistemas, durante las pruebas de sistema.

Existen cuatro técnicas principales de diseño de pruebas de caja blanca:

- **Pruebas de flujo de control:** Son pruebas que buscan los errores en la parte lógica del programa. Utilizando las condiciones simples (operador-relación) o con condiciones complejas (N x [operador-relación]). Por ejemplo sirve para encontrar errores en variables, en los paréntesis y hasta en operaciones aritméticas.
- **Pruebas de flujo de datos:** Es la técnica de diseño de pruebas en la que los casos de prueba se diseñan para probar las variables y definiciones en el programa.
- **Pruebas de bifurcación (branch testing):** Son pruebas para bucles o ligados a una bifurcación con la que se definen qué iteraciones están correctamente implementadas. Se verifica si las líneas de código que contengan alguna condición, deberían tenerla y por consiguiente tuvieran que ser modificadas.
- **Pruebas de caminos básicos:** Esta prueba se asegura de que cada sentencia del código sea ejecutada al menos una vez. En ella se hace uso de técnicas como los grafos, diagramas de flujo, complejidad ciclomática, etc.

2.7.4.1 Pruebas de cobertura

Para saber hasta qué punto se tiene que seguir aplicando el proceso de test de caja blanca, hace falta conocer con minuciosidad en qué punto se encuentran las pruebas que se hayan realizado hasta ese momento. El haber seguido un procedimiento sistemático no es garantía de que el programa esté libre de fallos y es imprescindible recurrir a una técnica que facilite una indicación de los test adicionales necesarios. Esta tarea se basa en analizar qué pruebas complementarias pueden ser efectivas y a su vez no redundantes.

Las pruebas de cobertura se basan en el análisis de la estructura del código. Sin embargo no están orientadas principalmente a la validación del programa, sino que se aseguran de la correcta ejecución del 100% del sistema. Si se dejara una parte del programa sin cubrir, se correría el riesgo de que esas líneas contuvieran fallos que no se conocieran hasta la puesta en marcha del sistema completo.

En esencia, las pruebas de cobertura se dividen en diferentes procesos en las que se analizan las sentencias ejecutadas, las posibles decisiones tomadas y las condiciones factibles que pueda llevar a ejecutar el programa.

La **cobertura de sentencias** (*ver ecuación 1*) o *Statement Coverage* es el proceso de covertura más simple de todos y se encarga de verificar que todas las líneas de código se han ejecutado al menos una vez.

$$\text{Cobertura de sentencias} = \frac{Número\ de\ sentencias\ ejecutadas\ al\ menos\ una\ vez}{Número\ de\ sentencias\ ejecutables}$$

Ecuación 1 - Cobertura de sentencias

La **cobertura de decisión** o *Branch Coverage* se centra en el análisis de la posible toma de decisiones del código durante la ejecución. Esta prueba se asegura de que cada estructura de control, como pueden ser los *IF*, *ELSE* o *WHILE* se haya ejecutado en todas sus variantes condicionales.

Por último, la **cobertura de condición** comprueba que toda condición lógica existente tenga repercusión en el resultado del sistema. Para ello cada condición se ejecuta de manera atómica en todas sus combinaciones lógicas posibles.

La cobertura de decisión de condición variable o *MCDC* se fundamenta en que a cada situación se le apliquen una serie de test, que realicen ejecuciones combinadas para asegurarse de que se hayan verificado todos los valores Booleanos de una condición atómica. Al mismo tiempo que se les aplica el mismo valor a todos las demás condiciones atómicas y se comprueban todas las condiciones Booleanas del resultado.

En la *ilustración 6* podemos visualizar un grafo de flujo que contempla las coberturas de sentencia y de decisión. Se consideraría una cobertura de sentencia completa si todos los nodos, tanto en forma de rombo (nodos condicionales) como en forma redonda (resto del código) consiguieran ser verificadas (color verde) al 100%. La representación del flujo en forma de flecha correspondería a la cobertura de decisión que también debería de cumplirse en su totalidad.

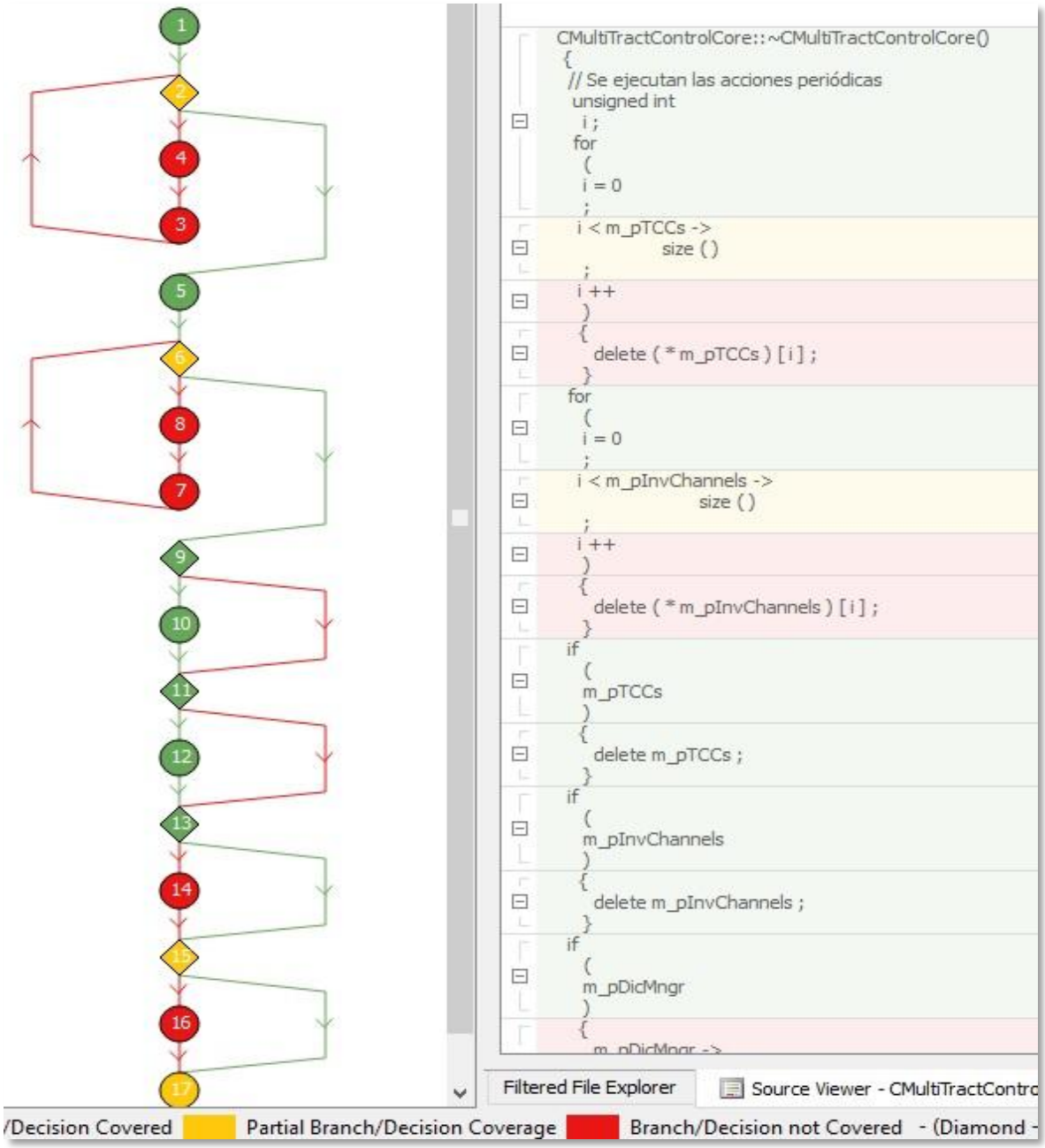


Ilustración 6 - Análisis de cobertura: Grafo de flujo en TBrun

2.8 Conceptos básicos para la interpretación de resultados post análisis

2.8.1 Bloques básicos

Un bloque básico es una secuencia de una o varias secuencias ejecutables consecutivas en un código fuente que no contengan:

- Un punto de entrada
- Un punto de salida
- Bifurcaciones internas (ramas)

2.8.2 Métricas de claridad

La claridad del software contempla medidas que calculan el nivel de entendimiento del programa y la sencillez con el que se define (*ver capítulo 5.2*).

Una de las características que se miden con las métricas de claridad son la cantidad de bloques básicos en el código. Estas se calculan en base a cuántas veces se llega a ejecutar cada línea en el conjunto de la función. Por supuesto, se puede calcular esta cuantía en relación al tamaño del propio procedimiento. LDRA establece un campo para medir esto mediante el "Average Length of Basic Blocks".

Por otro lado, la profundidad de anidamiento o el "Depth Loop Nesting" controla que no haya bucles que contengan otras iteraciones dentro de sí. En el caso de este proyecto, esta medida no ha sido especialmente problemático ya que los bucles con el anidamiento más profundo han sido de nivel 1.

Asimismo las líneas ejecutables (*Executable Lines*) juegan un papel muy importante en las métricas de claridad. Por una parte se mide la cantidad de comentarios del software en relación a este tipo de líneas, sabiendo que cuanta mayor sea esta cuantía en relación al código funcional más se entenderá el contenido del programa. Por otra parte LDRA permite también medir el número total de comentarios en las declaraciones del código.

Por último y al margen de esa relación de comentarios por líneas ejecutables, existe una medida respecto al total de notas escritas en el encabezado. Una descripción de lo que

hace el propio procedimiento es esencial para el entendimiento y la claridad de la clase *multitractcontrolcore*. Sin embargo no conviene que esta descripción sea muy extensa ya que puede llegar a abarcar una parte demasiado grande del código.

2.8.3 Métricas de mantenibilidad

LDRA se centra principalmente en dos características en cuanto a la mantenibilidad del código. Por una parte la complejidad ciclomática y por otro el numero de nudos.

- **Complejidad ciclomática:** La complejidad ciclomática (*ver ilustración 7*) de una función es una medida de la complejidad de un grafo, en base a su estructura de nodos y ramas descrita en la *Ecuación 2*. En cuanto a la complejidad ciclomática para el conjunto de un archivo se calcula restando "1" del valor de cada función, ejecutando la suma de estos valores y después añadiendo "1" al total. Esto elimina los efectos de las funciones, de modo que un mismo código dividido en tres funciones tiene la misma complejidad total, como si estuviera en un solo procedimiento. Esto hace que la complejidad ciclomática para todo el archivo sea independiente del número de funciones y sólo dependa de la propia complejidad del código.

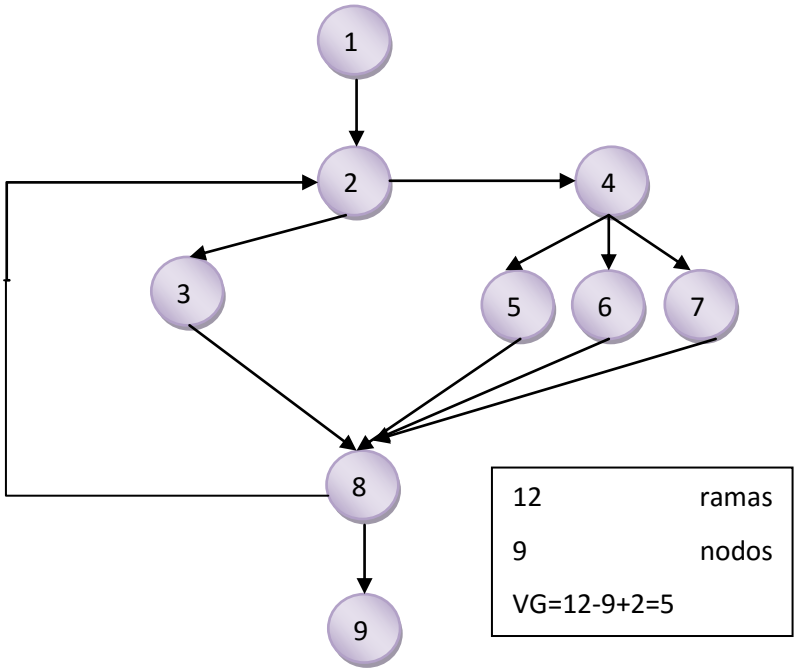


Ilustración 7 – Ejemplo de la complejidad ciclomática de un grafo

$$V(G) = Nº.\ Ramas - Nº.\ Nodos + 2$$

Ecuación 2 – Formula de la complejidad ciclomática

Nudos

La medida del nudo (*ver ilustración 8*) es muy sensible a la manera en el que se ordena el programa una vez es construido, es decir, la alineación del grafo que forma la estructura del código. Como tal, es un factor en la complejidad del programa. La gran cuantía de nudos facilita en elevado número de incoherencias en el código y por lo tanto la cantidad de "saltos" obligados a realizar al lector del código. Una cantidad de nudos excesiva puede significar que un programa deba ser reordenado para mejorar su legibilidad y reducir así su complejidad. Esto significa que mientras que los bloques básicos se pueden poner en una orden casi arbitraria, algunas formas de estructurar llegan a ser mejores que otras y esto se refleja en el análisis del nudo que en el caso de *multitractcontrolcore*, llega a haber algunos elementos con que casi llegan a alcanzar los 500 nudos. Sin duda es un punto que contempla un gran margen de mejora.

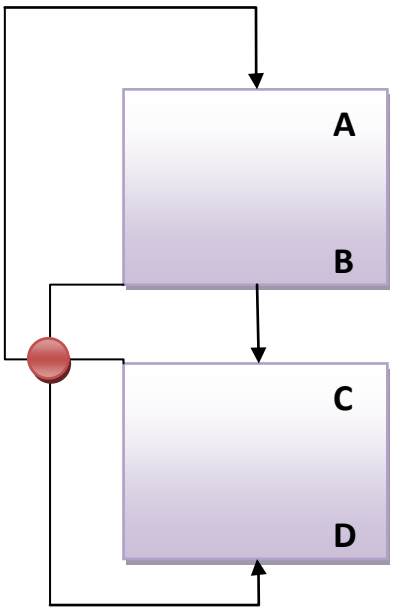


Ilustración 8 - Representación de un nudo

2.8.4 Métricas de testabilidad

Este tipo de métrica combina algunas cualidades que se exponen en los *apartados 2.8.2* y *2.8.3* y añade otras adicionales. Por ejemplo, a la hora de examinar el nivel de modularización se contempla el llevar una cuenta de las entradas a los procedimientos o *FAN IN.* Esta cualidad hace hincapié en el número de llamadas a otras funciones (ya sean llamadas normales o saltos de procedimiento en procedimiento). Por el contrario también se mide en número de salidas de los métodos o *FAN OUT*. En este caso se cuantifican funciones como *Returns*, *Exits* o *Gotos*.

Por último se calcula la cantidad de puntos de salida de los procedimientos y el número de bucles que hay en cada una.

2.8.5 Grafo de llamadas

Este grafo representa las características de los procedimientos como bloques con arcos dibujando llamadas entre ellos sin el código fuente bajo test. Un ejemplo de un grafo de llamadas se podría observar en la *ilustración 9*.

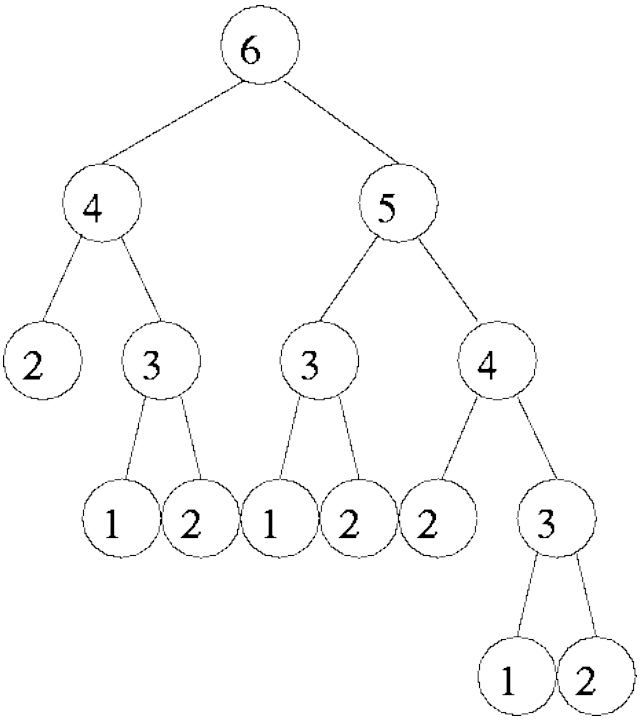


Ilustración 9 - Ejemplo de un grafo de llamadas recursivas

2.8.7 Grafo de flujo

El análisis de complejidad calcula las interconexiones entre los bloques básicos y produce una representación estática o *flowgraph* del programa (*ver ilustración 10*). En el grafo estático, los nodos son los elementos básicos y los arcos son las ramas.

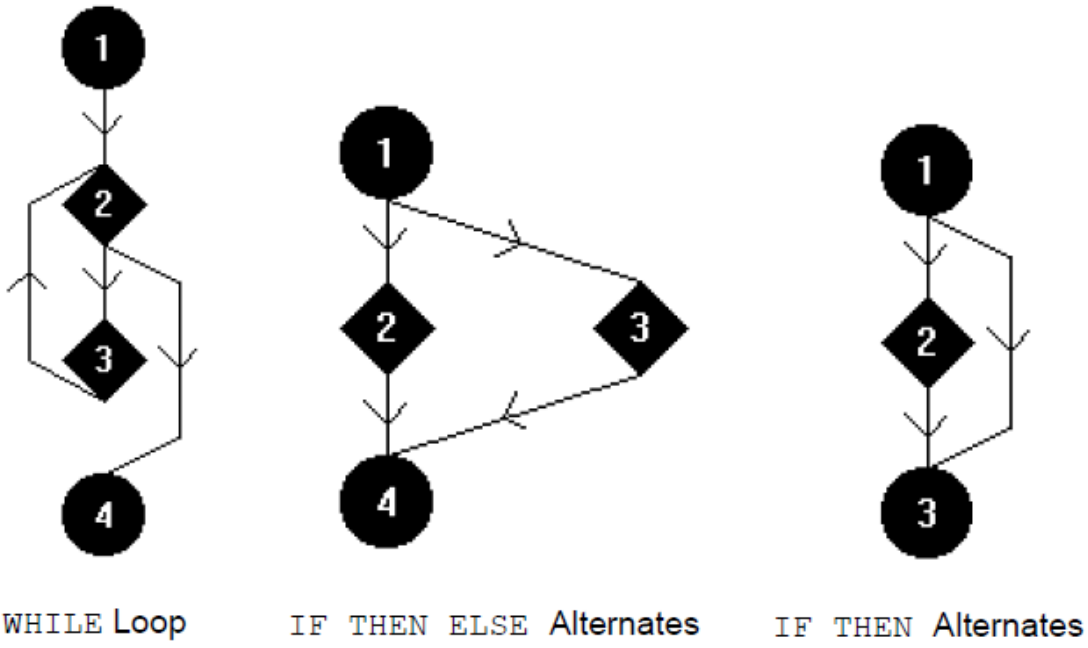


Ilustración 10 – Ejemplos básicos de grafos de flujo

Los grafos de flujo en LDRA se dibujan para cada procedimiento:

- Los bloques vinculados (nodos) se dibujan en secuencia hacia abajo y se conectan por arcos que a su vez representan las ramas.
- Los bucles tienen sus saltos hacia atrás representados hacia el lado izquierdo de los bloques.
- Los saltos hacia delante se representan en el lado derecho
- El algoritmo hace que las líneas no se crucen en ningún momento

2.9 Desarrollo en V

En el contexto de un proceso de evaluación de software se definen varios procedimientos en las etapas tempranas del ciclo de vida del proyecto. En él se resumen las principales medidas que deben adoptarse en relación con las prestaciones correspondientes en el marco del sistema informático de validación.

Se describen las actividades y resultados que deben producirse durante el desarrollo del producto. El lado izquierdo de la V representa la descomposición de las necesidades, y la creación de las especificaciones del sistema. El lado derecho de la V representa la integración de las piezas y su verificación.

Entre los objetivos del desarrollo en V se encuentra la minimización de los riesgos del proyecto, mejorando la transparencia y el control del desarrollo. También se busca especificar los enfoques estandarizados describiendo los resultados correspondientes y funciones de responsabilidad. Es importante su aplicación de cara a la detección temprana de las desviaciones y los riesgos, así como para mejorar la gestión de procesos, reduciendo de esta manera los riesgos del proyecto.

Otra finalidad de este desarrollo es la de la mejora y garantía de calidad. Como un modelo de proceso estándar, asegura que los resultados que se proporcionan sean completos y contengan la calidad deseada. Los resultados provisionales definidos se pueden comprobar en una fase temprana. La uniformidad en el contenido del producto mejora la legibilidad, comprensibilidad y verificabilidad.

Por otra parte se facilita la reducción de los gastos totales durante todo el proyecto y el ciclo de vida del desarrollo del sistema. El esfuerzo para la progresión, producción, operación y mantenimiento de un sistema puede ser calculado, estimado y controlado de forma transparente mediante la aplicación de un modelo de procesos estandarizados reduciendo la dependencia en los proveedores y el esfuerzo para las siguientes actividades y proyectos.

Por último se mejora la comunicación entre todos los participantes. La descripción estandarizada y uniforme de todos los elementos pertinentes y términos es la base para la comprensión mutua entre todos los participantes.

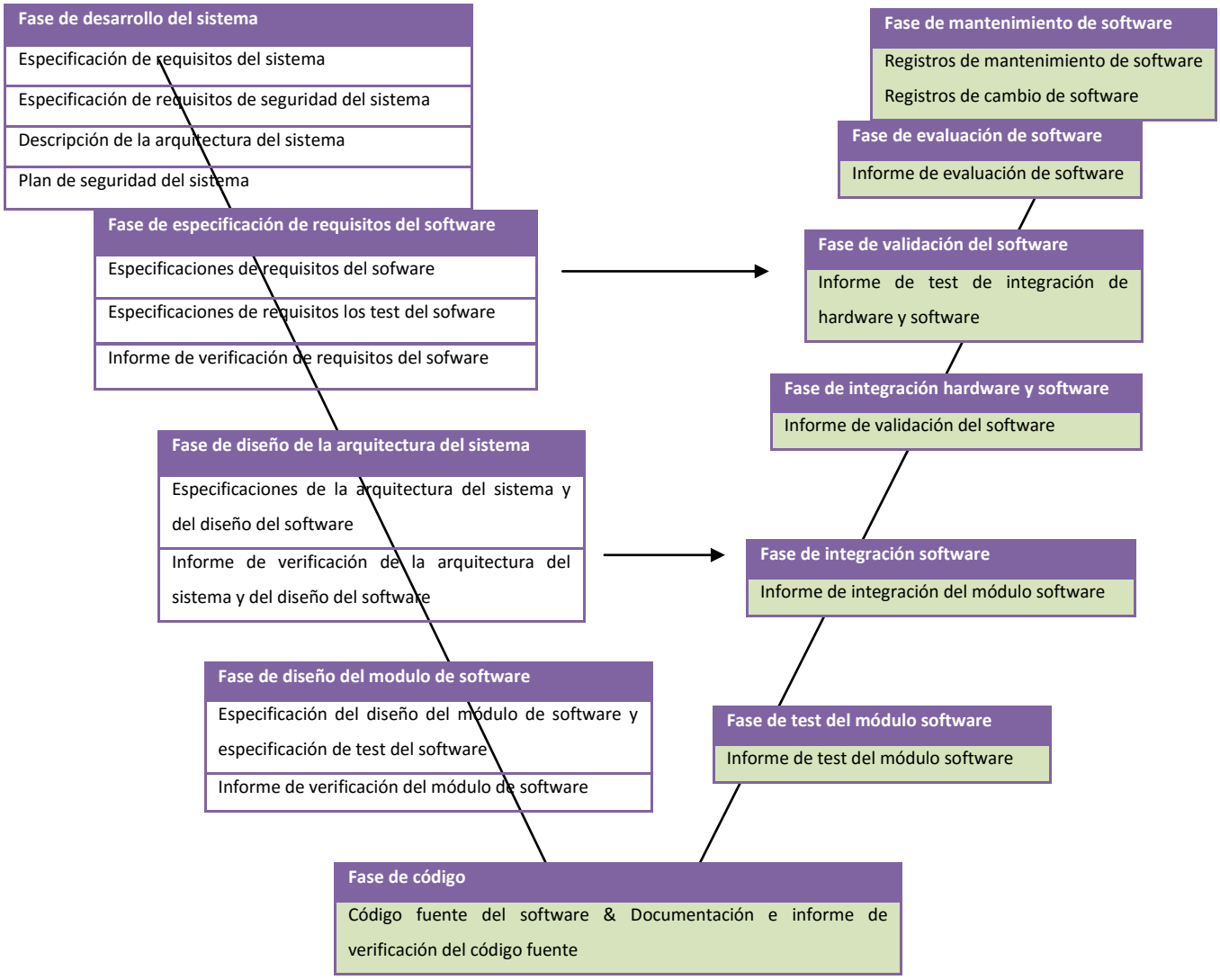


Ilustración 11 - Desarrollo en V

Dicho esto, en el proceso de validación y verificación de este proyecto no se ha llegado a aplicar el desarrollo en V en su totalidad, ya que se partía de un *Legacy Code* que ya estaba en la etapa final de su implementación. Este ha sido uno de los puntos críticos que han estado presentes en las sesiones de comunicación con el personal de CAF Power & Automation. De

esta manera se ha querido hacer llegar un feedback recomendando la futura aplicación del proceso de validación y verificación desde el comienzo del ciclo de vida del diseño.

Dicho esto y debido a necesidad de haber tenido que trabajar con este *Legacy Code*, se ha dedicado parte del desarrollo en aplicar la vertiente derecha del desarrollo en V (*ver ilustración 11*) a un coste importante tanto de esfuerzo como de tiempo.

2.10 Concepto de STUB

En el desarrollo de un proceso de V&V, un STUB representa un programa que simula el comportamiento de un componente de software o un módulo a que se le aplica un proceso de test.

Los test STUB son usados en principalmente en el testing incremental en la forma top-down. Estos programas temporales sustituyen un módulo proporcionando el mismo dato de salida que el modelo de software original.

3. DESCRIPCIÓN DEL DISPOSITIVO

3.1 Sistema VEGA

VEGA es un subsistema dentro del sistema electrónico de control de un equipo ferroviario embarcado que contempla la electrónica de control y el SW de bajo nivel a utilizar de forma genérica (*ver ilustración 12*).

Para gobernar un sistema electrónico de control ferroviario se necesitan estos equipos electrónicos de control sobre los cuales se instalan y ejecutan las aplicaciones específicas. Este sistema contempla principalmente 5 funcionalidades:

- Ejecución de la aplicación de control, con implicaciones en la seguridad.
- Ejecución de la aplicación de monitorización.
- Actualización remota de los dispositivos programables del propio sistema VEGA.
- Mapeo de los datos de proceso.
- Monitorización y diagnosis del estado del interno del sistema.

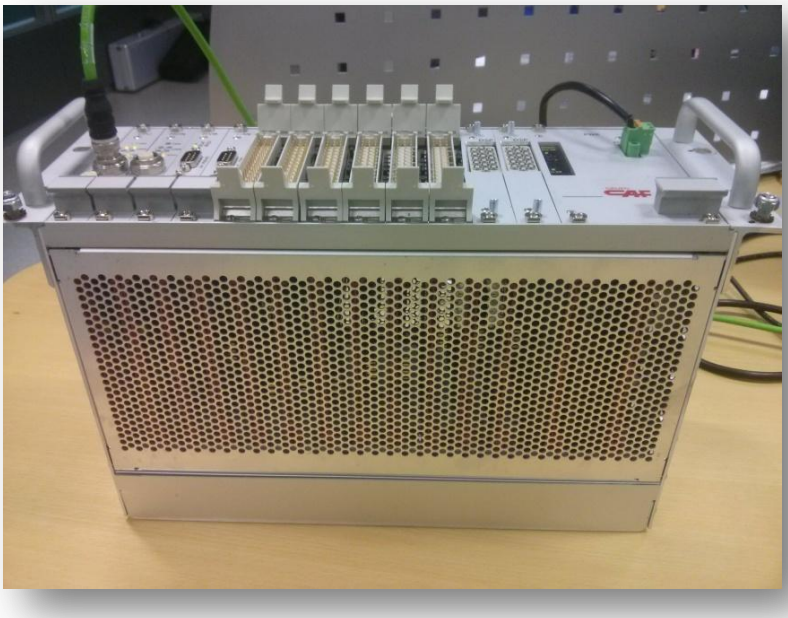


Ilustración 12 - Rack VEGA

El funcionamiento de este hardware se basa en adquirir entradas procedentes de diversos sensores y establecer salidas una vez ejecutadas series de instrucciones, actuando así sobre el sistema bajo control en base a una lógica definida por cada aplicación. Adicionalmente la ejecución requiere del intercambio de información con otros equipos del tren, así como de la lectura y escritura de archivos.

3.1.1 Arquitectura

El sistema se compone de varios subsistemas con funciones específicas que se conectan mediante un *backplane*. Los requisitos (tanto funcionales como los no funcionales) del sistema que se definen en los documentos de referencia se cumplen de manera distribuida en el subsistema.

Primeramente el hardware se compone de una unidad de procesamiento llamado sCPUT (*ver ilustración 13*) que ofrece la posibilidad de ejecutar distintas aplicaciones software. Se trata de la unidad de procesamiento principal del sistema para aplicaciones de tracción y TCMS. Lleva a cabo la tarea de ejecutar la aplicación del uP principal. También se encarga de gestionar todas las comunicaciones entre todos los subsistemas, tanto en el bus del CPU como en el procesador de señal digital. Es el subsistema en el que se ha validado y verificado el software genérico AX00 en este proyecto.

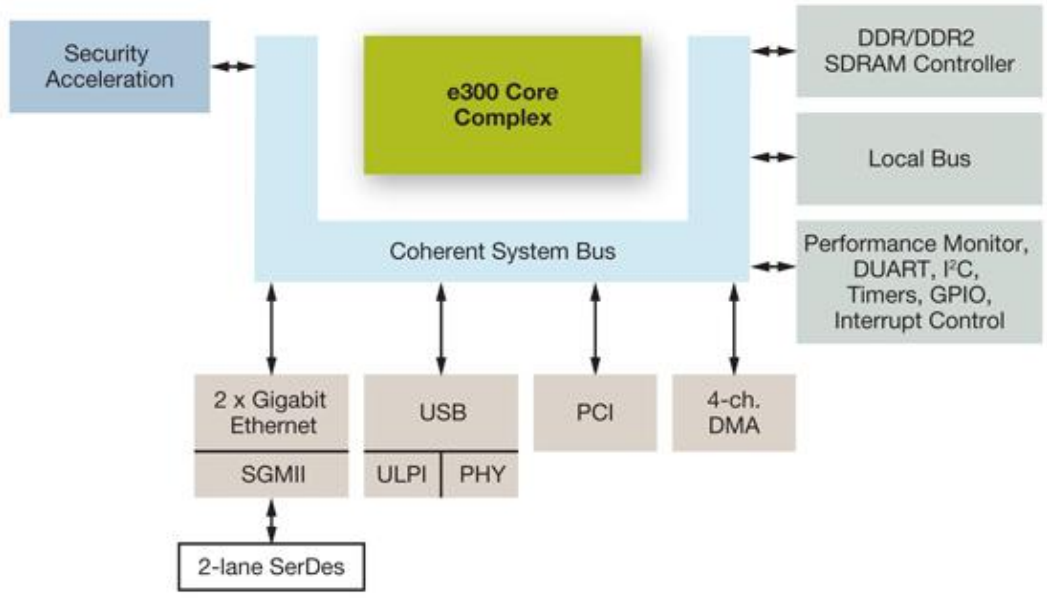


Ilustración 13 - Diagrama de bloques del MPC8313

Adicionalmente VEGA contiene varios periféricos de entrada y salida que necesite una aplicación para las distintas configuraciones de tracción, señalización y TCMS. Estos periféricos se definen y configuran en cada caso para la cual el sistema está constituido por distintos subsistemas de E/S. De esta manera se consigue una estructura modular y configurable con capacidad de adaptarse a las necesidades de cada aplicación.

3.1.2 Descripción detallada de la CPUT

El subsistema CPUT es una tarjeta electrónica basada en el procesador *MPC8313 de Freescale*. Es este procesador el encargado de ejecutar la aplicación del uP principal del sistema VEGA.

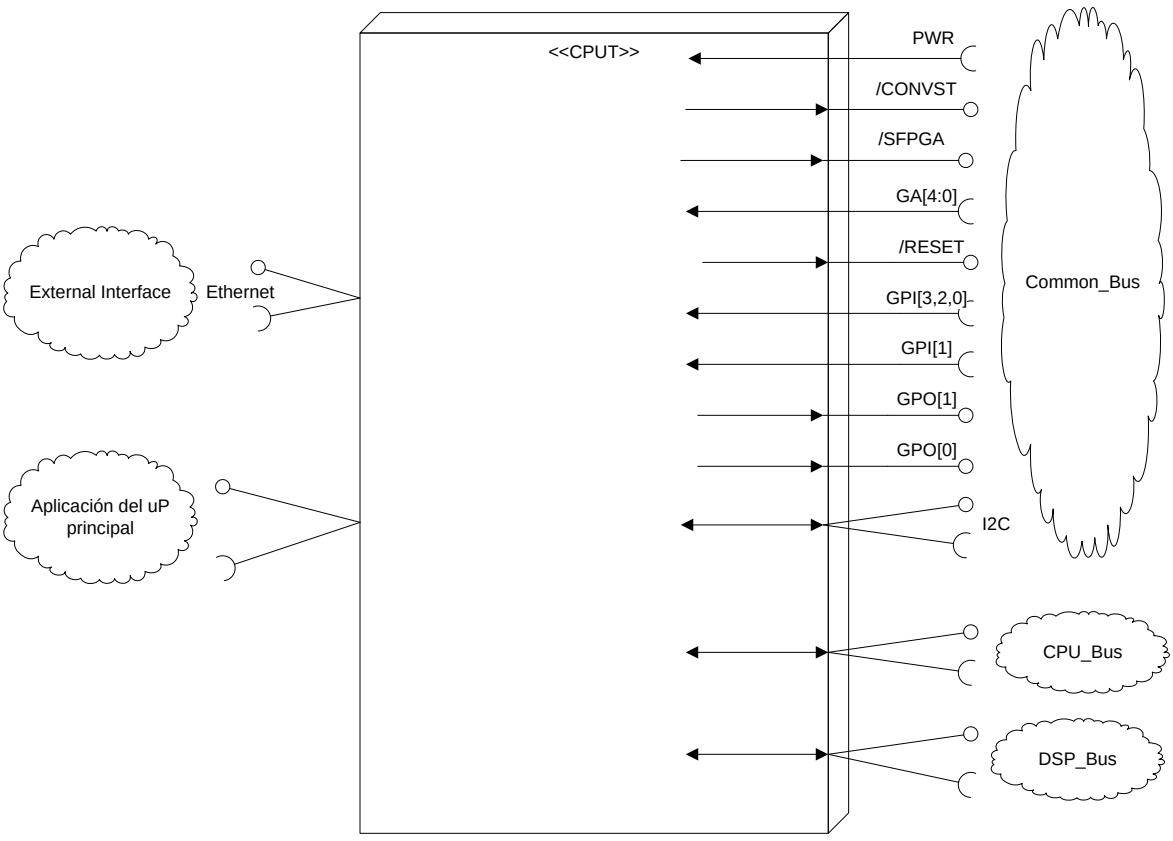


Ilustración 14 - Esquema CPUT

Las funciones principales de este subsistema son las siguientes:

- **Ejecución de la aplicación del uP principal:** Para que la aplicación del uP principal se pueda ejecutar en el sCPUT, éste debe ofrecer un soporte hardware compuesto por un procesador, memorias e interfaces tanto internos como externos.
- **Comunicaciones entre subsistemas:** La arquitectura del sistema se basa en dos buses independientes para la comunicación entre los distintos subsistemas. El sCPUT es el maestro en ambos buses y se encarga de la gestión de los mismos
- **Sincronización entre subsistemas:** El sCPUT se encarga de sincronizar distintos subsistemas.
- **Activación señal /RESET:** El sCPUT debe activar la señal /RESET en caso de que sea necesario pasar todo el sistema a un estado seguro.
- **Propagación GPI→GPO:** Para posibilitar la comunicación entre el sDI que lleva a cabo la inhibición de salidas de fibra óptica y los sDSP, el sCPUT debe propagar hacia la salida GPO la entrada GPI.
- **Alimentación:** El sCPUT utiliza la tensión de alimentación que se distribuye a través del Common_Bus para alimentar toda su circuitería electrónica.
- **Interfaz Ethernet:** El sCPUT debe ofrecer una interfaz externa, basada en un bus Ethernet, para la comunicación con el usuario para labores de mantenimiento, y con otros sistemas. La configuración de los distintos dispositivos programables se realiza a través de esta interfaz.
- **Funciones del SW de bajo nivel:** El sCPUT debe ofrecer cierta funcionalidad de SW de bajo nivel, que se encarga de comprobar la integridad del propio subsistema durante la fase de arranque, inicialización de periféricos, establecer un contexto de ejecución para código escrito en lenguaje de programación C/C++, gestionar el tiempo de CPU y la memoria asignada a los subprocesos de la aplicación del uP principal y hacer de interfaz con el hardware para dicha aplicación.
- **Fecha y hora:** El sCPUT debe mantener la fecha y hora definidas por la aplicación del uP principal.

4. HERRAMIENTAS DE DESARROLLO

4.1 Juego de herramientas de LDRA

LDRA se basa en un banco de pruebas llamado *LDRA Testbed* que ofrece un motor de análisis estáticos y dinámicos. Partiendo de este núcleo, proporciona varias herramientas en función de las tareas específicas que se quieran realizar. Por ejemplo, dispone del entorno TBvision para la visualización del cumplimiento de estándares, métricas de la calidad y cobertura de código. Por otra parte ofrece el entorno TBrun para la realización de test instrumentados en unidades Host y Target. Por último dispone de la posibilidad de hacer uso de otro tipo de herramientas como TBManager para la verificación de requisitos automatizados y la administración de objetivos. El uso de este último entorno ha quedado fuera de este proyecto.

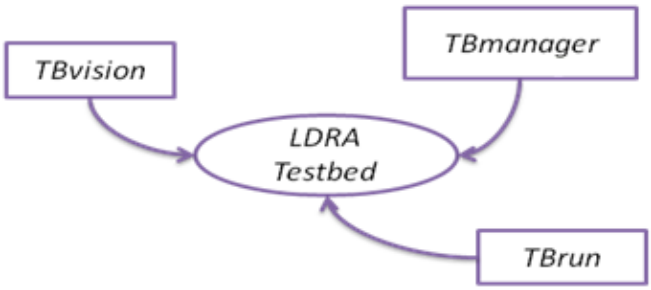


Ilustración 15 - Esquema de herramientas de LDRA

4.1.1 TBvision

TBvision es una herramienta de LDRA para generar informes de calidad del código (*ver ilustración 16*). Proporciona de forma fácil y rápida la capacidad de ver los resultados mediante gráficos de llamada, gráficos de flujo, informes de revisión de código, informes de revisión de calidad, informes de cobertura del código y varias opciones adicionales que resumen los análisis realizados. Estos informes permiten a los usuarios evaluar rápidamente aspectos como la portabilidad, fiabilidad, capacidad de prueba, mantenimiento, la complejidad y el estilo de código generado por los equipos de proyecto.

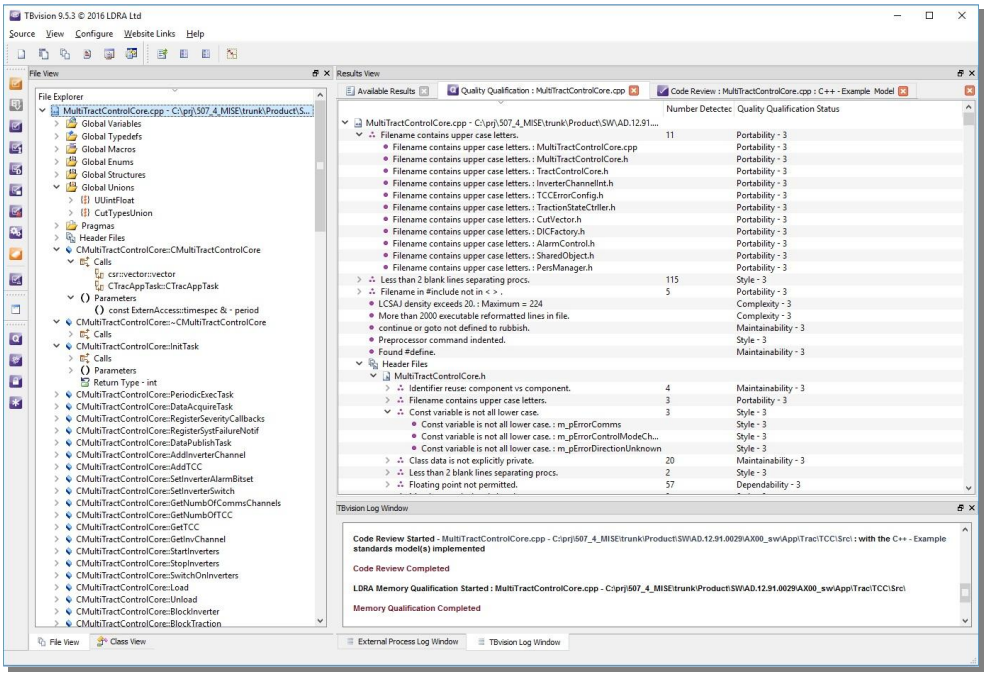


Ilustración 16 – Interfaz de TBvision

4.1.2 TBrun

TBrun (*ver ilustración 17*) es el generador de Test instrumentados de LDRA. Facilita la ejecución de procedimientos de código fuente o funciones (unidades) con los datos de prueba. Esta herramienta permite a los usuarios especificar las entradas a las unidades bajo prueba y aceptar o rechazar los resultados generados al ejecutarlos con los datos de prueba. La cobertura de código para dicho test también se puede analizar en este entorno.

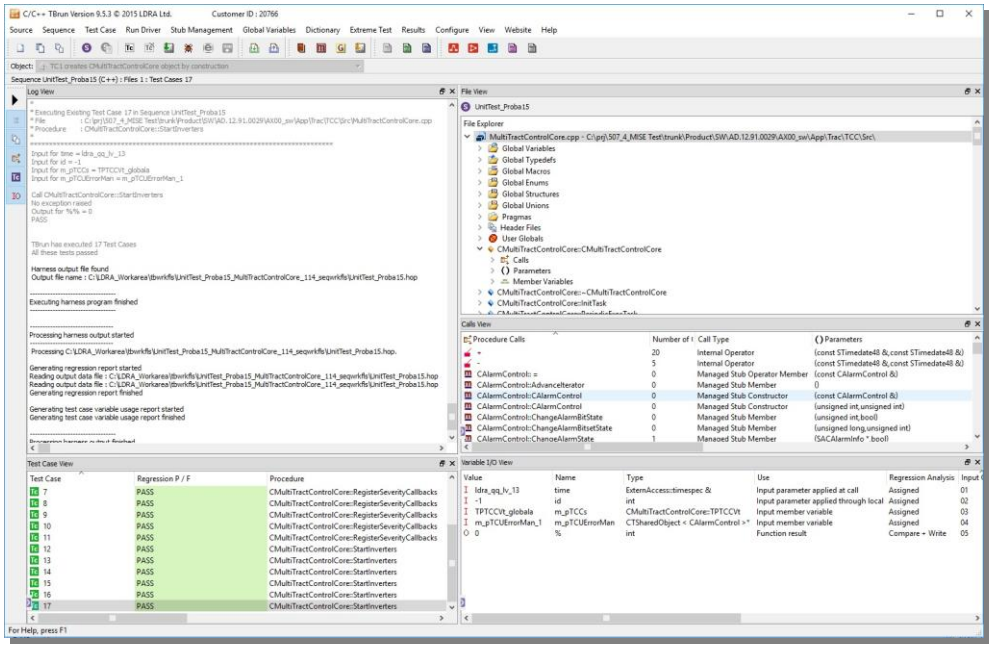


Ilustración 17 - TBrun

4.1.3 Microsoft Visual C++ 2010 Express

Entorno de desarrollo integrado Visual C++ (*ver ilustración 18*) está diseñado para el desarrollo de aplicaciones hechas en C, C++ y C++/CLI en el entorno Windows, habiéndose utilizado su versión Express para este proyecto. Incluye bibliotecas de Windows (WinApi), MFC y el entorno de desarrollo para .NET Framework. Esta IDE cuenta con un compilador propio y otras herramientas como IntelliSense, TeamFoundation Server, Debug, etc. Además provee de bibliotecas propias de cada versión del sistema operativo y sockets. Como otros compiladores, se le pueden añadir nuevas bibliotecas como DirectX, wxWidgets o SDL.

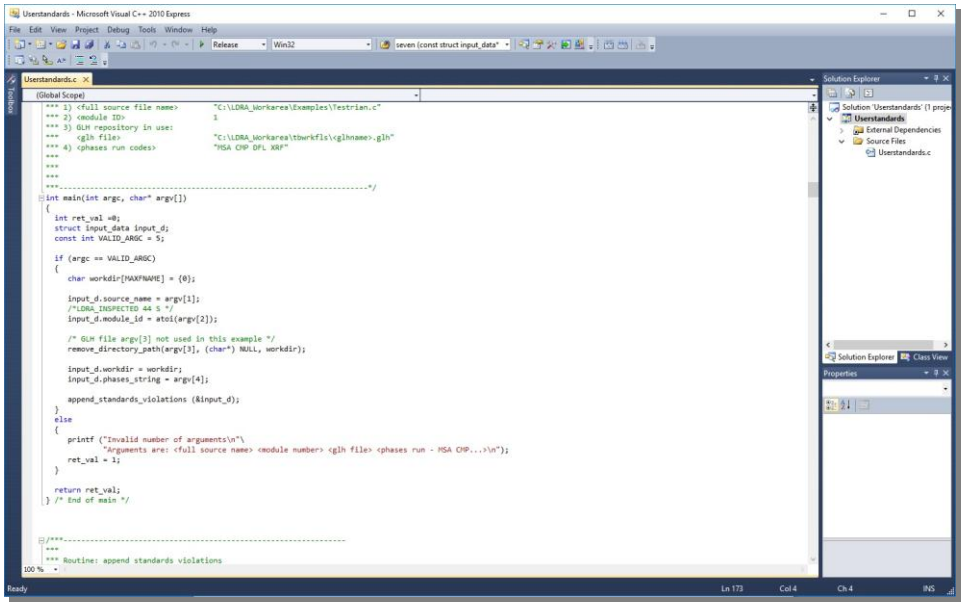


Ilustración 18 - Visual Studio C++ 2010 Express

4.1.4 Jenkins

Jenkins (*ver ilustración 19*) es un software libre *open source* escrito en Java. Esta herramienta basada en el proyecto Hudson proporciona integración continua para el desarrollo de software. Es un sistema corriendo en un servidor que es un contenedor de servlets, como Apache Tomcat. Soporta herramientas de control de versiones como CVS, Subversion, Git, Mercurial, Perforce y Clearcase y puede ejecutar proyectos basados en Apache Ant y Apache Maven, así como scripts de shell y programas batch de Windows. El desarrollador principal es Kohsuke Kawaguchi.

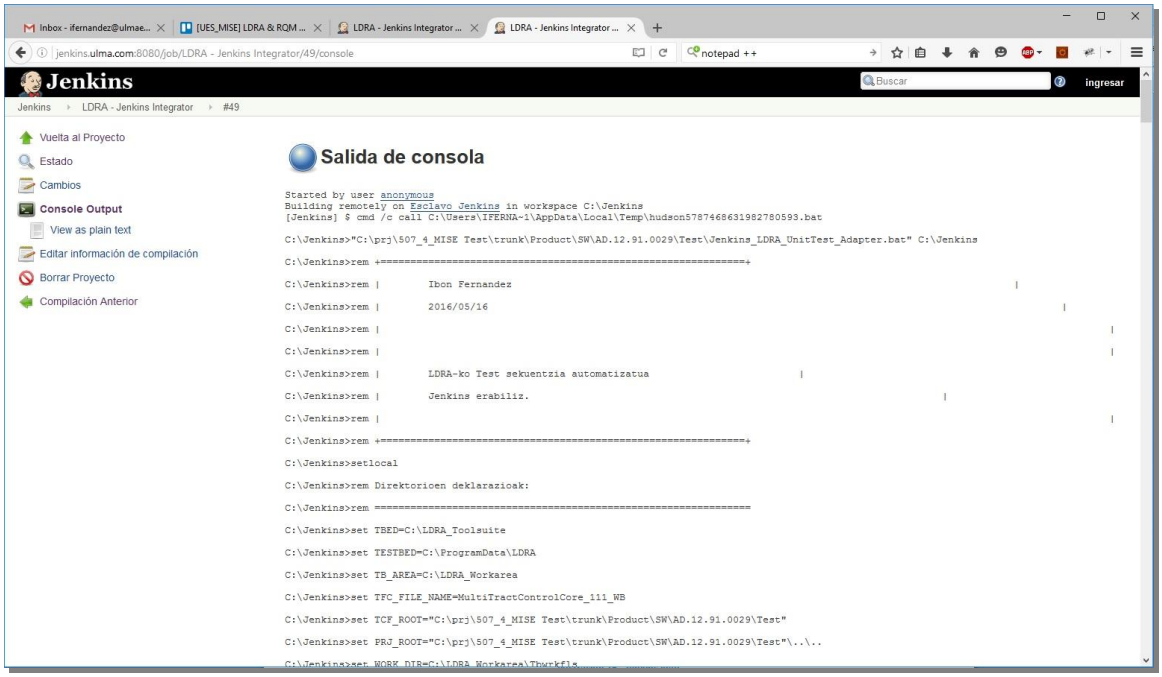


Ilustración 19 – Jenkins

4.1.5 MULTI

MULTI es un IDE (*ver ilustración 20*) multiplataforma para C y C ++ capaz de funcionar en Windows, Linux y Solaris. Está dirigido a ingenieros de sistemas embebidos y funciona con los compiladores de GreenHills en tareas de debugear errores. Entre sus características se encuentra un navegador CVS integrado, un visor de diferencias, el autocompletado automático del código, generadores gráficos de jerarquía de clases, la integración con Eclipse, un gestor de errores, y los puntos de interrupción de secuencias de comandos.

Con motivo de hacer compatible TBrun con el entorno de GreenHills, se tuvieron que realizar varias peticiones a los proveedores de LDRA. La primera para actualizar todo el entorno desde la versión 9.4.6 al 9.5.3 y la segunda para la descarga de una TLP (*LDRA TLP v953 C-C++ - GreenHills MULTI v6 INTEGRITY PPC Serial*) que pudiera instalar el compilador de Integrity en el área de trabajo.

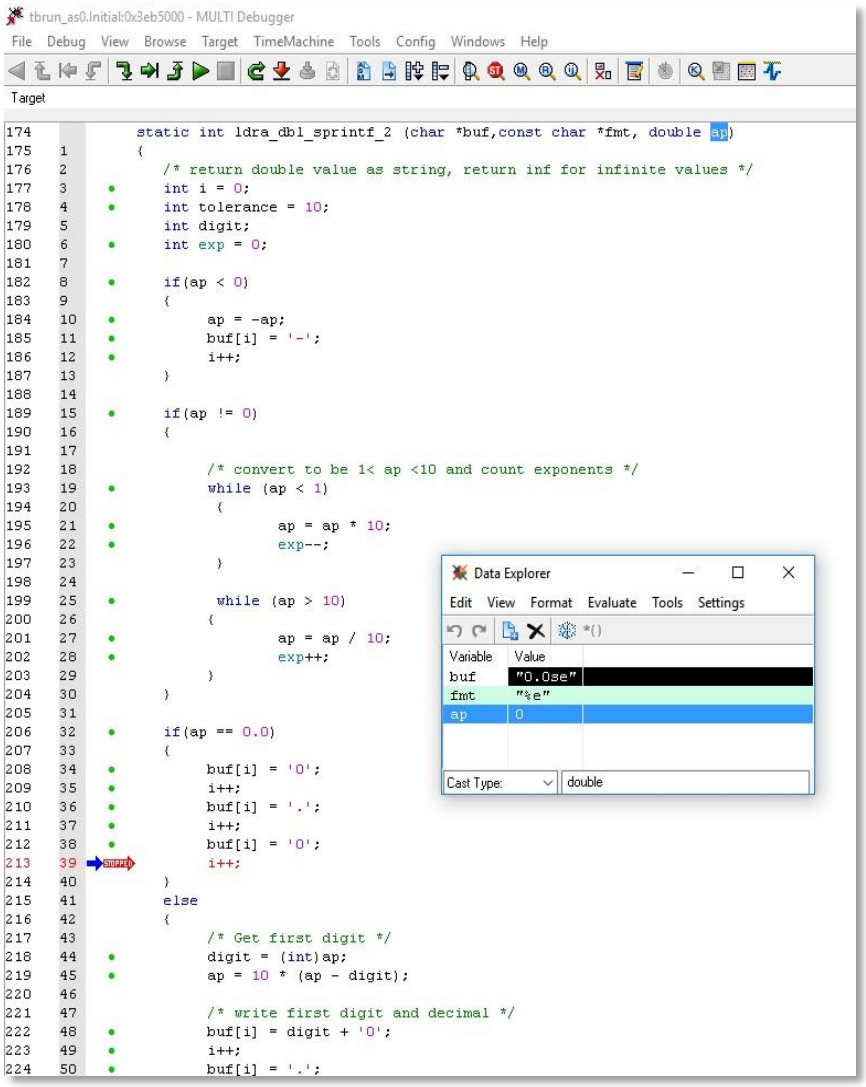


Ilustración 20 – Código de conversión de Float a String en MULTI

4.1.6 Otras herramientas software

En este proyecto se han utilizado editores de texto con soporte para varios lenguajes de programación como Eclipse Mars, Eclipse LUNA, Notepad++ y Sublime Text 2. También se han manejado otros programas como Araxis para comparar, comprender y combinar diferentes códigos fuentes. Para el control de versiones del proyecto se ha utilizado Apache Subversion. La obtención de licencias fuera de las redes en las que se encuentran los servidores de licencias de Integrity o LDRA, se ha realizado mediante varios VPN. Por otra parte, para el desarrollo de las configuraciones en el entorno del código fuente se ha empleado la aplicación Cygwin64, que proporciona un comportamiento similar a los sistemas Unix en Microsoft Windows.

5. PLAN DE VALIDACIÓN

En este capítulo se expondrá una parte del plan de validación diseñado al principio del ciclo de vida del proyecto. Se ha prescindido de fracciones de la misma ya que se repiten en otros puntos del documento.

5.1 Revisión del código

Se aplicará el estándar MISRA-C++:2008 para revisar las violaciones cometidas en el desarrollo del código. No será necesaria la corrección de todos los avisos ni requerimientos aparentes en la revisión, pero si se tendrán que enmendar una reducida parte de ellas en alguna etapa de la prueba, volviendo a rescatar el código original para la realización de la fase de prueba unitaria.

Se añadirán una serie de reglas adicionales de CAF P&A (en combinación con el estándar MISRA-C++:2008). Se implementará un número reducido de reglas en forma de Reglas de Usuario para LDRA y se comprobará el resultado de su verificación. A continuación se harán las correcciones necesarias para el cumplimiento de estas reglas de usuario, volviendo al código original para la realización de la fase de prueba unitaria.

Las reglas de CAF P&A a desarrollar serán estas tres:

- Formato adecuado de los identificadores de **tipo**.
- Formato adecuado de los identificadores de **clase**.
- Formato adecuado de los identificadores de **estructuras**.

5.2 Revisión de la calidad

Para el establecimiento de las métricas se marcará un criterio basado en las medidas del proyecto de la caja registradora de LDRA, adaptando algunas de ellas a la diferencia de tamaño entre las clases de un proyecto y otro. Estas van a ser las métricas que se van a aplicar para el análisis de calidad de la clase *multitractcontrolcore*:

5.2.1 Métricas del sistema

- **Nudos esenciales:** Mínimo de 0 y Máximo de 20.
- **Complejidad ciclomática esencial:** Mínimo de 1 y Máximo de 30.
- **Nudos:** Mínimo de 0 y Máximo de 50.
- **Complejidad ciclomática:** Mínimo de 0 y Máximo de 10.
- **Número de líneas ejecutables en el sistema:** Mínimo de 100 y Máximo de 5000.

5.2.2 Métricas de complejidad:

- **Nudos esenciales:** Mínimo de 0 y Máximo de 20.
- **Complejidad ciclomática esencial:** Mínimo de 1 y Máximo de 30.
- **Nudos:** Mínimo de 0 y Máximo de 50.
- **Complejidad ciclomática:** Mínimo de 1 y Máximo de 10.
- **Procedimientos estructurados:** Mínimo y Máximo de 1.

5.2.3 Comentarios relacionados con los procedimientos

- **Comentarios totales:** Mínimo de 1 y Máximo de 200.
- **Comentarios en encabezados:** Mínimo de 1 y Máximo de 50.
- **Comentarios en declaraciones:** Mínimo de 0 y Máximo de 100.
- **Comentarios en código ejecutable:** Mínimo de 1 y Máximo de 10.
- **Comentarios en blanco:** Mínimo de 0 y Máximo de 10.

5.2.4 Relación de comentarios a las líneas (%)

- **Número total de comentarios por líneas:** En un procedimiento: Mínimo de 10 y Máximo de 300. En la clase: Mínimo de 10 y Máximo de 300.
- **Número de comentarios en encabezados por líneas ejecutables :** En un procedimiento: Mínimo de 0 y Máximo de 1000. En la clase: Mínimo de 0 y Máximo de 0.
- **Número de comentarios en declaraciones por líneas ejecutables:** En un procedimiento: Mínimo de 1 y Máximo de 100. En la clase: Mínimo de 1 y Máximo de 100.
- **Número de comentarios en el código por líneas ejecutables:** En un procedimiento: Mínimo de 5 y Máximo de 200. En la clase: Mínimo de 5 y Máximo de 200.

5.2.5 Orientación a objetos basado en archivos

- **Número total de clases declarados:** Mínimo de 0 y Máximo de 10.
- **Número total de objetos creados:** Mínimo de 0 y Máximo de 10.

5.2.6 Orientación a objetos basado en clases

- **Número de hijos:** Mínimo de 0 y Máximo de 100.
- **Objetos creados:** Mínimo de 0 y Máximo de 100.
- **Número de datos miembros:** Mínimo de 0 y Máximo de 10.
- **Número de miembros:** Mínimo de 1 y Máximo de 20.

5.2.7 Clases base

- **Número total de clases base:** Mínimo de 0 y Máximo de 10.

- **Número total de miembros de dato:** Mínimo de 1 y Máximo de 30.
- **Número total de miembros:** Mínimo de 1 y Máximo de 40.
- **Profundidad de herencia:** Mínimo de 0 y Máximo de 5.

5.2.8 Clases base 2

- **Miembros estáticos:** Mínimo de 0 y Máximo de 10.
- **Uso de variables fuera de clases:** Mínimo de 0 y Máximo de 10.
- **Llamadas a procedimientos fuera de clases:** Mínimo de 0 y Máximo de 10.
- **Uso de métodos de clases externos:** Mínimo de 0 y Máximo de 10.
- **Uso de objetos fuera de clases:** Mínimo de 0 y Máximo de 10.

5.2.9 Información de procedimientos

- **Número de líneas reformadas ejecutables:** Mínimo de 2 y Máximo de 400.
- **Número de bloques básicos:** Mínimo de 1 y Máximo de 500.
- **Número medio de líneas por bloques básicos:** Mínimo de 1.0 y Máximo de 6.0.
- **Puntos de entrada a procedimientos:** Mínimo de 1y Máximo de 1.
- **Puntos de salida a procedimientos:** Mínimo de 0 y Máximo de 1.

5.2.10 Métricas *Halstead*

- **Operadores totales:** Mínimo de 0 y Máximo de 10000.
- **Operandos totales:** Mínimo de 0 y Máximo de 20000.
- **Operadores únicos:** Mínimo de 0 y Máximo de 100.

- **Operandos únicos:** Mínimo de 0 y Máximo de 300.
- **Vocabulario:** Mínimo de 0 y Máximo de 340.
- **Longitud:** Mínimo de 0 y Máximo de 15000.
- **Volumen:** Mínimo de 0 y Máximo de 150000.

5.2.11 Secuencia y saltos de código linear (LCSAJ) e inaccesibilidad de información

- **Total de LCSAJs:** Mínimo de 300 y Máximo de 2000.
- **LCSAJs accesibles:** Mínimo de 0 y Máximo de 2000.
- **LCSAJs inaccesibles:** Mínimo de 0 y Máximo de 0.
- **Densid máxima de LCSAJ:** Mínimo de 0 y Máximo de 1000.
- **Número de líneas inaccesibles:** Mínimo de 0 y Máximo de 0.
- **Número de ramas inaccesibles:** Mínimo de 0 y Máximo de 0.

5.2.12 Análisis bucle/intervalo

- **Número de bucles:** Mínimo de 0 y Máximo de 10.
- **Profundidad del bucle:** Mínimo de 0 y Máximo de 2.
- **Número de intervalos de orden 1:** Mínimo de 1 y Máximo de 5.
- **Profundidad máxima de intervalos:** Mínimo de 0 y Máximo de 3.
- **Intervalos reducibles:** Mínimo de 1 y Máximo de 1.

5.2.13 Información de código reformado

- **Total de líneas reformadas:** Mínimo de 1 y Máximo de 15000.

- **Número total de comentarios en referencia al código:** Mínimo de 50 y Máximo de 7500.
- **Número de líneas en blanco:** Mínimo de 0 y Máximo de 400.
- **Número de líneas reformadas ejecutables:** Mínimo de 0 y Máximo de 7500.
- **Número de líneas reformadas no ejecutables:** Mínimo de 1 y Máximo de 5000.
- **Número de procedimientos:** Mínimo de 0 y Máximo de 200.
- **Número de líneas fuente:** Mínimo de 5000 y Máximo de 13000.
- **Factor de expansión:** Mínimo de 1.0 y Máximo de 2.0.

5.2.14 Análisis de flujo de datos

- **Número de globales:** Mínimo de 0 y Máximo de 100.
- **Número de entradas:** Mínimo de 0 y Máximo de 15.
- **Número de salidas:** Mínimo de 0 y Máximo de 15.

5.3 Entornos

En Este proyecto se ha decidido ejecutar las pruebas de validación y verificación del código AX00 en los siguientes entornos:

- **Entorno 1:** Representa el entorno del Sistema Operativo Windows donde se desarrollarán las pruebas con herramientas de LDRA y ejecuciones mediante la plataforma Visual Studio 2010.
- **Entorno 2:** Representa al entorno del Rack VEGA y el soporte de la aplicación MULTI Compiler v5.2.4 en el PC.

5.4 Escenarios de las pruebas de validación y verificación

En este apartado se van a exponer los 12 escenarios distintos del proceso de validación. Se describen de la siguiente manera:

Descripción del análisis estático con Visual Studio 10

Identificador: V01

Entorno: Entorno 1

Responsables: Ibon Fernandez

Herramientas: TBvision

Descripción: Este escenario representa la situación en el que el SW se analiza de forma estática mediante TBvision. Se aplicará un análisis combinado de las opciones de "Main Static Analysis", "Complexity Analysis", "Static Data Flow Analysis" y "Cross Reference. El código tendrá que cumplir en lo posible el estándar MISRA-C++:2008, las reglas desarrolladas para CAF P&A y las métricas establecidas para la testabilidad, mantenivididad y claridad.

Resultado esperado: Se esperan encontrar resultados de Testabilidad, Mantenibilidad y Claridad inferiores al 100% y múltiples violaciones del código en los informes para el estándar MISRA-C++:2008 y las reglas de CAF P&A.

Resultado obtenido: La mayoría de métodos han cumplido al 100% el requerimiento de mantenivididad y testabilidad del código. No así con el campo de la claridad que ha fallado por norma general. Por otra parte, casi la totalidad de las funciones han manifestado errores y avisos al no cumplir los del estándar MISRA-C++:2008 y CAF P&A. Los errores en los informes de calificaciones de faltas, calificaciones de calidad y calificaciones de seguridad han mostrado fallos puntuales, mientras que el informe de calificación de memoria no ha mostrado ninguna violación del estándar.

Descripción de prueba unitaria en modo caja negra con Visual Studio 10

Identificador: V02

Entorno: Entorno 1

Responsables: Ibon Fernandez

Herramientas: TBrun + compilador Visual Studio 10

Descripción: Este escenario representa la situación en el que el SW se analiza mediante prueba unitaria con la técnica de prueba de caja negra usando TBrun. Los métodos que se pretenden testear son el constructor *CMultiTractControlCore*, métodos de inicialización de vectores como *AddTCC* y *AddInverterChannel* y procedimientos secundarios como *StartInverters*, *RegisterSeverityCallbacks*, etc.

Resultado esperado: Se espera que en un principio todos los casos de test consigan ser validados (PASS).

Resultado obtenido: Todos los test han sido validados correctamente.

Descripción de prueba unitaria en modo caja blanca con Visual Studio 10

Identificador: V03

Entorno: Entorno 1

Responsables: Ibon Fernandez

Herramientas: TBrun + compilador Visual Studio 10

Descripción: Este escenario representa la situación en el que el SW se analiza mediante prueba unitaria con la técnica de prueba de caja blanca usando TBrun. Los métodos que se pretenden testear son el constructor *CMultiTractControlCore*,

métodos de inicialización de vectores como *AddTCC* y *AddInverterChannel* y procesos secundarios como *StartInverters*, *RegisterSeverityCallbacks*, etc.

Resultado esperado: Se espera que en un principio todos los casos de test consigan ser validados (PASS), pero al mismo tiempo se prevé que en la mayoría de las pruebas no se consiga una cobertura de declaración ni una cobertura de decisión del 100%.

Resultado obtenido: Todos los test han conseguido ser validados correctamente, pero los procedimientos *CMultiTractControlCore*, *RegisterSeverityCallbacks* y *StartInverters* no han conseguido obtener una cobertura del 100%

Descripción del análisis estático con Integrity

Identificador: V04
Entorno: Entorno 1
Responsables: Ibon Fernandez
Herramientas: TBvision

Descripción: Este escenario representa la situación en el que el SW se analiza de forma estática mediante TBvision. Se aplicará un análisis combinado de las opciones de "Main Static Analysis", "Complexity Analysis", "Static Data Flow Analysis" y "Cross Reference. El código tendrá que cumplir en lo posible el estándar MISRA-C++:2008, las reglas desarrolladas para CAF P&A y las métricas establecidas para la testabilidad, mantenivididad y claridad.

Resultado esperado: Se esperan encontrar resultados de Testabilidad, Mantenibilidad y Claridad inferiores al 100% y múltiples violaciones del código en los informes para el estándar MISRA-C++:2008 y las reglas de CAF P&A.

Resultado obtenido: La mayoría de métodos han cumplido al 100% el requerimiento de mantenivididad y testabilidad del código. No así con el campo de la claridad que ha

fallado por norma general. Por otra parte, casi la totalidad de las funciones han manifestado errores y avisos al no cumplir los del estándar MISRA-C++:2008 y CAF P&A. Los errores en los informes de calificaciones de faltas, calificaciones de calidad y calificaciones de seguridad han mostrado errores puntuales, mientras que el informe de calificación de memoria no ha mostrado ninguna violación en el código.

Descripción de prueba unitaria en modo caja negra con Integrity

Identificador: V05
Entorno: Entorno 2
Responsables: Ibon Fernandez
Herramientas: TBrun + Integrity (GreenHills) + IDE MULTI

Descripción: Este escenario representa la situación en el que el SW se analiza mediante prueba unitaria con la técnica de prueba de caja negra usando TBrun. Los métodos que se pretenden testear son el constructor *CMultiTractControlCore*, métodos de inicialización de vectores como *AddTCC* y *AddInverterChannel* y procesos secundarios como *StartInverters*, *RegisterSeverityCallbacks*, etc.

Resultado esperado: Se espera que en un principio todos los casos de test consigan ser validados (PASS).

Resultado obtenido: Todos los test han sido validados correctamente.

Descripción de prueba unitaria en modo caja blanca con Integrity

Identificador: V06
Entorno: Entorno 2
Responsables: Ibon Fernandez
Herramientas: TBrun + Integrity (GreenHills) + IDE MULTI

Descripción: Este escenario representa la situación en el que el SW se analiza mediante prueba unitaria con la técnica de prueba de caja blanca usando TBrun. Los métodos que se pretenden testear son el constructor *CMultiTractControlCore*, métodos de inicialización de vectores como *AddTCC* y *AddInverterChannel* y procesos secundarios como *StartInverters*, *RegisterSeverityCallbacks*, etc.

Resultado esperado: Se espera que en un principio todos los casos de test consigan ser validados (PASS), pero al mismo tiempo se prevé que en la mayoría de las pruebas no se consiga una cobertura de declaración ni una cobertura de decisión del 100%.

Resultado obtenido: Todos los test han conseguido ser validados correctamente, pero los procedimientos *CMultiTractControlCore*, *RegisterSeverityCallbacks* y *StartInverters* no han conseguido obtener una cobertura del 100%

Descripción de prueba de integración continua en Jenkins

Identificador: V07
Entorno: Entorno 1
Responsables: Ibon Fernandez
Herramientas: Visual Studio 10 + Jenkins

Descripción:

Este escenario representa la situación en el que el SW se analiza de forma estática mediante Jenkins. Se aplicará un análisis combinado de las opciones de "Main Static Analysis", "Complexity Analysis", "Static Data Flow Analysis" y "Cross Reference. El código tendrá que cumplir en lo posible el estándar MISRA-C++:2008, las reglas desarrolladas para CAF P&A y las métricas establecidas para la testabilidad, mantenivididad y claridad. A continuación el SW se analizará mediante prueba unitaria con la técnica de prueba de caja negra usando. Los métodos que se pretenden testear son el constructor *CMultiTractControlCore*, métodos de inicialización de vectores como *AddTCC* y *AddInverterChannel* y procesos secundarios como *StartInverters*, *RegisterSeverityCallbacks*, etc que estarán exportados en formato (.tcf).

Resultado esperado: Se esperan encontrar resultados de Testabilidad, Mantenibilidad y Claridad inferiores al 100% y múltiples violaciones del código en los informes para el estándar MISRA-C++:2008 y las reglas de CAF P&A. Se confía también que en un principio todos los casos de test consigan ser validados (PASS), pero al mismo tiempo se prevé que en la mayoría de las pruebas no se consiga una cobertura de declaración ni una cobertura de decisión del 100%.

Resultado obtenido: La mayoría de métodos han cumplido al 100% el requerimiento de mantenivididad y testabilidad del código. No así con el campo de la claridad que ha fallado por norma general. Por otra parte, casi la totalidad de las funciones han manifestado errores y avisos al no cumplir los del estándar MISRA-C++:2008 y CAF P&A. Los errores en los informes de calificaciones de faltas, calificaciones de calidad y calificaciones de seguridad han mostrado errores puntuales, mientras que el informe de calificación de memoria no ha mostrado ninguna violación en el código. Todos los test han conseguido ser validados correctamente, pero los procedimientos *CMultiTractControlCore*, *RegisterSeverityCallbacks* y *StartInverters* no han conseguido obtener una cobertura del 100%

5.5 Tabla de tareas del proyecto

TAREAS	DESCRIPCIÓN	PARTICIPANTES
(1) Plan de validación y verificación P1	Diseñar y redactar la primera parte del plan de validación y verificación del software con herramientas avanzadas.	Ibon Fernandez, Ainhoa Aristimuño
(2) Configuración de las herramientas y el entorno.	Herramientas LDRA y entorno Visual Studio 10.	Ibon Fernandez, Ainhoa Aristimuño
(3) Análisis estático	Análisis del programa sin ejecución.	Ibon Fernandez
(4) Aplicación de métricas	Análisis, desarrollo e implementación de métricas.	Ibon Fernandez
(5) Resolver incidencias	Resolver incidencias surgidas en la ejecución del análisis estático.	Ibon Fernandez
(6) Repetir tarea	Repetir tareas (4)-(5) hasta superar el análisis estático satisfactoriamente.	Ibon Fernandez

(7) Aplicación del estándar MISRA-C++:2008	Análisis del estándar MISRA-C++:2008.	Ibon Fernandez
(8) Resolver incidencias	Resolver incidencias surgidas en la ejecución del análisis del estándar MISRA-C++:2008.	Ibon Fernandez
(9) Repetir tarea	Repetir tareas (7)-(8) hasta superar el análisis satisfactoriamente.	Ibon Fernandez
(10) Aplicación de reglas de CAF P&A	Análisis, desarrollo e implementación de las reglas de CAF P&A.	Ibon Fernandez
(11) Resolver incidencias	Resolver incidencias surgidas en la ejecución del análisis del estándar de CAF P&A.	Ibon Fernandez
(12) Repetir tarea	Repetir tareas (6)-(9)-(10)-(11) hasta superar el análisis satisfactoriamente.	Ibon Fernandez
(13) Compilación del programa en LDRA con VS10 para BB	Compilar el programa en TBrun con Visual Studio 10 para Black Box.	Ibon Fernandez
(14) Resolver incidencias	Resolver incidencias surgidas al compilar del programa en TBrun con Visual Studio 10 para Black Box.	Ibon Fernandez
(15) Repetir tarea	Repetir tareas (13)-(14) hasta compilar el programa satisfactoriamente con Visual Studio 10 para Black Box.	Ibon Fernandez
(16) Pruebas en BB en VS10	Realizar casos de test en Black Box con Visual Studio 10.	Ibon Fernandez
(17) Resolver incidencias	Resolver incidencias surgidas en la aplicación de casos de test en modo Black Box con Visual Studio 10.	Ibon Fernandez
(18) Repetir tarea	Repetir tareas (16)-(17) hasta superar los test satisfactoriamente en Black Box con Visual Studio 10.	Ibon Fernandez
(19) Compilación del programa en LDRA con VS10 para WB	Compilar el programa en TBrun con Visual Studio 10 para White Box.	Ibon Fernandez
(20) Resolver incidencias	Resolver incidencias surgidas al compilar del programa en TBrun con Visual Studio 10 para White Box.	Ibon Fernandez
(21) Repetir tarea	Repetir tareas (19)-(20) hasta compilar el programa satisfactoriamente con Visual Studio 10 para White Box.	Ibon Fernandez
(22) Pruebas en WB con VS10	Realizar casos de test en White Box con Visual	Ibon Fernandez

	Studio 10.	
(23) Pruebas de cobertura en VS10	Añadir casos de Test adicionales para conseguir pruebas de cobertura del 100% de cada método.	Ibon Fernandez
(24) Resolver incidencias	Resolver incidencias surgidas en la aplicación de casos de test en modo White Box con Visual Studio 10.	Ibon Fernandez
(25) Repetir tarea	Repetir tarea (22)-(23)-(24) hasta conseguir pruebas de cobertura del 100% de cada método.	Ibon Fernandez
(26) Compilar en Cygwin con Integrity	Compilar el código modificado para LDRA y VS10 en Cygwin con Integrity.	Ibon Fernandez
(27) Resolver incidencias	Resolver incidencias surgidas al compilar el código en Cygwin para Integrity.	Ibon Fernandez
(28) Repetir tarea	Repetir tareas (15)-(18)-(21)-(24) hasta compilar el código en LDRA con Visual Studio 10 y en Cygwin con Integrity.	Ibon Fernandez
(29) Instalar Integrity para LDRA	Instalación y puesta en marcha del compilador Greenhills MULTI v6 Integrity para LDRA.	Ibon Fernandez
(30) Compilación del programa en LDRA con VS10 para BB	Compilar el programa en TBrun con Integrity para Black Box.	Ibon Fernandez
(31) Resolver incidencias	Resolver incidencias surgidas al compilar del programa en TBrun con Integrity para Black Box.	Ibon Fernandez
(32) Repetir tarea	Repetir tareas (30)-(31) hasta compilar el programa satisfactoriamente con Integrity para Black Box.	Ibon Fernandez
(33) Pruebas en BB en VS10	Realizar casos de test en Black Box con Integrity.	Ibon Fernandez
(34) Resolver incidencias	Resolver incidencias surgidas en la aplicación de casos de test en modo Black Box con Integrity.	Ibon Fernandez
(35) Repetir tarea	Repetir tareas (33)-(34) hasta superar los test satisfactoriamente en Black Box con Integrity.	Ibon Fernandez
(36) Compilación del programa en LDRA con VS10 para WB	Compilar el programa en TBrun con Integrity para White Box.	Ibon Fernandez
(37) Resolver incidencias	Resolver incidencias surgidas al compilar del programa en TBrun con Integrity para White Box.	Ibon Fernandez
(38) Repetir tarea	Repetir tareas (36)-(37) hasta compilar el programa satisfactoriamente con Integrity para White Box.	Ibon Fernandez
(39) Pruebas en WB con VS10	Realizar casos de test en White Box con Integrity.	Ibon Fernandez
(40) Pruebas de cobertura en	Añadir casos de Test adicionales para conseguir	Ibon Fernandez

VS10	pruebas de cobertura del 100% de cada método.	
(41) Resolver incidencias	Resolver incidencias surgidas en la aplicación de casos de test en modo White Box con Integrity.	Ibon Fernandez
(42) Repetir tarea	Repetir tarea (39)-(40)-(41) hasta conseguir pruebas de cobertura del 100% de cada método.	Ibon Fernandez
(43) Compilar en Cygwin con Integrity	Compilar el código modificado para LDRA y Integrity en Cygwin con Integrity.	Ibon Fernandez
(44) Resolver incidencias	Resolver incidencias surgidas al compilar el código en Cygwin para Integrity.	Ibon Fernandez
(45) Repetir tarea	Repetir tareas (38)-(41)-(43)-(44) hasta compilar el código en LDRA con Integrity y en Cygwin con Integrity.	Ibon Fernandez
(46) Crear módulo integrado de varios clases	Crear un set de varios clases para una prueba de integración.	Ibon Fernandez
(47) Ejecutar análisis estático a los módulos integrados	Ejecutar un análisis estático al set de módulos, sin llegar a analizar los resultados.	Ibon Fernandez
(48) Compilar y ejecutar un test a los módulos integrados	Ejecutar un caso de test al set de módulos integrados sin llegar a analizar los resultados.	Ibon Fernandez
(49) Plan de validación y verificación P2	Redactar la segunda parte del plan de validación para la integración continua	Ibon Fernandez Ainhoa Aristimuño
(50) Configuración de las herramientas y el entorno	Configuración de Jenkins y proceso esclavo	Ibon Fernandez
(51) Desarrollo de programas	Implementación de scripts para automatización de proceso de V&V.	Ibon Fernandez
(52) Prueba de integración en Jenkins	Ejecución de la prueba de integración en Jenkins	Ibon Fernandez
(53) Repetir tarea	Repetir tareas (51)-(52) hasta conseguir una prueba de integración satisfactoria	Ibon Fernandez
(54) Desarrollo de entorno V&V	Desarrollo de un entorno que recopile todos los ficheros necesarios para la puesta en marcha completa del proyecto desde otro equipo.	Ibon Fernandez
(55) Preparación de una presentación	Preparar una presentación para exposición del proyecto al personal de LDRA vía Webex.	Ibon Fernandez, Ainhoa Aristimuño
(56) Preparación de una presentación + DEMO	Preparar una presentación + DEMO para exposición del proyecto al personal de CAF P&A en Irura.	Ibon Fernandez, Ainhoa Aristimuño, Oscar Berreteaga

(57) Preparación de una presentación	Preparar una presentación para exposición del proyecto en ULMA Embedded Sotutions	Ibon Fernandez, Ainhoa Aristimuño
(58) Preparación de una presentación	Preparar una presentación para exposición del proyecto para UPV/EHU	Ibon Fernandez

Tabla 2 - Tareas principales a realizar en el proyecto

5.6 Tabla de Coste de proceso de V&V en tiempo

La *tabla 3* pretende estimar el tiempo mínimo para desarrollar un proceso de pruebas con las herramientas de LDRA.

Estándares		
Registrar estándares de usuario	cppreport.dat	5 min
Configurar estándares de usuario	cpppen.dat	5 min
Programar estándares de usuario	Userstandars.c	20 min/erregelako
Crear ejecutable de estándares de usuario	Userstandars.exe	10 min
Ejegir estándares MISRA + Estándares personalizados	Tbvision	5 min
Análisis		
Confgurar fichero de importación	Sysearch.dat	15 min - 1h
Configurar opciones de análisis	Tbvision (LDRA Configuration Dialogs)	5 min
Ejecutar análisis + instrumentar (dependiendo de MCDC)	Tbvision	5 - 10 min
Analizar resultados de métricas	Tbvision (Quality Review)	15 min/metodoko
Corregir métricas	Código	15 min/metodoko
Analizar resultados de estándar	Tbvision (Code Review)	15 min/metodoko
Corregir estándar	Código	15 min/metodoko
Test		
Abrir módulo	Tbrun	0 min
Crear nuevas secuencias de test + Stubs + Variables globales	Tbrun	5 min
Configurar código mediante STUB	Tbrun	15 min - 1h
Crear casos de test (con variables necesarias)	Tbrun	15 min/por cada test
Ejecutar casos de test (BB)	Tbrun	0 min/por cada test
Ejecutar casos de test (WB)	Tbrun	5 min/por cada test
Visualizar grafo de flujo (WB)	TBrun	5 min
Exportar fichero TCF de casos de test	Tbrun	5 min
Importar fichero TCF de casos de test	Tbrun	0 min
En total		
Proceso por primera vez		(95 - 190) min fijos + 5 min (MCDC) + 20 min/regla + 60 min/método + 20 min/test
Uso de entorno preconfigurado		15 min + 5 min (MCDC)

Tabla 3 - Estimación de tiempos en proceso V&V

6. DESARROLLO DEL PROCESO DE VALIDACIÓN Y VERIFICACIÓN

El proceso de validación y verificación en LDRA ha contenido un primer paso de configuración y adaptación de los entornos de desarrollo, para después poder ejecutar la etapa de prueba. El desarrollo se ha dividido en tres pasos. Un primer análisis estático del código fuente, seguido la creación de secuencias de test y terminando con la automatización de los test mediante la integración en Jenkins.

6.1 ANALISIS ESTATICO DEL CODIGO

El análisis estático de software, que es aquel efectuado sin la puesta en marcha del programa, se puede ejecutar sobre un objeto, aunque en este caso se ha llevado a cabo sobre el propio código fuente. La información obtenida de esta tarea se ha llegado a usar para múltiples propósitos, como la indicación de errores de codificación.

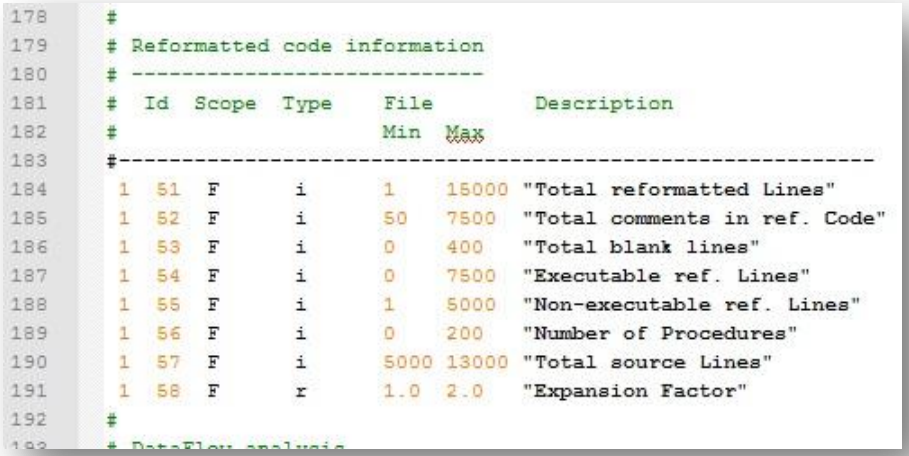
6.1.1 Configuración para la revisión de la calidad (métricas)

El informe de calidad de LDRA se basa en las métricas aplicadas las cuales se establecen mediante las tablas de configuración que ofrece el fichero de personalización de métricas *metpen.dat*. Este archivo clasifica las métricas en diferentes subgrupos (*ver apartado 5.2*) dependiendo del tipo de campo que se quiera medir.

En este proyecto se ha hecho uso de este recurso estableciendo una serie de métricas y observando a posteriori el grado de cumplimiento de la clase analizada en el informe de calidad. Debido a que el propietario del software no ha llegado a facilitar ningún tipo de directriz en cuanto al establecimiento de estas medidas, los criterios utilizados en la aplicación de medias, máximos y mínimos han sido intuidos a partir de la referencia de otro tipo de proyectos similares como el proyecto de *cashregister* de LDRA.

Las tablas de configuración de métricas de LDRA (*ver ilustración 21*) ofrecen la oportunidad de aplicar las medidas a todo el sistema, a nivel de fichero, por cada

procedimiento o en la clase entera. Permiten también la opción de configurar el tipo de dato que se quiera medir y otra clase de campos como descripciones e identificadores.



```
178  #
179  # Reformatted code information
180  # ---------------------------
181  #  Id  Scope  Type    File        Description
182  #                     Min  Max
183  #-------------------------------------------------------------
184   1  51  F      i      1    15000 "Total reformatted Lines"
185   1  52  F      i      50   7500  "Total comments in ref. Code"
186   1  53  F      i      0    400   "Total blank lines"
187   1  54  F      i      0    7500  "Executable ref. Lines"
188   1  55  F      i      1    5000  "Non-executable ref. Lines"
189   1  56  F      i      0    200   "Number of Procedures"
190   1  57  F      i      5000 13000 "Total source Lines"
191   1  58  F      r      1.0  2.0   "Expansion Factor"
192  #
```

Ilustración 21 - Ejemplo de tabla de configuración en *metpen.dat* (Información de código reformado)

Estos son los campos que LDRA permite modificar en el fichero *metpen.dat*, las cuales se han configurado en su totalidad:

- **Información del código reformateado para el Archivo:** Número total de líneas reformateadas, comentarios totales en el código reformateado, líneas ejecutables y no ejecutables en el código reformateado, número de procedimientos, líneas de código fuente totales y factor de expansión.

- **Información del procedimiento:** Líneas ejecutables en el código reformateado, número de bloques básicos, tamaño medio de los bloques básicos, puntos de entrada a procedimientos y puntos de salida de procedimientos.

- **Comentarios relacionados con los procedimientos (% del total):** Número total de comentarios, número de comentarios en cabeceras, número de comentarios en las declaraciones, comentarios en código ejecutable y cantidad de líneas en blanco.

- **Relación de los comentarios a las líneas ejecutables (%):** Líneas reformateadas ejecutables, número total de comentarios / líneas ejecutables, comentarios de cabecera / líneas ejecutables, comentarios en declaraciones / líneas ejecutables y comentarios en el código / líneas ejecutables.

- **Las métricas de complejidad:** Nudos, complejidad ciclomática, nudos esenciales, complejidad ciclomática esencial y procedimiento estructurados (SPV).

- **Métrica Halsteads:** Número total de operadores, cantidad de operandos, operadores únicos, operandos únicos, vocabulario, longitud y volumen.

- **Análisis del bucle / Intervalo:** Número de bucles, profundidad de los bucles, número de intervalos de orden 1, intervalo máximo de anidamiento de los bucles e intervalos reducibles.

- **LCSAJ y de inaccesibilidad:** LCSAJ totales, LCSAJ accesibles, LCSAJ inalcanzables, la densidad máxima de LCSAJ, líneas inalcanzables y ramas inalcanzable.

6.1.2 Configuración del informe del código (reglas y estándares)

Misra-C++:2008

El estándar que se ha seleccionado para realizar el análisis estático ha sido el MISRA-C++:2008, ya que es éste mismo el que se aplica en el código de control de tracción. Está disponible dentro de una lista en la que se encuentran otras normas para CPP como FSB582-C++, CERT-C++, HICC++, HICC++v4 entre otros. El menú de configuración de estos estándares permite no solo personalizar las reglas internacionales modificándolas al gusto, sino también la fuerza con la que se quieren aplicar en el análisis estático.

Regla “Example” basada en Misra-C++:2008

A raíz de la elección del estándar Misra-C++:2008, se ha contemplado la creación de un conjunto de reglas propias llamadas “Example” (*ver ilustración 22*), que si bien están basadas en la de MISRA, han permitido al mismo tiempo la incorporación de reglas adicionales de CAF Power & Automation.

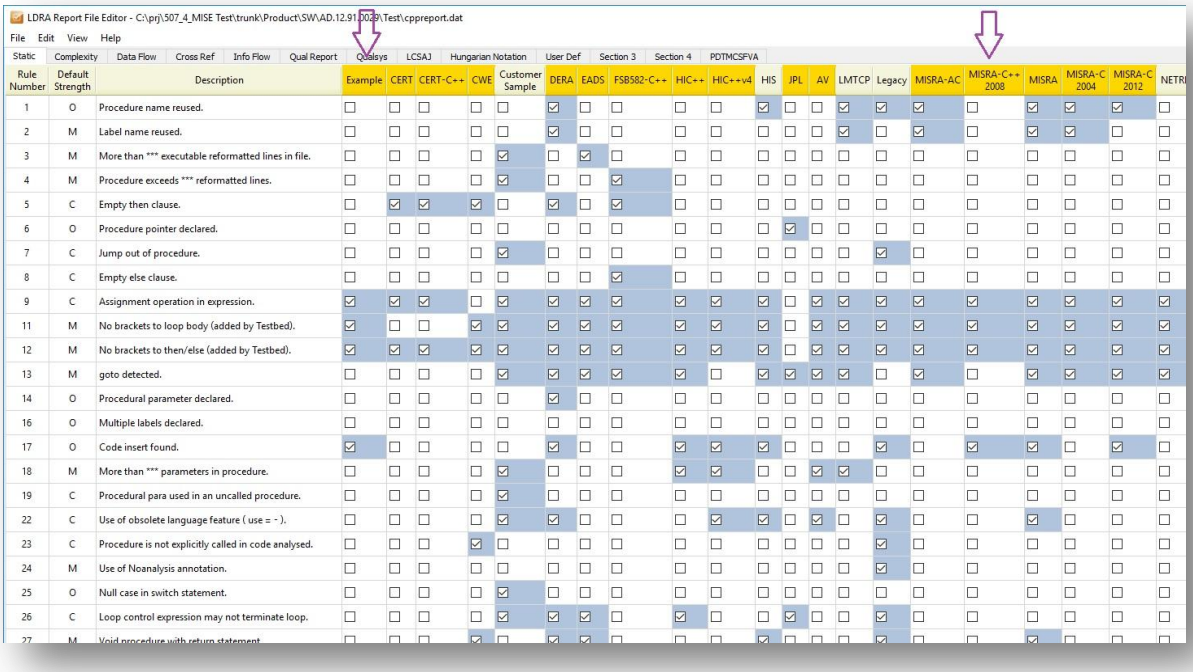


Ilustración 22 - Report File Editor

Reglas de CAF P&A

LDRA dispone de un sistema para desarrollar reglas propias e integrarlas con otros estándares. Las reglas de CAF P&A vienen en formato CPP interpretables para la herramienta de análisis estático QAC, pero no sirven para ninguna de las herramientas de LDRA. Por ello, este paso del proyecto ha requerido desarrollar una aplicación ejecutable que pueda servir para que TBvision verifique las reglas personalizadas de manera automática. En total CAF P&A dispone de 49 reglas aplicables para CPP, de las cuales se han implementado 3 para este proyecto.

Primeramente se ha tenido que modificar algunos ficheros de arranque de LDRA (*ver ilustración 23*) con motivo de configurar el sistema de reportes de TBvision. Dentro del archivo

cppreport.dat se han añadido las especificaciones de tres reglas de CAF P&A en el apartado de reglas de tipo "Z" que son las correspondientes a las desarrolladas por los usuarios. De la misma manera, se les ha adherido un identificador de regla (7, 8 y 9 respectivamente) para su referencia para el caso de que se cometiera una violación.

```
856 00 1    0    2 J Unused inspect annotation.
857 00 1    0    3 J All internal linkage calls unreachable.                MIS
858 00 1    0    1 Z File exceeds required size.
859 00 1    0    2 Z Chacks filename size is <= 8 characters.
860 00 1    0    3 Z Inclode name does not match source file name.
861 00 1    0    4 Z Source does not match layout template.
862 00 1    0    5 Z Hexadecimal number found.
863 00 1    0    6 Z Forbidden word found.
864 00 1    0    7 Z Caf rule 01 01 - Typedef identifiers shall end "_t".
865 00 1    0    8 Z Caf rule 01 02 - Struct identifiers incorrect format.
866 00 1    0    9 Z Caf rule 01 03 - Class identifier incorrect format.
867 00 1    0    1 H Global Variable does not conform to style g_<name>.    FSE
868 00 1    0    2 H Class Member does not conform to style m_<name>.       FSE
```

Ilustración 23 - Identificación de las reglas de usuario

En segundo lugar se ha configurado la elección de las reglas aplicables sobre el estándar personal "Example" creado anteriormente, la cual tiene la finalidad de unir Misra-C++:2008 con las reglas de CAF P&A. Para activar esta opción se ha tenido que modificar el fichero *cppreport.dat* a la que se le ha adherido la vinculación de dicha norma. Esta modificación se ve reflejada en el propio menú de estándares de TBvision una vez la herramienta lea el fichero de arranque (*ver ilustración 24*).

```
# 'Z' phase standards
# -------------------
#
   1 Z M "Example" "PDTMCSFVA:000000000" "K:style"
# File exceeds required size.
#
   2 Z M "Example" "PDTMCSFVA:000000000" "K:style"
# Chacks filename size is <= 8 characters.
#
   3 Z M "Example" "PDTMCSFVA:000000000" "K:style"
# Inclode name does not match source file name.
#
   4 Z M "Example" "PDTMCSFVA:000000000" "K:style"
# Source does not match layout template.
#
   5 Z M "Example" "PDTMCSFVA:000000000" "K:style"
# Hexadecimal number found.
#
   6 Z M "Example" "PDTMCSFVA:000000000" "K:style"
# Forbidden word found.
#
   7 Z M "Example" "PDTMCSFVA:000000000" "K:style"
# Caf rule 01 01 - Typedef identifiers shall end "_t".
#
   8 Z M "Example" "PDTMCSFVA:000000000" "K:style"
# Caf rule 01 02 - Struct identifiers incorrect format.
#
   9 Z M "Example" "PDTMCSFVA:000000000" "K:style"
# Caf rule 01 03 - Class identifier incorrect format.
```

LDRA Report File Editor - C:\LDRA_PRJ\507_4_MISE Test\trunk\Product\SW\AD.12

File Edit View Help

Static | Complexity | Data Flow | Cross Ref | Info Flow | Qual Report | Qua

Rule Number	Default Strength	Description	Example	CE
1	M	File exceeds required size.	☑	☐
2	M	Chacks filename size is <= 8 characters.	☑	☐
3	M	Inclode name does not match source file name.	☑	☐
4	M	Source does not match layout template.	☑	☐
5	M	Hexadecimal number found.	☑	☐
6	M	Forbidden word found.	☑	☐
7	M	Caf rule 01 01 - Typedef identifiers shall end "_t".	☑	☐
8	M	Caf rule 01 02 - Struct identifiers incorrect format.	☑	☐
9	M	Caf rule 01 03 - Class identifier incorrect format.	☑	☐

Ilustración 24 - Fichero de configuración cppreport.dat

Después de haber establecido las reglas personalizadas, se ha editado el fichero *userstandards.c* añadiendo la llamada e implementación de cada una de ellas complementando las 6 reglas que ya estaban creadas de antemano con las 3 de CAF P&A. Su funcionamiento se resume en la lectura línea por línea de una clase aplicando controles a nivel de String y verificando que se cumplan los estándares definidos. Si no lo hacen, generan una alarma de violación de regla que se reporta directamente en un mensaje predeterminado en el fichero de configuración *cppreport.dat*. Este mensaje aparece en el informe del código junto a las demás violaciones de estándar que se hayan dado en las reglas personalizadas o en el estándar de MISRA-C++:2008.

Por último, ha sido necesario generar un nuevo ejecutable para aplicar las reglas ya programadas. Para ello se ha tenido que depurar la implementación de *userstandards.c* con Microsoft Visual C++ 2010 Express (*ver ilustración 25*) y sustituir el *userstandars.exe* generado con el que pudiera haber en el área de trabajo de LDRA.

Ilustración 25 - Creación de Userstandards.exe con VS10

6.1.3 Ejecución del análisis estático

De la lista de opciones disponibles para ejecutar dicho análisis (*ver apartado 2.3.2*), en el menú de tipos de análisis de TBvision se han elegido las cuatro primeras opciones (*ver ilustración 26*): “Main Static Analysis”, “Complexity Analysis”, “Static Data Flow Analysis” y “Cross Reference”. Éste es el análisis mínimo que se le tiene que aplicar al código para que en la etapa posterior de la prueba unitaria se puedan ejecutar las pruebas de validación. Otras opciones como las condiciones compuestas no se han llegado a aplicar, sobre todo por su alto coste computacional, aunque el nivel SIL2 del estándar EN50128 ferroviario las contemple como “Recomendadas” (*ver tabla 4*).

	SIL 0	SIL 1	SIL 2	SIL 3	SIL 4
1. Statement Coverage	R	MR	MR	MR	MR
2. Branch Coverage	-	R	R	MR	MR
3. Composed conditions (MC/DC or MCC-Coverage)	-	R	R	MR	MR
4. Data Flow Analysis	-	R	R	MR	MR
5. Path Coverage	-	R	R	MR	MR

Tabla 4 – Algunos ejemplos dentro del EN 50128 "Railway applications", niveles de SIL

La ejecución del análisis estático se puede efectuar en diferentes configuraciones, de acuerdo a las necesidades de la verificación. Cuanto más exhaustiva sea ésta mayor será la probabilidad de que aparezcan errores y alertas en el informe posterior.

Con el "Main Static Analysis" se ha pretendido aplicar primero las métricas y después los estándares al igual que crear los grafos de llamadas. Mediante la opción de "Complexity Analysis" se ha buscado crear grafos de flujo y resultados de calidad de manera filtrada. Con la elección de "Static Data Flow Analysis" se ha producido la información sobre las llamadas de los procedimientos, anomalías en el flujo de datos y el análisis de la interfaz de los procedimientos. Por último, con la selección de "Cross Reference" se ha buscadp producir una referencia cruzada de todos los elementos de datos utilizados en el código fuente.

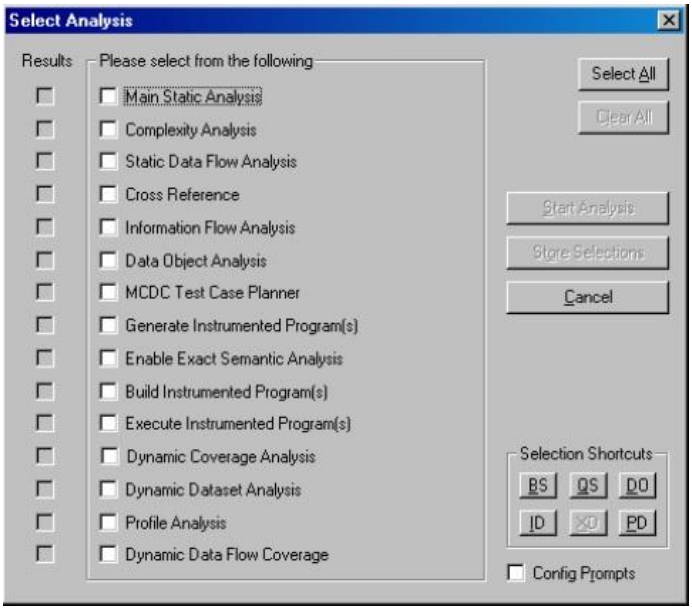


Ilustración 26 - Profile Analysis

Dicho esto, se han dejado fuera las opciones “Information Flow” para analizar las dependencias de las variables y “Data Object Analysis”, ya que no se ha planteado hacer un informe de variables específicas.

6.1.4 Resultados del análisis estático

Informe del código

El informe de revisión Código da una visión instantánea de la calidad del análisis de código fuente. En este caso, el informe ha reflejado la calidad de una sola clase (*multitractcontrolcore.cpp*), pero está pensado para que sea capaz de responder ante un sistema completo o un conjunto de clases integrados.

Los resultados del informe del código han desvelado un amplio abanico de errores y alarmas. Estos fallos han variado desde simples infracciones como el no uso de parámetros en diferentes procedimientos, hasta conversiones entre diferentes tipos de dato que se han producido sin su correspondiente casting. Si bien la finalidad de este proyecto no ha sido la corrección de la clase verificada, se ha intentado llevar a cabo un proceso de adaptación del código infractor con el fin de comparar los informes de los pre y post procesos de corrección.

En ese orden, se han corregido muchas de las faltas que mayormente han consistido en declarar tipos propios, eliminar procedimientos a los que no se les ha dado uso, declaración de miembros como constantes, etc. La presencia de otros errores sin embargo ha tenido que ser aceptada debido a la elevada dificultad de cumplir en su totalidad el estándar de MISRA. Ejemplo de ello pueden ser las directivas *#if*, *#elif*, *#else* y *#endif* que MISRA-C++:2008 no llega a permitir usar y el código de control de tracción contiene de manera generalizada para la distinción de las diferentes plataformas de compilación. Respecto a las reglas generadas de Caf Power & Automation, han aparecido también como violaciones de código al no haber sido tomadas en cuenta en la implementación del código.

Como ya se ha dicho, el recuento de errores y fallos en la clase *MultiTractControlCore* ha mostrado una cantidad elevada de violaciones del estándar MISRA-C++:2008 (*ver ilustración 27*), llegando a superar el millar de errores y llegando casi a las 500 alarmas:

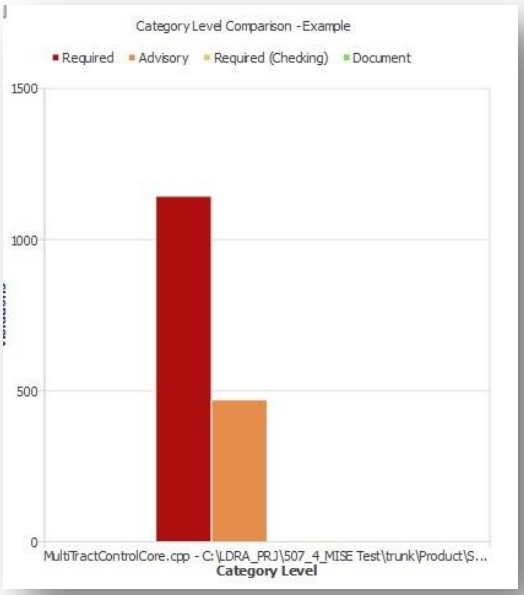


Ilustración 27 – Número de errores y alarmas en MultiTractControlCore.cpp

Entrando en más detalle, las violaciones del código se han podido diferenciar en distintas categorías de mayor a menor presencia, las cuales dan una idea de que muchos de estos errores han tendido a repetirse en diferentes procedimientos. Esto debería de facilitar la corrección sistemática de la gran parte de los fallos, siendo la mejor de las estrategias dejar los errores menos frecuentes para el último momento. La *tabla 5* muestra esta tendencia en una gran cantidad de normas.

Total Standards Violated - MultiTractControlCore.cpp	1614
Violated at Source Code Procedure level	**1292**
Violated Source Wide level	**163**
Violated at Header File level	**159**

Total Number of Unique Standards Violated - MultiTractControlCore.cpp	45
Unique Required Standards Violated	**43**
Unique Advisory Standards Violated	**2**

Standards Violated By Level - MultiTractControlCore.cpp	1614
Current Model (Example) Required Violated	1144
Current Model (Example) Advisory Violated	470

Frequency of Violated Standards - - MultiTractControlCore.cpp	1614
Basic type declaration used.	469
Caf rule 01 03 - Class identifier incorrect format.	154
No brackets to then/else.	149
Parameter should be declared const.	147
Literal value requires a U suffix.	119
Signed/unsigned conversion without cast.	109
No brackets to loop body.	74
Use of C type cast.	70
Procedure has more than one exit point.	67
Logical conjunctions need brackets.	61
Expression is not Boolean.	42
Use of mixed mode arithmetic.	21
Class data is not explicitly private.	20
DU anomaly, variable value is not used.	19
Narrower float conversion without cast.	15
Literal zero used in pointer context.	12
Use of banned function or variable.	6
Local variable should be declared const.	6
Function return type inconsistent.	5
Parameter should be declared * const.	4
Unused procedure parameter.	4
Unsigned integral type cast to signed.	4
Array passed as actual parameter.	3
Macro contains unacceptable items.	3
Member not declared virtual.	3
Type conversion without cast.	2
Use of bit operator on signed type.	2

Hexadecimal number found.	**2**
Macro parameter not in brackets.	**2**
(void) missing for discarded return value.	**2**
Found #if, #ifdef, #else, #elif .	**2**
Member function may be declared const.	**2**
Assignment operation in expression.	**2**
Expression needs brackets.	**1**
Equality comparison of floating point.	**1**
Scope of variable could be reduced.	**1**
More than one variable in declaration.	**1**
Else alternative missing in if.	**1**
Declaration does not specify an array.	**1**
Array has decayed to pointer.	**1**
Chacks filename size is <= 8 characters.	**1**
Caf rule 01 02 - Struct identifiers incorrect format.	**1**
Included file not protected with #define.	**1**
Use of function like macro.	**1**
Identifier is typographically ambiguous.	**1**

Tabla 5 – Violaciones de estándar

Como ya se ha expuesto en puntos anteriores, la revisión de un *Legacy Code* como es el caso, supone un esfuerzo muy costoso ya que el proceso de desarrollo del código fuente ha manifestado aspectos mejorables desde la etapa temprana del ciclo de vida del proyecto. La *tabla 6* refleja esta realidad mostrando cómo más del 95% de los procedimientos han llegado a manifestar algún tipo de violación del estándar.

Average Statistics - MultiTractControlCore.cpp	
Average Violation Per Function - Current Model (Example)	**9.06**
Average Unique Violation Per Function - Current Model (Example)	**0.24**
Percentage of Functions that Fail - Current Model (Example)	**95.80**

Tabla 6 – Media de violaciones de estándar

Informe del código

LDRA categoriza todos los fallos con un "Phase Code", la cual guía al usuario en el proceso de corrección del mismo. Esta opción despliega un *browser* que abre la descripción del error cometido al mismo tiempo que enseña un ejemplo aproximado del fallo generado. También asocia diferentes "Phase Code" de otros estándares a los que se puede acceder en caso de querer consultar una ayuda adicional.

En la *ilustración 28* se puede apreciar esta opción, la cual muestra el informe de código con la codificación de cada error y la solución del error *531 S* de MISRA-C++:2008. También se pueden apreciar los errores con codificación "Z" que corresponden a las reglas desarrolladas por el usuario. En este caso formadas por algunos ejemplos que ya venían desarrollados por defecto en LDRA (por ejemplo *5Z* y *2Z*) acompañados por reglas de la CAF como *8Z* y *9Z*. Este último es un error relacionado con el identificador de la clase la cual llega a fallar en 159 ocasiones. Es un buen indicador de que la corrección del código podría comenzar por ese punto.

Entrando en más detalle, se pueden encontrar diferentes tipos de codificación de las violaciones del estándar. Los *Phase Code* varían no solo en el identificador del error, sino también en la codificación que los acompaña. Aparte de los errores de los ya mencionados grupos "Z" (pertenecientes a las reglas personalizadas del usuario) y "S" (generados por el análisis estático), se puede apreciar un conjunto correspondiente a las variantes "C" por parte del análisis de complejidad, "D" a raíz del análisis estático de dato de flujo y "X" por la referencia cruzada. Los errores codificados con la letra "I" no aparecen ya que no se ha ejecutado la opción del análisis de flujo de información. Otros códigos representarían al informe del código ("Q"), *qualsys* ("U"), LCSAJ ("J") y las reglas húngaras ("H"), estas últimas también disponibles para ser desarrolladas por el usuario.

Paralelamente existe la posibilidad de aplicar más o menos fuerza a estas normas. El fichero de configuración de LDRA *Report.dat* ofrece cuatro niveles para configurar esta opción mediante codificaciones. "M" (norma obligatoria) que no permite en ningún caso que se viole la regla, "C" (norma obligatoria) que exige que la falta sea comprobada pero permite que se anule al mismo tiempo, "O" (estándar opcional) que en caso de error no afecta al resultado de

la ejecución aunque siga delante de manera condicional e "I" que solamente es un nivel informativo.

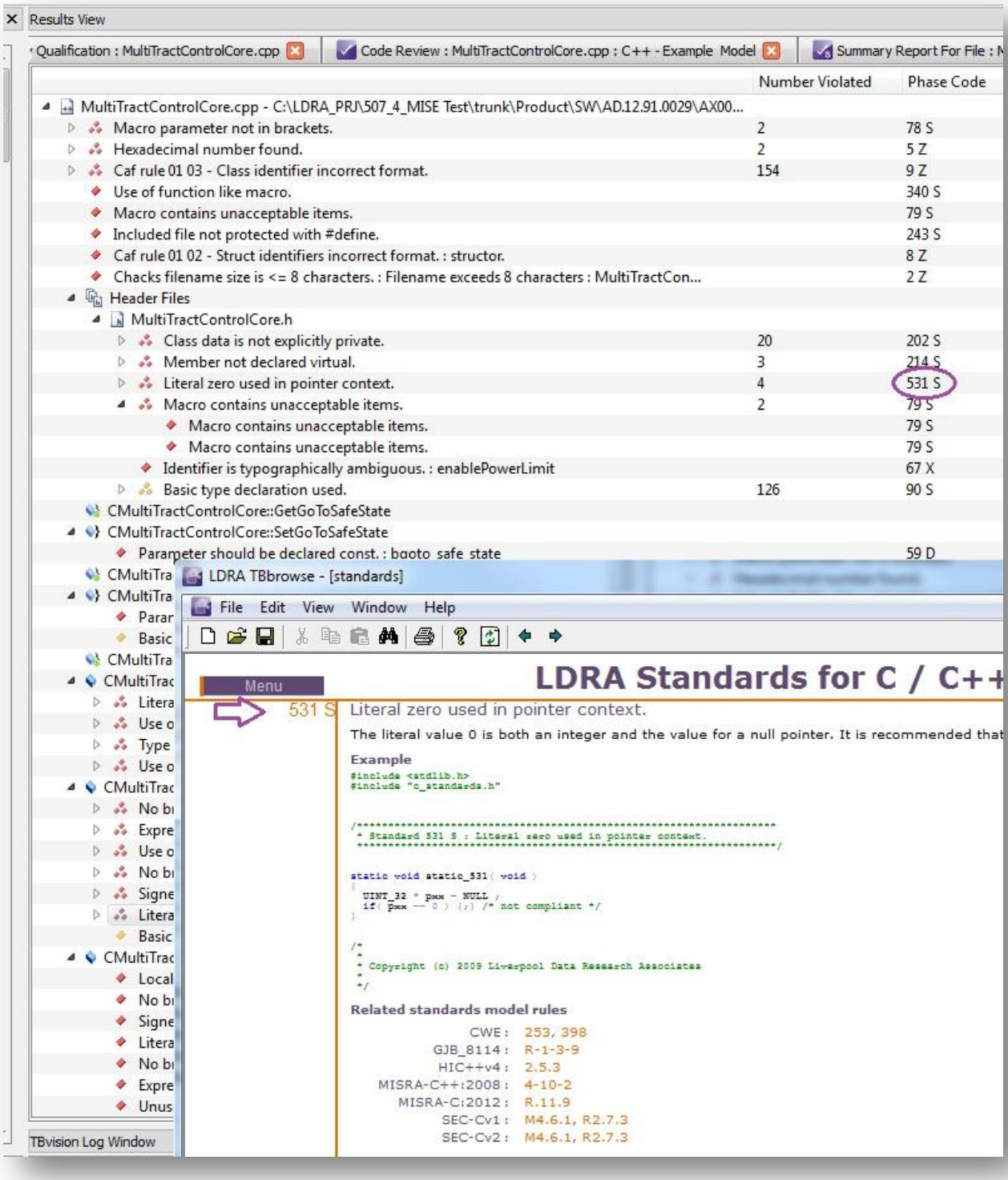


Ilustración 28 - Code Review y asistente de ayuda en LDRA

Calificación de fallos

LDRA TBvision dispone de diferentes recursos para visualizar opciones de calificación con distintas perspectivas del análisis estático, más allá del informe del código en sí. La opción de "Fault Qualification" (*ver ilustración 29*) muestra un informe centrado en las faltas cometidas en relación a tres niveles de gravedad:

- **Nivel 1:** Es el menos severo y se centra en las anulaciones o evitaciones (*Avoidances*).
- **Nivel 2:** Es un poco más restrictivo y muestra los defectos del código.
- **Nivel 3:** El nivel más duro que despliega la información de los fallos del código.

Como se puede apreciar en la *ilustración 29* de estos tres niveles solo se ha escogido el nivel 3 (*fault*) de la calificación de fallos.

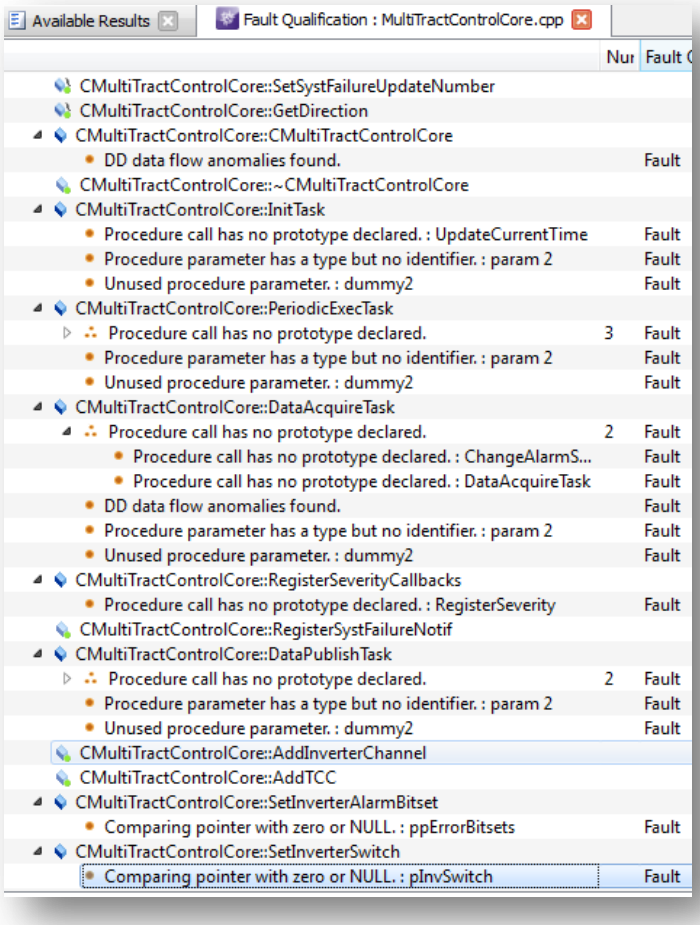


Ilustración 29 - Fault Qualification en LDRA

Calificación de seguridad

La calificación de seguridad (*ver ilustración 30*) es otro de los informes disponibles en TBvision, también dividido en tres niveles distintos acorde al nivel de intensidad en el que se quieran aplicar.

- **Nivel 1:** Bajo
- **Nivel 2:** Medio
- **Nivel 3:** Alto

En este proyecto se ha ejecutado el nivel 3 de seguridad y sin embargo no se ha contemplado gran cantidad de errores. Tal y como se observa en la *ilustración 30* la mayoría de ellos ha tenido que ver con la no designación de los Arrays como Unsigned. Sin embargo también se observan indicaciones de uso de funciones prohibidas por el estándar, como puede ser *"new"*.

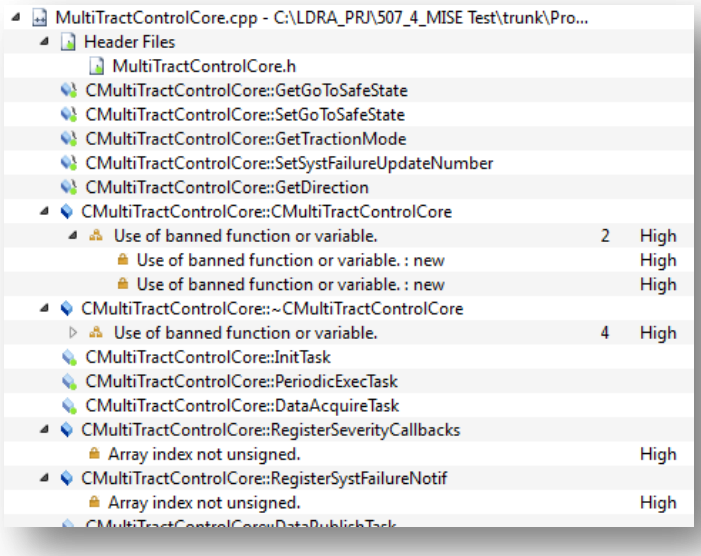


Ilustración 30 - Security Qualification en LDRA

Calificación de calidad

La calificación de calidad (*ver ilustración 31*) mide hasta qué punto nuestro código cumple con las propiedades de Portabilidad, Dependenciabilidad, Testabilidad, Mantenibilidad,

Complejidad y Estilo. Todos ellos disponibles en tres niveles de severidad al que se les ha aplicado el más estricto.

Los resultados han sido variados. La mayoría de los errores se han contabilizado por no cumplir con los mínimos de estilo. Por ejemplo, uno de los errores se ha manifestado porque el programa ha contenido código después del uso de una secuencia de caracter ";" en una misma línea.

Algunos errores sin embargo, han podido ser visualizados previamente en el informe del código (*ver apartado 6.1.4 – Informe del código*), dejando patente que ambas opciones contemplan a menudo el incumplimiento de un mismo tipo de fallo.

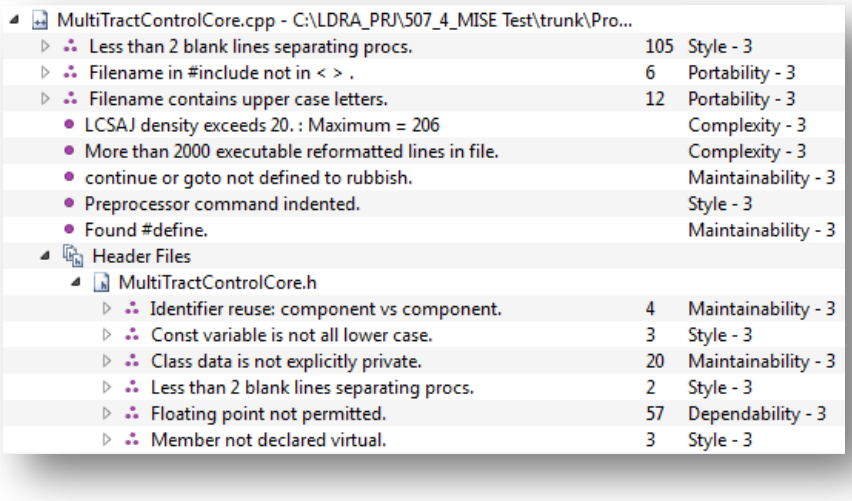


Ilustración 31 - Quality Qualification en LDRA

Informe de revisión de calidad

El informe de revisión de calidad (*ver ilustración 32*) da una visión instantánea sobre las medidas adoptadas para intentar medir diferentes aspectos del código fuente. El informe puede reflejar la métrica de un solo archivo, el sistema entero o una fuente no relacionada con el grupo de ficheros. Los resultados se pueden generar en formato ASCII o HTM y se les aplica un criterio de "PASS/FAIL" que puede ser configurado en las tablas de datos dentro de *metpen.dat*.

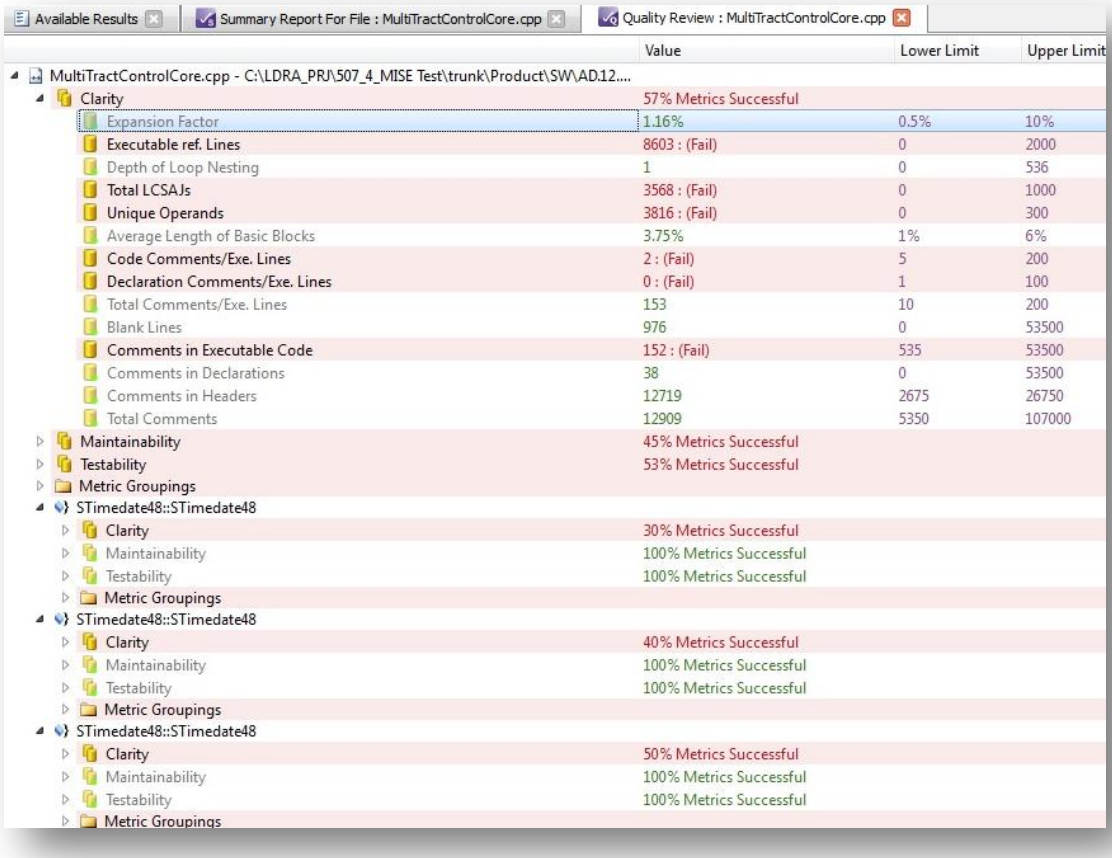


Ilustración 32 - Informe de calidad

Una vez ejecutado el análisis estático, se ha analizado el informe de calidad y los primeros datos han mostrado una tendencia de no cumplimiento en el campo de la **Claridad**. Este campo se centra mayormente en el número de comentarios que están establecidos en diferentes líneas del código fuente, tales como cabeceras, líneas ejecutables, interfaces de los procedimientos, etc. Sin embargo, analizando la clase *MultiTractControlCore* se puede observar que el número de citas comentando el código es relativamente alto. Puesto que las métricas establecidas en el campo de la claridad han estado basadas en el proyecto *CashRegister* de LDRA, con clases mucho más pequeñas, se ha llegado a la conclusión que las medidas programadas han sido demasiado estrictas, lo cual no quiere decir que el código fuente no sea claro. Por ejemplo, la cantidad máxima de operandos únicos está establecida en 300. Una cifra más que suficiente para una clase estándar del proyecto *Cashregister*. Sin embargo la clase *MultiTractControlCore* manifiesta una cantidad de operandos únicos de 3816 unidades. Pasa algo parecido con el número de referencias a líneas ejecutables, establecido en un máximo de 2000 y siendo superado por 8603 unidades.

En definitiva, se puede llegar a sacar una lectura de que la clase *multitractcontrolcore* es relativamente de fácil comprensión y necesita un ligero reajuste de las métricas para que su tasa de éxito aumente de manera considerable. Por otra parte, en lo que refiere a la claridad, no haría falta que la calidad llegase al 100% del cumplimiento, sino que se debería de buscar una tasa de consumación lo más elevado posible.

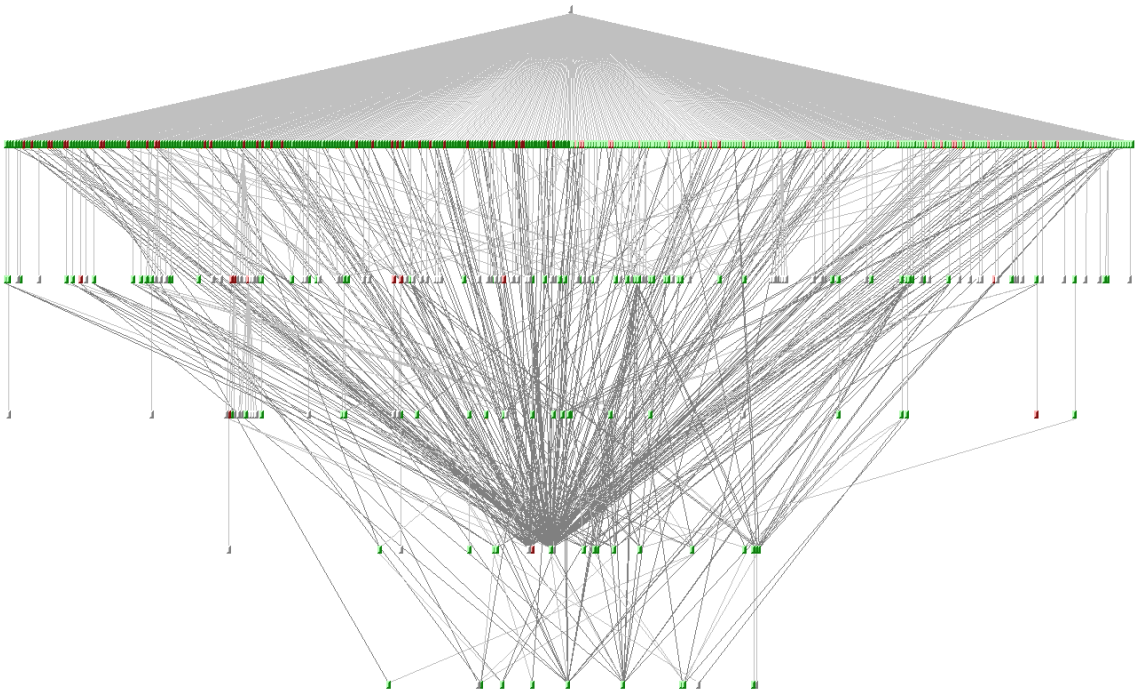
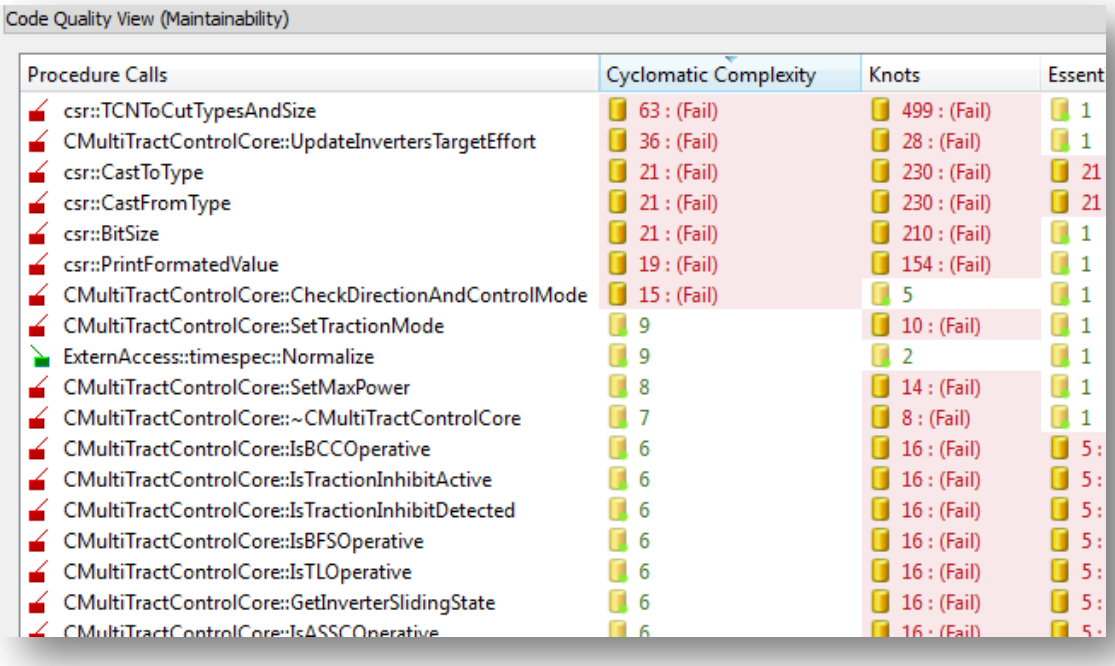


Ilustración 33 - Grafo estático de llamadas de la Mantenibilidad

De otra manera para el estudio de la revisión del informe de **Mantenibilidad**, se ha decidido consultar el grafo de llamadas (*ver ilustración 33*), pues permite visualizar el estado global de la clase de manera más efectiva. El propio grafo nos indica que la clase que estamos verificando contiene una estructura demasiado compleja. Está formado por 631 puntos de llamada, en las que se encuentran los procedimientos principales junto a funciones secundarias.

La mayor parte de los nodos se observan de color verde, la cual indica que han cumplido las métricas de mantenibilidad establecidas. Esto sería más que suficiente en el campo de la claridad, ya que se daría por hecho que la comprensión de la mayor parte del programa es satisfactoria. Sin embargo, si hablamos de lo mantenible que puede llegar ser un

programa, convendría revisar aquellos nodos que no han logrado pasar los criterios establecidos para el análisis estático. Para eso se ha accedido al menú de la vista de la claridad dentro del grafo de llamadas y se han ordenado los campos medidos de mayor a menor cumplimiento (*ver ilustración 34*) con el fin de visualizar los resultados de las llamadas que menos han cumplido las métricas de mantenibilidad. Si bien éstas representan una pequeña parte de todas las llamadas del sistema tal y como lo hemos podido observar en el grafo de llamadas, se puede apreciar un grupo de procedimientos que superan el límite máximo de complejidad ciclomática establecida en 10 (*ver apartado 5.2.1*). También se pueden observar nudos excesivamente grandes entre otro tipo de datos. Esto da a pensar que una parte del código, por muy pequeña que sea, es poco mantenible y que realizar cambios que podrían llegar a suponer la existencia de estructuras con grandes dependencias.


Code Quality View (Maintainability)

Procedure Calls	Cyclomatic Complexity	Knots	Essent
csr::TCNToCutTypesAndSize	63 : (Fail)	499 : (Fail)	1
CMultiTractControlCore::UpdateInvertersTargetEffort	36 : (Fail)	28 : (Fail)	1
csr::CastToType	21 : (Fail)	230 : (Fail)	21
csr::CastFromType	21 : (Fail)	230 : (Fail)	21
csr::BitSize	21 : (Fail)	210 : (Fail)	1
csr::PrintFormatedValue	19 : (Fail)	154 : (Fail)	1
CMultiTractControlCore::CheckDirectionAndControlMode	15 : (Fail)	5	1
CMultiTractControlCore::SetTractionMode	9	10 : (Fail)	1
ExternAccess::timespec::Normalize	9	2	1
CMultiTractControlCore::SetMaxPower	8	14 : (Fail)	1
CMultiTractControlCore::~CMultiTractControlCore	7	8 : (Fail)	1
CMultiTractControlCore::IsBCCOperative	6	16 : (Fail)	5 :
CMultiTractControlCore::IsTractionInhibitActive	6	16 : (Fail)	5 :
CMultiTractControlCore::IsTractionInhibitDetected	6	16 : (Fail)	5 :
CMultiTractControlCore::IsBFSOperative	6	16 : (Fail)	5 :
CMultiTractControlCore::IsTLOperative	6	16 : (Fail)	5 :
CMultiTractControlCore::GetInverterSlidingState	6	16 : (Fail)	5 :
CMultiTractControlCore::IsASSCOperative	6	16 : (Fail)	5 :

Ilustración 34 - Complejidad ciclomática en informe de mantenibilidad

Por último, se ha hecho uso del mismo tipo de informe para observar los resultados del análisis de **Testabilidad**. El grafo de llamadas, al igual que en el caso de la mantenibilidad, ha informado que la gran mayoría de elementos de la clase analizada han superado las métricas establecidas. Sin embargo, visualizando la lista de procedimientos y estableciendo una orden basada en los que más han fallado (*ver ilustración 35*), se ha observado un considerable grado de no cumplimiento en los puntos de salida de los procedimientos, el cual el máximo está establecido en “1”.

Code Quality View (Testability)

Procedure Calls	Fan Out	File Fan in	Procedure Exit Points
csr::TCNToCutTypesAndSize	0	1	64 : (Fail)
csr::BitSize	0	0	22 : (Fail)
CTriRedundant::GetValue	0	0	6 : (Fail)
TriRedundantAlgorithm	0	1	6 : (Fail)
CMultiTractControlCore::IsBCCOperative	4	0	5 : (Fail)
CMultiTractControlCore::IsTractionInhibitActive	4	0	5 : (Fail)
CMultiTractControlCore::IsTractionInhibitDetected	4	0	5 : (Fail)
CMultiTractControlCore::IsBFSOperative	4	0	5 : (Fail)
CMultiTractControlCore::IsTLOperative	4	0	5 : (Fail)
CMultiTractControlCore::IsASSCOperative	4	0	5 : (Fail)
CMultiTractControlCore::GetInverterSlidingState	4	0	5 : (Fail)
CMultiTractControlCore::GetSlidingSeverity	2	0	4 : (Fail)
CMultiTractControlCore::IsGlobalSpeedLimitationActive	2	0	4 : (Fail)
CMultiTractControlCore::IsLocalSpeedLimitationActive	2	0	4 : (Fail)
CMultiTractControlCore::GetInverterTrainSpeed	2	0	4 : (Fail)
CMultiTractControlCore::GetTCUTrainSpeed	2	0	4 : (Fail)
CMultiTractControlCore::GetCatenaryVoltage	2	0	4 : (Fail)

Ilustración 35 - Puntos de salida del procedimiento en informe de testabilidad

Un gran número de puntos de salida puede ser indicativo de una modularidad pobre y se intuye que es un factor con margen de mejora. La mayoría de esos procedimientos devuelven variables Booleanas mediante el uso de 4 o 5 funciones de retorno. El proyecto *Cashregister* de LDRA establecía el número máximo de "Procedure Exit Points" en 1 y esta sería la manera correcta de proceder, reestructurando el código de algunos de los procedimientos en *MultiTractControlCore*.

6.2 PRUEBA UNITARIA

En este proyecto se han llevado a cabo una serie de pruebas unitarias con la clase *MultiTractControlCore* y posteriormente se ha procedido a una prueba de integración de éste con las clases *TractControlCore* y *AlarmControl*. En estos últimos casos solo se ha comprobado la puesta en marcha de la secuencia de test, sin llegar a entrar en un proceso de validación.

La prueba unitaria de la clase *MultiTractControlCore* se ha basado en aislar dicha clase del sistema completo, por lo que han surgido una serie de problemas de dependencia al intentar ponerla en marcha. Esto se ha solucionado con la creación de una unidad de

secuencia y la adaptación de de la misma al código que se ha querido ejecutar, es decir, se han insertado líneas de código adicionales y se han usado STUBs dentro de la unidad de secuencia adherida a la clase para solucionar los problemas de dependencia, sin haber alterado la integridad del código fuente original.

6.2.1 Creación y puesta en marcha de la unidad de secuencia de los test

Para esta tarea ha sido necesaria una ejecución anterior del análisis estático y la instrumentación de la clase, ya que se ha decidido ejecutar test tanto en caja negra como en caja blanca.

En la creación de la secuencia de test, la herramienta de LDRA ha creado de manera automática un fichero (.cpp) que ha establecido una base para la ejecución de los test. Este archivo hace una llamada al código fuente que se quiera testear y contiene también los ajustes realizados para aislar la clase en el test unitario.

Adaptación de la unidad aislada a problemas de dependencia

Al realizar una prueba unitaria de una clase que a su vez necesita funciones y variables implementadas en el resto del sistema, se generan varios conflictos de dependencia que derivan en errores de compilación. Para solucionar estos errores existen diferentes estrategias a seguir. La primera es la de adaptar el código fuente a las necesidades de la prueba unitaria (aunque pueda ir en contra de los principios básicos del testing), modificando así el código que se quiera validar mediante los test. En este proyecto se ha tomado en parte este camino en el uso de la plataforma *Integrity* al no ser suficientes los conocimientos de uso de los recursos que ofrece TBrun para solucionar estos problemas de dependencia. Sin embargo, ésta no es la mejor de las opciones ya que al alterar el software se deja de verificar el sistema tal y como es en realidad.

Una de las opciones alternativas que ofrece LDRA es el uso de los STUBs, que no son más que recursos para la gestión de trozos del código con el fin de adaptarlos al entorno de la prueba unitaria. Un uso que se le ha dado a un STUB ha sido la de *checkear* un procedimiento como “CORRECTO”, que si bien causaba algún que otro error, no estaba prevista su validación

en la secuencia de test correspondiente. Con esta opción simplemente se le ha dicho a TBrun que no se ha querido tomar en cuenta esa funcionalidad, con lo cual ha dejado de ser atendida en el código de la secuencia de test y por consecuente, ha dejado de manifestar un error.

Otro uso que se le ha dado al sistema de STUBs ha sido el retorno de valores de funciones que no han estado contempladas en la clase verificada. Por ejemplo, en el caso del constructor existe una función que espera un valor de un semáforo en un punto condicional, al que se le ha aplicado un retorno de un valor personalizado en un caso de test concreto (*ver ilustración 36*).

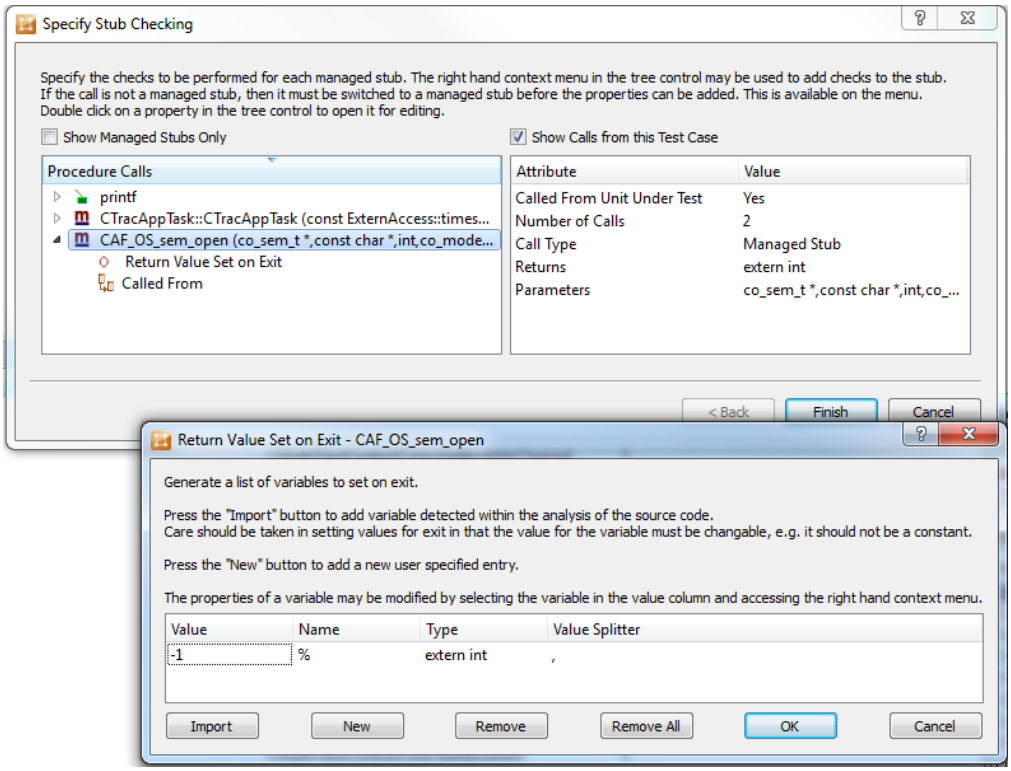


Ilustración 36 - Uso de STUB para retorno de valor

Al margen del uso de STUBs, en TBrun se permite la inserción de código adicional en la unidad de test, antes de que se le llame a la clase que se quiera verificar. De esta manera puede declararse las variables o funciones que hagan falta para el código que se vaya a validar sin que ésta sea modificada. Por ello existe una estrecha relación entre la unidad de test y la clase que se importa para la aplicación de test.

La inserción de código adicional a la unidad de test se realiza mediante un editor (ver *ilustración 37*) que posee la ventaja de no tener que generar el análisis estático cada vez que se quiera hacer un cambio. Esto es así porque en ningún momento se llega a modificar el código fuente de la clase *multitractcontrolcore*.

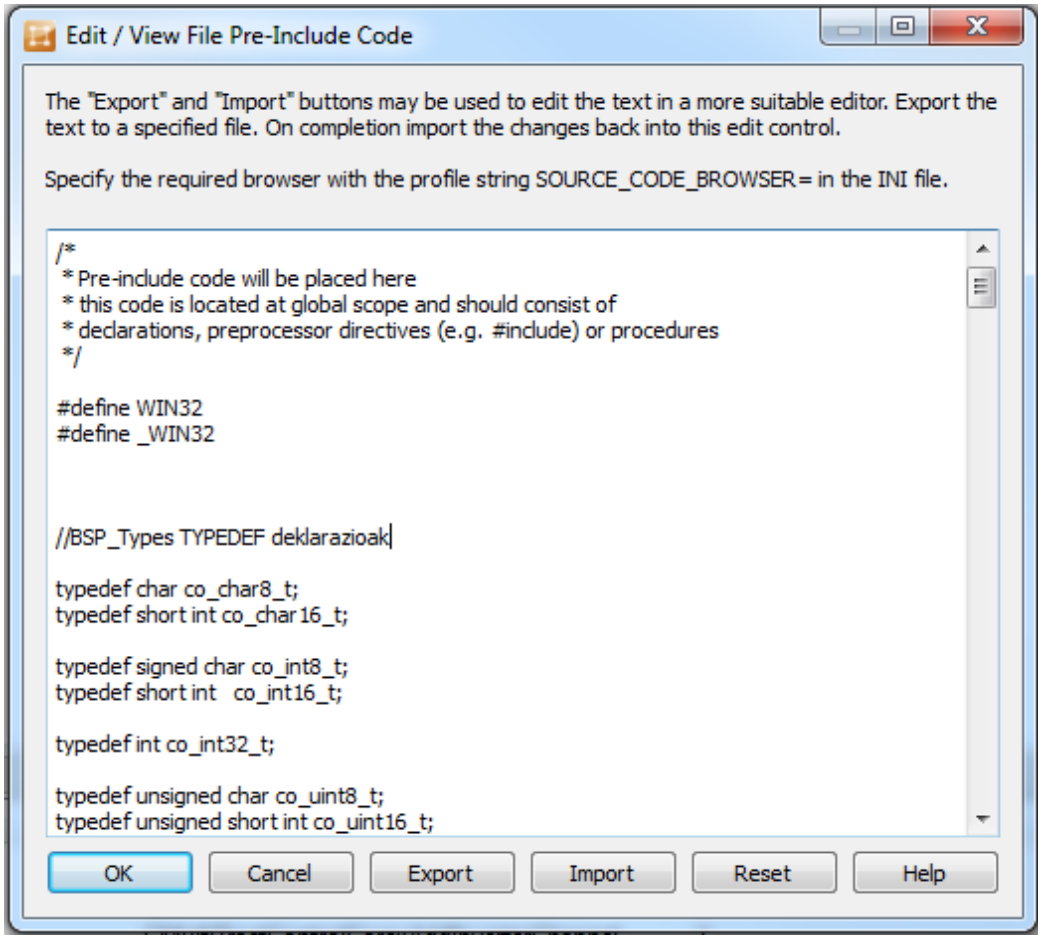


Ilustración 37 - Código pre llamada a la clase a verificar insertado en la secuencia de test

La versión del proyecto que se ha ejecutado en la plataforma Visual Studio 2010 x86 se ha adaptado a la unidad de test mediante estas técnicas por lo que se puede decir que en esta ocasión sí se ha llegado a validar el sistema de forma íntegra. Al contrario que en la versión testeada en la plataforma Integrity, ya que si bien se le han aplicado la mayor cantidad posible de usos de STUB e inserciones de código, se le ha tenido que modificar partes del programa original para poder ser compilado en el entorno de GreenHills.

La *ilustración 38* desarrolla la relación de la secuencia de test con la clase *multitractcontrolcore*, tanto con la versión de caja negra como la caja blanca (instrumentada).

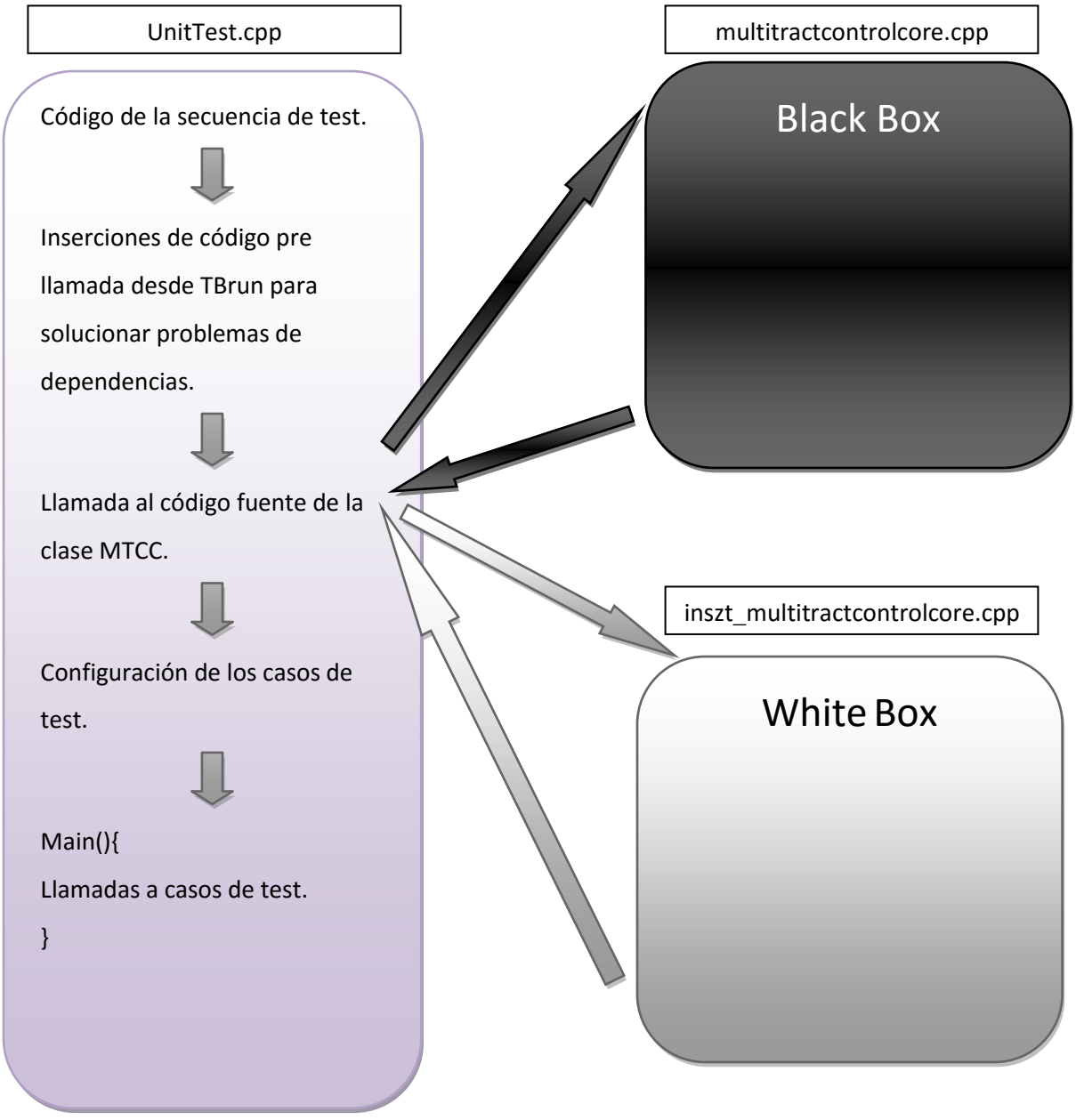


Ilustración 38 - Funcionamiento del código de la secuencia de test

6.2.2 Creación de casos de test

La secuencia de test debe de mantener una determinada orden si el sistema se quiere validar adecuadamente. El primer test debe de ir dedicado al constructor, seguido de las inicializaciones de todos los elementos que se vayan a utilizar en las pruebas posteriores. Por ejemplo, si se va a tener que hacer uso de un vector propio en un determinado caso de test, ese vector va a tener que ser creado e inicializado en una prueba anterior mediante la validación del procedimiento que verifique dicha tarea.

De esta manera en este proyecto se han creado casos de test para dos procedimientos en particular: *RegisterSeverityCallbacks* y *StartInverters*. Esta secuencia de test es la misma para las pruebas de caja negra como para las de caja blanca ya que LDRA permite esta práctica. Muchos de los test diseñados para caja blanca, se pueden considerar suplementarios para las pruebas de caja negra (puede llegar un punto en el que sean redundantes). A la hora de ejecutar la secuencia de test y para gestionar esta particularidad, TBrun permite suspender los test que se quieran, siempre que no afecten a las pruebas que se vayan a ejecutar en algún punto posterior de la secuencia.

Para las pruebas de caja negra se han tomado en cuenta las interfaces de los procedimientos, simulando el test con valores de entrada y salida. Se ha conseguido testear diferentes valores de entrada descritas en las interfaces al mismo tiempo que se han obtenido todos los valores de retorno que están establecidas en cada procedimiento. Por ello, todos los casos de test para las pruebas de caja negra han resultado satisfactorios.

En referencia a las pruebas de caja blanca, no solo se han tomado en cuenta los valores de entrada y salida de cada método, sino que también se ha analizado el comportamiento interno de las funciones. Con ello se han diseñado los casos de test con el objetivo de abarcar todo el contexto funcional de procedimiento. Se ha intentado diseñar los test para comprobar todos los nudos condicionales, en todas las combinaciones necesarias para validar el 100% del código.

Un buen ejemplo de esto se puede apreciar en la *ilustración 39* donde se ha decidido crear un test para la prueba de caja blanca, siendo el objetivo validar la ejecución de la segunda condición de dicho ejemplo. Se parte de que el vector *m_pTCCs* está inicializado en un test anterior por lo que su longitud se considera de "1". Al mismo tiempo se le aplica a la variable de entrada *id* un valor también de "1", cumpliendo así el requisito del segundo nudo condicional que deriva en la devolución de un entero con valor "-1" (un error). Como se puede apreciar, para verificar que este test haya pasado satisfactoriamente ha tenido que ser establecida un valor de salida de "-1". El objetivo del caso de test de *StartInverters* en este modo de caja blanca es la de aplicar cuantos test hagan falta para validar el 100% de las líneas de código del procedimiento y de esta manera conseguir ejecutar el 100% de la prueba de cobertura (tanto *Branch* como *Statement*). Esto no es necesario implementarlo en las pruebas de caja negra, ya que solo se tiene que validar el contexto referente a las interfaces.

```
729  ////////////////////////////////////////////////////////////////////////////////
730  // FUNTZIOA - FUNCIÓN: StartInverters
731  // SARRERAK - ENTRADAS:
732  //  Ninguna
733  // IRTEERAK - SALIDAS:
734  //   int: retorna -1, en caso de error, y 0 en caso de OK.
735  // AZALPENAK - COMENTARIOS:
736  // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
737  //   Encendido de los Inversores ( en caso que se tenga haya registrado el
738  //  controlador). Tras encender las ICU da comienzo de la comunicaciones. Si
739  //  todo es correcto el sistema alcanza el estado IDLE en cada ICU, tras
740  //  haberlas configurado.
741  ////////////////////////////////////////////////////////////////////////////////
742  int CMultiTractControlCore::StartInverters(const ExternAccess::timespec& time,int id)
743  {
744     // Actualiza el tiempo transcurrido
745     if(m_pTCUErrorMan) m_pTCUErrorMan->UpdateCurrentTime(time);
746
747     if(id >= (int) m_pTCCs->size() || id < -1) return -1; //ERROR
748     if (id < 0)
749     {
750        unsigned int i;
751        for (i=0;i<m_pTCCs->size();i++) (*m_pTCCs)[i]->StartInverter(time);
752     }
753     else (*m_pTCCs)[id]->StartInverter(time);
754     return 0;
755  }
```

Variable I/O View

Value	Name	Type	Use
I ldra_qq_lv_13	time	ExternAccess::timespec &	Input parame
I 1	id	int	Input parame
I TPTCCVt_globala	m_pTCCs	CMultiTractControlCore::TPT...	Input membe
I *** Value Retained ***	m_pTCUErrorMan	CTSharedObject < CAlarmCo...	Input membe
O -1	%	int	Function resu

Ilustración 39 - Configuración de variables de E/S para Test

6.2.3 Ejecución de la prueba unitaria en VS10

Las pruebas en Visual Studio 2010 se han ejecutado sin mayor complicación. Los test en modo de caja negra han generado sus reportes y han desplegado un mensaje de que el test ha llegado a pasar de forma correcta. Sin embargo, las pruebas en modo de caja blanca han tenido que llegar a validar todo el funcionamiento interno que describe el código. Debido a esto, han supuesto ser mucho más costosas desde el punto de vista de recursos computacionales y han exigido un tiempo de ejecución considerablemente mayor.

6.2.4 Ejecución de la prueba unitaria en Integrity

Como es de suponer, los Test en Integrity se han realizado de una manera distinta a la plataforma de Visual Studio 2010. La ejecución se la iniciado de la misma forma, pero una vez en marcha se ha comenzado el proceso de depuración *On-Target*, abriendo el proyecto adherido al fichero (.gpj) que configura la unión de TBrun y GreenHills. Es en ese momento cuando se ha procedido a depurar la secuencia de test en la IDE MULTI (*ver ilustración 40*) en vez de hacerlo en el Host (PC), como ha sido el caso de Visual Studio 2010.

Para este proceso, se ha tenido que pre-establecer la conexión entre el Rack VEGA y el PC por cable Ethernet y retocar ficheros de configuración dedicadas a gestionar la conexión entre LDRA y el rack en el momento de lanzar los test.

Una vez concluida la secuencia de test, la ejecución ha vuelto a TBrun donde le han esperado los resultados obtenidos. Una peculiaridad de la puesta en marcha de la prueba en MULTI ha sido que la IDE ha llegado a abrirse y posteriormente cerrarse en cada una de las ejecuciones de los casos de test existentes, en vez de haberlo hecho todo de una sola vez.

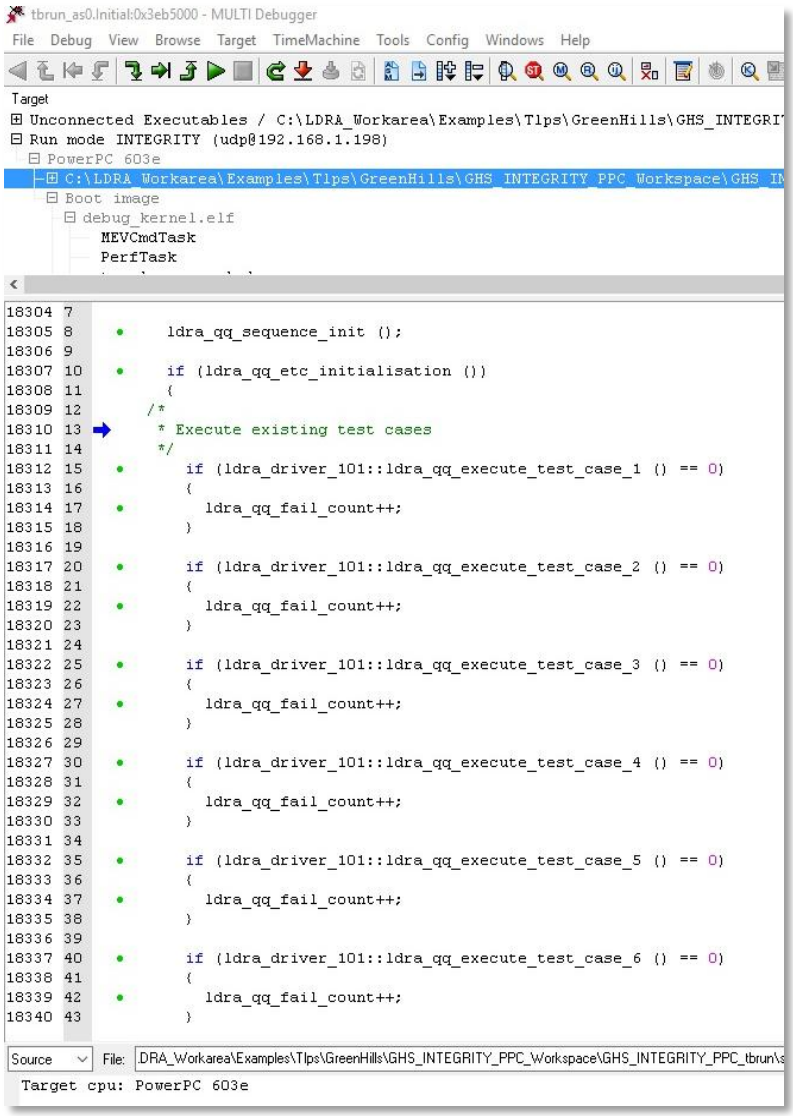


Ilustración 40 - Llamadas a casos de test en MULTI

6.2.5 Resultados de la prueba unitaria

Cada uno de los 18 test tanto en modo de caja negra como caja blanca han conseguido pasar satisfactoriamente, aunque solo han llegado a alcanzar una pequeña parte de la cobertura total de la clase *MultiTractControlCore* (*ver ilustración 41*). En cuanto a cada procedimiento, el constructor se ha ejecutado con el 100% de la cobertura gracias al segundo test con el valor de retorno establecido con el STUB (*ver ilustración 36*). Las pruebas de inicialización de vectores *AddTCC* y *AddInverterChannel* tampoco han supuesto un problema y han conseguido obtener el 100% del "Statement Coverage" (no se les ha calculado el "Branch Coverage" al no tener nodos condiciones que deriven en algún tipo de rama).

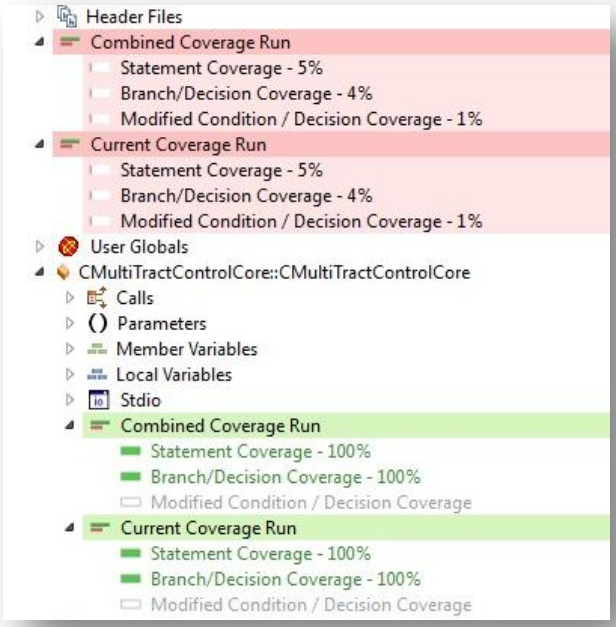


Ilustración 41 - Resultado de la cobertura total y cobertura del constructor

Por último, los dos procedimientos principales *RegisterSeverityCallbacks* y *StartInverters* han conseguido un gran porcentaje de cobertura, pero no han llegado al 100% de la misma. Para analizar este resultado se ha decidido consultar el grafo de flujo.

En el caso de *RegisterSeverityCallbacks* nos hemos encontrado con un "Statement Coverage" del 79% lo cual quiere decir que no todos los nodos han llegado a ejecutarse. El grafo de flujo muestra esta realidad tal y como se puede ver en la *ilustración 42*. Parece ser que el nodo condicional número 7 no se ha testeado en todas sus vertientes y el problema la ha generado un vector llamado *m_pTCUErrorMan*. La solución pasaría por implementar un test sobre *RegisterSeverityCallbacks* que contemplase el hecho de que éste vector estuviera inicializado y para ello debería de haber una prueba anterior en la secuencia de test que cumpliera tal labor. El problema está en que la clase *MultiTractControlCore* no posee un procedimiento para la inicialización del vector *m_pTCUErrorMan*, al contrario de los vectores *AddTCC* y *AddInverterChannel* que sí han podido ser verificados.

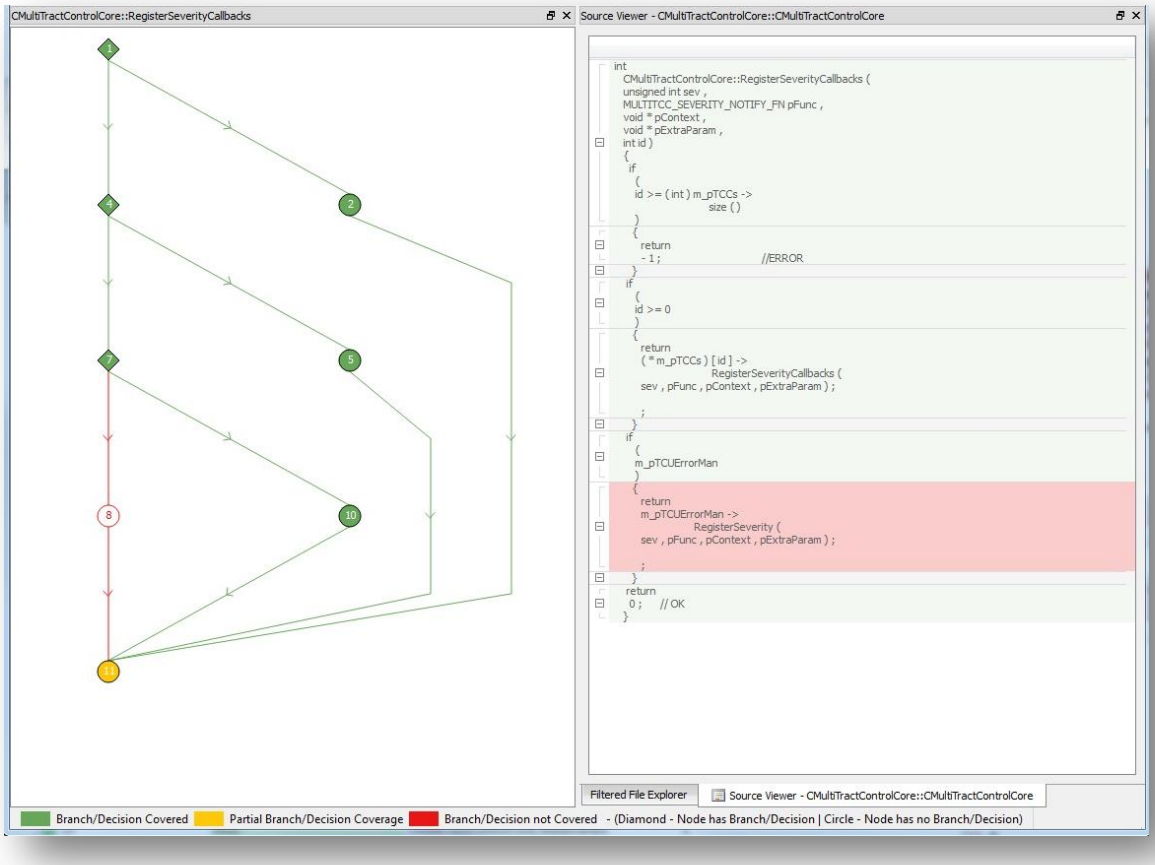


Ilustración 42 - Grafo de flujo de RegisterSeverityCallbacks

En el caso del procedimiento *StartInverters* nos encontramos con un caso parecido, de nuevo una condición dependiente del vector *m_pTCUErrorMan* nos impide superar el "Statement Coverage" del 90% y el "Branch Coverage" del 92%.

Lo primero que se puede deducir de estos resultados es que el código no es del todo testeable. Es probable que la inicialización de *m_pTCUErrorMan* se contemple en alguna otra clase, sin embargo esa implementación no se ha contemplado en esta prueba unitaria. Esto se podría haber solucionado desarrollando nodos condicionales con valores booleanos, las cuales se podrían haber modificado en un test mediante el uso de STUB.

TBrun nos permite visualizar grafos de flujo con resultados tanto de las pruebas unitarias como del análisis estático (de manera retroactiva) llevado a cabo con TBvision ya que todas las herramientas están relacionadas en torno al motor de LDRA Testbed. En la *ilustración 43* se observa cómo el informe del estándar mediante el grafo de flujo nos indica un error con

"PHASE code" *114S* de MISRA-C++:2008, referente a que una condición debe de ser de naturaleza Booleana. Esto ya nos da una pista de que no hemos ejecutado los pasos previos necesarios antes de crear pruebas unitarias (aunque como ya se ha dicho, la corrección del código mediante el análisis estático no entraba en los objetivos de este proyecto).

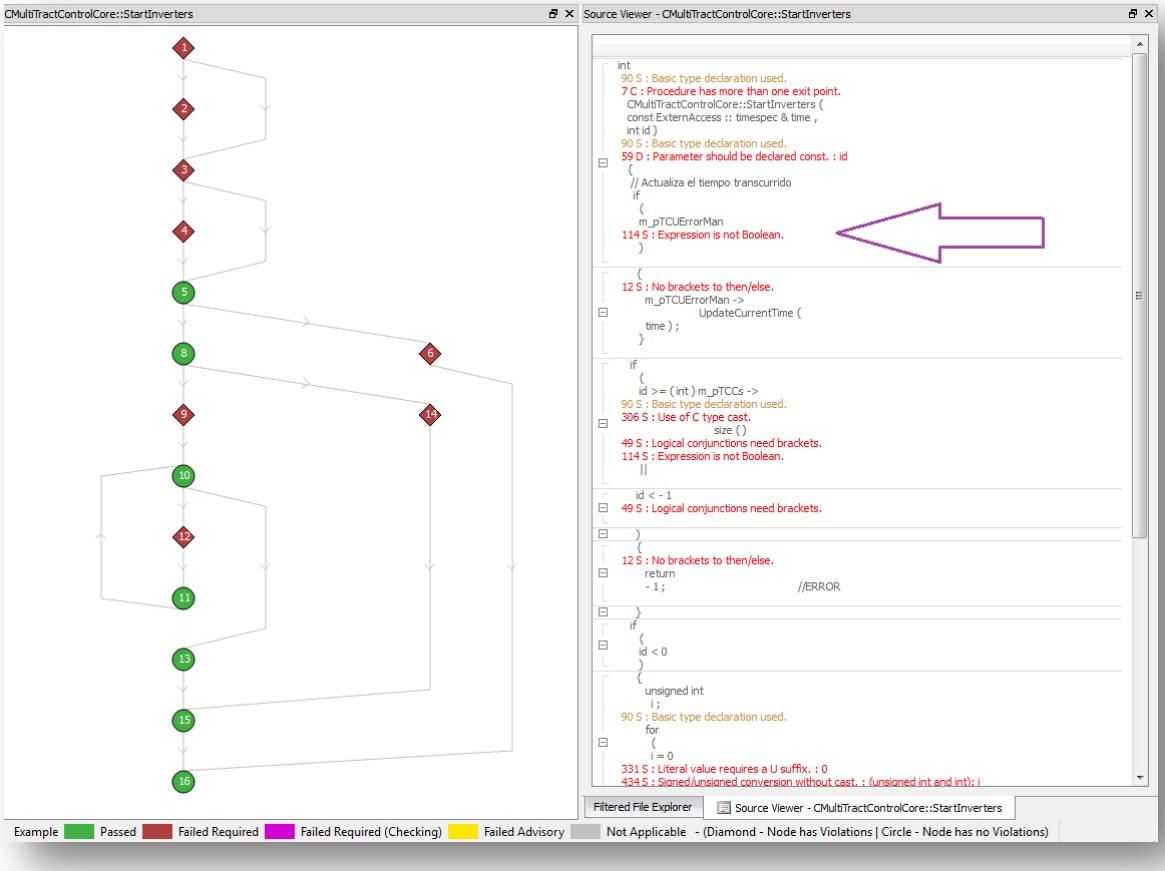


Ilustración 43 - Grafo de flujo de StartInverters (Informe de estandar)

Otra forma de resolver este problema puede ser la de corregir la imposibilidad de que el vector *m_pTCUErrorMan* acceda a la llamada en la otra clase. Para saber por dónde empezar con este método, TBrun proporciona un tipo de grafo llamado grafo de acoplamiento de control. En él se muestra de una manera aislada la relación del procedimiento requerido con las llamadas fuera de la prueba unitaria. En la *ilustración 44* se observa de manera clara cómo *StartInverters* no consigue comunicarse con otros nodos porque o bien aparecen como "no resueltos" (en blanco) o han fallado y no se han ejecutado (en rojo). Estos nodos proporcionan información de la clase origen del nodo al igual que su posición en el código reformado. En consecuencia, quedaría demostrado que para este caso, la prueba unitaria es insuficiente por

lo que la primera medida a tomar debería de ser la de una prueba de integración con las clases dependientes de los nodos que fallan.

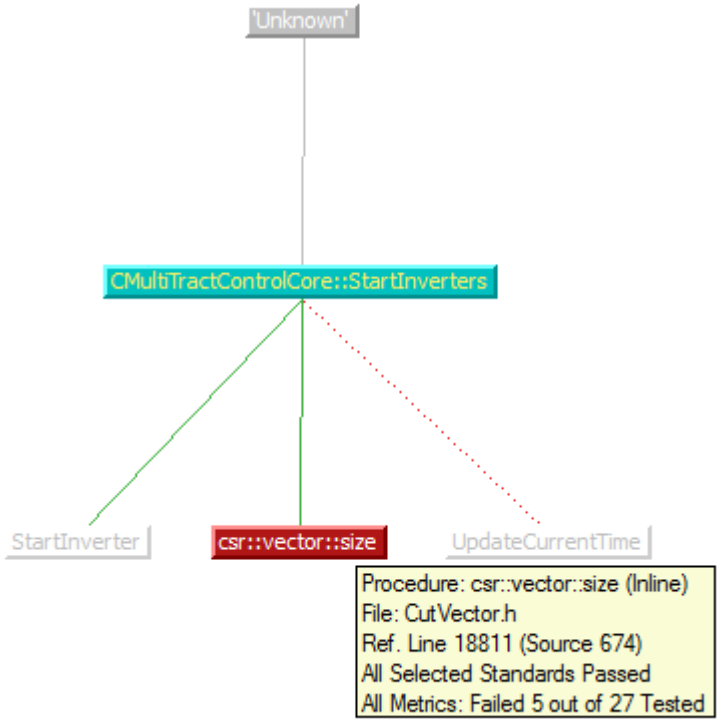


Ilustración 44 - Control coupling graph de StartInverters

6.2.6 Importación y Exportación de secuencias de test

A menudo una secuencia de pruebas de test puede servir para ser reutilizada por distintas fuentes. En el caso de LDRA se da la opción de exportar secuencias enteras para que otros usuarios puedan ejecutar unas pruebas ya implementadas.

El fichero exportado tiene un formato TCF y es un archivo que no solo contiene la información necesaria de las pruebas, sino que también abarca toda la configuración de los STUB e inserción de código en la secuencia de test que se ha desarrollado para la adaptación de la prueba unitaria a la clase que se ha separado del sistema completo. Esto es muy importante, ya que en muchos casos la complejidad de generar esta prueba unitaria supone invertir una gran cantidad de tiempo que llega a superar ampliamente la duración del proceso de diseño de las pruebas.

6.2.7 Ejecución de la unidad de test por *Command Prompt* y creación de un ejecutable

Para la creación de la secuencia de test es necesario el uso del entorno de TBrun y con ella la licencia de LDRA. Sin embargo el fichero que abarca la configuración de la unidad puede ser reutilizado sin el uso de ninguna herramienta. La idea parte de que cada vez que se modifique la clase que se esté validando, se compile la secuencia de test que contiene la implementación de los diferentes casos de test, y se cree un ejecutable para realizar test de regresión. Esto se ha llevado a la práctica con *MultiTractControlCore* y una de las secuencias generadas.

7. INTEGRACIÓN CONTÍNUA CON JENKINS

La integración entre LDRA y Jenkins se ha llevado a cabo mediante ficheros BATCH que contienen toda la funcionalidad de las propias herramientas de TBvision y TBrun ejecutadas por línea de comando.

La primera tarea a realizar para ejecutar estos scripts en Jenkins ha sido la de crear un servicio de un nodo esclavo que se ha encargado de todo el proceso de automatización de pruebas de V&V. El nodo esclavo se ha configurado de tal manera que se mantenga como proceso activo de forma permanente. De esta manera se mantiene a la espera hasta la solicitud de su ejecución. Para ello se ha generado un BATCH previo que activa dicho proceso en el sistema al inicio de cada sesión de Windows. A continuación se ha implementado la funcionalidad de LDRA en el BATCH principal.

Los comandos ejecutados en este script se han fundamentado en el proceso de importación de "un contexto" de una secuencia de test recopilado en un fichero TCF. Por ello la ejecución de una secuencia en Jenkins ha supuesto llevar a cabo un primer ciclo de validación y verificación en los entornos de TBvision y TBrun. De esta manera, previamente ha debido ser exportado la secuencia completa para su uso en Jenkins.

La funcionalidad del Script ha consistido en los siguientes puntos:

- Definición de rutas para el sistema de ficheros
- Definición de fichero TCF
- Preparación del entorno para generar reportes
- Configuración de ficheros de métricas y estándares
- Ejecución de análisis estático
- Ejecución de Test
- Generación de resultados

Una vez concluida la prueba, los reportes han sido enviados al espacio de trabajo de Jenkins en formato HTM. Al mismo tiempo, se ha copiado un fichero concreto generado en el área de trabajo de LDRA y se ha copiado en el área de trabajo de Jenkins. El fichero en cuestión es un Report simple escrito es ASCII (*ver ilustración 45*) que informa de manera breve del

resultado de la ejecución de la secuencia de test. El motivo de haberlo copiado en el área de trabajo de Jenkins ha sido el poder enviarlo mediante correo electrónico al usuario que ejecuta el test, para así poder poseer la información de si los test han pasado o no al instante. La idea es que una vez el usuario tenga un primer contacto con el resultado de la ejecución, decida si quiere consultar el informe completo en formato HTM (*ver ilustración 46*) accediendo al repositorio.

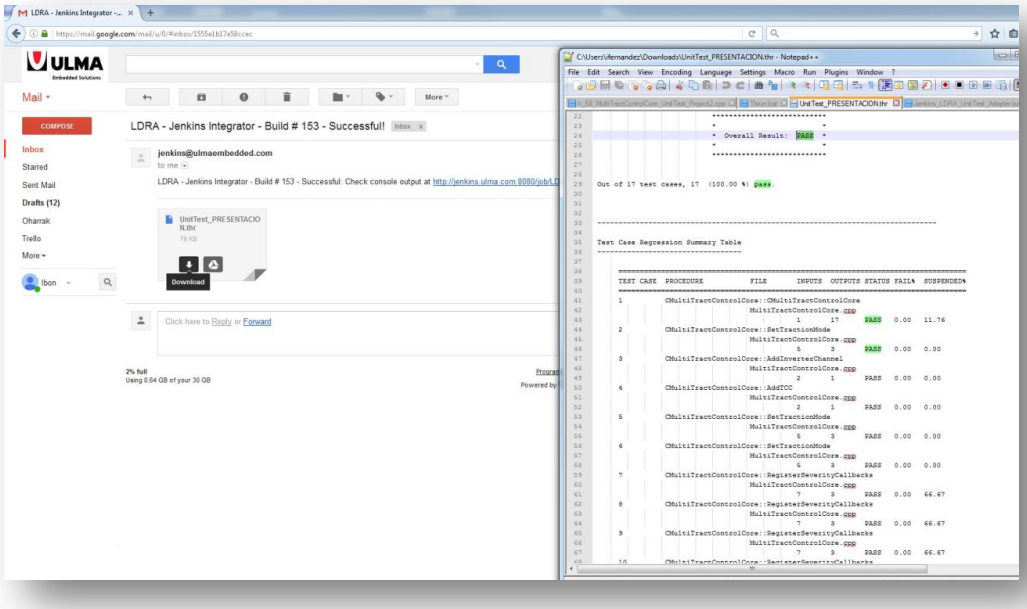


Ilustración 45 - Aviso de ejecución de Jenkins vía email

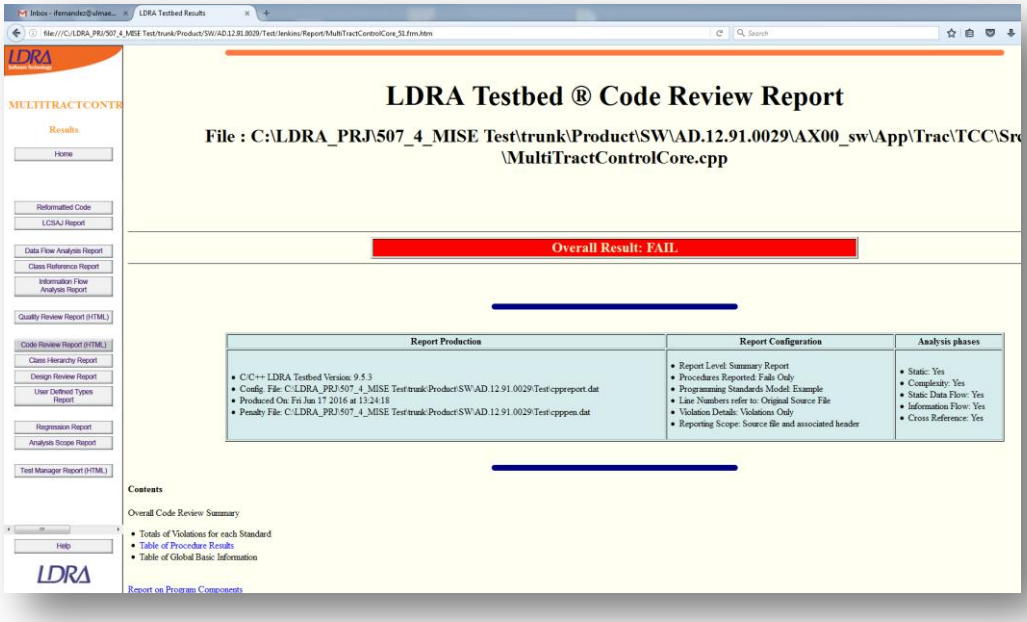


Ilustración 46 - Entorno de informes de LDRA en HTM

8. CREACIÓN DE ENTORNO ÚNICO DE TRABAJO

Con el fin de recopilar todo el funcionamiento que se ha expuesto anteriormente en este informe, se ha procedido a crear un directorio llamado TEST dentro del sistema de ficheros del código de tracción de CPA. En él se han depositado todas la configuración y secuencias generadas por un usuario de LDRA y Jenkins, compartiendo así su trabajo con personas que pueden hacer uso del mismo sin repetir la tarea completa de desarrollo o tener conocimientos avanzados de dichas herramientas.

La composición de este directorio es la siguiente:

- Ficheros de configuración de métricas y estándar: cpppen.dat, cppreport.dat, Sysearch.dat, Userstandards.c, Userstandards.exe y fichero de para macros sysppvar.dat.
- Archivos de secuencia de test en formato .cpp, .exe y .obj.
- Archivos del contexto de secuencias de test y configuración de pruebas unitarias .TCF.
- Archivos BATCH de arranque del servicio de nodo esclavo y ejecución de comandos.
- Código instrumentado.
- Un Archivo para añadir STUBs antes de instrumentar el código.
- Directorio de ejecución de secuencias de test por comando.
- Área de trabajo de Jenkins.
- Directorio con guías de uso del entorno de trabajo.

9. CONCLUSIONES

El proceso de validación y verificación de la clase *MultiTractControlCore* supone un esfuerzo muy costoso. Un código más simple y una clase de menor tamaño ahorrarían recursos computacionales y tiempo ya que LDRA es una herramienta que exige un hardware robusto, sobre todo para generar análisis estáticos en modo MCDC, pruebas unitarias en modo caja blanca y generar grafos de flujo.

Sin duda **conviene aplicar el proceso de análisis de verificación desde el comienzo del desarrollo del proyecto**. Especialmente la aplicación del análisis estático en una etapa temprana del desarrollo supondría un gran avance en el campo de la testabilidad del software.

Existe una gran cantidad de errores en el informe del código que informan del no cumplimiento del estándar MISRA-C++:2008 y las reglas de CPA. Tanto es así que la situación exige una estrategia a la hora aplicar soluciones o mejoras, siendo necesario centrarse primero en las violaciones del código que más tiendan a repetirse.

En definitiva se puede concluir que LDRA ayuda en el testeo de **Legacy Code** complejo, pero exige un conocimiento de la herramienta para un tipo de usuario avanzado. Se deduce que el periodo de aprendizaje de las diferentes funcionalidades de TBvision y TBrun conlleva una duración de varios meses, por lo que tiene que valorarse si el código a validar y verificar va a compensar dicha inversión de recursos. La herramienta ofrece ayudas en proyectos orientados al Safety o *Mission Critical software*, con miras a certificaciones y diferentes grados SIL. En el caso del código genérico de tracción queda claro que el trabajo a realizar es de una magnitud considerable.

Por último, el uso de entornos de integración continua como Jenkins en colaboración permanente con software de análisis como LDRA, multiplica de manera notable las posibilidades de uso y beneficios del proceso de validación y verificación.

10.REFERENCIAS

ULMA Embedded Solutions: *http://www.ulmaembedded.com/en/index.php*

LDRA Testbed User Guide

TBrun User Guide

TBvision User Guide

Using the LDRA tool suite with GreenHills MULTI 6 and INTEGRITY

Using the LDRA Tool Suite with Jenkins

LDRA_Implementing_EN 50128_2011

UNE-EN_50128=2012(EN)_commented – Railway applications Communication, signaling and processing systems Software for railway control and protection systems

LDRA Tool Training V9.4.x Workshops

Embedded Software Testing (Feabhas)

T-101 Verification and Validation for Managers Manual v1.1 (Feabhas)

11.AGRADECIMIENTOS

Eskerrik beroenak bidali nahi dizkiet proiektu hau aurrera eramaten lagundu didaten lankide guztiei.

El mayor de los agradecimientos a los compañeros que me han ayudado a sacar adelante este proyecto.

Todos "profesionales altamente cualificados".